Sage Reference Manual: Combinatorics

Release 7.1

The Sage Development Team

March 20, 2016

CONTENTS

CHAPTER

ONE

INTRODUCTORY MATERIAL

- *Combinatorics quickref*
- *Introduction to combinatorics in Sage*

CHAPTER

TWO

THEMATIC INDEXES

CHAPTER

THREE

UTILITIES

- *Output functions*
- *Rankers*
- `Combinatorial maps`
- *Miscellaneous*

CHAPTER

FOUR

RELATED TOPICS

- *Coding Theory*
- *Discrete dynamics*
- *Graph Theory*

CHAPTER

FIVE

COMPREHENSIVE MODULE LIST

5.1 Comprehensive Module list

Note: This list is currently sorted in alphabetical order w.r.t. the module names. It can be updated semi-automatically by running in `src/sage/combinat`:

```
find -name "*.py*" | sed 's|\.pyx\?$||; s|\./|    sage/combinat/|' | LANG=en_US.UTF-8 LC_COLLATE=C so
```

and copy pasting the result back there.

Todo

See trac ticket #17421 for desirable improvements.

5.1.1 Combinatorics

Introductory material

- *Combinatorics quickref*
- *Introduction to combinatorics in Sage*

Thematic indexes

- *Algebraic combinatorics*
  - *Cluster Algebras and Quivers*
  - *Crystals*
  - *Root Systems*
  - *Symmetric Functions*
- *Counting*
- *Enumerated sets and combinatorial objects*
- *Enumerated sets of partitions, tableaux, ...*
- *Finite State Machines, Automata, Transducers*

- *Combinatorial Species*
- *Combinatorial Designs and Incidence Structures*
- *Posets*
- *Words*

Utilities

- *Output functions*
- *Rankers*
- Combinatorial maps
- *Miscellaneous*

Related topics

- *Coding Theory*
- *Discrete dynamics*
- *Graph Theory*

5.1.2 Abstract Recursive Trees

The purpose of this class is to help implement trees with a specific structure on the children of each node. For instance, one could want to define a tree in which each node sees its children as linearly (see the Ordered Trees module) or cyclically ordered.

Tree structures

Conceptually, one can define a tree structure from any object that can contain others. Indeed, a list can contain lists which contain lists which contain lists, and thus define a tree ... The same can be done with sets, or any kind of iterable objects.

While any iterable is sufficient to encode trees, it can prove useful to have other methods available like isomorphism tests (see next section), conversions to DiGraphs objects (see as_digraph()) or computation of the number of automorphisms constrained by the structure on children. Providing such methods is the whole purpose of the AbstractTree class.

As a result, the AbstractTree class is not meant to be instantiated, but extended. It is expected that classes extending this one may also inherit from classes representing iterables, for instance ClonableArray or ClonableList

Constrained Trees

The tree built from a specific container will reflect the properties of the container. Indeed, if A is an iterable class whose elements are linearly ordered, a class B extending both of AbstractTree and A will be such that the children of a node will be linearly ordered. If A behaves like a set (i.e. if there is no order on the elements it contains), then two trees will be considered as equal if one can be obtained from the other through permutations between the children of a same node (see next section).

Paths and ID

It is expected that each element of a set of children should be identified by its index in the container. This way, any node of the tree can be identified by a word describing a path from the root node.

Canonical labellings

Equality between instances of classes extending both AbstractTree and A is entirely defined by the equality defined on the elements of A. A canonical labelling of such a tree, however, should be such that two trees a and b satisfying a == b have the same canonical labellings. On the other hand, the canonical labellings of trees a and b satisfying a != b are expected to be different.

For this reason, the values returned by the canonical_labelling method heavily depend on the data structure used for a node's children and **should be overridden** by most of the classes extending AbstractTree if it is incoherent with the data structure.

Authors

- Florent Hivert (2010-2011): initial revision
- Frédéric Chapoton (2011): contributed some methods

class sage.combinat.abstract_tree.**AbstractClonableTree**
Bases: sage.combinat.abstract_tree.AbstractTree

Abstract Clonable Tree.

An abstract class for trees with clone protocol (see list_clone). It is expected that classes extending this one may also inherit from classes like ClonableArray or ClonableList depending whether one wants to build trees where adding a child is allowed.

Note: Due to the limitation of Cython inheritance, one cannot inherit here from ClonableElement, because it would prevent us from later inheriting from ClonableArray or ClonableList.

How should this class be extended ?

A class extending AbstractClonableTree should satisfy the following assumptions:

- An instantiable class extending AbstractClonableTree should also extend the ClonableElement class or one of its subclasses generally, at least ClonableArray.
- To respect the Clone protocol, the AbstractClonableTree.check() method should be overridden by the new class.

See also the assumptions in AbstractTree.

check()
Check that self is a correct tree.

This method does nothing. It is implemented here because many extensions of AbstractClonableTree also extend sage.structure.list_clone.ClonableElement, which requires it.

It should be overridden in subclasses in order to check that the characterizing property of the respective kind of tree holds (eg: two children for binary trees).

EXAMPLES:

```
sage: OrderedTree([[],[[]]]).check()
sage: BinaryTree([[],[],[]]).check()
```

class sage.combinat.abstract_tree.**AbstractLabelledClonableTree**(*parent*, *children*, *label=None*, *check=True*)
Bases: sage.combinat.abstract_tree.AbstractLabelledTree, sage.combinat.abstract_tree.AbstractClonableTree

Abstract Labelled Clonable Tree

This class takes care of modification for the label by the clone protocol.

Note: Due to the limitation of Cython inheritance, one cannot inherit here from `ClonableArray`, because it would prevent us to inherit later from `ClonableList`.

---

map_labels(*f*)

Applies the function f to the labels of `self`

This method returns a copy of `self` on which the function f has been applied on all labels (a label x is replaced by $f(x)$).

EXAMPLES:

```
sage: LT = LabelledOrderedTree
sage: t = LT([LT([],label=1),LT([],label=7)],label=3); t
3[1[], 7[]]
sage: t.map_labels(lambda z:z+1)
4[2[], 8[]]

sage: LBT = LabelledBinaryTree
sage: bt = LBT([LBT([],label=1),LBT([],label=4)],label=2); bt
2[1[., .], 4[., .]]
sage: bt.map_labels(lambda z:z+1)
3[2[., .], 5[., .]]
```

set_label(*path*, *label*)

Changes the label of subtree indexed by `path` to `label`

INPUT:

- **path – None (default) or a path (list or tuple of children** index in the tree)
- `label` – any sage object

OUTPUT: Nothing, `self` is modified in place

Note: `self` must be in a mutable state. See `sage.structure.list_clone` for more details about mutability.

EXAMPLES:

```
sage: t = LabelledOrderedTree([[],[[],[]]])
sage: t.set_label((0,), 4)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with t.clone() as t:
....:     t.set_label((0,), 4)
sage: t
None[4[], None[None[], None[]]]
sage: with t.clone() as t:
....:     t.set_label((1,0), label = 42)
sage: t
None[4[], None[42[], None[]]]
```

Todo

Do we want to implement the following syntactic sugar:

```
with t.clone() as tt:
    tt.labels[1,2] = 3 ?
```

set_root_label(*label*)

Sets the label of the root of `self`

INPUT: `label` – any Sage object

OUTPUT: `None`, `self` is modified in place

Note: `self` must be in a mutable state. See `sage.structure.list_clone` for more details about mutability.

EXAMPLES:

```
sage: t = LabelledOrderedTree([[],[[],[]]])
sage: t.set_root_label(3)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with t.clone() as t:
....:     t.set_root_label(3)
sage: t.label()
3
sage: t
3[None[], None[None[], None[]]]
```

This also works for binary trees:

```
sage: bt = LabelledBinaryTree([[],[]])
sage: bt.set_root_label(3)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with bt.clone() as bt:
....:     bt.set_root_label(3)
sage: bt.label()
3
sage: bt
3[None[., .], None[., .]]
```

TESTS:

```
sage: with t.clone() as t:
....:     t[0] = LabelledOrderedTree(t[0], label = 4)
sage: t
3[4[], None[None[], None[]]]
sage: with t.clone() as t:
....:     t[1,0] = LabelledOrderedTree(t[1,0], label = 42)
sage: t
3[4[], None[42[], None[]]]
```

class sage.combinat.abstract_tree.**AbstractLabelledTree**(*parent*, *children*, *label=None*, *check=True*)

Bases: sage.combinat.abstract_tree.AbstractTree

Abstract Labelled Tree.

Typically a class for labelled trees is contructed by inheriting from a class for unlabelled trees and AbstractLabelledTree.

How should this class be extended ?

A class extending AbstractLabelledTree should respect the following assumptions:

- For a labelled tree T the call T.parent().unlabelled_trees() should return a parent for unlabelled trees of the same kind: for example,
  - if T is a binary labelled tree, T.parent() is LabelledBinaryTrees() and T.parent().unlabelled_trees() is BinaryTrees()
  - if T is an ordered labelled tree, T.parent() is LabelledOrderedTrees() and T.parent().unlabelled_trees() is OrderedTrees()
- In the same vein, the class of T should contain an attribute _UnLabelled which should be the class for the corresponding unlabelled trees.

See also the assumptions in AbstractTree.

See also:

AbstractTree

as_digraph()

Returns a directed graph version of self.

> **Warning:** At this time, the output makes sense only if self is a labelled binary tree with no repeated labels and no None labels.

EXAMPLES:

```
sage: LT = LabelledOrderedTrees()
sage: t1 = LT([LT([],label=6),LT([],label=1)],label=9)
sage: t1.as_digraph()
Digraph on 3 vertices

sage: t = BinaryTree([[None, None],[[],None]]);
sage: lt = t.canonical_labelling()
sage: lt.as_digraph()
Digraph on 4 vertices
```

label(*path=None*)

Return the label of self.

INPUT:

- **path – None (default) or a path (list or tuple of children index** in the tree)

OUTPUT: the label of the subtree indexed by path

EXAMPLES:

```
sage: t = LabelledOrderedTree([[],[]], label = 3)
sage: t.label()
3
sage: t[0].label()
sage: t = LabelledOrderedTree([LabelledOrderedTree([], 5),[]], label = 3)
sage: t.label()
3
sage: t[0].label()
5
sage: t[1].label()
```

```
sage: t.label([0])
5
```

labels()

Return the list of labels of `self`.

EXAMPLES:

```
sage: LT = LabelledOrderedTree
sage: t = LT([LT([],label='b'),LT([],label='c')],label='a')
sage: t.labels()
['a', 'b', 'c']

sage: LBT = LabelledBinaryTree
sage: LBT([LBT([],label=1),LBT([],label=4)],label=2).labels()
[2, 1, 4]
```

leaf_labels()

Return the list of labels of the leaves of `self`.

In case of a labelled binary tree, these "leaves" are not actually the leaves of the binary trees, but the nodes whose both children are leaves!

EXAMPLES:

```
sage: LT = LabelledOrderedTree
sage: t = LT([LT([],label='b'),LT([],label='c')],label='a')
sage: t.leaf_labels()
['b', 'c']

sage: LBT = LabelledBinaryTree
sage: bt = LBT([LBT([],label='b'),LBT([],label='c')],label='a')
sage: bt.leaf_labels()
['b', 'c']
sage: LBT([], label='1').leaf_labels()
['1']
sage: LBT(None).leaf_labels()
[]
```

shape()

Return the unlabelled tree associated to `self`.

EXAMPLES:

```
sage: t = LabelledOrderedTree([[],[[]]], label = 25).shape(); t
[[], [[]]]

sage: LabelledBinaryTree([[],[[],[]]], label = 25).shape()
[[., .], [[., .], [., .]]]

sage: LRT = LabelledRootedTree
sage: tb = LRT([],label='b')
sage: LRT([tb, tb], label='a').shape()
[[], []]
```

TESTS:

```
sage: t.parent()
Ordered trees
sage: type(t)
<class 'sage.combinat.ordered_tree.OrderedTrees_all_with_category.element_class'>
```

class sage.combinat.abstract_tree.**AbstractTree**

Bases: object

Abstract Tree.

There is no data structure defined here, as this class is meant to be extended, not instantiated.

How should this class be extended?

A class extending AbstractTree should respect several assumptions:

- For a tree T, the call iter(T) should return an iterator on the children of the root T.
- The canonical_labelling method should return the same value for trees that are considered equal (see the "canonical labellings" section in the documentation of the AbstractTree class).
- For a tree T the call T.parent().labelled_trees() should return a parent for labelled trees of the same kind: for example,
  - if T is a binary tree, T.parent() is BinaryTrees() and T.parent().labelled_trees() is LabelledBinaryTrees()
  - if T is an ordered tree, T.parent() is OrderedTrees() and T.parent().labelled_trees() is LabelledOrderedTrees()

TESTS:

```
sage: TestSuite(OrderedTree()).run()
sage: TestSuite(BinaryTree()).run()
```

breadth_first_order_traversal(*action=None*)

Run the breadth-first post-order traversal algorithm and subject every node encountered to some procedure action. The algorithm is:

```
queue <- [ root ];
while the queue is not empty:
    node <- pop( queue );
    manipulate the node;
    prepend to the queue the list of all subtrees of
        the node (from the rightmost to the leftmost).
```

INPUT:

- action – (optional) a function which takes a node as input, and does something during the exploration

OUTPUT:

None. (This is *not* an iterator.)

EXAMPLES:

For example, on the following binary tree *b*:

```
|    ___3_____       |
|   /         \      |
| 1          _7_    |
|   \        /   \   |
|    2      5     8  |
|          / \       |
|         4   6      |
```

the breadth-first order traversal algorithm explores b in the following order of nodes: $3, 1, 7, 2, 5, 8, 4, 6$.

TESTS:

```
sage: b = BinaryTree([[None,[]],[[[],[]],[]]]).canonical_labelling()
sage: l = []
sage: b.breadth_first_order_traversal(lambda node: l.append(node.label()))
sage: l
[3, 1, 7, 2, 5, 8, 4, 6]

sage: t = OrderedTree([[[[[],[]]],[[]],[[],[]]]).canonical_labelling()
sage: t
1[2[3[4[], 5[]]], 6[7[]], 8[9[], 10[]]]
sage: l = []
sage: t.breadth_first_order_traversal(lambda node: l.append(node.label()))
sage: l
[1, 2, 6, 8, 3, 7, 9, 10, 4, 5]

sage: l = []
sage: BinaryTree().canonical_labelling().\
....:     breadth_first_order_traversal(
....:         lambda node: l.append(node.label()))
sage: l
[]
sage: OrderedTree([]).canonical_labelling().\
....:     breadth_first_order_traversal(
....:         lambda node: l.append(node.label()))
sage: l
[1]
```

canonical_labelling(*shift=1*)

Returns a labelled version of `self`.

The actual canonical labelling is currently unspecified. However, it is guaranteed to have labels in $1...n$ where n is the number of nodes of the tree. Moreover, two (unlabelled) trees compare as equal if and only if their canonical labelled trees compare as equal.

EXAMPLES:

```
sage: t = OrderedTree([[], [[], [[], []], [[], []]], [[], []]])
sage: t.canonical_labelling()
1[2[], 3[4[], 5[6[], 7[]], 8[9[], 10[]]], 11[12[], 13[]]]

sage: BinaryTree([]).canonical_labelling()
1[., .]
sage: BinaryTree([[],[[],[]]]).canonical_labelling()
2[1[., .], 4[3[., .], 5[., .]]]
```

TESTS:

```
sage: BinaryTree().canonical_labelling()
.
```

depth()

The depth of `self`.

EXAMPLES:

```
sage: OrderedTree().depth()
1
sage: OrderedTree([]).depth()
1
sage: OrderedTree([[],[]]).depth()
2
sage: OrderedTree([[],[[]]]).depth()
3
sage: OrderedTree([[], [[], [[], []], [[], []]], [[], [[]]]).depth()
4

sage: BinaryTree().depth()
0
sage: BinaryTree([[],[[],[]]]).depth()
3
```

iterative_post_order_traversal(*action=None*)

Run the depth-first post-order traversal algorithm (iterative implementation) and subject every node encountered to some procedure `action`. The algorithm is:

```
explore each subtree (by the algorithm) from the
    leftmost one to the rightmost one;
then manipulate the root with function `action` (in the
    case of a binary tree, only if the root is not a leaf).
```

INPUT:

- `action` – (optional) a function which takes a node as input, and does something during the exploration

OUTPUT:

`None`. (This is *not* an iterator.)

See also:

- `post_order_traversal_iter()`

TESTS:

```
sage: l = []
sage: b = BinaryTree([[None,[]],[[[],[]],[]]]).canonical_labelling()
sage: b
3[1[., 2[., .]], 7[5[4[., .], 6[., .]], 8[., .]]]
sage: b.iterative_post_order_traversal(lambda node: l.append(node.label()))
sage: l
[2, 1, 4, 6, 5, 8, 7, 3]

sage: t = OrderedTree([[[[],[]],[[]],[[],[]]]).canonical_labelling()
sage: t
1[2[3[4[], 5[]]], 6[7[]], 8[9[], 10[]]]
sage: l = []
sage: t.iterative_post_order_traversal(lambda node: l.append(node.label()))
sage: l
[4, 5, 3, 2, 7, 6, 9, 10, 8, 1]

sage: l = []
sage: BinaryTree().canonical_labelling().\
....:     iterative_post_order_traversal(
....:         lambda node: l.append(node.label()))
```

```
sage: l
[]
sage: OrderedTree([]).canonical_labelling().\
....:     iterative_post_order_traversal(
....:         lambda node: l.append(node.label()))
sage: l
[1]
```

The following test checks that things do not go wrong if some among the descendants of the tree are equal or even identical:

```
sage: u = BinaryTree(None)
sage: v = BinaryTree([u, u])
sage: w = BinaryTree([v, v])
sage: t = BinaryTree([w, w])
sage: t.node_number()
7
sage: l = []
sage: t.iterative_post_order_traversal(lambda node: l.append(1))
sage: len(l)
7
```

iterative_pre_order_traversal(*action=None*)

Run the depth-first pre-order traversal algorithm (iterative implementation) and subject every node encountered to some procedure `action`. The algorithm is:

```
manipulate the root with function `action` (in the case
    of a binary tree, only if the root is not a leaf);
then explore each subtree (by the algorithm) from the
    leftmost one to the rightmost one.
```

INPUT:

- `action` – (optional) a function which takes a node as input, and does something during the exploration

OUTPUT:

`None`. (This is *not* an iterator.)

See also:

- `pre_order_traversal_iter()`
- `pre_order_traversal()`

TESTS:

```
sage: l = []
sage: b = BinaryTree([[None,[]],[[[],[]],[]]]).canonical_labelling()
sage: b
3[1[., 2[., .]], 7[5[4[., .], 6[., .]], 8[., .]]]
sage: b.iterative_pre_order_traversal(lambda node: l.append(node.label()))
sage: l
[3, 1, 2, 7, 5, 4, 6, 8]

sage: t = OrderedTree([[[[],[]]],[[]],[[],[]]]).canonical_labelling()
sage: t
1[2[3[4[], 5[]]], 6[7[]], 8[9[], 10[]]]
sage: l = []
```

```
sage: t.iterative_pre_order_traversal(lambda node: l.append(node.label()))
sage: l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: l = []

sage: BinaryTree().canonical_labelling().\
....:     pre_order_traversal(lambda node: l.append(node.label()))
sage: l
[]
sage: OrderedTree([]).canonical_labelling().\
....:     iterative_pre_order_traversal(lambda node: l.append(node.label()))
sage: l
[1]
```

The following test checks that things do not go wrong if some among the descendants of the tree are equal or even identical:

```
sage: u = BinaryTree(None)
sage: v = BinaryTree([u, u])
sage: w = BinaryTree([v, v])
sage: t = BinaryTree([w, w])
sage: t.node_number()
7
sage: l = []
sage: t.iterative_pre_order_traversal(lambda node: l.append(1))
sage: len(l)
7
```

node_number()

The number of nodes of `self`.

EXAMPLES:

```
sage: OrderedTree().node_number()
1
sage: OrderedTree([]).node_number()
1
sage: OrderedTree([[],[]]).node_number()
3
sage: OrderedTree([[],[[]]]).node_number()
4
sage: OrderedTree([[], [[], [[], []], [[], []]], [[], [[]]]).node_number()
13
```

EXAMPLES:

```
sage: BinaryTree(None).node_number()
0
sage: BinaryTree([]).node_number()
1
sage: BinaryTree([[], None]).node_number()
2
sage: BinaryTree([[None, [[], []]], None]).node_number()
5
```

paths()

Return a generator for all paths to nodes of `self`.

OUTPUT:

This method returns a list of sequences of integers. Each of these sequences represents a path from the root node to some node. For instance, $(1, 3, 2, 5, 0, 3)$ represents the node obtained by choosing the 1st child of the root node (in the ordering returned by `iter`), then the 3rd child of its child, then the 2nd child of the latter, etc. (where the labelling of the children is zero-based).

The root element is represented by the empty tuple `()`.

EXAMPLES:

```
sage: list(OrderedTree([]).paths())
[()]
sage: list(OrderedTree([[],[[]]]).paths())
[(), (0,), (1,), (1, 0)]

sage: list(BinaryTree([[],[[],[]]]).paths())
[(), (0,), (1,), (1, 0), (1, 1)]
```

TESTS:

```
sage: t = OrderedTree([[], [[], [[], []], [[], []]], [[], []]])
sage: t.node_number() == len(list(t.paths()))
True
sage: list(BinaryTree().paths())
[]
sage: bt = BinaryTree([[],[[],[]]])
sage: bt.node_number() == len(list(bt.paths()))
True
```

post_order_traversal(*action=None*)

Run the depth-first post-order traversal algorithm (recursive implementation) and subject every node encountered to some procedure `action`. The algorithm is:

```
explore each subtree (by the algorithm) from the
    leftmost one to the rightmost one;
then manipulate the root with function 'action' (in the
    case of a binary tree, only if the root is not a leaf).
```

INPUT:

- `action` – (optional) a function which takes a node as input, and does something during the exploration

OUTPUT:

`None`. (This is *not* an iterator.)

See also:

- `post_order_traversal_iter()`
- `iterative_post_order_traversal()`

TESTS:

```
sage: l = []
sage: b = BinaryTree([[None,[]],[[[],[]],[[]]]]).canonical_labelling()
sage: b
3[1[., 2[., .]], 7[5[4[., .], 6[., .]], 8[., .]]]
sage: b.post_order_traversal(lambda node: l.append(node.label()))
sage: l
[2, 1, 4, 6, 5, 8, 7, 3]
```

```
sage: t = OrderedTree([[[[],[]]],[[]],[[],[]]]).\
....:     canonical_labelling(); t
1[2[3[4[], 5[]]], 6[7[]], 8[9[], 10[]]]
sage: l = []
sage: t.post_order_traversal(lambda node: l.append(node.label()))
sage: l
[4, 5, 3, 2, 7, 6, 9, 10, 8, 1]

sage: l = []
sage: BinaryTree().canonical_labelling().\
....:     post_order_traversal(lambda node: l.append(node.label()))
sage: l
[]
sage: OrderedTree([]).canonical_labelling().\
....:     post_order_traversal(lambda node: l.append(node.label()))
sage: l
[1]
```

The following test checks that things do not go wrong if some among the descendants of the tree are equal or even identical:

```
sage: u = BinaryTree(None)
sage: v = BinaryTree([u, u])
sage: w = BinaryTree([v, v])
sage: t = BinaryTree([w, w])
sage: t.node_number()
7
sage: l = []
sage: t.post_order_traversal(lambda node: l.append(1))
sage: len(l)
7
```

post_order_traversal_iter()

The depth-first post-order traversal iterator.

This method iters each node following the depth-first post-order traversal algorithm (recursive implementation). The algorithm is:

```
explore each subtree (by the algorithm) from the
    leftmost one to the rightmost one;
then yield the root (in the case of binary trees, only if
    it is not a leaf).
```

EXAMPLES:

For example on the following binary tree b:

```
|    ___3____       |
|   /        \      |
| 1          _7_    |
|  \        /   \   |
|   2      5     8  |
|         / \       |
|        4   6      |
```

(only the nodes are shown), the depth-first post-order traversal algorithm explores b in the following order of nodes: $2, 1, 4, 6, 5, 8, 7, 3$.

For another example, consider the labelled tree:

```
|      __1____ |
|     /  /    / |
|    2  6    8_ |
|    |  |   / / |
|    3_ 7 9 10 |
|   / /         |
|  4 5          |
```

The algorithm explores this tree in the following order: $4, 5, 3, 2, 7, 6, 9, 10, 8, 1$.

TESTS:

```
sage: b = BinaryTree([[None,[]],[[[],[]],[]]]).canonical_labelling()
sage: ascii_art([b])
[    ___3____       ]
[   /        \      ]
[ 1          _7_    ]
[  \        /   \   ]
[   2      5     8  ]
[         / \       ]
[        4   6      ]
sage: [node.label() for node in b.post_order_traversal_iter()]
[2, 1, 4, 6, 5, 8, 7, 3]

sage: t = OrderedTree([[[[],[]],[]],[[]],[[],[]]]).canonical_labelling()
sage: ascii_art([t])
[      __1____    ]
[     /  /    /    ]
[    2  6    8_   ]
[    |  |   / /    ]
[    3_ 7 9 10    ]
[   / /           ]
[  4 5            ]
sage: [node.label() for node in t.post_order_traversal_iter()]
[4, 5, 3, 2, 7, 6, 9, 10, 8, 1]

sage: [node.label() for node in BinaryTree().canonical_labelling().\
....:     post_order_traversal_iter()]
[]
sage: [node.label() for node in OrderedTree([]).\
....:     canonical_labelling().post_order_traversal_iter()]
[1]
```

The following test checks that things do not go wrong if some among the descendants of the tree are equal or even identical:

```
sage: u = BinaryTree(None)
sage: v = BinaryTree([u, u])
sage: w = BinaryTree([v, v])
sage: t = BinaryTree([w, w])
sage: t.node_number()
7
sage: l = [1 for i in t.post_order_traversal_iter()]
sage: len(l)
7
```

pre_order_traversal(*action=None*)

Run the depth-first pre-order traversal algorithm (recursive implementation) and subject every node encountered to some procedure `action`. The algorithm is:

```
manipulate the root with function 'action' (in the case
    of a binary tree, only if the root is not a leaf);
then explore each subtree (by the algorithm) from the
    leftmost one to the rightmost one.
```

INPUT:

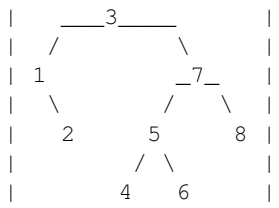
- `action` – (optional) a function which takes a node as input, and does something during the exploration

OUTPUT:

`None`. (This is *not* an iterator.)

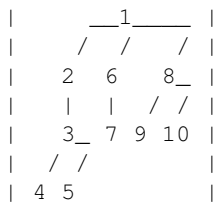
EXAMPLES:

For example, on the following binary tree b:



the depth-first pre-order traversal algorithm explores b in the following order of nodes: $3, 1, 2, 7, 5, 4, 6, 8$.

Another example:



The algorithm explores this tree in the following order: $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$.

See also:

- `pre_order_traversal_iter()`
- `iterative_pre_order_traversal()`

TESTS:

```
sage: l = []
sage: b = BinaryTree([[None,[]],[[[],[]],[]]]).canonical_labelling()
sage: b
3[1[., 2[., .]], 7[5[4[., .], 6[., .]], 8[., .]]]
sage: b.pre_order_traversal(lambda node: l.append(node.label()))
sage: l
[3, 1, 2, 7, 5, 4, 6, 8]
sage: li = []
sage: b.iterative_pre_order_traversal(lambda node: li.append(node.label()))
sage: l == li
True

sage: t = OrderedTree([[[[],[]]],[[]],[[],[]]]).canonical_labelling()
```

```
sage: t
1[2[3[4[], 5[]]], 6[7[]], 8[9[], 10[]]]
sage: l = []
sage: t.pre_order_traversal(lambda node: l.append(node.label()))
sage: l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: li = []
sage: t.iterative_pre_order_traversal(lambda node: li.append(node.label()))
sage: l == li
True

sage: l = []
sage: BinaryTree().canonical_labelling().\
....:     pre_order_traversal(lambda node: l.append(node.label()))
sage: l
[]
sage: OrderedTree([]).canonical_labelling().\
....:     pre_order_traversal(lambda node: l.append(node.label()))
sage: l
[1]
```

The following test checks that things do not go wrong if some among the descendants of the tree are equal or even identical:

```
sage: u = BinaryTree(None)
sage: v = BinaryTree([u, u])
sage: w = BinaryTree([v, v])
sage: t = BinaryTree([w, w])
sage: t.node_number()
7
sage: l = []
sage: t.pre_order_traversal(lambda node: l.append(1))
sage: len(l)
7
```

pre_order_traversal_iter()

The depth-first pre-order traversal iterator.

This method iters each node following the depth-first pre-order traversal algorithm (recursive implementation). The algorithm is:

```
yield the root (in the case of binary trees, if it is not
    a leaf);
then explore each subtree (by the algorithm) from the
    leftmost one to the rightmost one.
```

EXAMPLES:

For example, on the following binary tree b:

```
|    ___3____       |
|   /        \      |
| 1          _7_    |
|   \       /   \   |
|    2     5     8  |
|         / \       |
|        4   6      |
```

(only the nodes shown), the depth-first pre-order traversal algorithm explores b in the following order of

nodes: $3, 1, 2, 7, 5, 4, 6, 8$.

Another example:

```
|      __1_____ |
|     /  /    / |
|    2  6    8_ |
|    |  |   / / |
|    3_ 7 9 10  |
|   / /         |
|  4 5          |
```

The algorithm explores this labelled tree in the following order: $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$.

TESTS:

```
sage: b = BinaryTree([[None,[]],[[[],[]],[]]]).canonical_labelling()
sage: ascii_art([b])
[    ___3_____       ]
[   /         \      ]
[  1           _7_   ]
[   \         /   \  ]
[    2       5     8 ]
[           / \       ]
[          4   6      ]
sage: [n.label() for n in b.pre_order_traversal_iter()]
[3, 1, 2, 7, 5, 4, 6, 8]

sage: t = OrderedTree([[[[],[]]],[[]],[[],[]]]).canonical_labelling()
sage: ascii_art([t])
[      __1_____    ]
[     /  /    /    ]
[    2  6    8_    ]
[    |  |   / /    ]
[    3_ 7 9 10     ]
[   / /            ]
[  4 5             ]
sage: [n.label() for n in t.pre_order_traversal_iter()]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

sage: [n for n in BinaryTree(None).pre_order_traversal_iter()]
[]
```

The following test checks that things do not go wrong if some among the descendants of the tree are equal or even identical:

```
sage: u = BinaryTree(None)
sage: v = BinaryTree([u, u])
sage: w = BinaryTree([v, v])
sage: t = BinaryTree([w, w])
sage: t.node_number()
7
sage: l = [1 for i in t.pre_order_traversal_iter()]
sage: len(l)
7
```

subtrees()

Return a generator for all nonempty subtrees of `self`.

The number of nonempty subtrees of a tree is its number of nodes. (The word "nonempty" makes a difference only in the case of binary trees. For ordered trees, for example, all trees are nonempty.)

EXAMPLES:

```
sage: list(OrderedTree([]).subtrees())
[[]]
sage: list(OrderedTree([[],[[]]]).subtrees())
[[[], [[]]], [], [[]], []]

sage: list(OrderedTree([[],[[]]]).canonical_labelling().subtrees())
[1[2[], 3[4[]]], 2[], 3[4[]], 4[]]

sage: list(BinaryTree([[],[[],[]]]).subtrees())
[[[., .], [[., .], [., .]]], [., .], [[., .], [., .]], [., .], [., .]]

sage: v = BinaryTree([[],[]])
sage: list(v.canonical_labelling().subtrees())
[2[1[., .], 3[., .]], 1[., .], 3[., .]]
```

TESTS:

```
sage: t = OrderedTree([[], [[], [[], []], [[], []]], [[], []]])
sage: t.node_number() == len(list(t.subtrees()))
True
sage: list(BinaryTree().subtrees())
[]
sage: bt = BinaryTree([[],[[],[]]])
sage: bt.node_number() == len(list(bt.subtrees()))
True
```

to_hexacode()

Transform a tree into an hexadecimal string.

The definition of the hexacode is recursive. The first letter is the valence of the root as an hexadecimal (up to 15), followed by the concatenation of the hexacodes of the subtrees.

This method only works for trees where every vertex has valency at most 15.

See `from_hexacode()` for the reverse transformation.

EXAMPLES:

```
sage: from sage.combinat.abstract_tree import from_hexacode
sage: LT = LabelledOrderedTrees()
sage: from_hexacode('2010', LT).to_hexacode()
'2010'
sage: LT.an_element().to_hexacode()
'3020010'
sage: t = from_hexacode('a0000000000000000', LT)
sage: t.to_hexacode()
'a0000000000'

sage: OrderedTrees(6).an_element().to_hexacode()
'500000'
```

TESTS:

```
sage: one = LT([], label='@')
sage: LT([one for _ in range(15)], label='@').to_hexacode()
'f000000000000000'
sage: LT([one for _ in range(16)], label='@').to_hexacode()
Traceback (most recent call last):
...
ValueError: the width of the tree is too large
```

tree_factorial()

Return the tree-factorial of `self`.

Definition:

The tree-factorial $T!$ of a tree T is the product $\prod_{v \in T} \#\text{children}(v)$.

EXAMPLES:

```
sage: LT = LabelledOrderedTrees()
sage: t = LT([LT([],label=6),LT([],label=1)],label=9)
sage: t.tree_factorial()
3

sage: BinaryTree([[],[[],[]]]).tree_factorial()
15
```

TESTS:

```
sage: BinaryTree().tree_factorial()
1
```

sage.combinat.abstract_tree.**from_hexacode**(*ch*, *parent=None*, *label='@'*)

Transform an hexadecimal string into a tree.

INPUT:

- `ch` – an hexadecimal string
- `parent` – kind of trees to be produced. If `None`, this will be `LabelledOrderedTrees`
- `label` – a label (default: `'@'`) to be used for every vertex of the tree

See `AbstractTree.to_hexacode()` for the description of the encoding

See `_from_hexacode_aux()` for the actual code

EXAMPLES:

```
sage: from sage.combinat.abstract_tree import from_hexacode
sage: from_hexacode('12000', LabelledOrderedTrees())
@[@[@[], @[]]]

sage: from_hexacode('1200', LabelledOrderedTrees())
@[@[@[], @[]]]
```

It can happen that only a prefix of the word is used:

```
sage: from_hexacode('a'+14*'0', LabelledOrderedTrees())
@[@[], @[], @[], @[], @[], @[], @[], @[], @[], @[]]
```

One can choose the label:

```
sage: from_hexacode('1200', LabelledOrderedTrees(), label='o')
o[o[o[], o[]]]
```

One can also create other kinds of trees:

```
sage: from_hexacode('1200', OrderedTrees())
[[[], []]]
```

5.1.3 Affine Permutations

class sage.combinat.affine_permutation.**AffinePermutation**(*parent*, *lst*, *check=True*)
Bases: sage.structure.list_clone.ClonableArray

An affine permutation, representated in the window notation, and considered as a bijection from **Z** to **Z**.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
```

apply_simple_reflection(*i*, *side='right'*)
Applies a simple reflection.

INPUT:

- i – an integer.
- side – Determines whether to apply the reflection on the 'right' or 'left'. Default 'right'.

EXAMPLES:

```
sage: p=AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.apply_simple_reflection(3)
Type A affine permutation with window [3, -1, 6, 0, 5, 4, 10, 9]
sage: p.apply_simple_reflection(11)
Type A affine permutation with window [3, -1, 6, 0, 5, 4, 10, 9]
sage: p.apply_simple_reflection(3, 'left')
Type A affine permutation with window [4, -1, 0, 6, 5, 3, 10, 9]
sage: p.apply_simple_reflection(11, 'left')
Type A affine permutation with window [4, -1, 0, 6, 5, 3, 10, 9]
```

grassmannian_quotient(*i=0*, *side='right'*)
Return Grassmannian quotient.

Factors self into a unique product of a Grassmannian and a finite-type element. Returns a tuple containing the Grassmannain and finite elements, in order according to side.

INPUT:

- i – An element of the index set; the descent checked for. Defaults to 0.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: gq=p.grassmannian_quotient()
sage: gq
(Type A affine permutation with window [-1, 0, 3, 4, 5, 6, 9, 10], Type A affine permutation
sage: gq[0].is_i_grassmannian()
True
sage: 0 not in gq[1].reduced_word()
True
sage: prod(gq)==p
True

sage: gqLeft=p.grassmannian_quotient(side='left')
sage: 0 not in gqLeft[0].reduced_word()
True
sage: gqLeft[1].is_i_grassmannian(side='left')
```

```
True
sage: prod(gqLeft)==p
True
```

index_set()
Index set of the affine permutation group.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: A.index_set()
(0, 1, 2, 3, 4, 5, 6, 7)
```

inverse()
Finds the inverse affine permutation.

EXAMPLES:

```
sage: p=AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.inverse()
Type A affine permutation with window [0, -1, 1, 6, 5, 4, 10, 11]
```

is_i_grassmannian(*i=0*, *side='right'*)
Test whether `self` is i-grassmannian, ie, either is the identity or has `i` as the sole descent.

INPUT:

- `i` – An element of the index set.
- `side` – determines the side on which to check the descents.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.is_i_grassmannian()
False
sage: q=A.from_word([3,2,1,0])
sage: q.is_i_grassmannian()
True
sage: q=A.from_word([2,3,4,5])
sage: q.is_i_grassmannian(5)
True
sage: q.is_i_grassmannian(2, side='left')
True
```

is_one()
Tests whether the affine permutation is the identity.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.is_one()
False
sage: q=A.one()
sage: q.is_one()
True
```

lower_covers(*side='right'*)
Return lower covers of `self`.

The set of affine permutations of one less length related by multiplication by a simple transposition on the indicated side. These are the elements that `self` covers in weak order.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.lower_covers()
[Type A affine permutation with window [-1, 3, 0, 6, 5, 4, 10, 9], Type A affine permutation
```

reduced_word()

Returns a reduced word for the affine permutation.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.reduced_word()
[0, 7, 4, 1, 0, 7, 5, 4, 2, 1]
```

signature()

Signature of the affine permutation, $(-1)^l$, where l is the length of the permutation.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.signature()
1
```

to_weyl_group_element()

The affine Weyl group element corresponding to the affine permutation.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.to_weyl_group_element()
[ 0 -1  0  1  0  0  1  0]
[ 1 -1  0  1  0  0  1 -1]
[ 1 -1  0  1  0  0  0  0]
[ 0  0  0  1  0  0  0  0]
[ 0  0  0  1  0 -1  1  0]
[ 0  0  0  1 -1  0  1  0]
[ 0  0  0  0  0  0  1  0]
[ 0 -1  1  0  0  0  1  0]
```

sage.combinat.affine_permutation.**AffinePermutationGroup**(*cartan_type*)

Wrapper function for specific affine permutation groups.

These are combinatorial implmentations of the affine Weyl groups of types A, B, C, D, and G as permutations of the set of all integers. the basic algorithms are derived from [BjBr] and [Erik].

REFERENCES:

EXAMPLES:

```
sage: ct=CartanType(['A',7,1])
sage: A=AffinePermutationGroup(ct)
sage: A
The group of affine permutations of type ['A', 7, 1]
```

We define an element of `A`:

```
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
```

We find the value $p(1)$, considering p as a bijection on the integers. This is the same as calling the 'value' method:

```
sage: p.value(1)
3
sage: p(1)==p.value(1)
True
```

We can also find the position of the integer 3 in p considered as a sequence, equivalent to finding $p^{-1}(3)$:

```
sage: p.position(3)
1
sage: (p^-1)(3)
1
```

Since the affine permutation group is a group, we demonstrate its group properties:

```
sage: A.one()
Type A affine permutation with window [1, 2, 3, 4, 5, 6, 7, 8]

sage: q=A([0, 2, 3, 4, 5, 6, 7, 9])
sage: p*q
Type A affine permutation with window [1, -1, 0, 6, 5, 4, 10, 11]
sage: q*p
Type A affine permutation with window [3, -1, 1, 6, 5, 4, 10, 8]

sage: p^-1
Type A affine permutation with window [0, -1, 1, 6, 5, 4, 10, 11]
sage: p^-1*p==A.one()
True
sage: p*p^-1==A.one()
True
```

If we decide we prefer the Weyl Group implementation of the affine Weyl group, we can easily get it:

```
sage: p.to_weyl_group_element()
[ 0 -1  0  1  0  0  1  0]
[ 1 -1  0  1  0  0  1 -1]
[ 1 -1  0  1  0  0  0  0]
[ 0  0  0  1  0  0  0  0]
[ 0  0  0  1  0 -1  1  0]
[ 0  0  0  1 -1  0  1  0]
[ 0  0  0  0  0  0  1  0]
[ 0 -1  1  0  0  0  1  0]
```

We can find a reduced word and do all of the other things one expects in a Coxeter group:

```
sage: p.has_right_descent(1)
True
sage: p.apply_simple_reflection(1)
Type A affine permutation with window [-1, 3, 0, 6, 5, 4, 10, 9]
sage: p.apply_simple_reflection(0)
Type A affine permutation with window [1, -1, 0, 6, 5, 4, 10, 11]
sage: p.reduced_word()
[0, 7, 4, 1, 0, 7, 5, 4, 2, 1]
sage: p.length()
```

```
10
```

The following methods are particular to Type A. We can check if the element is fully commutative:

```
sage: p.is_fully_commutative()
False
sage: q.is_fully_commutative()
True
```

And we can also compute the affine Lehmer code of the permutation, a weak composition with $k+1$ entries:

```
sage: p.to_lehmer_code()
[0, 3, 3, 0, 1, 2, 0, 1]
```

Once we have the Lehmer code, we can obtain a k-bounded partition by sorting the Lehmer code, and then reading the row lengths. There is a unique 0-Grassmanian (dominant) affine permutation associated to this k-bounded partition, and a k-core as well.

```
sage: p.to_bounded_partition()
[5, 3, 2]
sage: p.to_dominant()
Type A affine permutation with window [-2, -1, 1, 3, 4, 8, 10, 13]
sage: p.to_core()
[5, 3, 2]
```

Finally, we can take a reduced word for p and insert it to find a standard composition tableau associated uniquely to that word.

```
sage: p.tableau_of_word(p.reduced_word())
[[], [1, 6, 9], [2, 7, 10], [], [3], [4, 8], [], [5]]
```

We can also form affine permutation groups in types B, C, D, and G.

```
sage: B=AffinePermutationGroup(['B',4,1])
sage: B.an_element()
Type B affine permutation with window [-1, 3, 4, 11]

sage: C=AffinePermutationGroup(['C',4,1])
sage: C.an_element()
Type C affine permutation with window [2, 3, 4, 10]

sage: D=AffinePermutationGroup(['D',4,1])
sage: D.an_element()
Type D affine permutation with window [-1, 3, 11, 5]

sage: G=AffinePermutationGroup(['G',2,1])
sage: G.an_element()
Type G affine permutation with window [0, 4, -1, 8, 3, 7]
```

class sage.combinat.affine_permutation.**AffinePermutationGroupGeneric**(*cartan_type*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The generic affine permutation group class, in which we define all type-free methods for the specific affine permutation groups.

cartan_matrix()

Returns the Cartan matrix of `self`.

EXAMPLES:

```
sage: AffinePermutationGroup(['A',7,1]).cartan_matrix()
[ 2 -1  0  0  0  0  0 -1]
[-1  2 -1  0  0  0  0  0]
[ 0 -1  2 -1  0  0  0  0]
[ 0  0 -1  2 -1  0  0  0]
[ 0  0  0 -1  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[-1  0  0  0  0  0 -1  2]
```

cartan_type()

Returns the Cartan type of `self`.

EXAMPLES:

```
sage: AffinePermutationGroup(['A',7,1]).cartan_type()
['A', 7, 1]
```

classical()

Returns the finite permutation group.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: A.classical()
Symmetric group of order 8! as a permutation group
```

from_word(*w*)

Builds an affine permutation from a given word. Note: Already in category as `from_reduced_word`, but this is less typing!

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: A.from_word([0, 7, 4, 1, 0, 7, 5, 4, 2, 1])
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
```

index_set()

EXAMPLES:

```
sage: AffinePermutationGroup(['A',7,1]).index_set()
(0, 1, 2, 3, 4, 5, 6, 7)
```

is_crystallographic()

Tells whether the affine permutation group is crystallographic.

EXAMPLES:

```
sage: AffinePermutationGroup(['A',7,1]).is_crystallographic()
True
```

random_element(*n=None*)

Return a random affine permutation of length `n`.

If `n` is not specified, then `n` is chosen as a random non-negative integer in $[0, 1000]$.

Starts at the identity, then chooses an upper cover at random. Not very uniform: actually constructs a uniformly random reduced word of length n. Thus we most likely get elements with lots of reduced words!

For the actual code, see `sage.categories.coxeter_group.random_element_of_length()`.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',7,1])
sage: A.random_element() # random
Type A affine permutation with window [-12, 16, 19, -1, -2, 10, -3, 9]
sage: p = A.random_element(10)
sage: p.length() == 10
True
```

rank()

Rank of the affine permutation group, equal to $k+1$.

EXAMPLES:

```
sage: AffinePermutationGroup(['A',7,1]).rank()
8
```

reflection_index_set()

EXAMPLES:

```
sage: AffinePermutationGroup(['A',7,1]).reflection_index_set()
(0, 1, 2, 3, 4, 5, 6, 7)
```

weyl_group()

Returns the Weyl Group of the same type as `self`.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: A.weyl_group()
Weyl Group of type ['A', 7, 1] (as a matrix group acting on the root space)
```

class sage.combinat.affine_permutation.**AffinePermutationGroupTypeA**(*cartan_type*)

Bases: sage.combinat.affine_permutation.AffinePermutationGroupGeneric

TESTS:

```
sage: AffinePermutationGroup(['A',7,1])
The group of affine permutations of type ['A', 7, 1]
```

Element

alias of AffinePermutationTypeA

from_lehmer_code(*C*, *typ='decreasing'*, *side='right'*)

Returns the affine permutation with the supplied Lehmer code (a weak composition with $k+1$ parts, at least one of which is 0).

INPUT:

- **typ – 'increasing' or 'decreasing': type of product.** (default: 'decreasing'.)
- **side – 'right' or 'left': Whether the decomposition is from** the right or left. (default: 'right'.)

EXAMPLES:

```
sage: import itertools
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.to_lehmer_code()
[0, 3, 3, 0, 1, 2, 0, 1]
sage: A.from_lehmer_code(p.to_lehmer_code())==p
True
```

```
sage: orders = ('increasing','decreasing')
sage: sides = ('left','right')
sage: for o,s in itertools.product(orders,sides):
....:   A.from_lehmer_code(p.to_lehmer_code(o,s),o,s)==p
True
True
True
True
```

one()
Returns the identity element.

EXAMPLES:

```
sage: AffinePermutationGroup(['A',7,1]).one()
Type A affine permutation with window [1, 2, 3, 4, 5, 6, 7, 8]
```

TESTS:

```
sage: A=AffinePermutationGroup(['A',5,1])
sage: A==loads(dumps(A))
True
sage: TestSuite(A).run()
```

class sage.combinat.affine_permutation.**AffinePermutationGroupTypeB**(*cartan_type*)
Bases: sage.combinat.affine_permutation.AffinePermutationGroupTypeC

TESTS:

```
sage: AffinePermutationGroup(['A',7,1])
The group of affine permutations of type ['A', 7, 1]
```

Element
alias of AffinePermutationTypeB

class sage.combinat.affine_permutation.**AffinePermutationGroupTypeC**(*cartan_type*)
Bases: sage.combinat.affine_permutation.AffinePermutationGroupTypeC

TESTS:

```
sage: AffinePermutationGroup(['A',7,1])
The group of affine permutations of type ['A', 7, 1]
```

Element
alias of AffinePermutationTypeC

class sage.combinat.affine_permutation.**AffinePermutationGroupTypeD**(*cartan_type*)
Bases: sage.combinat.affine_permutation.AffinePermutationGroupTypeC

TESTS:

```
sage: AffinePermutationGroup(['A',7,1])
The group of affine permutations of type ['A', 7, 1]
```

Element
alias of AffinePermutationTypeD

class sage.combinat.affine_permutation.**AffinePermutationGroupTypeG**(*cartan_type*)
Bases: sage.combinat.affine_permutation.AffinePermutationGroupGeneric

TESTS:

```
sage: AffinePermutationGroup(['A',7,1])
The group of affine permutations of type ['A', 7, 1]
```

Element
alias of AffinePermutationTypeG

one()
Returns the identity element.

EXAMPLES:
```
sage: AffinePermutationGroup(['G',2,1]).one()
Type G affine permutation with window [1, 2, 3, 4, 5, 6]
```

TESTS:
```
sage: G=AffinePermutationGroup(['G',2,1])
sage: G==loads(dumps(G))
True
sage: TestSuite(G).run()
```

class sage.combinat.affine_permutation.**AffinePermutationTypeA**(*parent*, *lst*, *check=True*)
Bases: sage.combinat.affine_permutation.AffinePermutation

Initialize `self`

INPUT:

- `parent` – The parent affine permutation group.
- `lst` – List giving the base window of the affine permutation.
- `check`– Chooses whether to test that the affine permutation is legit.

EXAMPLES:
```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9]) #indirect doctest
sage: p
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
```

apply_simple_reflection_left(*i*)
Applies simple reflection to the values $i, i+1$.

EXAMPLES:
```
sage: p=AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.apply_simple_reflection_left(3)
Type A affine permutation with window [4, -1, 0, 6, 5, 3, 10, 9]
sage: p.apply_simple_reflection_left(11)
Type A affine permutation with window [4, -1, 0, 6, 5, 3, 10, 9]
```

apply_simple_reflection_right(*i*)
Applies the simple reflection to positions $i, i+1$. i is allowed to be any integer.

EXAMPLES:
```
sage: p=AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.apply_simple_reflection_right(3)
Type A affine permutation with window [3, -1, 6, 0, 5, 4, 10, 9]
sage: p.apply_simple_reflection_right(11)
Type A affine permutation with window [3, -1, 6, 0, 5, 4, 10, 9]
```

check()
Check that `self` is an affine permutation.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
sage: q=A([1,2,3])  # indirect doctest
Traceback (most recent call last):
...
ValueError: Length of list must be k+1=8.
sage: q=A([1,2,3,4,5,6,7,0])  # indirect doctest
Traceback (most recent call last):
...
ValueError: Window does not sum to 36.
sage: q=A([1,1,3,4,5,6,7,9])  # indirect doctest
Traceback (most recent call last):
...
ValueError: Entries must have distinct residues.
```

flip_automorphism()
The Dynkin diagram automorphism which fixes s_0 and reverses all other indices.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.flip_automorphism()
Type A affine permutation with window [0, -1, 5, 4, 3, 9, 10, 6]
```

has_left_descent(i)
Determines whether there is a descent at i.

INPUT:

- `i` – an integer.

EXAMPLES:

```
sage: p=AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.has_left_descent(1)
True
sage: p.has_left_descent(9)
True
sage: p.has_left_descent(0)
True
```

has_right_descent(i)
Determines whether there is a descent at i.

INPUT:

- `i` – an integer.

EXAMPLES:

```
sage: p=AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.has_right_descent(1)
True
sage: p.has_right_descent(9)
True
```

```
sage: p.has_right_descent(0)
False
```

is_fully_commutative()

Determines whether `self` is fully commutative, ie, has no reduced words with a braid.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.is_fully_commutative()
False
sage: q=A([-3, -2, 0, 7, 9, 2, 11, 12])
sage: q.is_fully_commutative()
True
```

maximal_cyclic_decomposition(*typ='decreasing'*, *side='right'*, *verbose=False*)

Finds the unique maximal decomposition of `self` into cyclically decreasing/increasing elements.

INPUT:

- `typ` – 'increasing' or 'decreasing' (default: 'decreasing'.) Chooses whether to find increasing or deacreasing sets.
- `side` – 'right' or 'left' (default: 'right'.) Chooses whether to find maximal sets starting from the left or the right.
- `verbose` – Print extra information while finding the decomposition.

EXAMPLES:

```
sage: p=AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.maximal_cyclic_decomposition()
[[0, 7], [4, 1, 0], [7, 5, 4, 2, 1]]
sage: p.maximal_cyclic_decomposition(side='left')
[[1, 0, 7, 5, 4], [1, 0, 5], [2, 1]]
sage: p.maximal_cyclic_decomposition(typ='increasing', side='right')
[[1], [5, 0, 1, 2], [4, 5, 7, 0, 1]]
sage: p.maximal_cyclic_decomposition(typ='increasing', side='left')
[[0, 1, 2, 4, 5], [4, 7, 0, 1], [7]]
```

TESTS:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: S=p.maximal_cyclic_decomposition()
sage: p==prod(A.from_word(l) for l in S)
True
sage: S=p.maximal_cyclic_decomposition(typ='increasing', side='left')
sage: p==prod(A.from_word(l) for l in S)
True
sage: S=p.maximal_cyclic_decomposition(typ='increasing', side='right')
sage: p==prod(A.from_word(l) for l in S)
True
sage: S=p.maximal_cyclic_decomposition(typ='decreasing', side='right')
sage: p==prod(A.from_word(l) for l in S)
True
```

maximal_cyclic_factor(*typ='decreasing'*, *side='right'*, *verbose=False*)

For an affine permutation x, finds the unique maximal subset A of the index set such that $x = yd_A$ is a reduced product.

INPUT:

- `typ` – 'increasing' or 'decreasing.' Determines the type of maximal cyclic element found.
- `side` – 'right' or 'left'.
- `verbose` – True or False. If True, outputs information about how the cyclically increasing element was found.

EXAMPLES:

```
sage: p=AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.maximal_cyclic_factor()
[7, 5, 4, 2, 1]
sage: p.maximal_cyclic_factor(side='left')
[1, 0, 7, 5, 4]
sage: p.maximal_cyclic_factor('increasing','right')
[4, 5, 7, 0, 1]
sage: p.maximal_cyclic_factor('increasing','left')
[0, 1, 2, 4, 5]
```

position(*i*)

Find the position j such the `self.value(j)=i`

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.position(3)
1
sage: p.position(11)
9
```

promotion()

The Dynkin diagram automorphism which sends s_i to s_{i+1}.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.promotion()
Type A affine permutation with window [2, 4, 0, 1, 7, 6, 5, 11]
```

tableau_of_word(*w*, *typ='decreasing'*, *side='right'*, *alpha=None*)

Finds a tableau on the Lehmer code of `self` corresponding to the given reduced word.

For a full description of this algorithm, see [D2012].

INPUT:

- `w` – a reduced word for self.
- `typ` – 'increasing' or 'decreasing.' The type of Lehmer code used.
- `side` – 'right' or 'left.'
- `alpha` – A content vector. w should be of type alpha. Specifying alpha produces semistandard tableaux.

REFERENCES:

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
```

```
sage: p.tableau_of_word(p.reduced_word())
[[], [1, 6, 9], [2, 7, 10], [], [3], [4, 8], [], [5]]
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: w=p.reduced_word()
sage: w
[0, 7, 4, 1, 0, 7, 5, 4, 2, 1]
sage: alpha=[5,3,2]
sage: p.tableau_of_word(p.reduced_word(), alpha=alpha)
[[], [1, 2, 3], [1, 2, 3], [], [1], [1, 2], [], [1]]
sage: p.tableau_of_word(p.reduced_word(), side='left')
[[1, 4, 9], [6], [], [], [3, 7], [8], [], [2, 5, 10]]
sage: p.tableau_of_word(p.reduced_word(), typ='increasing', side='right')
[[9, 10], [1, 2], [], [], [3, 4], [8], [], [5, 6, 7]]
sage: p.tableau_of_word(p.reduced_word(), typ='increasing', side='left')
[[1, 2], [4, 5, 6], [9, 10], [], [3], [7, 8], [], []]
```

to_bounded_partition(*typ='decreasing'*, *side='right'*)

Returns the k-bounded partition associated to the dominant element obtained by sorting the Lehmer code.

INPUT:

- `typ` – 'increasing' or 'decreasing' (default: 'decreasing'.) Chooses whether to find increasing or deacreasing sets.
- `side` – 'right' or 'left' (default: 'right'.) Chooses whether to find maximal sets starting from the left or the right.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',2,1])
sage: p=A.from_lehmer_code([4,1,0])
sage: p.to_bounded_partition()
[2, 1, 1, 1]
```

to_core(*typ='decreasing'*, *side='right'*)

Returns the core associated to the dominant element obtained by sorting the Lehmer code.

INPUT:

- `typ` – 'increasing' or 'decreasing' (default: 'decreasing'.)
- `side` – 'right' or 'left' (default: 'right'.) Chooses whether to find maximal sets starting from the left or the right.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',2,1])
sage: p=A.from_lehmer_code([4,1,0])
sage: p.to_bounded_partition()
[2, 1, 1, 1]
sage: p.to_core()
[4, 2, 1, 1]
```

to_dominant(*typ='decreasing'*, *side='right'*)

Finds the Lehmer code and then sorts it. Returns the affine permutation with the given sorted Lehmer code; this element is 0-dominant.

INPUT:

- `typ` – 'increasing' or 'decreasing' (default: 'decreasing'.) Chooses whether to find increasing or deacreasing sets.

- `side` – 'right' or 'left' (default: 'right'.) Chooses whether to find maximal sets starting from the left or the right.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.to_dominant()
Type A affine permutation with window [-2, -1, 1, 3, 4, 8, 10, 13]
sage: p.to_dominant(typ='increasing', side='left')
Type A affine permutation with window [3, 4, -1, 5, 0, 9, 6, 10]
```

to_lehmer_code(*typ='decreasing'*, *side='right'*)

Returns the affine Lehmer code.

There are four such codes; the options `typ` and `side` determine which code is generated. The codes generated are the shape of the maximal cyclic decompositions of `self` according to the given `typ` and `side` options.

INPUT:

- `typ` – 'increasing' or 'decreasing' (default: 'decreasing'.) Chooses whether to find increasing or deacreasing sets.
- `side` – 'right' or 'left' (default: 'right'.) Chooses whether to find maximal sets starting from the left or the right.

EXAMPLES:

```
sage: import itertools
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: orders = ('increasing','decreasing')
sage: sides = ('left','right')
sage: for o,s in itertools.product(orders, sides):
....:   p.to_lehmer_code(o,s)
[2, 3, 2, 0, 1, 2, 0, 0]
[2, 2, 0, 0, 2, 1, 0, 3]
[3, 1, 0, 0, 2, 1, 0, 3]
[0, 3, 3, 0, 1, 2, 0, 1]
sage: for a in itertools.product(orders, sides):
....:   A.from_lehmer_code(p.to_lehmer_code(a[0],a[1]), a[0],a[1])==p
True
True
True
True
```

to_type_a()

Returns an embedding of `self` into the affine permutation group of type A. (For Type A, just returns self.)

EXAMPLES:

```
sage: p=AffinePermutationGroup(['A',7,1])([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.to_type_a()==p
True
```

value(*i*, *base_window=False*)

Return the image of the integer `i` under this permutation.

INPUT:

- `base_window` – a Boolean, indicating whether i is in the base window. If True, will run a bit faster, but the method will screw up if i is not actually in the index set.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9])
sage: p.value(1)
3
sage: p.value(9)
11
```

class sage.combinat.affine_permutation.**AffinePermutationTypeB**(*parent*, *lst*, *check=True*)

Bases: sage.combinat.affine_permutation.AffinePermutationTypeC

Initialize `self`

INPUT:

- `parent` – The parent affine permutation group.
- `lst` – List giving the base window of the affine permutation.
- `check`– Chooses whether to test that the affine permutation is legit.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9]) #indirect doctest
sage: p
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
```

apply_simple_reflection_left(*i*)

Applies simple reflection indexed by i on values.

EXAMPLES:

```
sage: B=AffinePermutationGroup(['B',4,1])
sage: p=B([-5,1,6,-2])
sage: p.apply_simple_reflection_left(0)
Type B affine permutation with window [-5, -2, 6, 1]
sage: p.apply_simple_reflection_left(2)
Type B affine permutation with window [-5, 1, 7, -3]
sage: p.apply_simple_reflection_left(4)
Type B affine permutation with window [-4, 1, 6, -2]
```

apply_simple_reflection_right(*i*)

Applies the simple reflection indexed by i on positions.

EXAMPLES:

```
sage: B=AffinePermutationGroup(['B',4,1])
sage: p=B([-5,1,6,-2])
sage: p.apply_simple_reflection_right(1)
Type B affine permutation with window [1, -5, 6, -2]
sage: p.apply_simple_reflection_right(0)
Type B affine permutation with window [-1, 5, 6, -2]
sage: p.apply_simple_reflection_right(4)
Type B affine permutation with window [-5, 1, 6, 11]
```

check()

Check that `self` is an affine permutation.

EXAMPLES:

```
sage: B=AffinePermutationGroup(['B',4,1])
sage: x=B([-5,1,6,-2])
sage: x
Type B affine permutation with window [-5, 1, 6, -2]
```

has_left_descent(*i*)

Determines whether there is a descent at i.

INPUT:

- `i` – an integer.

EXAMPLES:

```
sage: B=AffinePermutationGroup(['B',4,1])
sage: p=B([-5,1,6,-2])
sage: [p.has_left_descent(i) for i in B.index_set()]
[True, True, False, False, True]
```

has_right_descent(*i*)

Determines whether there is a descent at index i.

INPUT:

- `i` – an integer.

EXAMPLES:

```
sage: B=AffinePermutationGroup(['B',4,1])
sage: p=B([-5,1,6,-2])
sage: [p.has_right_descent(i) for i in B.index_set()]
[True, False, False, True, False]
```

class sage.combinat.affine_permutation.**AffinePermutationTypeC**(*parent*, *lst*, *check=True*)

Bases: sage.combinat.affine_permutation.AffinePermutation

Initialize `self`

INPUT:

- `parent` – The parent affine permutation group.
- `lst` – List giving the base window of the affine permutation.
- `check`– Chooses whether to test that the affine permutation is legit.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9]) #indirect doctest
sage: p
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
```

apply_simple_reflection_left(*i*)

Applies simple reflection indexed by i on values.

EXAMPLES:

```
sage: C=AffinePermutationGroup(['C',4,1])
sage: x=C([-1,5,3,7])
sage: for i in C.index_set(): x.apply_simple_reflection_left(i)
Type C affine permutation with window [1, 5, 3, 7]
```

```
Type C affine permutation with window [-2, 5, 3, 8]
Type C affine permutation with window [-1, 5, 2, 6]
Type C affine permutation with window [-1, 6, 4, 7]
Type C affine permutation with window [-1, 4, 3, 7]
```

apply_simple_reflection_right(*i*)

Applies the simple reflection indexed by i on positions.

EXAMPLES:

```
sage: C=AffinePermutationGroup(['C',4,1])
sage: x=C([-1,5,3,7])
sage: for i in C.index_set(): x.apply_simple_reflection_right(i)
Type C affine permutation with window [1, 5, 3, 7]
Type C affine permutation with window [5, -1, 3, 7]
Type C affine permutation with window [-1, 3, 5, 7]
Type C affine permutation with window [-1, 5, 7, 3]
Type C affine permutation with window [-1, 5, 3, 2]
```

check()

Check that `self` is an affine permutation.

EXAMPLES:

```
sage: C=AffinePermutationGroup(['C',4,1])
sage: x=C([-1,5,3,7])
sage: x
Type C affine permutation with window [-1, 5, 3, 7]
```

has_left_descent(*i*)

Determines whether there is a descent at i.

INPUT:

- `i` – an integer.

EXAMPLES:

```
sage: C=AffinePermutationGroup(['C',4,1])
sage: x=C([-1,5,3,7])
sage: for i in C.index_set(): x.has_left_descent(i)
True
False
True
False
True
```

has_right_descent(*i*)

Determines whether there is a descent at index i.

INPUT:

- `i` – an integer.

EXAMPLES:

```
sage: C=AffinePermutationGroup(['C',4,1])
sage: x=C([-1,5,3,7])
sage: for i in C.index_set(): x.has_right_descent(i)
True
False
True
```

```
False
True
```

position(*i*)

Find the position j such the `self.value(j)=i`

EXAMPLES:

```
sage: C=AffinePermutationGroup(['C',4,1])
sage: x=C.one()
sage: [x.position(i) for i in range(-10,10)]==range(-10,10)
True
```

to_type_a()

Returns an embedding of `self` into the affine permutation group of type A.

EXAMPLES:

```
sage: C=AffinePermutationGroup(['C',4,1])
sage: x=C([-1,5,3,7])
sage: x.to_type_a()
Type A affine permutation with window [-1, 5, 3, 7, 2, 6, 4, 10, 9]
```

value(*i*)

Returns the image of the integer i under this permutation.

EXAMPLES:

```
sage: C=AffinePermutationGroup(['C',4,1])
sage: x=C.one()
sage: [x.value(i) for i in range(-10,10)]==range(-10,10)
True
```

class sage.combinat.affine_permutation.**AffinePermutationTypeD**(*parent*, *lst*, *check=True*)

Bases: sage.combinat.affine_permutation.AffinePermutationTypeC

Initialize `self`

INPUT:

- `parent` – The parent affine permutation group.
- `lst` – List giving the base window of the affine permutation.
- `check`– Chooses whether to test that the affine permutation is legit.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9]) #indirect doctest
sage: p
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
```

apply_simple_reflection_left(*i*)

Applies simple reflection indexed by i on values.

EXAMPLES:

```
sage: D=AffinePermutationGroup(['D',4,1])
sage: p=D([1,-6,5,-2])
sage: p.apply_simple_reflection_left(0)
Type D affine permutation with window [-2, -6, 5, 1]
```

```
sage: p.apply_simple_reflection_left(1)
Type D affine permutation with window [2, -6, 5, -1]
sage: p.apply_simple_reflection_left(4)
Type D affine permutation with window [1, -4, 3, -2]
```

apply_simple_reflection_right(*i*)

Applies the simple reflection indexed by i on positions.

EXAMPLES:

```
sage: D=AffinePermutationGroup(['D',4,1])
sage: p=D([1,-6,5,-2])
sage: p.apply_simple_reflection_right(0)
Type D affine permutation with window [6, -1, 5, -2]
sage: p.apply_simple_reflection_right(1)
Type D affine permutation with window [-6, 1, 5, -2]
sage: p.apply_simple_reflection_right(4)
Type D affine permutation with window [1, -6, 11, 4]
```

check()

Check that `self` is an affine permutation.

EXAMPLES:

```
sage: D=AffinePermutationGroup(['D',4,1])
sage: p=D([1,-6,5,-2])
sage: p
Type D affine permutation with window [1, -6, 5, -2]
```

has_left_descent(*i*)

Determines whether there is a descent at i.

INPUT:

- `i` – an integer.

EXAMPLES:

```
sage: D=AffinePermutationGroup(['D',4,1])
sage: p=D([1,-6,5,-2])
sage: [p.has_left_descent(i) for i in D.index_set()]
[True, True, False, True, True]
```

has_right_descent(*i*)

Determines whether there is a descent at index i.

INPUT:

- `i` – an integer.

EXAMPLES:

```
sage: D=AffinePermutationGroup(['D',4,1])
sage: p=D([1,-6,5,-2])
sage: [p.has_right_descent(i) for i in D.index_set()]
[True, True, False, True, False]
```

class sage.combinat.affine_permutation.**AffinePermutationTypeG**(*parent*, *lst*, *check=True*)

Bases: sage.combinat.affine_permutation.AffinePermutation

Initialize `self`

INPUT:

- `parent` – The parent affine permutation group.
- `lst` – List giving the base window of the affine permutation.
- `check`– Chooses whether to test that the affine permutation is legit.

EXAMPLES:

```
sage: A=AffinePermutationGroup(['A',7,1])
sage: p=A([3, -1, 0, 6, 5, 4, 10, 9]) #indirect doctest
sage: p
Type A affine permutation with window [3, -1, 0, 6, 5, 4, 10, 9]
```

apply_simple_reflection_left(i)

Applies simple reflection indexed by i on values.

EXAMPLES:

```
sage: G=AffinePermutationGroup(['G',2,1])
sage: p=G([2, 10, -5, 12, -3, 5])
sage: p.apply_simple_reflection_left(0)
Type G affine permutation with window [0, 10, -7, 14, -3, 7]
sage: p.apply_simple_reflection_left(1)
Type G affine permutation with window [1, 9, -4, 11, -2, 6]
sage: p.apply_simple_reflection_left(2)
Type G affine permutation with window [3, 11, -5, 12, -4, 4]
```

apply_simple_reflection_right(i)

Applies the simple reflection indexed by i on positions.

EXAMPLES:

```
sage: G=AffinePermutationGroup(['G',2,1])
sage: p=G([2, 10, -5, 12, -3, 5])
sage: p.apply_simple_reflection_right(0)
Type G affine permutation with window [-9, -1, -5, 12, 8, 16]
sage: p.apply_simple_reflection_right(1)
Type G affine permutation with window [10, 2, 12, -5, 5, -3]
sage: p.apply_simple_reflection_right(2)
Type G affine permutation with window [2, -5, 10, -3, 12, 5]
```

check()

Check that `self` is an affine permutation.

EXAMPLES:

```
sage: G=AffinePermutationGroup(['G',2,1])
sage: p=G([2, 10, -5, 12, -3, 5])
sage: p
Type G affine permutation with window [2, 10, -5, 12, -3, 5]
```

has_left_descent(i)

Determines whether there is a descent at i.

INPUT:

- `i` – an integer.

EXAMPLES:

```
sage: G=AffinePermutationGroup(['G',2,1])
sage: p=G([2, 10, -5, 12, -3, 5])
```

```
sage: [p.has_left_descent(i) for i in G.index_set()]
[False, True, False]
```

has_right_descent(*i*)
Determines whether there is a descent at index i.

INPUT:

- `i` – an integer.

EXAMPLES:

```
sage: G=AffinePermutationGroup(['G',2,1])
sage: p=G([2, 10, -5, 12, -3, 5])
sage: [p.has_right_descent(i) for i in G.index_set()]
[False, False, True]
```

position(*i*)
Find the position j such the `self.value(j)=i`

EXAMPLES:

```
sage: G=AffinePermutationGroup(['G',2,1])
sage: p=G([2, 10, -5, 12, -3, 5])
sage: [p.position(i) for i in p._lst]
[1, 2, 3, 4, 5, 6]
```

to_type_a()
Returns an embedding of `self` into the affine permutation group of type A.

EXAMPLES:

```
sage: G=AffinePermutationGroup(['G',2,1])
sage: p=G([2, 10, -5, 12, -3, 5])
sage: p.to_type_a()
Type A affine permutation with window [2, 10, -5, 12, -3, 5]
```

value(*i*, *base_window=False*)
Returns the image of the integer i under this permutation.

INPUT:

- `base_window` – a Boolean indicating whether i is between 1 and $k+1$. If True, will run a bit faster, but the method will screw up if i is not actually in the index set.

EXAMPLES:

```
sage: G=AffinePermutationGroup(['G',2,1])
sage: p=G([2, 10, -5, 12, -3, 5])
sage: [p.value(i) for i in [1..12]]
[2, 10, -5, 12, -3, 5, 8, 16, 1, 18, 3, 11]
```

5.1.4 Algebraic combinatorics

Quickref

Todo

write it!

Thematic tutorials

Todo
get Sphinx to create those cross links properly

- Algebraic Combinatorics in Sage
- Lie Methods and Related Combinatorics in Sage
- Linear Programming (Mixed Integer)

Enumerated sets of combinatorial objects

- *Enumerated sets of partitions, tableaux, ...*
- `GelfandTsetlinPattern`, `GelfandTsetlinPatterns`
- `KnutsonTaoPuzzleSolver`

Combinatorial Hopf Algebras

- *Symmetric Functions*
- *Non-Commutative Symmetric Functions and Quasi-Symmetric Functions*
- *Schubert Polynomials*
- *Symmetric Functions in Non-Commuting Variables*

Groups and Algebras

Todo
add link to the catalog of algebras when it exists

- *Groups*
- `SymmetricGroup`, `CoxeterGroup`, `WeylGroup`
- `PartitionAlgebra`
- `IwahoriHeckeAlgebra`
- `SymmetricGroupAlgebra`
- `NilCoxeterAlgebra`
- `AffineNilTemperleyLiebTypeA`
- *Descent Algebras*
- *Diagram and Partition Algebras*

Combinatorial Representation Theory

- *Root Systems*
- *Crystals*
- *Rigged Configurations*
- *Cluster Algebras and Quivers*
- KazhdanLusztigPolynomial
- SymmetricGroupRepresentation
- *Yang-Baxter Graphs*
- *Hall Polynomials*

Operads and their algebras

- *Free Pre-Lie Algebras*

5.1.5 Combinatorics features that are imported by default in the interpreter namespace

sage.combinat.all.**SetPartitionsAk**(*k*)

Returns the combinatorial class of set partitions of type A_k.

EXAMPLES:

```
sage: A3 = SetPartitionsAk(3); A3
Set partitions of {1, ..., 3, -1, ..., -3}

sage: A3.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: A3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
sage: A3.random_element()  #random
{{1, 3, -3, -1}, {2, -2}}

sage: A3.cardinality()
203

sage: A2p5 = SetPartitionsAk(2.5); A2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block
sage: A2p5.cardinality()
52

sage: A2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: A2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
sage: A2p5.random_element() #random
{{-1}, {-2}, {3, -3}, {1, 2}}
```

sage.combinat.all.**SetPartitionsBk**(*k*)

Returns the combinatorial class of set partitions of type B_k. These are the set partitions where every block has size 2.

EXAMPLES:

```
sage: B3 = SetPartitionsBk(3); B3
Set partitions of {1, ..., 3, -1, ..., -3} with block size 2

sage: B3.first() #random
{{2, -2}, {1, -3}, {3, -1}}
sage: B3.last() #random
{{1, 2}, {3, -2}, {-3, -1}}
sage: B3.random_element() #random
{{2, -1}, {1, -3}, {3, -2}}

sage: B3.cardinality()
15

sage: B2p5 = SetPartitionsBk(2.5); B2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and with block size 2

sage: B2p5.first() #random
{{2, -1}, {3, -3}, {1, -2}}
sage: B2p5.last() #random
{{1, 2}, {3, -3}, {-1, -2}}
sage: B2p5.random_element() #random
{{2, -2}, {3, -3}, {1, -1}}

sage: B2p5.cardinality()
3
```

sage.combinat.all.**SetPartitionsIk**(*k*)

Returns the combinatorial class of set partitions of type I_k. These are set partitions with a propagating number of less than k. Note that the identity set partition {{1, -1}, ..., {k, -k}} is not in I_k.

EXAMPLES:

```
sage: I3 = SetPartitionsIk(3); I3
Set partitions of {1, ..., 3, -1, ..., -3} with propagating number < 3
sage: I3.cardinality()
197

sage: I3.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: I3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
sage: I3.random_element() #random
{{-1}, {-3, -2}, {2, 3}, {1}}

sage: I2p5 = SetPartitionsIk(2.5); I2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and propagating numbe
sage: I2p5.cardinality()
50

sage: I2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: I2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
sage: I2p5.random_element() #random
{{-1}, {-2}, {1, 3, -3}, {2}}
```

sage.combinat.all.**SetPartitionsPRk**(*k*)

sage.combinat.all.**SetPartitionsPk**(*k*)

Returns the combinatorial class of set partitions of type P_k. These are the planar set partitions.

EXAMPLES:

```
sage: P3 = SetPartitionsPk(3); P3
Set partitions of {1, ..., 3, -1, ..., -3} that are planar
sage: P3.cardinality()
132

sage: P3.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: P3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
sage: P3.random_element() #random
{{1, 2, -1}, {-3}, {3, -2}}

sage: P2p5 = SetPartitionsPk(2.5); P2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and that are planar
sage: P2p5.cardinality()
42

sage: P2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: P2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
sage: P2p5.random_element() #random
{{1, 2, 3, -3}, {-1, -2}}
```

sage.combinat.all.**SetPartitionsRk**(*k*)

sage.combinat.all.**SetPartitionsSk**(*k*)

Returns the combinatorial class of set partitions of type S_k. There is a bijection between these set partitions and the permutations of 1, ..., k.

EXAMPLES:

```
sage: S3 = SetPartitionsSk(3); S3
Set partitions of {1, ..., 3, -1, ..., -3} with propagating number 3
sage: S3.cardinality()
6

sage: S3.list()  #random
[{{2, -2}, {3, -3}, {1, -1}},
 {{1, -1}, {2, -3}, {3, -2}},
 {{2, -1}, {3, -3}, {1, -2}},
 {{1, -2}, {2, -3}, {3, -1}},
 {{1, -3}, {2, -1}, {3, -2}},
 {{1, -3}, {2, -2}, {3, -1}}]
sage: S3.first() #random
{{2, -2}, {3, -3}, {1, -1}}
sage: S3.last() #random
{{1, -3}, {2, -2}, {3, -1}}
sage: S3.random_element() #random
{{1, -3}, {2, -1}, {3, -2}}

sage: S3p5 = SetPartitionsSk(3.5); S3p5
Set partitions of {1, ..., 4, -1, ..., -4} with 4 and -4 in the same block and propagating numbe
sage: S3p5.cardinality()
6
```

```
sage: S3p5.list() #random
[{{2, -2}, {3, -3}, {1, -1}, {4, -4}},
 {{2, -3}, {1, -1}, {4, -4}, {3, -2}},
 {{2, -1}, {3, -3}, {1, -2}, {4, -4}},
 {{2, -3}, {1, -2}, {4, -4}, {3, -1}},
 {{1, -3}, {2, -1}, {4, -4}, {3, -2}},
 {{1, -3}, {2, -2}, {4, -4}, {3, -1}}]
sage: S3p5.first() #random
{{2, -2}, {3, -3}, {1, -1}, {4, -4}}
sage: S3p5.last() #random
{{1, -3}, {2, -2}, {4, -4}, {3, -1}}
sage: S3p5.random_element() #random
{{1, -3}, {2, -2}, {4, -4}, {3, -1}}
```

sage.combinat.all.**SetPartitionsTk**(*k*)

Returns the combinatorial class of set partitions of type T_k. These are planar set partitions where every block is of size 2.

EXAMPLES:

```
sage: T3 = SetPartitionsTk(3); T3
Set partitions of {1, ..., 3, -1, ..., -3} with block size 2 and that are planar
sage: T3.cardinality()
5

sage: T3.first() #random
{{1, -3}, {2, 3}, {-1, -2}}
sage: T3.last() #random
{{1, 2}, {3, -1}, {-3, -2}}
sage: T3.random_element() #random
{{1, -3}, {2, 3}, {-1, -2}}

sage: T2p5 = SetPartitionsTk(2.5); T2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and with block size 2
sage: T2p5.cardinality()
2

sage: T2p5.first() #random
{{2, -2}, {3, -3}, {1, -1}}
sage: T2p5.last() #random
{{1, 2}, {3, -3}, {-1, -2}}
```

5.1.6 Alternating Sign Matrices

AUTHORS:

- Mike Hansen (2007): Initial version
- Pierre Cange, Luis Serrano (2012): Added monotone triangles
- Travis Scrimshaw (2013-28-03): Added element class for ASM's and made `MonotoneTriangles` inherit from `GelfandTsetlinPatterns`
- Jessica Striker (2013): Added additional methods

class sage.combinat.alternating_sign_matrix.**AlternatingSignMatrices**(*n*, *use_monotone_triangles=None*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Class of all $n \times n$ alternating sign matrices.

An alternating sign matrix of size n is an $n \times n$ matrix of 0's, 1's and -1's such that the sum of each row and column is 1 and the non-zero entries in each row and column alternate in sign.

Alternating sign matrices of size n are in bijection with monotone triangles with n rows.

INPUT:

- n – an integer, the size of the matrices.
- use_monotone_triangle – deprecated

EXAMPLES:

This will create an instance to manipulate the alternating sign matrices of size 3:

```
sage: A = AlternatingSignMatrices(3)
sage: A
Alternating sign matrices of size 3
sage: A.cardinality()
7
```

Notably, this implementation allows to make a lattice of it:

```
sage: L = A.lattice()
sage: L
Finite lattice containing 7 elements
sage: L.category()
Join of Category of finite lattice posets
 and Category of finite enumerated sets
 and Category of facade sets
```

Element

alias of AlternatingSignMatrix

cardinality()

Return the cardinality of self.

The number of $n \times n$ alternating sign matrices is equal to

$$\prod_{k=0}^{n-1} \frac{(3k+1)!}{(n+k)!} = \frac{1!4!7!10!\cdots(3n-2)!}{n!(n+1)!(n+2)!(n+3)!\cdots(2n-1)!}$$

EXAMPLES:

```
sage: [AlternatingSignMatrices(n).cardinality() for n in range(0, 11)]
[1, 1, 2, 7, 42, 429, 7436, 218348, 10850216, 911835460, 129534272700]
```

cover_relations()

Iterate on the cover relations between the alternating sign matrices.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: for (a,b) in A.cover_relations():
....:     eval('a, b')
(
[1 0 0]  [0 1 0]
[0 1 0]  [1 0 0]
[0 0 1], [0 0 1]
)
(
```

```
[1 0 0]  [1 0 0]
[0 1 0]  [0 0 1]
[0 0 1], [0 1 0]
)
(
[0 1 0]  [ 0  1  0]
[1 0 0]  [ 1 -1  1]
[0 0 1], [ 0  1  0]
)
(
[1 0 0]  [ 0  1  0]
[0 0 1]  [ 1 -1  1]
[0 1 0], [ 0  1  0]
)
(
[ 0  1  0]  [0 0 1]
[ 1 -1  1]  [1 0 0]
[ 0  1  0], [0 1 0]
)
(
[ 0  1  0]  [0 1 0]
[ 1 -1  1]  [0 0 1]
[ 0  1  0], [1 0 0]
)
(
[0 0 1]  [0 0 1]
[1 0 0]  [0 1 0]
[0 1 0], [1 0 0]
)
(
[0 1 0]  [0 0 1]
[0 0 1]  [0 1 0]
[1 0 0], [1 0 0]
)
```

from_contre_tableau(*comps*)

Return an alternating sign matrix from a contre-tableau.

EXAMPLES:

```
sage: ASM = AlternatingSignMatrices(3)
sage: ASM.from_contre_tableau([[1, 2, 3], [1, 2], [1]])
[0 0 1]
[0 1 0]
[1 0 0]
sage: ASM.from_contre_tableau([[1, 2, 3], [2, 3], [3]])
[1 0 0]
[0 1 0]
[0 0 1]
```

from_corner_sum(*corner*)

Return an alternating sign matrix from a corner sum matrix.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A.from_corner_sum(matrix([[0,0,0,0],[0,1,1,1],[0,1,2,2],[0,1,2,3]]))
[1 0 0]
[0 1 0]
```

```
[0 0 1]
sage: A.from_corner_sum(matrix([[0,0,0,0],[0,0,1,1],[0,1,1,2],[0,1,2,3]]))
[ 0  1  0]
[ 1 -1  1]
[ 0  1  0]
```

from_height_function(*height*)

Return an alternating sign matrix from a height function.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A.from_height_function(matrix([[0,1,2,3],[1,2,1,2],[2,3,2,1],[3,2,1,0]]))
[0 0 1]
[1 0 0]
[0 1 0]
sage: A.from_height_function(matrix([[0,1,2,3],[1,2,1,2],[2,1,2,1],[3,2,1,0]]))
[ 0  1  0]
[ 1 -1  1]
[ 0  1  0]
```

from_monotone_triangle(*triangle*)

Return an alternating sign matrix from a monotone triangle.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A.from_monotone_triangle([[3, 2, 1], [2, 1], [1]])
[1 0 0]
[0 1 0]
[0 0 1]
sage: A.from_monotone_triangle([[3, 2, 1], [3, 2], [3]])
[0 0 1]
[0 1 0]
[1 0 0]
```

gyration_orbit_sizes()

Return the sizes of gyration orbits of `self`.

EXAMPLES:

```
sage: AlternatingSignMatrices(3).gyration_orbit_sizes()
[3, 2, 2]
sage: AlternatingSignMatrices(4).gyration_orbit_sizes()
[4, 8, 2, 8, 8, 8, 2, 2]

sage: A = AlternatingSignMatrices(5)
sage: li = [5,10,10,10,10,10,2,5,10,10,10,10,10,10,10,10,10,10,10,\
4,10,10,10,10,10,10,4,5,10,10,10,10,10,10,10,2,4,5,10,10,10,10,10,10,\
4,5,10,10,2,2]
sage: A.gyration_orbit_sizes() == li
True
```

gyration_orbits()

Return the list of gyration orbits of `self`.

EXAMPLES:

```
sage: AlternatingSignMatrices(3).gyration_orbits()
((
  [1 0 0]  [0 0 1]  [ 0  1  0]
```

```
 [0 1 0]  [0 1 0]  [ 1 -1  1]
 [0 0 1], [1 0 0], [ 0  1  0]
),
(
 [0 1 0]  [1 0 0]
 [1 0 0]  [0 0 1]
 [0 0 1], [0 1 0]
),
(
 [0 0 1]  [0 1 0]
 [1 0 0]  [0 0 1]
 [0 1 0], [1 0 0]
))
```

lattice()
Return the lattice of the alternating sign matrices of size n, created by `LatticePoset`.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: L = A.lattice()
sage: L
Finite lattice containing 7 elements
```

matrix_space()
Return the underlying matrix space.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A.matrix_space()
Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
```

size()
Return the size of the matrices in `self`.

TESTS:

```
sage: A = AlternatingSignMatrices(4)
sage: A.size()
4
```

class sage.combinat.alternating_sign_matrix.**AlternatingSignMatrix**(*parent*, *asm*)
Bases: sage.structure.element.Element

An alternating sign matrix.

An alternating sign matrix is a square matrix of 0's, 1's and -1's such that the sum of each row and column is 1 and the non-zero entries in each row and column alternate in sign.

These were introduced in [MiRoRu].

REFERENCES:

ASM_compatible(B)
Return `True` if `self` and `B` are compatible alternating sign matrices in the sense of [EKLP92]. (If `self` is of size n, `B` must be of size $n+1$.)

In [EKLP92], there is a notion of a pair of ASM's with sizes differing by 1 being compatible, in the sense that they can be combined to encode a tiling of the Aztec Diamond.

REFERENCES:

EXAMPLES:

```
sage: A = AlternatingSignMatrix(matrix([[0,0,1,0],[0,1,-1,1],[1,0,0,0],[0,0,1,0]]))
sage: B = AlternatingSignMatrix(matrix([[0,0,1,0,0],[0,0,0,1,0],[1,0,0,-1,1],[0,1,0,0,0],[0,
sage: A.ASM_compatible(B)
True
sage: A = AlternatingSignMatrix(matrix([[0,1,0],[1,-1,1],[0,1,0]]))
sage: B = AlternatingSignMatrix(matrix([[0,0,1,0],[0,0,0,1],[1,0,0,0],[0,1,0,0]]))
sage: A.ASM_compatible(B)
False
```

ASM_compatible_bigger()

Return all ASM's compatible with `self` that are of size one greater than `self`.

Given an $n \times n$ alternating sign matrix A, there are as many ASM's of size $n+1$ compatible with A as 2 raised to the power of the number of 1's in A [EKLP92].

EXAMPLES:

```
sage: A = AlternatingSignMatrix(matrix([[1,0],[0,1]]))
sage: A.ASM_compatible_bigger()
[
[ 0  1  0]  [1 0 0]  [0 1 0]  [1 0 0]
[ 1 -1  1]  [0 0 1]  [1 0 0]  [0 1 0]
[ 0  1  0], [0 1 0], [0 0 1], [0 0 1]
]
sage: B = AlternatingSignMatrix(matrix([[0,1],[1,0]]))
sage: B.ASM_compatible_bigger()
[
[0 0 1]  [0 0 1]  [0 1 0]  [ 0  1  0]
[0 1 0]  [1 0 0]  [0 0 1]  [ 1 -1  1]
[1 0 0], [0 1 0], [1 0 0], [ 0  1  0]
]
```

ASM_compatible_smaller()

Return the list of all ASMs compatible with `self` that are of size one smaller than `self`.

Given an alternating sign matrix A of size n, there are as many ASM's of size $n-1$ compatible with it as 2 raised to the power of the number of -1's in A [EKLP92].

EXAMPLES:

```
sage: A = AlternatingSignMatrix(matrix([[0,0,1,0],[0,1,-1,1],[1,0,0,0],[0,0,1,0]]))
sage: A.ASM_compatible_smaller()
[
[0 0 1]  [ 0  1  0]
[1 0 0]  [ 1 -1  1]
[0 1 0], [ 0  1  0]
]
sage: B = AlternatingSignMatrix(matrix([[1,0,0],[0,0,1],[0,1,0]]))
sage: B.ASM_compatible_smaller()
[
[1 0]
[0 1]
]
```

corner_sum_matrix()

Return the corner sum matrix from `self`.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).corner_sum_matrix()
[0 0 0 0]
[0 1 1 1]
[0 1 2 2]
[0 1 2 3]
sage: asm = A([[0, 1, 0],[1, -1, 1],[0, 1, 0]])
sage: asm.corner_sum_matrix()
[0 0 0 0]
[0 0 1 1]
[0 1 1 2]
[0 1 2 3]
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.corner_sum_matrix()
[0 0 0 0]
[0 0 0 1]
[0 1 1 2]
[0 1 2 3]
```

TESTS:

Some non-symmetric tests:

```
sage: A = AlternatingSignMatrices(3)
sage: asm = A([[0, 1, 0], [0, 0, 1], [1, 0, 0]])
sage: asm.corner_sum_matrix()
[0 0 0 0]
[0 0 1 1]
[0 0 1 2]
[0 1 2 3]
sage: B = AlternatingSignMatrices(4)
sage: asm = B([[0, 0, 1, 0], [1, 0, 0, 0], [0, 1, -1, 1], [0, 0, 1, 0]])
sage: asm.corner_sum_matrix()
[0 0 0 0 0]
[0 0 0 1 1]
[0 1 1 2 2]
[0 1 2 2 3]
[0 1 2 3 4]
```

gyration()

Return the alternating sign matrix obtained by applying gyration to the height function in bijection with `self`.

Gyration acts on height functions as follows. Go through the entries of the matrix, first those for which the sum of the row and column indices is even, then for those for which it is odd, and increment or decrement the squares by 2 wherever possible such that the resulting matrix is still a height function. Gyration was first defined in [Wieland00] as an action on fully-packed loops.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).gyration()
[0 0 1]
[0 1 0]
[1 0 0]
sage: asm = A([[0, 1, 0],[1, -1, 1],[0, 1, 0]])
sage: asm.gyration()
[1 0 0]
[0 1 0]
[0 0 1]
```

```
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.gyration()
[0 1 0]
[0 0 1]
[1 0 0]
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).gyration().gyration()
[ 0  1  0]
[ 1 -1  1]
[ 0  1  0]
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).gyration().gyration().gyration()
[1 0 0]
[0 1 0]
[0 0 1]

sage: A = AlternatingSignMatrices(4)
sage: M = A([[0,0,1,0],[1,0,0,0],[0,1,-1,1],[0,0,1,0]])
sage: for i in range(5):
....:     M = M.gyration()
sage: M
[1 0 0 0]
[0 0 0 1]
[0 1 0 0]
[0 0 1 0]
```

gyration_orbit()

Return the gyration orbit of `self` (including `self`)

EXAMPLES:

```
sage: AlternatingSignMatrix([[0,1,0],[1,-1,1],[0,1,0]]).gyration_orbit()
[
[ 0  1  0]  [1 0 0]  [0 0 1]
[ 1 -1  1]  [0 1 0]  [0 1 0]
[ 0  1  0], [0 0 1], [1 0 0]
]

sage: AlternatingSignMatrix([[0,1,0,0],[1,-1,1,0],[0,1,-1,1],[0,0,1,0]]).gyration_orbit()
[
[ 0  1  0  0]  [1 0 0 0]  [ 0  0  1  0]  [0 0 0 1]
[ 1 -1  1  0]  [0 1 0 0]  [ 0  1 -1  1]  [0 0 1 0]
[ 0  1 -1  1]  [0 0 1 0]  [ 1 -1  1  0]  [0 1 0 0]
[ 0  0  1  0], [0 0 0 1], [ 0  1  0  0], [1 0 0 0]
]

sage: len(AlternatingSignMatrix([[0,1,0,0,0,0],[0,0,1,0,0,0],[1,-1,0,0,0,1],\
[0,1,0,0,0,0],[0,0,0,1,0,0],[0,0,0,0,1,0]]).gyration_orbit())
12
```

height_function()

Return the height function from `self`. A height function corresponding to an $n \times n$ ASM is an $(n+1) \times (n+1)$ matrix such that the first row is $0, 1, \ldots, n$, the last row is $n, n-1, \ldots, 1, 0$, and the difference between adjacent entries is 1.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).height_function()
[0 1 2 3]
```

```
[1 0 1 2]
[2 1 0 1]
[3 2 1 0]
sage: asm = A([[0, 1, 0],[1, -1, 1],[0, 1, 0]])
sage: asm.height_function()
[0 1 2 3]
[1 2 1 2]
[2 1 2 1]
[3 2 1 0]
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.height_function()
[0 1 2 3]
[1 2 1 2]
[2 3 2 1]
[3 2 1 0]
```

inversion_number()

Return the inversion number of `self`.

If we denote the entries of the alternating sign matrix as $a_{i,j}$, the inversion number is defined as $\sum_{i>k} \sum_{j<l} a_{i,j} a_{k,l}$. When restricted to permutation matrices, this gives the usual inversion number of the permutation.

This definition is equivalent to the one given in [MiRoRu].

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).inversion_number()
0
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.inversion_number()
2
sage: asm = A([[0, 1, 0],[1, -1, 1],[0, 1, 0]])
sage: asm.inversion_number()
2
sage: P=Permutations(5)
sage: all(p.number_of_inversions()==AlternatingSignMatrix(p.to_matrix()).inversion_number()
True
```

is_permutation()

Return `True` if `self` is a permutation matrix and `False` otherwise.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: asm = A([[0,1,0],[1,0,0],[0,0,1]])
sage: asm.is_permutation()
True
sage: asm = A([[0,1,0],[1,-1,1],[0,1,0]])
sage: asm.is_permutation()
False
```

left_key()

Return the left key of the alternating sign matrix `self`.

The left key of an alternating sign matrix was defined by Lascoux in [LascouxPreprint] and is obtained by successively removing all the -1'suntil what remains is a permutation matrix. This notion corresponds to the notion of left key for semistandard tableaux. So our algorithm proceeds as follows: we map `self` to

its corresponding monotone triangle, view that monotone triangle as a semistandard tableaux, take its left key, and then map back through monotone triangles to the permutation matrix which is the left key.

REFERENCES:

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[0,0,1],[1,0,0],[0,1,0]]).left_key()
[0 0 1]
[1 0 0]
[0 1 0]
sage: t = A([[0,1,0],[1,-1,1],[0,1,0]]).left_key(); t
[1 0 0]
[0 0 1]
[0 1 0]
sage: parent(t)
Alternating sign matrices of size 3
```

left_key_as_permutation()

Return the permutation of the left key of `self`.

See also:

- left_key()

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[0,0,1],[1,0,0],[0,1,0]]).left_key_as_permutation()
[3, 1, 2]
sage: t = A([[0,1,0],[1,-1,1],[0,1,0]]).left_key_as_permutation(); t
[1, 3, 2]
sage: parent(t)
Standard permutations
```

link_pattern()

Return the link pattern corresponding to the fully packed loop corresponding to self.

EXAMPLES:

We can extract the underlying link pattern (a non-crossing partition) from a fully packed loop:

```
sage: A = AlternatingSignMatrix([[0, 1, 0], [1, -1, 1], [0, 1, 0]])
sage: A.link_pattern()
[(1, 2), (3, 6), (4, 5)]

sage: B = AlternatingSignMatrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
sage: B.link_pattern()
[(1, 6), (2, 5), (3, 4)]
```

number_negative_ones()

Return the number of entries in `self` equal to -1.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: asm = A([[0,1,0],[1,0,0],[0,0,1]])
sage: asm.number_negative_ones()
0
sage: asm = A([[0,1,0],[1,-1,1],[0,1,0]])
```

```
sage: asm.number_negative_ones()
1
```

rotate_ccw()
Return the counterclockwise quarter turn rotation of `self`.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).rotate_ccw()
[0 0 1]
[0 1 0]
[1 0 0]
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.rotate_ccw()
[1 0 0]
[0 0 1]
[0 1 0]
```

rotate_cw()
Return the clockwise quarter turn rotation of `self`.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).rotate_cw()
[0 0 1]
[0 1 0]
[1 0 0]
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.rotate_cw()
[0 1 0]
[1 0 0]
[0 0 1]
```

to_dyck_word(*algorithm*)
Return a Dyck word determined by the specified algorithm.

The algorithm 'last_diagonal' uses the last diagonal of the monotone triangle corresponding to `self`. The algorithm 'link_pattern' returns the Dyck word in bijection with the link pattern of the fully packed loop.

Note that these two algorithms in general yield different Dyck words for a given alternating sign matrix.

INPUT:

- `algorithm` -
  - `'last_diagonal'`
  - `'link_pattern'`

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[0,1,0],[1,0,0],[0,0,1]]).to_dyck_word(algorithm = 'last_diagonal')
[1, 1, 0, 0, 1, 0]
sage: d = A([[0,1,0],[1,-1,1],[0,1,0]]).to_dyck_word(algorithm = 'last_diagonal'); d
[1, 1, 0, 1, 0, 0]
sage: parent(d)
Complete Dyck words
sage: A = AlternatingSignMatrices(3)
sage: asm = A([[0,1,0],[1,0,0],[0,0,1]])
```

```
sage: asm.to_dyck_word(algorithm = 'link_pattern')
[1, 0, 1, 0, 1, 0]
sage: asm = A([[0,1,0],[1,-1,1],[0,1,0]])
sage: asm.to_dyck_word(algorithm = 'link_pattern')
[1, 0, 1, 1, 0, 0]
sage: A = AlternatingSignMatrices(4)
sage: asm = A([[0,0,1,0],[1,0,0,0],[0,1,-1,1],[0,0,1,0]])
sage: asm.to_dyck_word(algorithm = 'link_pattern')
[1, 1, 1, 0, 1, 0, 0, 0]
sage: asm.to_dyck_word()
Traceback (most recent call last):
...
TypeError: to_dyck_word() takes exactly 2 arguments (1 given)
sage: asm.to_dyck_word(algorithm = 'notamethod')
Traceback (most recent call last):
...
ValueError: unknown algorithm 'notamethod'
```

to_fully_packed_loop()

Return the fully packed loop configuration from `self`.

See also:

:class:FullyPackedLoop

EXAMPLES:

```
sage: asm = AlternatingSignMatrix([[1,0,0],[0,1,0],[0,0,1]])
sage: fpl = asm.to_fully_packed_loop()
sage: fpl
    |         |
    |         |
    +   + -- +
    |   |
    |   |
 -- +   +   + --
       |   |
       |   |
    + -- +   +
    |        |
    |        |
```

to_matrix()

Return `self` as a regular matrix.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: asm = A([[1, 0, 0],[0, 1, 0],[0, 0, 1]])
sage: m = asm.to_matrix(); m
[1 0 0]
[0 1 0]
[0 0 1]
sage: m.parent()
Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
```

to_monotone_triangle()

Return a monotone triangle from `self`.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).to_monotone_triangle()
[[3, 2, 1], [2, 1], [1]]
sage: asm = A([[0, 1, 0],[1, -1, 1],[0, 1, 0]])
sage: asm.to_monotone_triangle()
[[3, 2, 1], [3, 1], [2]]
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.to_monotone_triangle()
[[3, 2, 1], [3, 1], [3]]
sage: A.from_monotone_triangle(asm.to_monotone_triangle()) == asm
True
```

to_permutation()

Return the corresponding permutation if `self` is a permutation matrix.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: asm = A([[0,1,0],[1,0,0],[0,0,1]])
sage: p = asm.to_permutation(); p
[2, 1, 3]
sage: parent(p)
Standard permutations
sage: asm = A([[0,1,0],[1,-1,1],[0,1,0]])
sage: asm.to_permutation()
Traceback (most recent call last):
...
ValueError: Not a permutation matrix
```

to_semistandard_tableau()

Return the semistandard tableau corresponding the monotone triangle corresponding to `self`.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[0,0,1],[1,0,0],[0,1,0]]).to_semistandard_tableau()
[[1, 1, 3], [2, 3], [3]]
sage: t = A([[0,1,0],[1,-1,1],[0,1,0]]).to_semistandard_tableau(); t
[[1, 1, 2], [2, 3], [3]]
sage: parent(t)
Semistandard tableaux
```

to_six_vertex_model()

Return the six vertex model configuration from `self`. This method calls `sage.combinat.six_vertex_model.from_alternating_sign_matrix()`.

EXAMPLES:

```
sage: asm = AlternatingSignMatrix([[0,1,0],[1,-1,1],[0,1,0]])
sage: asm.to_six_vertex_model()
    ^    ^    ^
    |    |    |
--> # -> # <- # <--
    ^    |    ^
    |    V    |
--> # <- # -> # <--
    |    ^    |
    V    |    V
--> # -> # <- # <--
    |    |    |
```

```
    V    V    V
```

TESTS:

```
sage: ASM = AlternatingSignMatrices(5)
sage: all((x.to_six_vertex_model()).to_alternating_sign_matrix() == x
....:     for x in ASM)
True
```

transpose()

Return the counterclockwise quarter turn rotation of `self`.

EXAMPLES:

```
sage: A = AlternatingSignMatrices(3)
sage: A([[1, 0, 0],[0, 1, 0],[0, 0, 1]]).transpose()
[1 0 0]
[0 1 0]
[0 0 1]
sage: asm = A([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: asm.transpose()
[0 1 0]
[0 0 1]
[1 0 0]
```

class sage.combinat.alternating_sign_matrix.**ContreTableaux**

Bases: sage.structure.parent.Parent

Factory class for the combinatorial class of contre tableaux of size n.

EXAMPLES:

```
sage: ct4 = ContreTableaux(4); ct4
Contre tableaux of size 4
sage: ct4.cardinality()
42
```

class sage.combinat.alternating_sign_matrix.**ContreTableaux_n**(n)

Bases: sage.combinat.alternating_sign_matrix.ContreTableaux

TESTS:

```
sage: ct2 = ContreTableaux(2); ct2
Contre tableaux of size 2
sage: ct2 == loads(dumps(ct2))
True
```

cardinality()

EXAMPLES:

```
sage: [ ContreTableaux(n).cardinality() for n in range(0, 11)]
[1, 1, 2, 7, 42, 429, 7436, 218348, 10850216, 911835460, 129534272700]
```

class sage.combinat.alternating_sign_matrix.**MonotoneTriangles**(n)

Bases: sage.combinat.gelfand_tsetlin_patterns.GelfandTsetlinPatternsTopRow

Monotone triangles with n rows.

A monotone triangle is a number triangle $(a_{i,j})_{1 \leq i \leq n, 1 \leq j \leq i}$ on $\{1, \ldots, n\}$ such that:

- $a_{i,j} < a_{i,j+1}$
- $a_{i+1,j} < a_{i,j} \leq a_{i+1,j+1}$

This notably requires that the bottom column is `[1,...,n]`.

Alternatively a monotone triangle is a strict Gelfand-Tsetlin pattern with top row $(n, \ldots, 2, 1)$.

INPUT:

- `n` – The number of rows in the monotone triangles

EXAMPLES:

This represents the monotone triangles with base `[3,2,1]`:

```
sage: M = MonotoneTriangles(3)
sage: M
Monotone triangles with 3 rows
sage: M.cardinality()
7
```

The monotone triangles are a lattice:

```
sage: M.lattice()
Finite lattice containing 7 elements
```

Monotone triangles can be converted to alternating sign matrices and back:

```
sage: M = MonotoneTriangles(5)
sage: A = AlternatingSignMatrices(5)
sage: all(A.from_monotone_triangle(m).to_monotone_triangle() == m for m in M)
True
```

cardinality()

Cardinality of `self`.

The number of monotone triangles with n rows is equal to

$$\prod_{k=0}^{n-1} \frac{(3k+1)!}{(n+k)!} = \frac{1!4!7!10!\cdots(3n-2)!}{n!(n+1)!(n+2)!(n+3)!\cdots(2n-1)!}$$

EXAMPLES:

```
sage: M = MonotoneTriangles(4)
sage: M.cardinality()
42
```

cover_relations()

Iterate on the cover relations in the set of monotone triangles with n rows.

EXAMPLES:

```
sage: M = MonotoneTriangles(3)
sage: for (a,b) in M.cover_relations():
....:   eval('a, b')
([[3, 2, 1], [2, 1], [1]], [[3, 2, 1], [2, 1], [2]])
([[3, 2, 1], [2, 1], [1]], [[3, 2, 1], [3, 1], [1]])
([[3, 2, 1], [2, 1], [2]], [[3, 2, 1], [3, 1], [2]])
([[3, 2, 1], [3, 1], [1]], [[3, 2, 1], [3, 1], [2]])
([[3, 2, 1], [3, 1], [2]], [[3, 2, 1], [3, 1], [3]])
([[3, 2, 1], [3, 1], [2]], [[3, 2, 1], [3, 2], [2]])
([[3, 2, 1], [3, 1], [3]], [[3, 2, 1], [3, 2], [3]])
([[3, 2, 1], [3, 2], [2]], [[3, 2, 1], [3, 2], [3]])
```

lattice()

Return the lattice of the monotone triangles with n rows.

EXAMPLES:

```
sage: M = MonotoneTriangles(3)
sage: P = M.lattice()
sage: P
Finite lattice containing 7 elements
```

class sage.combinat.alternating_sign_matrix.**TruncatedStaircases**

Bases: sage.structure.parent.Parent

Factory class for the combinatorial class of truncated staircases of size n with last column last_column.

EXAMPLES:

```
sage: t4 = TruncatedStaircases(4, [2,3]); t4
Truncated staircases of size 4 with last column [2, 3]
sage: t4.cardinality()
4
```

class sage.combinat.alternating_sign_matrix.**TruncatedStaircases_nlastcolumn**(*n*, *last_column*)

Bases: sage.combinat.alternating_sign_matrix.TruncatedStaircases

TESTS:

```
sage: t4 = TruncatedStaircases(4, [2,3]); t4
Truncated staircases of size 4 with last column [2, 3]
sage: t4 == loads(dumps(t4))
True
```

cardinality()

EXAMPLES:

```
sage: T = TruncatedStaircases(4, [2,3])
sage: T.cardinality()
4
```

sage.combinat.alternating_sign_matrix.**nw_corner_sum**(*M*, *i*, *j*)

Return the sum of entries to the northwest of (i, j) in matrix.

EXAMPLES:

```
sage: from sage.combinat.alternating_sign_matrix import nw_corner_sum
sage: A = matrix.ones(3,3)
sage: nw_corner_sum(A,2,2)
4
```

5.1.7 Backtracking

This library contains generic tools for constructing large sets whose elements can be enumerated by exploring a search space with a (lazy) tree or graph structure.

- GenericBacktracker: Depth first search through a tree described by a children function, with branch pruning, etc.

Deprecated classes (use RecursivelyEnumeratedSet() instead):

- SearchForest: Depth and breadth first search through a tree described by a children function.
- TransitiveIdeal: Depth first search through a graph described by a neighbours relation.
- TransitiveIdealGraded: Breadth first search through a graph described by a neighbours relation.

Deprecation details:

- `SearchForest(seeds, succ)` keeps the same behavior as before trac ticket #6637 and is now the same as `RecursivelyEnumeratedSet(seeds, succ, structure='forest', enumeration='depth')`.
- `TransitiveIdeal(succ, seeds)` keeps the same behavior as before trac ticket #6637 and is now the same as `RecursivelyEnumeratedSet(seeds, succ, structure=None, enumeration='naive')`.
- `TransitiveIdealGraded(succ, seeds, max_depth)` keeps the same behavior as before trac ticket #6637 and is now the same as `RecursivelyEnumeratedSet(seeds, succ, structure=None, enumeration='breadth', max_depth=max_depth)`.

TODO:

- For now the code of `SearchForest` is still in `sage/combinat/backtrack.py`. It should be moved in `sage/sets/recursively_enumerated_set.pyx` into a class named `RecursivelyEnumeratedSet_forest` in a later ticket.
- `TransitiveIdeal` and `TransitiveIdealGraded` are used in the code of `categories/finitely_generated_semigroups.py` (at least). These should be updated to use `RecursivelyEnumeratedSet` in a later ticket for speed improvements and `TransitiveIdeal` and `TransitiveIdealGraded` may be deprecated.
- Once the deprecation has been there for enough time: delete `TransitiveIdeal` and `TransitiveIdealGraded`.

class sage.combinat.backtrack.**GenericBacktracker**(*initial_data*, *initial_state*)
Bases: object

A generic backtrack tool for exploring a search space organized as a tree, with branch pruning, etc.

See also `SearchForest` and `TransitiveIdeal` for handling simple special cases.

class sage.combinat.backtrack.**PositiveIntegerSemigroup**
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.combinat.backtrack.SearchForest

The commutative additive semigroup of positive integers.

This class provides an example of algebraic structure which inherits from `SearchForest`. It builds the positive integers a la Peano, and endows it with its natural commutative additive semigroup structure.

EXAMPLES:

```
sage: from sage.combinat.backtrack import PositiveIntegerSemigroup
sage: PP = PositiveIntegerSemigroup()
sage: PP.category()
Join of Category of monoids and Category of commutative additive semigroups and Category of infi
sage: PP.cardinality()
+Infinity
sage: PP.one()
1
sage: PP.an_element()
1
sage: some_elements = list(PP.some_elements()); some_elements
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
```

TESTS:

```
sage: from sage.combinat.backtrack import PositiveIntegerSemigroup
sage: PP = PositiveIntegerSemigroup()
```

We factor out the long test from the TestSuite:

```
sage: TestSuite(PP).run(skip='_test_enumerated_set_contains')
sage: PP._test_enumerated_set_contains()  # long time
```

children(*x*)

Return the single child x+1 of the integer x

EXAMPLES:

```
sage: from sage.combinat.backtrack import PositiveIntegerSemigroup
sage: PP = PositiveIntegerSemigroup()
sage: list(PP.children(1))
[2]
sage: list(PP.children(42))
[43]
```

one()

Return the unit of self.

EXAMPLES:

```
sage: from sage.combinat.backtrack import PositiveIntegerSemigroup
sage: PP = PositiveIntegerSemigroup()
sage: PP.one()
1
```

roots()

Return the single root of self.

EXAMPLES:

```
sage: from sage.combinat.backtrack import PositiveIntegerSemigroup
sage: PP = PositiveIntegerSemigroup()
sage: list(PP.roots())
[1]
```

class sage.combinat.backtrack.**SearchForest**(*roots=None*, *children=None*, *post_process=None*, *algorithm='depth'*, *facade=None*, *category=None*)

Bases: sage.structure.parent.Parent

The enumerated set of the nodes of the forest having the given roots, and where children(x) returns the children of the node x of the forest.

See also GenericBacktracker, TransitiveIdeal, and TransitiveIdealGraded.

INPUT:

- roots – a list (or iterable)
- children – a function returning a list (or iterable, or iterator)
- post_process – a function defined over the nodes of the forest (default: no post processing)
- algorithm – 'depth' or 'breadth' (default: 'depth')
- category – a category (default: EnumeratedSets)

The option post_process allows for customizing the nodes that are actually produced. Furthermore, if f(x) returns None, then x won't be output at all.

EXAMPLES:

We construct the set of all binary sequences of length at most three, and list them:

```
sage: from sage.combinat.backtrack import SearchForest
sage: S = SearchForest( [[]],
....:     lambda l: [l+[0], l+[1]] if len(l) < 3 else [],
....:     category=FiniteEnumeratedSets())
sage: S.list()
[[],
 [0], [0, 0], [0, 0, 0], [0, 0, 1], [0, 1], [0, 1, 0], [0, 1, 1],
 [1], [1, 0], [1, 0, 0], [1, 0, 1], [1, 1], [1, 1, 0], [1, 1, 1]]
```

`SearchForest` needs to be explicitly told that the set is finite for the following to work:

```
sage: S.category()
Category of finite enumerated sets
sage: S.cardinality()
15
```

We proceed with the set of all lists of letters in `0,1,2` without repetitions, ordered by increasing length (i.e. using a breadth first search through the tree):

```
sage: from sage.combinat.backtrack import SearchForest
sage: tb = SearchForest( [[]],
....:       lambda l: [l + [i] for i in range(3) if i not in l],
....:       algorithm = 'breadth',
....:       category=FiniteEnumeratedSets())
sage: tb[0]
[]
sage: tb.cardinality()
16
sage: list(tb)
[[],
 [0], [1], [2],
 [0, 1], [0, 2], [1, 0], [1, 2], [2, 0], [2, 1],
 [0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]
```

For infinite sets, this option should be set carefully to ensure that all elements are actually generated. The following example builds the set of all ordered pairs (i, j) of nonnegative integers such that $j \leq 1$:

```
sage: from sage.combinat.backtrack import SearchForest
sage: I = SearchForest([(0,0)],
....:                  lambda l: [(l[0]+1, l[1]), (l[0], 1)]
....:                            if l[1] == 0 else [(l[0], l[1]+1)])
```

With a depth first search, only the elements of the form $(i, 0)$ are generated:

```
sage: depth_search = I.depth_first_search_iterator()
sage: [next(depth_search) for i in range(7)]
[(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0)]
```

Using instead breadth first search gives the usual anti-diagonal iterator:

```
sage: breadth_search = I.breadth_first_search_iterator()
sage: [next(breadth_search) for i in range(15)]
[(0, 0),
 (1, 0), (0, 1),
 (2, 0), (1, 1), (0, 2),
 (3, 0), (2, 1), (1, 2), (0, 3),
 (4, 0), (3, 1), (2, 2), (1, 3), (0, 4)]
```

Deriving subclasses

The class of a parent A may derive from `SearchForest` so that A can benefit from enumeration tools. As a running example, we consider the problem of enumerating integers whose binary expansion have at most three nonzero digits. For example, $3 = 2^1 + 2^0$ has two nonzero digits. $15 = 2^3 + 2^2 + 2^1 + 2^0$ has four nonzero digits. In fact, 15 is the smallest integer which is not in the enumerated set.

To achieve this, we use `SearchForest` to enumerate binary tuples with at most three nonzero digits, apply a post processing to recover the corresponding integers, and discard tuples finishing by zero.

A first approach is to pass the `roots` and `children` functions as arguments to `SearchForest.__init__()`:

```
sage: from sage.combinat.backtrack import SearchForest
sage: class A(UniqueRepresentation, SearchForest):
....:     def __init__(self):
....:         SearchForest.__init__(self, [()],
....:             lambda x : [x+(0,), x+(1,)] if sum(x) < 3 else [],
....:             lambda x : sum(x[i]*2^i for i in range(len(x))) if sum(x) != 0 and x[-1] != 0
....:             algorithm = 'breadth',
....:             category=InfiniteEnumeratedSets())
sage: MyForest = A(); MyForest
An enumerated set with a forest structure
sage: MyForest.category()
Category of infinite enumerated sets
sage: p = iter(MyForest)
sage: [next(p) for i in range(30)]
[1, 2, 3, 4, 6, 5, 7, 8, 12, 10, 14, 9, 13, 11, 16, 24, 20, 28, 18, 26, 22, 17, 25, 21, 19, 32,
```

An alternative approach is to implement `roots` and `children` as methods of the subclass (in fact they could also be attributes of A). Namely, `A.roots()` must return an iterable containing the enumeration generators, and `A.children(x)` must return an iterable over the children of x. Optionally, A can have a method or attribute such that `A.post_process(x)` returns the desired output for the node `x` of the tree:

```
sage: from sage.combinat.backtrack import SearchForest
sage: class A(UniqueRepresentation, SearchForest):
....:     def __init__(self):
....:         SearchForest.__init__(self, algorithm = 'breadth',
....:                               category=InfiniteEnumeratedSets())
....:
....:     def roots(self):
....:         return [()]
....:
....:     def children(self, x):
....:         if sum(x) < 3:
....:             return [x+(0,), x+(1,)]
....:         else:
....:             return []
....:
....:     def post_process(self, x):
....:         if sum(x) == 0 or x[-1] == 0:
....:             return None
....:         else:
....:             return sum(x[i]*2^i for i in range(len(x)))
sage: MyForest = A(); MyForest
An enumerated set with a forest structure
sage: MyForest.category()
Category of infinite enumerated sets
sage: p = iter(MyForest)
```

```
sage: [next(p) for i in range(30)]
[1, 2, 3, 4, 6, 5, 7, 8, 12, 10, 14, 9, 13, 11, 16, 24, 20, 28, 18, 26, 22, 17, 25, 21, 19, 32,
```

Warning: A SearchForest instance is picklable if and only if the input functions are themselves picklable. This excludes anonymous or interactively defined functions:

```
sage: def children(x):
....:     return [x+1]
sage: S = SearchForest( [1], children, category=InfiniteEnumeratedSets())
sage: dumps(S)
Traceback (most recent call last):
....:
PicklingError: Can't pickle <type 'function'>: attribute lookup __builtin__.function failed
```

Let us now fake children being defined in a Python module:

```
sage: import __main__
sage: __main__.children = children
sage: S = SearchForest( [1], children, category=InfiniteEnumeratedSets())
sage: loads(dumps(S))
An enumerated set with a forest structure
```

breadth_first_search_iterator()

Return a breadth first search iterator over the elements of self

EXAMPLES:

```
sage: from sage.combinat.backtrack import SearchForest
sage: f = SearchForest([[]],
....:                  lambda l: [l+[0], l+[1]] if len(l) < 3 else [])
sage: list(f.breadth_first_search_iterator())
[[], [0], [1], [0, 0], [0, 1], [1, 0], [1, 1], [0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [
sage: S = SearchForest([(0,0)],
....: lambda x : [(x[0], x[1]+1)] if x[1] != 0 else [(x[0]+1,0), (x[0],1)],
....: post_process = lambda x: x if ((is_prime(x[0]) and is_prime(x[1])) and ((x[0] - x[1])
sage: p = S.breadth_first_search_iterator()
sage: [next(p), next(p), next(p), next(p), next(p), next(p), next(p)]
[(5, 3), (7, 5), (13, 11), (19, 17), (31, 29), (43, 41), (61, 59)]
```

children(*x*)

Return the children of the element x

The result can be a list, an iterable, an iterator, or even a generator.

EXAMPLES:

```
sage: from sage.combinat.backtrack import SearchForest
sage: I = SearchForest([(0,0)], lambda l: [(l[0]+1, l[1]), (l[0], 1)] if l[1] == 0 else [(l[
sage: [i for i in I.children((0,0))]
[(1, 0), (0, 1)]
sage: [i for i in I.children((1,0))]
[(2, 0), (1, 1)]
sage: [i for i in I.children((1,1))]
[(1, 2)]
sage: [i for i in I.children((4,1))]
[(4, 2)]
sage: [i for i in I.children((4,0))]
[(5, 0), (4, 1)]
```

depth_first_search_iterator()

Return a depth first search iterator over the elements of `self`

EXAMPLES:

```
sage: from sage.combinat.backtrack import SearchForest
sage: f = SearchForest([[]],
....:                  lambda l: [l+[0], l+[1]] if len(l) < 3 else [])
sage: list(f.depth_first_search_iterator())
[[], [0], [0, 0], [0, 0, 0], [0, 0, 1], [0, 1], [0, 1, 0], [0, 1, 1], [1], [1, 0], [1, 0, 0]
```

elements_of_depth_iterator(*depth=0*)

Return an iterator over the elements of `self` of given depth. An element of depth n can be obtained applying n times the children function from a root.

EXAMPLES:

```
sage: from sage.combinat.backtrack import SearchForest
sage: S = SearchForest([(0,0)] ,
....:        lambda x : [(x[0], x[1]+1)] if x[1] != 0 else [(x[0]+1,0), (x[0],1)],
....:        post_process = lambda x: x if ((is_prime(x[0]) and is_prime(x[1]))
....:                                       and ((x[0] - x[1]) == 2)) else None)
sage: p = S.elements_of_depth_iterator(8)
sage: next(p)
(5, 3)
sage: S = SearchForest(NN, lambda x : [],
....:                       lambda x: x^2 if x.is_prime() else None)
sage: p = S.elements_of_depth_iterator(0)
sage: [next(p), next(p), next(p), next(p), next(p)]
[4, 9, 25, 49, 121]
```

roots()

Return an iterable over the roots of `self`.

EXAMPLES:

```
sage: from sage.combinat.backtrack import SearchForest
sage: I = SearchForest([(0,0)], lambda l: [(l[0]+1, l[1]), (l[0], 1)] if l[1] == 0 else [(l[
sage: [i for i in I.roots()]
[(0, 0)]
sage: I = SearchForest([(0,0),(1,1)], lambda l: [(l[0]+1, l[1]), (l[0], 1)] if l[1] == 0 els
sage: [i for i in I.roots()]
[(0, 0), (1, 1)]
```

class sage.combinat.backtrack.**TransitiveIdeal**(*succ*, *generators*)

Bases: `sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_generic`

Generic tool for constructing ideals of a relation.

INPUT:

- `relation` – a function (or callable) returning a list (or iterable)
- `generators` – a list (or iterable)

Returns the set S of elements that can be obtained by repeated application of `relation` on the elements of `generators`.

Consider `relation` as modeling a directed graph (possibly with loops, cycles, or circuits). Then S is the ideal generated by `generators` under this relation.

Enumerating the elements of S is achieved by depth first search through the graph. The time complexity is $O(n+m)$ where n is the size of the ideal, and m the number of edges in the relation. The memory complexity is the depth, that is the maximal distance between a generator and an element of S.

See also SearchForest and TransitiveIdealGraded.

EXAMPLES:

```
sage: from sage.combinat.backtrack import TransitiveIdeal
sage: [i for i in TransitiveIdeal(lambda i: [i+1] if i<10 else [], [0])]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

sage: [i for i in TransitiveIdeal(lambda i: [mod(i+1,3)], [0])]
[0, 1, 2]
sage: [i for i in TransitiveIdeal(lambda i: [mod(i+2,3)], [0])]
[0, 2, 1]
sage: [i for i in TransitiveIdeal(lambda i: [mod(i+2,10)], [0])]
[0, 2, 4, 6, 8]
sage: [i for i in TransitiveIdeal(lambda i: [mod(i+3,10),mod(i+5,10)], [0])]
[0, 3, 8, 1, 4, 5, 6, 7, 9, 2]
sage: [i for i in TransitiveIdeal(lambda i: [mod(i+4,10),mod(i+6,10)], [0])]
[0, 4, 8, 2, 6]
sage: [i for i in TransitiveIdeal(lambda i: [mod(i+3,9)], [0,1])]
[0, 1, 3, 4, 6, 7]

sage: [p for p in TransitiveIdeal(lambda x:[x],[Permutation([3,1,2,4]), Permutation([2,1,3,4])])
[[2, 1, 3, 4], [3, 1, 2, 4]]
```

We now illustrate that the enumeration is done lazily, by depth first search:

```
sage: C = TransitiveIdeal(lambda x: [x-1, x+1], (-10, 0, 10))
sage: f = C.__iter__()
sage: [ next(f) for i in range(6) ]
[0, 1, 2, 3, 4, 5]
```

We compute all the permutations of 3:

```
sage: [p for p in TransitiveIdeal(attrcall("permutohedron_succ"), [Permutation([1,2,3])])]
[[1, 2, 3], [2, 1, 3], [1, 3, 2], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

We compute all the permutations which are larger than [3,1,2,4], [2,1,3,4] in the right permutohedron:

```
sage: [p for p in TransitiveIdeal(attrcall("permutohedron_succ"), [Permutation([3,1,2,4]), Permu
[[2, 1, 3, 4], [3, 1, 2, 4], [2, 1, 4, 3], [3, 1, 4, 2],
 [2, 3, 1, 4], [3, 4, 1, 2], [3, 4, 2, 1], [2, 3, 4, 1],
 [2, 4, 1, 3], [3, 2, 1, 4], [4, 3, 1, 2], [4, 3, 2, 1],
 [3, 2, 4, 1], [4, 2, 1, 3], [2, 4, 3, 1], [4, 2, 3, 1]]
```

Using TransitiveIdeal people have been using the __contains__ method provided from the __iter__ method. We need to make sure that this continues to work:

```
sage: T = TransitiveIdeal(lambda a:[a+7,a+5], [0])
sage: 12 in T
True
```

class sage.combinat.backtrack.**TransitiveIdealGraded**(*succ*, *generators*, *max_depth=inf*)

Bases: sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_generic

Generic tool for constructing ideals of a relation.

INPUT:

- relation – a function (or callable) returning a list (or iterable)
- generators – a list (or iterable)
- max_depth – (Default: infinity) Specifies the maximal depth to which elements are computed

Return the set S of elements that can be obtained by repeated application of relation on the elements of generators.

Consider relation as modeling a directed graph (possibly with loops, cycles, or circuits). Then S is the ideal generated by generators under this relation.

Enumerating the elements of S is achieved by breadth first search through the graph; hence elements are enumerated by increasing distance from the generators. The time complexity is $O(n + m)$ where n is the size of the ideal, and m the number of edges in the relation. The memory complexity is the depth, that is the maximal distance between a generator and an element of S.

See also SearchForest and TransitiveIdeal.

EXAMPLES:

```
sage: from sage.combinat.backtrack import TransitiveIdealGraded
sage: [i for i in TransitiveIdealGraded(lambda i: [i+1] if i<10 else [], [0])]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We now illustrate that the enumeration is done lazily, by breadth first search:

```
sage: C = TransitiveIdealGraded(lambda x: [x-1, x+1], (-10, 0, 10))
sage: f = C.__iter__()
```

The elements at distance 0 from the generators:

```
sage: sorted([ next(f) for i in range(3) ])
[-10, 0, 10]
```

The elements at distance 1 from the generators:

```
sage: sorted([ next(f) for i in range(6) ])
[-11, -9, -1, 1, 9, 11]
```

The elements at distance 2 from the generators:

```
sage: sorted([ next(f) for i in range(6) ])
[-12, -8, -2, 2, 8, 12]
```

The enumeration order between elements at the same distance is not specified.

We compute all the permutations which are larger than [3,1,2,4] or [2,1,3,4] in the permutohedron:

```
sage: [p for p in TransitiveIdealGraded(attrcall("permutohedron_succ"), [Permutation([3,1,2,4]),
[[3, 1, 2, 4], [2, 1, 3, 4], [2, 3, 1, 4], [2, 1, 4, 3],
 [3, 2, 1, 4], [3, 1, 4, 2], [3, 2, 4, 1], [2, 4, 1, 3],
 [3, 4, 1, 2], [2, 3, 4, 1], [4, 3, 1, 2], [3, 4, 2, 1],
 [4, 2, 1, 3], [2, 4, 3, 1], [4, 3, 2, 1], [4, 2, 3, 1]]
```

sage.combinat.backtrack.**search_forest_iterator**(*roots*, *children*, *algorithm='depth'*)

Return an iterator on the nodes of the forest having the given roots, and where children(x) returns the children of the node x of the forest. Note that every node of the tree is returned, not simply the leaves.

INPUT:

- roots – a list (or iterable)
- children – a function returning a list (or iterable)

- algorithm – 'depth' or 'breadth' (default: 'depth')

EXAMPLES:

We construct the prefix tree of binary sequences of length at most three, and enumerate its nodes:

```
sage: from sage.combinat.backtrack import search_forest_iterator
sage: list(search_forest_iterator([[]], lambda l: [l+[0], l+[1]]
....:                                   if len(l) < 3 else []))
[[], [0], [0, 0], [0, 0, 0], [0, 0, 1], [0, 1], [0, 1, 0],
 [0, 1, 1], [1], [1, 0], [1, 0, 0], [1, 0, 1], [1, 1], [1, 1, 0], [1, 1, 1]]
```

By default, the nodes are iterated through by depth first search. We can instead use a breadth first search (increasing depth):

```
sage: list(search_forest_iterator([[]], lambda l: [l+[0], l+[1]]
....:                                   if len(l) < 3 else [],
....:                              algorithm='breadth'))
[[],
 [0], [1],
 [0, 0], [0, 1], [1, 0], [1, 1],
 [0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1],
 [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]]
```

This allows for iterating trough trees of infinite depth:

```
sage: it = search_forest_iterator([[]], lambda l: [l+[0], l+[1]], algorithm='breadth')
sage: [ next(it) for i in range(16) ]
[[],
 [0], [1], [0, 0], [0, 1], [1, 0], [1, 1],
 [0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1],
 [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1],
 [0, 0, 0, 0]]
```

Here is an iterator through the prefix tree of sequences of letters in $0, 1, 2$ without repetitions, sorted by length; the leaves are therefore permutations:

```
sage: list(search_forest_iterator([[]], lambda l: [l + [i] for i in range(3) if i not in l],
....:                              algorithm='breadth'))
[[],
 [0], [1], [2],
 [0, 1], [0, 2], [1, 0], [1, 2], [2, 0], [2, 1],
 [0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]
```

5.1.8 Baxter permutations

class sage.combinat.baxter_permutations.**BaxterPermutations**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The combinatorial class of Baxter permutations.

A Baxter permutation is a permutation avoiding the generalized permutation patterns $2-41-3$ and $3-14-2$. In other words, a permutation σ is a Baxter permutation if for any subword $u := u_1u_2u_3u_4$ of σ such that the letters u_2 and u_3 are adjacent in σ, the standardized version of u is neither 2413 nor 3142.

See [Gir12] for a study of Baxter permutations.

INPUT:

- n – (default: None) a nonnegative integer, the size of the permutations.

OUTPUT:

Return the combinatorial class of the Baxter permutations of size `n` if `n` is not `None`. Otherwise, return the combinatorial class of all Baxter permutations.

EXAMPLES:

```
sage: BaxterPermutations(5)
Baxter permutations of size 5
sage: BaxterPermutations()
Baxter permutations
```

REFERENCES:

class sage.combinat.baxter_permutations.**BaxterPermutations_all**(*n=None*)

Bases: sage.sets.disjoint_union_enumerated_sets.DisjointUnionEnumeratedSets, sage.combinat.baxter_permutations.BaxterPermutations

The enumerated set of all Baxter permutations.

See `BaxterPermutations` for the definition of Baxter permutations.

EXAMPLES:

```
sage: from sage.combinat.baxter_permutations import BaxterPermutations_all
sage: BaxterPermutations_all()
Baxter permutations
```

to_pair_of_twin_binary_trees(*p*)

Apply a bijection between Baxter permutations of size `self._n` and the set of pairs of twin binary trees with `self._n` nodes.

INPUT:

- `p` – a Baxter permutation.

OUTPUT:

The pair of twin binary trees (T_L, T_R) where T_L (resp. T_R) is obtained by inserting the letters of `p` from left to right (resp. right to left) following the the binary search tree insertion algorithm. This is called the *Baxter P-symbol* in [Gir12] Definition 4.1.

Note: This method only works when `p` is a permutation. For words with repeated letters, it would return two "right binary search trees" (in the terminology of [Gir12]), which conflicts with the definition in [Gir12].

EXAMPLES:

```
sage: BaxterPermutations().to_pair_of_twin_binary_trees(Permutation([]))
(., .)
sage: BaxterPermutations().to_pair_of_twin_binary_trees(Permutation([1, 2, 3]))
(1[., 2[., 3[., .]]], 3[2[1[., .], .], .])
sage: BaxterPermutations().to_pair_of_twin_binary_trees(Permutation([3, 4, 1, 2]))
(3[1[., 2[., .]], 4[., .]], 2[1[., .], 4[3[., .], .]])
```

class sage.combinat.baxter_permutations.**BaxterPermutations_size**(*n*)

Bases: sage.combinat.baxter_permutations.BaxterPermutations

The enumerated set of Baxter permutations of a given size.

See `BaxterPermutations` for the definition of Baxter permutations.

EXAMPLES:

```
sage: from sage.combinat.baxter_permutations import BaxterPermutations_size
sage: BaxterPermutations_size(5)
Baxter permutations of size 5
```

cardinality()

Return the number of Baxter permutations of size `self._n`.

For any positive integer n, the number of Baxter permutations of size n equals

$$\sum_{k=1}^{n} \frac{\binom{n+1}{k-1}\binom{n+1}{k}\binom{n+1}{k+1}}{\binom{n+1}{1}\binom{n+1}{2}}.$$

This is OEIS sequence A001181.

EXAMPLES:

```
sage: [BaxterPermutations(n).cardinality() for n in xrange(13)]
[1, 1, 2, 6, 22, 92, 422, 2074, 10754, 58202, 326240, 1882960, 11140560]

sage: BaxterPermutations(3r).cardinality()
6
sage: parent(_)
Integer Ring
```

5.1.9 Binary Recurrence Sequences.

This class implements several methods relating to general linear binary recurrence sequences, including a sieve to find perfect powers in integral linear binary recurrence sequences.

EXAMPLES:

```
sage: R = BinaryRecurrenceSequence(1,1)        #the Fibonacci sequence
sage: R(137)        #the 137th term of the Fibonacci sequence
19134702400093278081449423917
sage: R(137) == fibonacci(137)
True
sage: [R(i) % 4 for i in xrange(12)]
[0, 1, 1, 2, 3, 1, 0, 1, 1, 2, 3, 1]
sage: R.period(4)        #the period of the fibonacci sequence modulo 4
6
sage: R.pthpowers(2, 10**30)        # long time (7 seconds) -- in fact these are all squares, c.f. [1
[0, 1, 2, 12]

sage: S = BinaryRecurrenceSequence(8,1) #a Lucas sequence
sage: S.period(73)
148
sage: S(5) % 73 == S(5 +148) %73
True
sage: S.pthpowers(3,10**30)    # long time (3 seconds) -- provably finds the indices of all 3rd powe
[0, 1, 2]

sage: T = BinaryRecurrenceSequence(2,0,1,2)
sage: [T(i) for i in xrange(10)]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
sage: T.is_degenerate()
True
sage: T.is_geometric()
```

```
True
sage: T.pthpowers(7,10**30)
Traceback (most recent call last):
...
ValueError: The degenerate binary recurrence sequence is geometric or quasigeometric and has many pth
```

AUTHORS:

-Isabel Vogt (2013): initial version

REFERENCES:

class sage.combinat.binary_recurrence_sequences.**BinaryRecurrenceSequence**(*b*, *c*, *u0=0*, *u1=1*)

Bases: sage.structure.sage_object.SageObject

Create a linear binary recurrence sequence defined by initial conditions u_0 and u_1 and recurrence relation $u_{n+2} = b * u_{n+1} + c * u_n$.

INPUT:

- b – an integer (partially determining the recurrence relation)
- c – an integer (partially determining the recurrence relation)
- u0 – an integer (the 0th term of the binary recurrence sequence)
- u1 – an integer (the 1st term of the binary recurrence sequence)

OUTPUT:

- An integral linear binary recurrence sequence defined by u0, u1, and $u_{n+2} = b * u_{n+1} + c * u_n$

See also:

fibonacci(), lucas_number1(), lucas_number2()

EXAMPLES:

```
sage: R = BinaryRecurrenceSequence(3,3,2,1)
sage: R
Binary recurrence sequence defined by: u_n = 3 * u_{n-1} + 3 * u_{n-2};
With initial conditions: u_0 = 2, and u_1 = 1
```

is_arithmetic()

Decide whether the sequence is degenerate and an arithmetic sequence.

The sequence is arithmetic if and only if $u_1 - u_0 = u_2 - u_1 = u_3 - u_2$.

This corresponds to the matrix $F = [[0, 1], [c, b]]$ being nondiagonalizable and $\alpha/\beta = 1$.

EXAMPLES:

```
sage: S = BinaryRecurrenceSequence(2,-1)
sage: [S(i) for i in xrange(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: S.is_arithmetic()
True
```

is_degenerate()

Decide whether the binary recurrence sequence is degenerate.

Let α and β denote the roots of the characteristic polynomial $p(x) = x^2 - bx - c$. Let $a = u_1 - u_0\beta/(\beta - \alpha)$ and $b = u_1 - u_0\alpha/(\beta - \alpha)$. The sequence is, thus, given by $u_n = a\alpha^n - b\beta^n$. Then we say that the sequence is nondegenerate if and only if $a * b * \alpha * \beta \neq 0$ and α/β is not a root of unity.

More concretely, there are 4 classes of degeneracy, that can all be formulated in terms of the matrix $F = [[0, 1], [c, b]]$.

- F is singular – this corresponds to `c` = 0, and thus $\alpha * \beta = 0$. This sequence is geometric after term `u0` and so we call it `quasigeometric`.
- $v = [[u_0], [u_1]]$ is an eigenvector of F – this corresponds to a `geometric` sequence with $a * b = 0$.
- F is nondiagonalizable – this corresponds to $\alpha = \beta$. This sequence will be the point-wise product of an arithmetic and geometric sequence.
- F^k is scaler, for some $k > 1$ – this corresponds to α/β a k th root of unity. This sequence is a union of several geometric sequences, and so we again call it `quasigeometric`.

EXAMPLES:

```
sage: S = BinaryRecurrenceSequence(0,1)
sage: S.is_degenerate()
True
sage: S.is_geometric()
False
sage: S.is_quasigeometric()
True

sage: R = BinaryRecurrenceSequence(3,-2)
sage: R.is_degenerate()
False

sage: T = BinaryRecurrenceSequence(2,-1)
sage: T.is_degenerate()
True
sage: T.is_arithmetic()
True
```

is_geometric()

Decide whether the binary recurrence sequence is geometric - ie a geometric sequence.

This is a subcase of a degenerate binary recurrence sequence, for which $ab = 0$, i.e. $u_n/u_{n-1} = r$ for some value of r. See `is_degenerate` for a description of degeneracy and definitions of a and b.

EXAMPLES:

```
sage: S = BinaryRecurrenceSequence(2,0,1,2)
sage: [S(i) for i in xrange(10)]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
sage: S.is_geometric()
True
```

is_quasigeometric()

Decide whether the binary recurrence sequence is degenerate and similar to a geometric sequence, i.e. the union of multiple geometric sequences, or geometric after term `u0`.

If α/β is a k th root of unity, where $k > 1$, then necessarily $k = 2, 3, 4, 6$. Then $F = [[0, 1], [c, b]$ is diagonalizable, and $F^k = [[\alpha^k, 0], [0, \beta^k]]$ is scaler matrix. Thus for all values of j mod k, the j mod k terms of u_n form a geometric series.

If α or β is zero, this implies that $c = 0$. This is the case when F is singular. In this case, $u_1, u_2, u_3, ...$ is geometric.

EXAMPLES:

```
sage: S = BinaryRecurrenceSequence(0,1)
sage: [S(i) for i in xrange(10)]
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
sage: S.is_quasigeometric()
True

sage: R = BinaryRecurrenceSequence(3,0)
sage: [R(i) for i in xrange(10)]
[0, 1, 3, 9, 27, 81, 243, 729, 2187, 6561]
sage: R.is_quasigeometric()
True
```

period(*m*)

Return the period of the binary recurrence sequence modulo an integer m.

If n_1 is congruent to n_2 modulu `period(m)`, then u_{n_1} is is congruent to u_{n_2} modulo m.

INPUT:

- m – an integer (modulo which the period of the recurrence relation is calculated).

OUTPUT:

- The integer (the period of the sequence modulo m)

EXAMPLES:

If $p = \pm 1 \mod 5$, then the period of the Fibonacci sequence mod p is $p-1$ (c.f. Lemma 3.3 of [BMS06]).

```
sage: R = BinaryRecurrenceSequence(1,1)
sage: R.period(31)
30

sage: [R(i) % 4 for i in xrange(12)]
[0, 1, 1, 2, 3, 1, 0, 1, 1, 2, 3, 1]
sage: R.period(4)
6
```

This function works for degenerate sequences as well.

```
sage: S = BinaryRecurrenceSequence(2,0,1,2)
sage: S.is_degenerate()
True
sage: S.is_geometric()
True
sage: [S(i) % 17 for i in xrange(16)]
[1, 2, 4, 8, 16, 15, 13, 9, 1, 2, 4, 8, 16, 15, 13, 9]
sage: S.period(17)
8
```

Note: the answer is cached.

pthpowers(*p*, *Bound*)

Find the indices of proveably all pth powers in the recurrence sequence bounded by Bound.

Let u_n be a binary recurrence sequence. A p th power in u_n is a solution to $u_n = y^p$ for some integer y. There are only finitely many p th powers in any recurrence sequence [SS].

INPUT:

- p - a rational prime integer (the fixed p in $u_n = y^p$)

- Bound - a natural number (the maximum index n in $u_n = y^p$ that is checked).

OUTPUT:

- A list of the indices of all p th powers less bounded by Bound. If the sequence is degenerate and there are many p th powers, raises ValueError.

EXAMPLES:

```
sage: R = BinaryRecurrenceSequence(1,1)        #the Fibonacci sequence
sage: R.pthpowers(2, 10**30)        # long time (7 seconds) -- in fact these are all squares
[0, 1, 2, 12]

sage: S = BinaryRecurrenceSequence(8,1) #a Lucas sequence
sage: S.pthpowers(3,10**30)     # long time (3 seconds) -- provably finds the indices of all
[0, 1, 2]

sage: Q = BinaryRecurrenceSequence(3,3,2,1)
sage: Q.pthpowers(11,10**30)          # long time (7.5 seconds)
[1]
```

If the sequence is degenerate, and there are are no p th powers, returns []. Otherwise, if there are many p th powers, raises ValueError.

```
sage: T = BinaryRecurrenceSequence(2,0,1,2)
sage: T.is_degenerate()
True
sage: T.is_geometric()
True
sage: T.pthpowers(7,10**30)
Traceback (most recent call last):
...
ValueError: The degenerate binary recurrence sequence is geometric or quasigeometric and has

sage: L = BinaryRecurrenceSequence(4,0,2,2)
sage: [L(i).factor() for i in xrange(10)]
[2, 2, 2^3, 2^5, 2^7, 2^9, 2^11, 2^13, 2^15, 2^17]
sage: L.is_quasigeometric()
True
sage: L.pthpowers(2,10**30)
[]
```

NOTE: This function is primarily optimized in the range where Bound is much larger than p.

5.1.10 Binary Trees

This module deals with binary trees as mathematical (in particular immutable) objects.

Note: If you need the data-structure for example to represent sets or hash tables with AVL trees, you should have a look at sage.misc.sagex_ds.

AUTHORS:

- Florent Hivert (2010-2011): initial implementation.

REFERENCES:

class sage.combinat.binary_tree.**BinaryTree**(*parent*, *children=None*, *check=True*)

Bases: sage.combinat.abstract_tree.AbstractClonableTree, sage.structure.list_clone.ClonableArray

Binary trees.

Binary trees here mean ordered (a.k.a. plane) finite binary trees, where "ordered" means that the children of each node are ordered.

Binary trees contain nodes and leaves, where each node has two children while each leaf has no children. The number of leaves of a binary tree always equals the number of nodes plus 1.

INPUT:

- `children` – `None` (default) or a list, tuple or iterable of length 2 of binary trees or convertible objects. This corresponds to the standard recursive definition of a binary tree as either a leaf or a pair of binary trees. Syntactic sugar allows leaving out all but the outermost calls of the `BinaryTree()` constructor, so that, e. g., `BinaryTree([BinaryTree(None),BinaryTree(None)])` can be shortened to `BinaryTree([None,None])`. It is also allowed to abbreviate `[None, None]` by `[]`.
- `check` – (default: `True`) whether check for binary should be performed or not.

EXAMPLES:

```
sage: BinaryTree()
.
sage: BinaryTree(None)
.
sage: BinaryTree([])
[., .]
sage: BinaryTree([None, None])
[., .]
sage: BinaryTree([None, []])
[., [., .]]
sage: BinaryTree([[], None])
[[., .], .]
sage: BinaryTree("[[], .]")
[[., .], .]
sage: BinaryTree([None, BinaryTree([None, None])])
[., [., .]]

sage: BinaryTree([[], None, []])
Traceback (most recent call last):
...
ValueError: this is not a binary tree
```

TESTS:

```
sage: t1 = BinaryTree([[None, [],[[], None]],[[],[]]])
sage: t2 = BinaryTree([[[],[]],[]])
sage: with t1.clone() as t1c:
....:     t1c[1,1,1] = t2
sage: t1 == t1c
False
```

as_ordered_tree(*with_leaves=True*)

Return the same tree seen as an ordered tree. By default, leaves are transformed into actual nodes, but this can be avoided by setting the optional variable `with_leaves` to `False`.

EXAMPLES:

```
sage: bt = BinaryTree([]); bt
[., .]
sage: bt.as_ordered_tree()
[[], []]
sage: bt.as_ordered_tree(with_leaves = False)
```

```
[]
sage: bt = bt.canonical_labelling(); bt
1[., .]
sage: bt.as_ordered_tree()
1[None[], None[]]
sage: bt.as_ordered_tree(with_leaves=False)
1[]
```

canonical_labelling(*shift=1*)

Return a labelled version of `self`.

The canonical labelling of a binary tree is a certain labelling of the nodes (not the leaves) of the tree. The actual canonical labelling is currently unspecified. However, it is guaranteed to have labels in $1...n$ where n is the number of nodes of the tree. Moreover, two (unlabelled) trees compare as equal if and only if their canonical labelled trees compare as equal.

EXAMPLES:

```
sage: BinaryTree().canonical_labelling()
.
sage: BinaryTree([]).canonical_labelling()
1[., .]
sage: BinaryTree([[[], []], [[], None]]).canonical_labelling()
5[2[1[., .], 4[3[., .], .]], 7[6[., .], 8[., .]]]
```

canopee()

Return the canopee of `self`.

The *canopee* of a non-empty binary tree T with n internal nodes is the list l of 0 and 1 of length $n-1$ obtained by going along the leaves of T from left to right except the two extremal ones, writing 0 if the leaf is a right leaf and 1 if the leaf is a left leaf.

EXAMPLES:

```
sage: BinaryTree([]).canopee()
[]
sage: BinaryTree([None, []]).canopee()
[1]
sage: BinaryTree([[], None]).canopee()
[0]
sage: BinaryTree([[], []]).canopee()
[0, 1]
sage: BinaryTree([[[], []], [[], None]]).canopee()
[0, 1, 0, 0, 1, 0, 1]
```

The number of pairs (t_1, t_2) of binary trees of size n such that the canopee of t_1 is the complementary of the canopee of t_2 is also the number of Baxter permutations (see [DG94], see also OEIS sequence A001181). We check this in small cases:

```
sage: [len([(u,v) for u in BinaryTrees(n) for v in BinaryTrees(n)
....:       if map(lambda x:1-x, u.canopee()) == v.canopee()])
....:    for n in range(1, 5)]
[1, 2, 6, 22]
```

Here is a less trivial implementation of this:

```
sage: from sage.sets.finite_set_map_cy import fibers
sage: from sage.misc.all import attrcall
sage: def baxter(n):
....:     f = fibers(lambda t: tuple(t.canopee()),
```

```
....:                     BinaryTrees(n))
....:     return sum(len(f[i])*len(f[tuple(1-x for x in i)])
....:                for i in f)
sage: [baxter(n) for n in range(1, 7)]
[1, 2, 6, 22, 92, 422]
```

TESTS:

```
sage: t = BinaryTree().canopee()
Traceback (most recent call last):
...
ValueError: canopee is only defined for non empty binary trees
```

REFERENCES:

check()

Check that `self` is a binary tree.

EXAMPLES:

```
sage: BinaryTree([[], []])      # indirect doctest
[[., .], [., .]]
sage: BinaryTree([[], [], []]) # indirect doctest
Traceback (most recent call last):
...
ValueError: this is not a binary tree
sage: BinaryTree([[]])          # indirect doctest
Traceback (most recent call last):
...
ValueError: this is not a binary tree
```

graph(*with_leaves=True*)

Convert `self` to a digraph. By default, this graph contains both nodes and leaves, hence is never empty. To obtain a graph which contains only the nodes, the `with_leaves` optional keyword variable has to be set to `False`.

INPUT:

- `with_leaves` – (default: `True`) a Boolean, determining whether the resulting graph will be formed from the leaves and the nodes of `self` (if `True`), or only from the nodes of `self` (if `False`)

EXAMPLES:

```
sage: t1 = BinaryTree([[], None])
sage: t1.graph()
Digraph on 5 vertices
sage: t1.graph(with_leaves=False)
Digraph on 2 vertices

sage: t1 = BinaryTree([[], [[], None]])
sage: t1.graph()
Digraph on 9 vertices
sage: t1.graph().edges()
[(0, 1, None), (0, 4, None), (1, 2, None), (1, 3, None), (4, 5, None), (4, 8, None), (5, 6,
sage: t1.graph(with_leaves=False)
Digraph on 4 vertices
sage: t1.graph(with_leaves=False).edges()
[(0, 1, None), (0, 2, None), (2, 3, None)]

sage: t1 = BinaryTree()
sage: t1.graph()
```

```
Digraph on 1 vertex
sage: t1.graph(with_leaves=False)
Digraph on 0 vertices

sage: BinaryTree([]).graph()
Digraph on 3 vertices
sage: BinaryTree([]).graph(with_leaves=False)
Digraph on 1 vertex

sage: t1 = BinaryTree([[], [[], []]])
sage: t1.graph(with_leaves=False)
Digraph on 5 vertices
sage: t1.graph(with_leaves=False).edges()
[(0, 1, None), (0, 2, None), (2, 3, None), (2, 4, None)]
```

in_order_traversal(*node_action=None*, *leaf_action=None*)

Explore the binary tree `self` using the depth-first infix-order traversal algorithm, executing the `node_action` function whenever traversing a node and executing the `leaf_action` function whenever traversing a leaf.

In more detail, what this method does to a tree T is the following:

```
if the root of `T` is a node:
    apply in_order_traversal to the left subtree of `T`
        (with the same node_action and leaf_action);
    apply node_action to the root of `T`;
    apply in_order_traversal to the right subtree of `T`
        (with the same node_action and leaf_action);
else:
    apply leaf_action to the root of `T`.
```

For example on the following binary tree T, where we denote leaves by $a, b, c, \ldots$ and nodes by $1, 2, 3, \ldots$:

```
|      _____3_____           |
|     /           \          |
|    1           __7__       |
|   / \         /     \      |
|  a   2       _5_     8     |
|     / \     /   \   / \    |
|    b   c   4     6 h   i   |
|           / \   / \        |
|          d   e f   g       |
```

this method first applies `leaf_action` to a, then applies `node_action` to 1, then `leaf_action` to b, then `node_action` to 2, etc., with the vertices being traversed in the order $a, 1, b, 2, c, 3, d, 4, e, 5, f, 6, g, 7, h, 8, i$.

See `in_order_traversal_iter()` for a version of this algorithm which only iterates through the vertices rather than applying any function to them.

INPUT:

- `node_action` – (optional) a function which takes a node in input and does something during the exploration
- `leaf_action` – (optional) a function which takes a leaf in input and does something during the exploration

TESTS:

```
sage: nb_leaf = 0
sage: def l_action(_):
....:     global nb_leaf
....:     nb_leaf += 1
sage: nb_node = 0
sage: def n_action(_):
....:     global nb_node
....:     nb_node += 1

sage: BinaryTree().in_order_traversal(n_action, l_action)
sage: nb_leaf, nb_node
(1, 0)

sage: nb_leaf, nb_node = 0, 0
sage: b = BinaryTree([[],[[],[]]]); b
[[., .], [[., .], [., .]]]
sage: b.in_order_traversal(n_action, l_action)
sage: nb_leaf, nb_node
(6, 5)
sage: nb_leaf, nb_node = 0, 0
sage: b = b.canonical_labelling()
sage: b.in_order_traversal(n_action, l_action)
sage: nb_leaf, nb_node
(6, 5)
sage: l = []
sage: b.in_order_traversal(lambda node: l.append( node.label() ))
sage: l
[1, 2, 3, 4, 5]

sage: leaf = 'a'
sage: l = []
sage: def l_action(_):
....:     global leaf, l
....:     l.append(leaf)
....:     leaf = chr( ord(leaf)+1 )
sage: n_action = lambda node: l.append( node.label() )
sage: b = BinaryTree([[None,[]],[[[],[]],[]]]).\
....:      canonical_labelling()
sage: b.in_order_traversal(n_action, l_action)
sage: l
['a', 1, 'b', 2, 'c', 3, 'd', 4, 'e', 5, 'f', 6, 'g', 7, 'h', 8,
 'i']
```

in_order_traversal_iter()

The depth-first infix-order traversal iterator for the binary tree `self`.

This method iters each vertex (node and leaf alike) of the given binary tree following the depth-first infix order traversal algorithm.

The *depth-first infix order traversal algorithm* iterates through a binary tree as follows:

```
iterate through the left subtree (by the depth-first infix
    order traversal algorithm);
yield the root;
iterate through the right subtree (by the depth-first infix
    order traversal algorithm).
```

For example on the following binary tree T, where we denote leaves by $a, b, c, \ldots$ and nodes by $1, 2, 3, \ldots$:

```
|      ____3____          |
|     /         \         |
|    1          __7__     |
|   / \        /     \    |
|  a   2      _5_     8   |
|     / \    /   \   / \  |
|    b   c  4     6 h   i |
|          / \   / \      |
|         d   e f   g     |
```

the depth-first infix-order traversal algorithm iterates through the vertices of T in the following order: $a, 1, b, 2, c, 3, d, 4, e, 5, f, 6, g, 7, h, 8, i$.

See `in_order_traversal()` for a version of this algorithm which not only iterates through, but actually does something at the vertices of tree.

TESTS:

```
sage: b = BinaryTree([[],[[],[]]]); ascii_art([b])
[   _o_     ]
[  /   \    ]
[ o     o   ]
[      / \  ]
[     o   o ]
sage: ascii_art(list(b.in_order_traversal_iter()))
[ , o, ,   _o_    , , o, ,   o  , , o,  ]
[         /   \             / \         ]
[        o     o           o   o        ]
[             / \                       ]
[            o   o                      ]
sage: ascii_art(filter(lambda node: node.label() is not None,
....:     b.canonical_labelling().in_order_traversal_iter()))
[ 1,   _2_    , 3,   4  , 5 ]
[     /   \          / \     ]
[    1     4        3   5    ]
[          / \               ]
[         3   5              ]

sage: list(BinaryTree(None).in_order_traversal_iter())
[.]
```

is_complete()

Return `True` if `self` is complete, else return `False`.

In a nutshell, a complete binary tree is a perfect binary tree except possibly in the last level, with all nodes in the last level "flush to the left".

In more detail: A complete binary tree (also called binary heap) is a binary tree in which every level, except possibly the last one (the deepest), is completely filled. At depth n, all nodes must be as far left as possible.

For example:

```
|          ___o___    |
|         /       \   |
|      __o__        o |
|     /     \         |
|    o       o        |
|   / \     / \       |
|  o   o   o   o      |
```

is not complete but the following ones are:

```
|      __o__           _o_             ___o___      |
|     /     \         /   \           /       \     |
|    o       o       o     o       __o__       o    |
|   / \     / \     / \           /     \     / \   |
|  o   o   o   o, o   o    ,    o       o   o   o  |
|                              / \     /            |
|                             o   o   o             |
```

EXAMPLES:

```
sage: lst = lambda i: filter(lambda bt: bt.is_complete(), BinaryTrees(i))
sage: for i in range(9): ascii_art(lst(i)) # long time
[  ]
[ o ]
[   o ]
[  /  ]
[ o   ]
[   o   ]
[  / \  ]
[ o   o ]
[     o   ]
[    / \  ]
[   o   o ]
[  /      ]
[ o       ]
[     _o_   ]
[    /   \  ]
[   o     o ]
[  / \      ]
[ o   o     ]
[     __o__   ]
[    /     \  ]
[   o       o ]
[  / \     /  ]
[ o   o   o   ]
[     __o__     ]
[    /     \    ]
[   o       o   ]
[  / \     / \  ]
[ o   o   o   o ]
[       __o__     ]
[      /     \    ]
[     o       o   ]
[    / \     / \  ]
[   o   o   o   o ]
[  /              ]
[ o               ]
```

is_empty()

Return whether `self` is empty.

The notion of emptiness employed here is the one which defines a binary tree to be empty if its root is a leaf. There is precisely one empty binary tree.

EXAMPLES:

```
sage: BinaryTree().is_empty()
True
sage: BinaryTree([]).is_empty()
False
sage: BinaryTree([[], None]).is_empty()
False
```

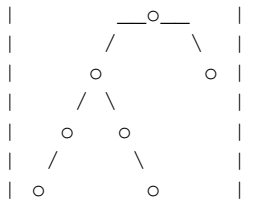
is_full()

Return `True` if `self` is full, else return `False`.

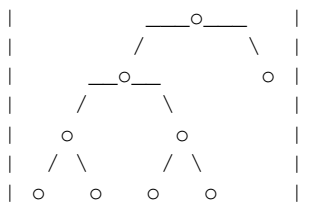
A full binary tree is a tree in which every node either has two child nodes or has two child leaves.

This is also known as *proper binary tree* or *2-tree* or *strictly binary tree*.

For example:



is not full but the next one is:



EXAMPLES:

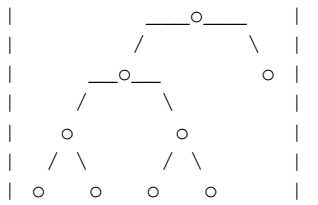
```
sage: BinaryTree([[[[],None],[None,[]]], []]).is_full()
False
sage: BinaryTree([[[[],[]],[[],[]]], []]).is_full()
True
sage: ascii_art(filter(lambda bt: bt.is_full(), BinaryTrees(5)))
[   _o_     ,      _o_    ]
[  /   \         /   \   ]
[ o     o       o     o  ]
[      / \     / \       ]
[     o   o   o   o      ]
```

is_perfect()

Return `True` if `self` is perfect, else return `False`.

A perfect binary tree is a full tree in which all leaves are at the same depth.

For example:



is not perfect but the next one is:

```
|      __o__      |
|     /     \     |
|    o       o    |
|   / \     / \   |
|  o   o   o   o  |
```

EXAMPLES:

```
sage: lst = lambda i: filter(lambda bt: bt.is_perfect(), BinaryTrees(i))
sage: for i in range(10): ascii_art(lst(i)) # long time
[  ]
[ o ]
[  ]
[   o   ]
[  / \  ]
[ o   o ]
[  ]
[  ]
[  ]
[      __o__      ]
[     /     \     ]
[    o       o    ]
[   / \     / \   ]
[  o   o   o   o  ]
[  ]
[  ]
```

left_border_symmetry()

Return the tree where a symmetry has been applied recursively on all left borders. If a tree is made of three trees $[T_1, T_2, T_3]$ on its left border, it becomes $[T_3', T_2', T_1']$ where same symmetry has been applied to T_1, T_2, T_3.

EXAMPLES:

```
sage: BinaryTree().left_border_symmetry()
.
sage: BinaryTree([]).left_border_symmetry()
[., .]
sage: BinaryTree([[None,[]],None]).left_border_symmetry()
[[., .], [., .]]
sage: BinaryTree([[None,[None,[]]],None]).left_border_symmetry()
[[., .], [., [., .]]]
sage: bt = BinaryTree([[None,[None,[]]],None]).canonical_labelling()
sage: bt
4[1[., 2[., 3[., .]]], .]
sage: bt.left_border_symmetry()
1[4[., .], 2[., 3[., .]]]
```

left_right_symmetry()

Return the left-right symmetrized tree of `self`.

EXAMPLES:

```
sage: BinaryTree().left_right_symmetry()
.
sage: BinaryTree([]).left_right_symmetry()
[., .]
```

```
sage: BinaryTree([[],None]).left_right_symmetry()
[., [., .]]
sage: BinaryTree([[None, []],None]).left_right_symmetry()
[., [[., .], .]]
```

left_rotate()

Return the result of left rotation applied to the binary tree `self`.

Left rotation on binary trees is defined as follows: Let T be a binary tree such that the right child of the root of T is a node. Let A be the left child of the root of T, and let B and C be the left and right children of the right child of the root of T. (Keep in mind that nodes of trees are identified with the subtrees consisting of their descendants.) Then, the left rotation of T is the binary tree in which the right child of the root is C, whereas the left child of the root is a node whose left and right children are A and B. In pictures:

```
|   *                          *   |
|  / \                        / \  |
| A   *  -left-rotate->      *   C |
|    / \                    / \    |
|   B   C                  A   B   |
```

where asterisks signify a single node each (but A, B and C might be empty).

For example,

```
|   _o_                          o |
|  /   \                        /  |
| o     o  -left-rotate->      o   |
|      /                      / \  |
|     o                      o   o |
<BLANKLINE>
|       __o__                       o |
|      /     \                     /  |
|     o       o  -left-rotate->    o   |
|    / \                         /    |
|   o   o                       o     |
|  /     \                     / \    |
| o       o                   o   o   |
|                            /     \  |
|                           o       o |
```

Left rotation is the inverse operation to right rotation (`right_rotate()`).

See also:

`right_rotate()`

EXAMPLES:

```
sage: b = BinaryTree([[],[[],None]]); ascii_art([b])
[   _o_   ]
[  /   \  ]
[ o     o ]
[      /  ]
[     o   ]
sage: ascii_art([b.left_rotate()])
[     o ]
[    /  ]
[   o   ]
[  / \  ]
[ o   o ]
```

```
sage: b.left_rotate().right_rotate() == b
True
```

make_leaf()

Modify `self` so that it becomes a leaf (i. e., an empty tree).

Note: `self` must be in a mutable state.

See also:

`make_node`

EXAMPLES:

```
sage: t = BinaryTree([None, None])
sage: t.make_leaf()
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with t.clone() as t1:
....:     t1.make_leaf()
sage: t, t1
([., .], .)
```

make_node(*child_list=[None, None]*)

Modify `self` so that it becomes a node with children `child_list`.

INPUT:

- `child_list` – a pair of binary trees (or objects convertible to)

Note: `self` must be in a mutable state.

See also:

`make_leaf`

EXAMPLES:

```
sage: t = BinaryTree()
sage: t.make_node([None, None])
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with t.clone() as t1:
....:     t1.make_node([None, None])
sage: t, t1
(., [., .])
sage: with t.clone() as t:
....:     t.make_node([BinaryTree(), BinaryTree(), BinaryTree([])])
Traceback (most recent call last):
...
ValueError: the list must have length 2
sage: with t1.clone() as t2:
....:     t2.make_node([t1, t1])
sage: with t2.clone() as t3:
....:     t3.make_node([t1, t2])
sage: t1, t2, t3
([., .], [[., .], [., .]], [[., .], [[., .], [., .]]])
```

over(*bt*)

Return `self` over `bt`, where "over" is the `over` (/) operation.

If T and T' are two binary trees, then T over T' (written T/T') is defined as the tree obtained by grafting T' on the rightmost leaf of T. More precisely, T/T' is defined by identifying the root of the T' with the rightmost leaf of T. See section 4.5 of [HNT05].

If T is empty, then $T/T' = T'$.

The definition of this "over" operation goes back to Loday-Ronco [LodRon0102066] (Definition 2.2), but it is denoted by $\backslash$ and called the "under" operation there. In fact, trees in sage have their root at the top, contrary to the trees in [LodRon0102066] which are growing upwards. For this reason, the names of the over and under operations are swapped, in order to keep a graphical meaning. (Our notation follows that of section 4.5 of [HNT05].)

See also:

`under()`

EXAMPLES:

Showing only the nodes of a binary tree, here is an example for the over operation:

```
|   o        __o__        _o_          |
|  / \  /   /     \  =   /   \         |
| o   o  o        o   o     o          |
|          \     /           \        |
|           o   o           __o__     |
|                          /     \    |
|                         o       o   |
|                          \     /    |
|                           o   o     |
```

A Sage example:

```
sage: b1 = BinaryTree([[],[[],[]]])
sage: b2 = BinaryTree([[None, []],[]])
sage: ascii_art((b1, b2, b1/b2))
(   _o_     ,   _o_  ,   _o_           )
(  /   \       /   \    /   \          )
( o     o     o     o  o     o_        )
(      / \     \             /  \       )
(     o   o     o           o    o      )
(                             \         )
(                            _o_       )
(                           /   \      )
(                          o     o     )
(                           \          )
(                            o         )
```

TESTS:

```
sage: b1 = BinaryTree([[],[]]); ascii_art([b1])
[   o   ]
[  / \  ]
[ o   o ]
sage: b2 = BinaryTree([[None,[]],[[],None]]); ascii_art([b2])
[   __o__   ]
[  /     \  ]
[ o       o ]
[  \     /  ]
[   o   o   ]
```

```
sage: ascii_art([b1.over(b2)])
[   _o_          ]
[  /   \         ]
[ o     o        ]
[        \       ]
[       __o__    ]
[      /     \   ]
[     o       o  ]
[      \     /   ]
[       o   o    ]
```

The same in the labelled case:

```
sage: b1 = b1.canonical_labelling()
sage: b2 = b2.canonical_labelling()
sage: ascii_art([b1.over(b2)])
[   _2_          ]
[  /   \         ]
[ 1     3        ]
[        \       ]
[       __3__    ]
[      /     \   ]
[     1       5  ]
[      \     /   ]
[       2   4    ]
```

q_hook_length_fraction(*q=None*, *q_factor=False*)

Compute the q-hook length fraction of the binary tree `self`, with an additional "q-factor" if desired.

If T is a (plane) binary tree and q is a polynomial indeterminate over some ring, then the q-hook length fraction $h_q(T)$ of T is defined by

$$h_q(T) = \frac{[|T|]_q!}{\prod_{t \in T} [|T_t|]_q},$$

where the product ranges over all nodes t of T, where T_t denotes the subtree of T consisting of t and its all descendants, and where for every tree S, we denote by $|S|$ the number of nodes of S. While this definition only shows that $h_q(T)$ is a rational function in T, it is in fact easy to show that $h_q(T)$ is actually a polynomial in T, and thus makes sense when any element of a commutative ring is substituted for q. This can also be explicitly seen from the following recursive formula for $h_q(T)$:

$$h_q(T) = \binom{|T|-1}{|T_1|}_q h_q(T_1) h_q(T_2),$$

where T is any nonempty binary tree, and T_1 and T_2 are the two child trees of the root of T, and where $\binom{a}{b}_q$ denotes a q-binomial coefficient.

A variation of the q-hook length fraction is the following "q-hook length fraction with q-factor":

$$f_q(T) = h_q(T) \cdot \prod_{t \in T} q^{|T_{\mathrm{right}(t)}|},$$

where for every node t, we denote by $\mathrm{right}(t)$ the right child of t. This $f_q(T)$ differs from $h_q(T)$ only in a multiplicative factor, which is a power of q.

When $q = 1$, both $f_q(T)$ and $h_q(T)$ equal the number of permutations whose binary search tree (see [HNT05] for the definition) is T (after dropping the labels). For example, there are 20 permutations which give a binary tree of the following shape:

```
|      __o__    |
|     /     \   |
|    o       o  |
|   / \     /   |
|  o   o   o    |
```

by the binary search insertion algorithm, in accordance with the fact that this tree satisfies $f_1(T) = 20$.

When q is considered as a polynomial indeterminate, $f_q(T)$ is the generating function for all permutations whose binary search tree is T (after dropping the labels) with respect to the number of inversions (i. e., the Coxeter length) of the permutations.

Objects similar to $h_q(T)$ also make sense for general ordered forests (rather than just binary trees), see e. g. [BW88], Theorem 9.1.

INPUT:

- `q` – a ring element which is to be substituted as q into the q-hook length fraction (by default, this is set to be the indeterminate q in the polynomial ring $\mathbf{Z}[q]$)
- `q_factor` – a Boolean (default: `False`) which determines whether to compute $h_q(T)$ or to compute $f_q(T)$ (namely, $h_q(T)$ is obtained when `q_factor == False`, and $f_q(T)$ is obtained when `q_factor == True`)

REFERENCES:

EXAMPLES:

Let us start with a simple example. Actually, let us start with the easiest possible example – the binary tree with only one vertex (which is a leaf):

```
sage: b = BinaryTree()
sage: b.q_hook_length_fraction()
1
sage: b.q_hook_length_fraction(q_factor=True)
1
```

Nothing different for a tree with one node and two leaves:

```
sage: b = BinaryTree([]); b
[., .]
sage: b.q_hook_length_fraction()
1
sage: b.q_hook_length_fraction(q_factor=True)
1
```

Let us get to a more interesting tree:

```
sage: b = BinaryTree([[[],[]],[[],None]]); b
[[[., .], [., .]], [[., .], .]]
sage: b.q_hook_length_fraction()(q=1)
20
sage: b.q_hook_length_fraction()
q^7 + 2*q^6 + 3*q^5 + 4*q^4 + 4*q^3 + 3*q^2 + 2*q + 1
sage: b.q_hook_length_fraction(q_factor=True)
q^10 + 2*q^9 + 3*q^8 + 4*q^7 + 4*q^6 + 3*q^5 + 2*q^4 + q^3
sage: b.q_hook_length_fraction(q=2)
465
sage: b.q_hook_length_fraction(q=2, q_factor=True)
3720
sage: q = PolynomialRing(ZZ, 'q').gen()
```

```
sage: b.q_hook_length_fraction(q=q**2)
q^14 + 2*q^12 + 3*q^10 + 4*q^8 + 4*q^6 + 3*q^4 + 2*q^2 + 1
```

Let us check the fact that $f_q(T)$ is the generating function for all permutations whose binary search tree is T (after dropping the labels) with respect to the number of inversions of the permutations:

```
sage: def q_hook_length_fraction_2(T):
....:     P = PolynomialRing(ZZ, 'q')
....:     q = P.gen()
....:     res = P.zero()
....:     for w in T.sylvester_class():
....:         res += q ** Permutation(w).length()
....:     return res
sage: def test_genfun(i):
....:     return all( q_hook_length_fraction_2(T)
....:                 == T.q_hook_length_fraction(q_factor=True)
....:                 for T in BinaryTrees(i) )
sage: test_genfun(4)
True
```

right_rotate()

Return the result of right rotation applied to the binary tree `self`.

Right rotation on binary trees is defined as follows: Let T be a binary tree such that the left child of the root of T is a node. Let C be the right child of the root of T, and let A and B be the left and right children of the left child of the root of T. (Keep in mind that nodes of trees are identified with the subtrees consisting of their descendants.) Then, the right rotation of T is the binary tree in which the left child of the root is A, whereas the right child of the root is a node whose left and right children are B and C. In pictures:

```
|      *                          *      |
|     / \                        / \     |
|    *   C -right-rotate->      A   *    |
|   / \                            / \   |
|  A   B                          B   C  |
```

where asterisks signify a single node each (but A, B and C might be empty).

For example,

```
|      o                        _o_     |
|     /                        /   \    |
|    o     -right-rotate->     o     o  |
|   / \                            /    |
|  o   o                          o     |
<BLANKLINE>
|        __o__                          _o__       |
|       /     \                        /    \      |
|      o       o  -right-rotate->     o     _o_    |
|     / \                            /     /   \   |
|    o   o                          o     o     o  |
|   /     \                               \        |
|  o       o                               o       |
```

Right rotation is the inverse operation to left rotation (`left_rotate()`).

The right rotation operation introduced here is the one defined in Definition 2.1 of [CP12].

See also:

`left_rotate()`

EXAMPLES:

```
sage: b = BinaryTree([[[],[]], None]); ascii_art([b])
[      o ]
[     /  ]
[    o   ]
[   / \  ]
[  o   o ]
sage: ascii_art([b.right_rotate()])
[    _o_   ]
[   /   \  ]
[  o     o ]
[       /  ]
[      o   ]
sage: b = BinaryTree([[[[],None],[None,[]]], []]); ascii_art([b])
[         __o__    ]
[        /     \   ]
[       o       o  ]
[      / \          ]
[     o   o         ]
[    /     \        ]
[   o       o       ]
sage: ascii_art([b.right_rotate()])
[      _o__        ]
[     /    \       ]
[    o     _o_     ]
[   /     /   \    ]
[  o     o     o   ]
[         \        ]
[          o       ]
```

show(*with_leaves=False*)

Show the binary tree `show`, with or without leaves depending on the Boolean keyword variable `with_leaves`.

Warning: Left and right children might get interchanged in the actual picture. Moreover, for a labelled binary tree, the labels shown in the picture are not (in general) the ones given by the labelling! Use `_latex_()`, `view`, `_ascii_art_()` or `pretty_print` for more faithful representations of the data of the tree.

TESTS:

```
sage: t1 = BinaryTree([[], [[], None]])
sage: t1.show()
```

single_edge_cut_shapes()

Return the list of possible single-edge cut shapes for the binary tree.

This is used in `sage.combinat.interval_posets.TamariIntervalPoset.is_new()`.

OUTPUT:

a list of triples (m, i, n) of integers

This is a list running over all inner edges (i.e., edges joining two non-leaf vertices) of the binary tree. The removal of each inner edge defines two binary trees (connected components), the root-tree and the sub-tree. Thus, to every inner edge, we can assign three positive integers: m is the node number of the root-tree R, and n is the node number of the sub-tree S. The integer i is the index of the leaf of R on which S is grafted to obtain the original tree. The leaves of R are numbered starting from 1 (from left to right), hence $1 \leq i \leq m + 1$.

In fact, each of m and n determines the other, as the total node number of R and S is the node number of `self`.

EXAMPLES:

```
sage: BT = BinaryTrees(3)
sage: [t.single_edge_cut_shapes() for t in BT]
[[(2, 3, 1), (1, 2, 2)],
[(2, 2, 1), (1, 2, 2)],
[(2, 1, 1), (2, 3, 1)],
[(2, 2, 1), (1, 1, 2)],
[(2, 1, 1), (1, 1, 2)]]

sage: BT = BinaryTrees(2)
sage: [t.single_edge_cut_shapes() for t in BT]
[[(1, 2, 1)], [(1, 1, 1)]]

sage: BT = BinaryTrees(1)
sage: [t.single_edge_cut_shapes() for t in BT]
[[]]
```

sylvester_class(*left_to_right=False*)

Iterate over the sylvester class corresponding to the binary tree `self`.

The sylvester class of a tree T is the set of permutations σ whose right-to-left binary search tree (a notion defined in [HNT05], Definition 7) is T after forgetting the labels. This is an equivalence class of the sylvester congruence (the congruence on words which holds two words $uacvbw$ and $ucavbw$ congruent whenever a, b, c are letters satisfying $a \leq b < c$, and extends by transitivity) on the symmetric group.

For example the following tree's sylvester class consists of the permutations $(1, 3, 2)$ and $(3, 1, 2)$:

```
[   o   ]
[  / \  ]
[ o   o ]
```

(only the nodes are drawn here).

The right-to-left binary search tree of a word is constructed by an RSK-like insertion algorithm which proceeds as follows: Start with an empty labelled binary tree, and read the word from right to left. Each time a letter is read from the word, insert this letter in the existing tree using binary search tree insertion (`binary_search_insert()`). This is what the `binary_search_tree()` method computes if it is given the keyword `left_to_right=False`.

Here are two more descriptions of the sylvester class of a binary search tree:

- The sylvester class of a binary search tree T is the set of all linear extensions of the poset corresponding to T (that is, of the poset whose Hasse diagram is T, with the root on top), provided that the nodes of T are labelled with $1, 2, \ldots, n$ in a binary-search-tree way (i.e., every left descendant of a node has a label smaller than that of the node, and every right descendant of a node has a label higher than that of the node).
- The sylvester class of a binary search tree T (with vertex labels $1, 2, \ldots, n$) is the interval $[u, v]$ in the right permutohedron order (`permutohedron_lequal()`), where u is the 312-avoiding permutation corresponding to T (`to_312_avoiding_permutation()`), and where v is the 132-avoiding permutation corresponding to T (`to_132_avoiding_permutation()`).

If the optional keyword variable `left_to_right` is set to `True`, then the *left* sylvester class of `self` is returned instead. This is the set of permutations σ whose left-to-right binary search tree (that is, the result of the `binary_search_tree()` with `left_to_right` set to `True`) is `self`. It is an equivalence class of the left sylvester congruence.

Warning: This method yields the elements of the sylvester class as raw lists, not as permutations!

EXAMPLES:

Verifying the claim that the right-to-left binary search trees of the permutations in the sylvester class of a tree t all equal t:

```
sage: def test_bst_of_sc(n, left_to_right):
....:     for t in BinaryTrees(n):
....:         for p in t.sylvester_class(left_to_right=left_to_right):
....:             p_per = Permutation(p)
....:             tree = p_per.binary_search_tree(left_to_right=left_to_right)
....:             if not BinaryTree(tree) == t:
....:                 return False
....:     return True
sage: test_bst_of_sc(4, False)
True
sage: test_bst_of_sc(5, False)    # long time
True
sage: test_bst_of_sc(6, False)    # long time
True
```

The same with the left-to-right version of binary search:

```
sage: test_bst_of_sc(4, True)
True
sage: test_bst_of_sc(5, True)    # long time
True
sage: test_bst_of_sc(6, True)    # long time
True
```

Checking that the sylvester class is the set of linear extensions of the poset of the tree:

```
sage: all( sorted(t.canonical_labelling().sylvester_class())
....:      == sorted(list(v) for v in t.canonical_labelling().to_poset().linear_extensions()
....:      for t in BinaryTrees(4) )
True
```

TESTS:

```
sage: list(BinaryTree([[],[]]).sylvester_class())
[[1, 3, 2], [3, 1, 2]]
sage: bt = BinaryTree([[[],None],[[],[]]])
sage: l = list(bt.sylvester_class()); l
[[1, 2, 4, 6, 5, 3],
 [1, 4, 2, 6, 5, 3],
 [1, 4, 6, 2, 5, 3],
 [1, 4, 6, 5, 2, 3],
 [4, 1, 2, 6, 5, 3],
 [4, 1, 6, 2, 5, 3],
 [4, 1, 6, 5, 2, 3],
 [4, 6, 1, 2, 5, 3],
 [4, 6, 1, 5, 2, 3],
 [4, 6, 5, 1, 2, 3],
 [1, 2, 6, 4, 5, 3],
 [1, 6, 2, 4, 5, 3],
 [1, 6, 4, 2, 5, 3],
 [1, 6, 4, 5, 2, 3],
 [6, 1, 2, 4, 5, 3],
 [6, 1, 4, 2, 5, 3],
```

```
 [6, 1, 4, 5, 2, 3],
 [6, 4, 1, 2, 5, 3],
 [6, 4, 1, 5, 2, 3],
 [6, 4, 5, 1, 2, 3]]
sage: len(l) == Integer(bt.q_hook_length_fraction()(q=1))
True
```

Border cases:

```
sage: list(BinaryTree().sylvester_class())
[[]]
sage: list(BinaryTree([]).sylvester_class())
[[1]]
```

tamari_greater()

The list of all trees greater or equal to `self` in the Tamari order.

This is the order filter of the Tamari order generated by `self`.

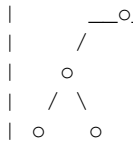
See `tamari_lequal()` for the definition of the Tamari poset.

See also:

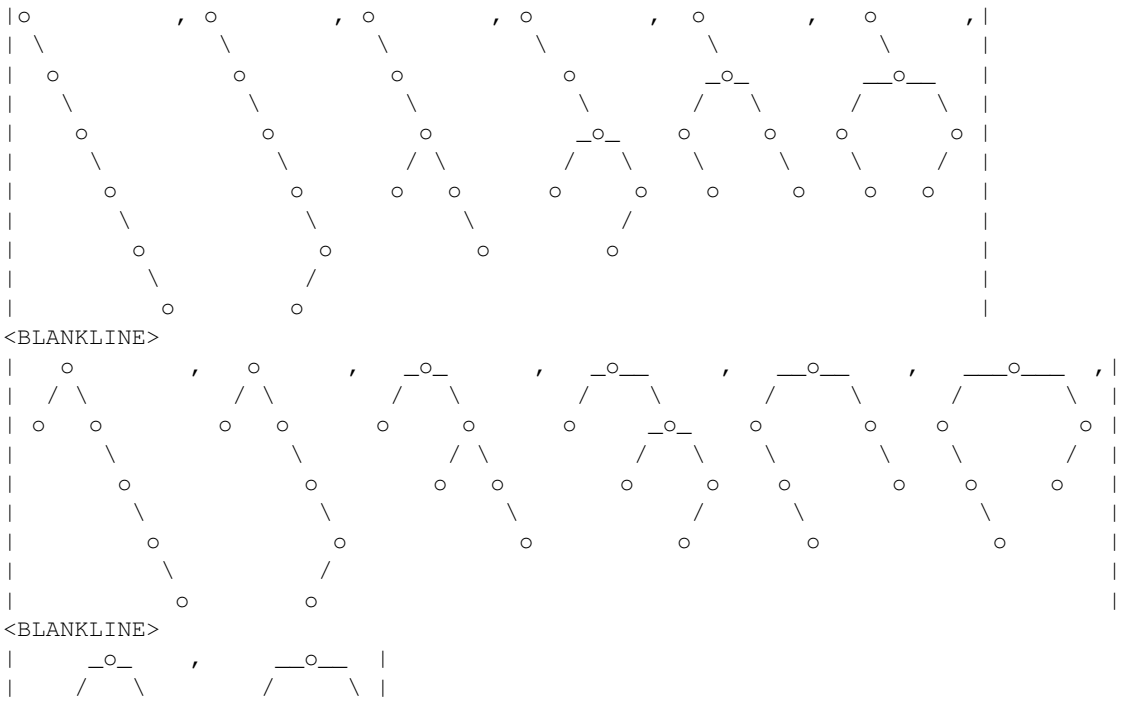
`tamari_smaller()`

EXAMPLES:

For example, the tree:



has these trees greater or equal to it:



```
|   o     o      o       o|
|  / \     \    / \     / |
| o   o     o  o   o   o  |
```

TESTS:

```
sage: B = BinaryTree
sage: b = B([None, B([None, B([None, B([])])])]);b
[., [., [., [., .]]]]
sage: b.tamari_greater()
[[., [., [., [., .]]]]]
sage: b = B([B([B([B([]), None]), None]), None]);b
[[[[., .], .], .], .]
sage: b.tamari_greater()
[[., [., [., [., .]]]], [., [., [[., .], .]]],
[., [[., .], [., .]]], [., [[., [., .]], .]],
[., [[[., .], .], .]], [[., .], [., [., .]]],
[[., .], [[., .], .]], [[., [., .]], [., .]],
[[., [., [., .]]], .], [[., [[., .], .]], .],
[[[., .], .], [., .]], [[[., .], [., .]], .],
[[[., [., .]], .], .], [[[[., .], .], .], .]]
```

tamari_interval(*other*)

Return the Tamari interval between `self` and `other` as a `TamariIntervalPoset`.

A "Tamari interval" is an interval in the Tamari poset. See `tamari_lequal()` for the definition of the Tamari poset.

INPUT:

- `other` – a binary tree greater or equal to `self` in the Tamari order

EXAMPLES:

```
sage: bt = BinaryTree([[None, [[], None]], None])
sage: ip = bt.tamari_interval(BinaryTree([None, [[None, []], None]])); ip
The Tamari interval of size 4 induced by relations [(2, 4), (3, 4), (3, 1), (2, 1)]
sage: ip.lower_binary_tree()
[[., [[., .], .]], .]
sage: ip.upper_binary_tree()
[., [[., [., .]], .]]
sage: ip.interval_cardinality()
4
sage: ip.number_of_tamari_inversions()
2
sage: list(ip.binary_trees())
[[., [[., [., .]], .]],
 [[., [., [., .]]], .],
 [., [[[., .], .], .]],
 [[., [[., .], .]], .]]
sage: bt.tamari_interval(BinaryTree([[None,[]],[]]))
Traceback (most recent call last):
...
ValueError: The two binary trees are not comparable on the Tamari lattice.
```

TESTS:

Setting `other` equal to `bt` gives an interval consisting of just one element:

```
sage: ip = bt.tamari_interval(bt)
sage: ip
The Tamari interval of size 4 induced by relations [(1, 4), (2, 3), (3, 4), (3, 1), (2, 1)]
sage: list(ip.binary_trees())
[[., [[., .], .]], .]]
```

Empty trees work well:

```
sage: bt = BinaryTree()
sage: ip = bt.tamari_interval(bt)
sage: ip
The Tamari interval of size 0 induced by relations []
sage: list(ip.binary_trees())
[.]
```

tamari_join(*other*)

Return the join of the binary trees `self` and `other` (of equal size) in the n-th Tamari poset (where n is the size of these trees).

The n-th Tamari poset (defined in `tamari_lequal()`) is known to be a lattice, and the map from the n-th symmetric group S_n to the n-th Tamari poset defined by sending every permutation $p \in S_n$ to the binary search tree of p (more precisely, to `p.binary_search_tree_shape()`) is a lattice homomorphism. (See Theorem 6.2 in [Read04].)

See also:

`tamari_lequal()`, `tamari_meet()`.

AUTHORS:

Viviane Pons and Darij Grinberg, 18 June 2014.

EXAMPLES:

```
sage: a = BinaryTree([None, [None, []]])
sage: b = BinaryTree([None, [[], None]])
sage: c = BinaryTree([[None, []], None])
sage: d = BinaryTree([[[], None], None])
sage: e = BinaryTree([[], []])
sage: a.tamari_join(c) == a
True
sage: b.tamari_join(c) == b
True
sage: c.tamari_join(e) == a
True
sage: d.tamari_join(e) == e
True
sage: e.tamari_join(b) == a
True
sage: e.tamari_join(a) == a
True

sage: b1 = BinaryTree([None, [[], None], None])
sage: b2 = BinaryTree([[], None], []])
sage: b1.tamari_join(b2)
[., [[., .], [., .]]]
sage: b3 = BinaryTree([[], [[], None]])
sage: b1.tamari_join(b3)
[., [., [[., .], .]]]
sage: b2.tamari_join(b3)
[[., .], [., [., .]]]
```

The universal property of the meet operation is satisfied:

```
sage: def test_uni_join(p, q):
....:     j = p.tamari_join(q)
....:     if not p.tamari_lequal(j):
....:         return False
....:     if not q.tamari_lequal(j):
....:         return False
....:     for r in p.tamari_greater():
....:         if q.tamari_lequal(r) and not j.tamari_lequal(r):
....:             return False
....:     return True
sage: all( test_uni_join(p, q) for p in BinaryTrees(3) for q in BinaryTrees(3) )
True
sage: p = BinaryTrees(6).random_element(); q = BinaryTrees(6).random_element(); test_uni_joi
True
```

Border cases:

```
sage: b = BinaryTree(None)
sage: b.tamari_join(b)
.
sage: b = BinaryTree([])
sage: b.tamari_join(b)
[., .]
```

REFERENCES:

tamari_lequal(*t2*)

Return `True` if `self` is less or equal to another binary tree `t2` (of the same size as `self`) in the Tamari order.

The Tamari order on binary trees of size n is the partial order on the set of all binary trees of size n generated by the following requirement: If a binary tree T' is obtained by right rotation (see `right_rotate()`) from a binary tree T, then $T < T'$. This not only is a well-defined partial order, but actually is a lattice structure on the set of binary trees of size n, and is a quotient of the weak order on the n-th symmetric group (also known as the right permutohedron order, see `permutohedron_lequal()`). See [CP12]. The set of binary trees of size n equipped with the Tamari order is called the n-th Tamari poset.

The Tamari order can equivalently be defined as follows:

If T and S are two binary trees of size n, then the following four statements are equivalent:

- We have $T \leq S$ in the Tamari order.
- There exist elements t and s of the Sylvester classes (`sylvester_class()`) of T and S, respectively, such that $t \leq s$ in the weak order on the symmetric group.
- The 132-avoiding permutation corresponding to T (see `to_132_avoiding_permutation()`) is $\leq$ to the 132-avoiding permutation corresponding to S in the weak order on the symmetric group.
- The 312-avoiding permutation corresponding to T (see `to_312_avoiding_permutation()`) is $\leq$ to the 312-avoiding permutation corresponding to S in the weak order on the symmetric group.

See also:

`tamari_smaller()`, `tamari_greater()`, `tamari_pred()`, `tamari_succ()`, `tamari_interval()`

EXAMPLES:

This tree:

```
|        o     |
|       / \    |
|      o   o   |
|     /        |
|    o         |
|   / \        |
|  o   o       |
```

is Tamari-$\leq$ to the following tree:

```
|      _o_      |
|     /   \     |
|    o     o    |
|   / \     \   |
|  o   o     o  |
```

Checking this:

```
sage: b = BinaryTree([[[[], []], None], [])
sage: c = BinaryTree([[[],[]],[None,[]]])
sage: b.tamari_lequal(c)
True
```

TESTS:

```
sage: for T in BinaryTrees(4):
....:     for S in T.tamari_smaller():
....:         if S != T and T.tamari_lequal(S):
....:             print "FAILURE"
....:         if not S.tamari_lequal(T):
....:             print "FAILURE"
```

tamari_meet (*other*, *side='right'*)

Return the meet of the binary trees `self` and `other` (of equal size) in the n-th Tamari poset (where n is the size of these trees).

The n-th Tamari poset (defined in `tamari_lequal()`) is known to be a lattice, and the map from the n-th symmetric group S_n to the n-th Tamari poset defined by sending every permutation $p \in S_n$ to the binary search tree of p (more precisely, to `p.binary_search_tree_shape()`) is a lattice homomorphism. (See Theorem 6.2 in [Read04].)

See also:

`tamari_lequal()`, `tamari_join()`.

AUTHORS:

Viviane Pons and Darij Grinberg, 18 June 2014.

EXAMPLES:

```
sage: a = BinaryTree([None, [None, []]])
sage: b = BinaryTree([None, [[], None]])
sage: c = BinaryTree([[None, []], None])
sage: d = BinaryTree([[[], None], None])
sage: e = BinaryTree([[], []])
sage: a.tamari_meet(c) == c
True
sage: b.tamari_meet(c) == c
True
```

```
sage: c.tamari_meet(e) == d
True
sage: d.tamari_meet(e) == d
True
sage: e.tamari_meet(b) == d
True
sage: e.tamari_meet(a) == e
True

sage: b1 = BinaryTree([None, [[], None], None])
sage: b2 = BinaryTree([[], None], [])
sage: b1.tamari_meet(b2)
[[[., .], .], .], .]
sage: b3 = BinaryTree([[], [[], None]])
sage: b1.tamari_meet(b3)
[[[., .], .], .], .]
sage: b2.tamari_meet(b3)
[[[., .], .], .], .]
```

The universal property of the meet operation is satisfied:

```
sage: def test_uni_meet(p, q):
....:     m = p.tamari_meet(q)
....:     if not m.tamari_lequal(p):
....:         return False
....:     if not m.tamari_lequal(q):
....:         return False
....:     for r in p.tamari_smaller():
....:         if r.tamari_lequal(q) and not r.tamari_lequal(m):
....:             return False
....:     return True
sage: all( test_uni_meet(p, q) for p in BinaryTrees(3) for q in BinaryTrees(3) )
True
sage: p = BinaryTrees(6).random_element(); q = BinaryTrees(6).random_element(); test_uni_mee
True
```

Border cases:

```
sage: b = BinaryTree(None)
sage: b.tamari_meet(b)
.
sage: b = BinaryTree([])
sage: b.tamari_meet(b)
[., .]
```

tamari_pred()

Compute the list of predecessors of `self` in the Tamari poset.

This list is computed by performing all left rotates possible on its nodes.

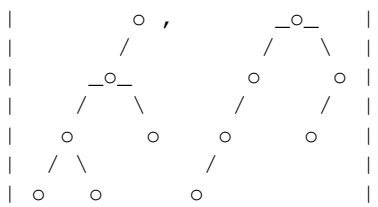
See `tamari_lequal()` for the definition of the Tamari poset.

EXAMPLES:

For this tree:

```
|      __o__    |
|     /     \   |
|    o       o  |
|   / \     /   |
|  o   o   o    |
```

the list is:



TESTS:

```
sage: B = BinaryTree
sage: b = B([B([B([B([]), None]), None]), None]);b
[[[[., .], .], .], .]
sage: b.tamari_pred()
[]
sage: b = B([None, B([None, B([None, B([])])])]);b
[., [., [., [., .]]]]
sage: b.tamari_pred()
[[[., .], [., [., .]]], [., [[., .], [., .]]], [., [., [[., .], .]]]]
```

tamari_smaller()

The list of all trees smaller or equal to `self` in the Tamari order.

This is the order ideal of the Tamari order generated by `self`.

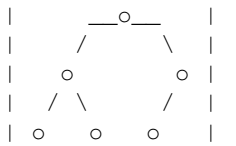
See `tamari_lequal()` for the definition of the Tamari poset.

See also:

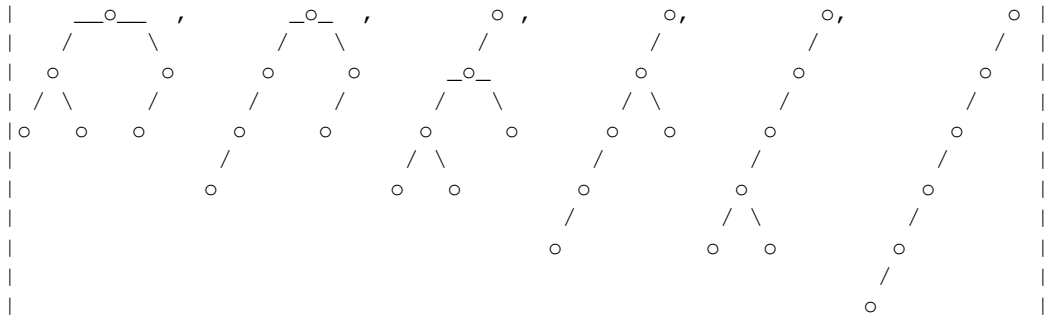
`tamari_greater()`

EXAMPLES:

The tree:



has these trees smaller or equal to it:



TESTS:

```
sage: B = BinaryTree
sage: b = B([None, B([None, B([None, B([])])])]);b
[., [., [., [., .]]]]
sage: b.tamari_smaller()
[[., [., [., [., .]]]], [., [., [[., .], .]]],
[., [[., .], [., .]]], [., [[., [., .]], .]],
[., [[[., .], .], .]], [[., .], [., [., .]]],
[[., .], [[., .], .]], [[., [., .]], [., .]],
[[., [., [., .]]], .], [[., [[., .], .]], .],
[[[., .], .], [., .]], [[[., .], [., .]], .],
[[[., [., .]], .], .], [[[[., .], .], .], .]]
sage: b = B([B([B([B([]), None]), None]), None]);b
[[[[., .], .], .], .]
sage: b.tamari_smaller()
[[[[[., .], .], .], .]]
```

tamari_succ()

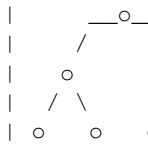
Compute the list of successors of `self` in the Tamari poset.

This is the list of all trees obtained by a right rotate of one of its nodes.

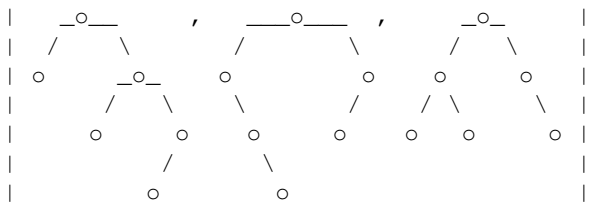
See `tamari_lequal()` for the definition of the Tamari poset.

EXAMPLES:

The list of successors of:



is:



TESTS:

```
sage: B = BinaryTree
sage: b = B([B([B([B([]), None]), None]), None]);b
[[[[., .], .], .], .]
sage: b.tamari_succ()
[[[[., .], .], [., .]], [[[., .], [., .]], .], [[[., [., .]], .], .]]

sage: b = B([])
sage: b.tamari_succ()
[]

sage: b = B([[],[]])
sage: b.tamari_succ()
[[., [., [., .]]]]
```

to_132_avoiding_permutation()

Return a 132-avoiding permutation corresponding to the binary tree.

The linear extensions of a binary tree form an interval of the weak order called the sylvester class of the tree. This permutation is the maximal element of this sylvester class.

EXAMPLES:

```
sage: bt = BinaryTree([[],[]])
sage: bt.to_132_avoiding_permutation()
[3, 1, 2]
sage: bt = BinaryTree([[[], [[], None]], [[], []]])
sage: bt.to_132_avoiding_permutation()
[8, 6, 7, 3, 4, 1, 2, 5]
```

TESTS:

```
sage: bt = BinaryTree([[],[]])
sage: bt == bt.to_132_avoiding_permutation().binary_search_tree_shape(left_to_right=False)
True
sage: bt = BinaryTree([[[], [[], None]], [[], []]])
sage: bt == bt.to_132_avoiding_permutation().binary_search_tree_shape(left_to_right=False)
True
```

to_312_avoiding_permutation()

Return a 312-avoiding permutation corresponding to the binary tree.

The linear extensions of a binary tree form an interval of the weak order called the sylvester class of the tree. This permutation is the minimal element of this sylvester class.

EXAMPLES:

```
sage: bt = BinaryTree([[],[]])
sage: bt.to_312_avoiding_permutation()
[1, 3, 2]
sage: bt = BinaryTree([[[], [[], None]], [[], []]])
sage: bt.to_312_avoiding_permutation()
[1, 3, 4, 2, 6, 8, 7, 5]
```

TESTS:

```
sage: bt = BinaryTree([[],[]])
sage: bt == bt.to_312_avoiding_permutation().binary_search_tree_shape(left_to_right=False)
True
sage: bt = BinaryTree([[[], [[], None]], [[], []]])
sage: bt == bt.to_312_avoiding_permutation().binary_search_tree_shape(left_to_right=False)
True
```

to_dyck_word(*usemap='1L0R'*)

Return the Dyck word associated with `self` using the given map.

INPUT:

- `usemap` – a string, either `1L0R`, `1R0L`, `L1R0`, `R1L0`

The bijection is defined recursively as follows:

- a leaf is associated to the empty Dyck Word
- a tree with children l, r is associated with the Dyck word described by `usemap` where L and R are respectively the Dyck words associated with the trees l and r.

EXAMPLES:

```
sage: BinaryTree().to_dyck_word()
[]
sage: BinaryTree([]).to_dyck_word()
[1, 0]
sage: BinaryTree([[[], [[], None]], [[], []]]).to_dyck_word()
[1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0]
sage: BinaryTree([[None,[]],None]).to_dyck_word()
[1, 1, 0, 1, 0, 0]
sage: BinaryTree([[None,[]],None]).to_dyck_word("1R0L")
[1, 0, 1, 1, 0, 0]
sage: BinaryTree([[None,[]],None]).to_dyck_word("L1R0")
[1, 1, 0, 0, 1, 0]
sage: BinaryTree([[None,[]],None]).to_dyck_word("R1L0")
[1, 1, 0, 1, 0, 0]
sage: BinaryTree([[None,[]],None]).to_dyck_word("R10L")
Traceback (most recent call last):
...
ValueError: R10L is not a correct map
```

TESTS:

```
sage: bt = BinaryTree([[[], [[], None]], [[], []]])
sage: bt == bt.to_dyck_word().to_binary_tree()
True
sage: bt == bt.to_dyck_word("1R0L").to_binary_tree("1R0L")
True
sage: bt == bt.to_dyck_word("L1R0").to_binary_tree("L1R0")
True
sage: bt == bt.to_dyck_word("R1L0").to_binary_tree("R1L0")
True
```

to_dyck_word_tamari()

Return the Dyck word associated with `self` in consistency with the Tamari order on Dyck words and binary trees.

The bijection is defined recursively as follows:

- a leaf is associated with an empty Dyck word;
- a tree with children l, r is associated with the Dyck word $T(l)1T(r)0$.

EXAMPLES:

```
sage: BinaryTree().to_dyck_word_tamari()
[]
sage: BinaryTree([]).to_dyck_word_tamari()
[1, 0]
sage: BinaryTree([[None,[]],None]).to_dyck_word_tamari()
[1, 1, 0, 0, 1, 0]
sage: BinaryTree([[[], [[], None]], [[], []]]).to_dyck_word_tamari()
[1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0]
```

to_ordered_tree_left_branch()

Return an ordered tree of size $n + 1$ by the following recursive rule:

- if x is the left child of y, x becomes the left brother of y
- if x is the right child of y, x becomes the last child of y

EXAMPLES:

```
sage: bt = BinaryTree([[],[]])
sage: bt.to_ordered_tree_left_branch()
[[], [[]]]
sage: bt = BinaryTree([[[], [[], None]], [[], [[]]])
sage: bt.to_ordered_tree_left_branch()
[[], [[], []], [[], [[[]]]]]
```

to_ordered_tree_right_branch()

Return an ordered tree of size $n + 1$ by the following recursive rule:

- if x is the right child of y, x becomes the right brother of y
- if x is the left child of y, x becomes the first child of y

EXAMPLES:

```
sage: bt = BinaryTree([[],[]])
sage: bt.to_ordered_tree_right_branch()
[[[]], []]
sage: bt = BinaryTree([[[], [[], None]], [[], [[]]])
sage: bt.to_ordered_tree_right_branch()
[[[[]], [[]]], [[]], []]
```

to_poset(*with_leaves=False*, *root_to_leaf=False*)

Return the poset obtained by interpreting the tree as a Hasse diagram.

The default orientation is from leaves to root but you can pass `root_to_leaf=True` to obtain the inverse orientation.

Leaves are ignored by default, but one can set `with_leaves` to `True` to obtain the poset of the complete tree.

INPUT:

- `with_leaves` – (default: `False`) a Boolean, determining whether the resulting poset will be formed from the leaves and the nodes of `self` (if `True`), or only from the nodes of `self` (if `False`)
- `root_to_leaf` – (default: `False`) a Boolean, determining whether the poset orientation should be from root to leaves (if `True`) or from leaves to root (if `False`).

EXAMPLES:

```
sage: bt = BinaryTree([])
sage: bt.to_poset()
Finite poset containing 1 elements
sage: bt.to_poset(with_leaves=True)
Finite poset containing 3 elements
sage: P1 = bt.to_poset(with_leaves=True)
sage: len(P1.maximal_elements())
1
sage: len(P1.minimal_elements())
2
sage: bt = BinaryTree([])
sage: P2 = bt.to_poset(with_leaves=True,root_to_leaf=True)
sage: len(P2.maximal_elements())
2
sage: len(P2.minimal_elements())
1
```

If the tree is labelled, we use its labelling to label the poset. Otherwise, we use the poset canonical labelling:

```
sage: bt = BinaryTree([[],[None,[]]]).canonical_labelling()
sage: bt
2[1[., .], 3[., 4[., .]]]
sage: bt.to_poset().cover_relations()
[[4, 3], [3, 2], [1, 2]]
```

Let us check that the empty binary tree is correctly handled:

```
sage: bt = BinaryTree()
sage: bt.to_poset()
Finite poset containing 0 elements
sage: bt.to_poset(with_leaves=True)
Finite poset containing 1 elements
```

to_undirected_graph(*with_leaves=False*)

Return the undirected graph obtained from the tree nodes and edges.

Leaves are ignored by default, but one can set `with_leaves` to `True` to obtain the graph of the complete tree.

INPUT:

- `with_leaves` – (default: `False`) a Boolean, determining whether the resulting graph will be formed from the leaves and the nodes of `self` (if `True`), or only from the nodes of `self` (if `False`)

EXAMPLES:

```
sage: bt = BinaryTree([])
sage: bt.to_undirected_graph()
Graph on 1 vertex
sage: bt.to_undirected_graph(with_leaves=True)
Graph on 3 vertices

sage: bt = BinaryTree()
sage: bt.to_undirected_graph()
Graph on 0 vertices
sage: bt.to_undirected_graph(with_leaves=True)
Graph on 1 vertex
```

If the tree is labelled, we use its labelling to label the graph. Otherwise, we use the graph canonical labelling which means that two different trees can have the same graph.

EXAMPLES:

```
sage: bt = BinaryTree([[],[None,[]]])
sage: bt.canonical_labelling().to_undirected_graph() == bt.to_undirected_graph()
False
sage: BinaryTree([[],[]]).to_undirected_graph() == BinaryTree([[[],None],None]).to_undirecte
True
```

under(*bt*)

Return `self` under `bt`, where "under" is the `under` ($\backslash$) operation.

If T and T' are two binary trees, then T under T' (written $T \backslash T'$) is defined as the tree obtained by grafting T on the leftmost leaf of T'. More precisely, $T \backslash T'$ is defined by identifying the root of T with the leftmost leaf of T'.

If T' is empty, then $T \backslash T' = T$.

The definition of this "under" operation goes back to Loday-Ronco [LodRon0102066] (Definition 2.2), but it is denoted by $/$ and called the "over" operation there. In fact, trees in sage have their root at the top,

contrary to the trees in [LodRon0102066] which are growing upwards. For this reason, the names of the over and under operations are swapped, in order to keep a graphical meaning. (Our notation follows that of section 4.5 of [HNT05].)

See also:

`over()`

EXAMPLES:

Showing only the nodes of a binary tree, here is an example for the under operation:

```
sage: b1 = BinaryTree([[],[]])
sage: b2 = BinaryTree([None,[]])
sage: ascii_art((b1, b2, b1 \ b2))
(   o  , o  ,     _o_   )
(  / \    \      /   \  )
( o   o    o    o     o )
(              / \      )
(             o   o     )
```

TESTS:

```
sage: b1 = BinaryTree([[],[[None,[]],None]]); ascii_art([b1])
[   _o_   ]
[  /   \  ]
[ o     o ]
[      /  ]
[     o   ]
[      \  ]
[       o ]
sage: b2 = BinaryTree([[],[None,[]]]); ascii_art([b2])
[   o     ]
[  / \    ]
[ o   o   ]
[      \  ]
[       o ]
sage: ascii_art([b1.under(b2)])
[        o_     ]
[       /  \    ]
[      o    o   ]
[     /      \  ]
[   _o_       o ]
[  /   \        ]
[ o     o       ]
[      /        ]
[     o         ]
[      \        ]
[       o       ]
```

The same in the labelled case:

```
sage: b1 = b1.canonical_labelling()
sage: b2 = b2.canonical_labelling()
sage: ascii_art([b1.under(b2)])
[        2_     ]
[       /  \    ]
[      1    3   ]
[     /      \  ]
[   _2_       4 ]
[  /   \        ]
```

```
[ 1     5       ]
[       /       ]
[     3         ]
[       \       ]
[        4      ]
```

class sage.combinat.binary_tree.**BinaryTrees**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Factory for binary trees.

INPUT:

- size – (optional) an integer

OUTPUT:

- the set of all binary trees (of the given size if specified)

EXAMPLES:

```
sage: BinaryTrees()
Binary trees

sage: BinaryTrees(2)
Binary trees of size 2
```

Note: this is a factory class whose constructor returns instances of subclasses.

Note: the fact that BinaryTrees is a class instead of a simple callable is an implementation detail. It could be changed in the future and one should not rely on it.

---

leaf()

Return a leaf tree with self as parent.

EXAMPLES:

```
sage: BinaryTrees().leaf()
.
```

TEST:

```
sage: (BinaryTrees().leaf() is
....:  sage.combinat.binary_tree.BinaryTrees_all().leaf())
True
```

class sage.combinat.binary_tree.**BinaryTrees_all**

Bases: sage.sets.disjoint_union_enumerated_sets.DisjointUnionEnumeratedSets, sage.combinat.binary_tree.BinaryTrees

TESTS:

```
sage: from sage.combinat.binary_tree import BinaryTrees_all
sage: B = BinaryTrees_all()
sage: B.cardinality()
+Infinity

sage: it = iter(B)
sage: (next(it), next(it), next(it), next(it), next(it))
```

```
(., [., .], [., [., .]], [[., .], .], [., [., [., .]]])
sage: next(it).parent()
Binary trees
sage: B([])
[., .]

sage: B is BinaryTrees_all()
True
sage: TestSuite(B).run() # long time
```

Element
alias of BinaryTree

labelled_trees()
Return the set of labelled trees associated to self.

EXAMPLES:
```
sage: BinaryTrees().labelled_trees()
Labelled binary trees
```

unlabelled_trees()
Return the set of unlabelled trees associated to self.

EXAMPLES:
```
sage: BinaryTrees().unlabelled_trees()
Binary trees
```

class sage.combinat.binary_tree.**BinaryTrees_size**(*size*)
Bases: sage.combinat.binary_tree.BinaryTrees

The enumerated sets of binary trees of given size

TESTS:
```
sage: from sage.combinat.binary_tree import BinaryTrees_size
sage: for i in range(6): TestSuite(BinaryTrees_size(i)).run()
```

cardinality()
The cardinality of self

This is a Catalan number.

TESTS:
```
sage: BinaryTrees(0).cardinality()
1
sage: BinaryTrees(5).cardinality()
42
```

element_class()
TESTS:
```
sage: S = BinaryTrees(3)
sage: S.element_class
<class 'sage.combinat.binary_tree.BinaryTrees_all_with_category.element_class'>
sage: S.first().__class__ == BinaryTrees().first().__class__
True
```

random_element()
Return a random BinaryTree with uniform probability.

This method generates a random DyckWord and then uses a bijection between Dyck words and binary trees.

EXAMPLES:

```
sage: BinaryTrees(5).random_element() # random
[., [., [., [., [., .]]]]]
sage: BinaryTrees(0).random_element()
.
sage: BinaryTrees(1).random_element()
[., .]
```

TESTS:

```
sage: all([BinaryTrees(10).random_element() in BinaryTrees(10) for i in range(20)])
True
```

class sage.combinat.binary_tree.**LabelledBinaryTree**(*parent*, *children*, *label=None*, *check=True*)

Bases: sage.combinat.abstract_tree.AbstractLabelledClonableTree, sage.combinat.binary_tree.BinaryTree

Labelled binary trees.

A labelled binary tree is a binary tree (see BinaryTree for the meaning of this) with a label assigned to each node. The labels need not be integers, nor are they required to be distinct. None can be used as a label.

Warning: While it is possible to assign values to leaves (not just nodes) using this class, these labels are disregarded by various methods such as labels(), map_labels(), and (ironically) leaf_labels().

INPUT:

- children – None (default) or a list, tuple or iterable of length 2 of labelled binary trees or convertible objects. This corresponds to the standard recursive definition of a labelled binary tree as being either a leaf, or a pair of:
  - a pair of labelled binary trees,
  - and a label.

  (The label is specified in the keyword variable label; see below.)

  Syntactic sugar allows leaving out all but the outermost calls of the LabelledBinaryTree() constructor, so that, e. g., LabelledBinaryTree([LabelledBinaryTree(None),LabelledBinaryTree(None)]) can be shortened to LabelledBinaryTree([None,None]). However, using this shorthand, it is impossible to label any vertex of the tree other than the root (because there is no way to pass a label variable without calling LabelledBinaryTree explicitly).

  It is also allowed to abbreviate [None, None] by [] if one does not want to label the leaves (which one should not do anyway!).
- label – (default: None) the label to be put on the root of this tree.
- check – (default: True) whether checks should be performed or not.

Todo

It is currently not possible to use LabelledBinaryTree() as a shorthand for LabelledBinaryTree(None) (in analogy to similar syntax in the BinaryTree class).

EXAMPLES:

```
sage: LabelledBinaryTree(None)
.
sage: LabelledBinaryTree(None, label="ae")     # not well supported
'ae'
sage: LabelledBinaryTree([])
None[., .]
sage: LabelledBinaryTree([], label=3)     # not well supported
3[., .]
sage: LabelledBinaryTree([None, None])
None[., .]
sage: LabelledBinaryTree([None, None], label=5)
5[., .]
sage: LabelledBinaryTree([None, []])
None[., None[., .]]
sage: LabelledBinaryTree([None, []], label=4)
4[., None[., .]]
sage: LabelledBinaryTree([[], None])
None[None[., .], .]
sage: LabelledBinaryTree("[[], .]", label=False)
False[None[., .], .]
sage: LabelledBinaryTree([None, LabelledBinaryTree([None, None], label=4)], label=3)
3[., 4[., .]]
sage: LabelledBinaryTree([None, BinaryTree([None, None])], label=3)
3[., None[., .]]

sage: LabelledBinaryTree([[], None, []])
Traceback (most recent call last):
...
ValueError: this is not a binary tree

sage: LBT = LabelledBinaryTree
sage: t1 = LBT([[LBT([], label=2), None], None], label=4); t1
4[None[2[., .], .], .]
```

TESTS:

```
sage: t1 = LabelledBinaryTree([[None, [[],[[], None]]],[[],[]]])
sage: t2 = LabelledBinaryTree([[[[],[]],[]],[]])
sage: with t1.clone() as t1c:
....:     t1c[1,1,1] = t2
sage: t1 == t1c
False
```

We check for trac ticket #16314:

```
sage: t1 = LBT([ LBT([LBT([], label=2),
....:                 LBT([], label=5)], label=6),
....:            None], label=4); t1
4[6[2[., .], 5[., .]], .]
sage: class Foo(LabelledBinaryTree):
....:     pass
sage: t2 = Foo(t1.parent(), t1); t2
4[6[2[., .], 5[., .]], .]
sage: t2.label()
4
sage: t2[0].label()
6
sage: t2.__class__, t2[0].__class__
(<class '__main__.Foo'>, <class '__main__.Foo'>)
```

binary_search_insert (*letter*)

Return the result of inserting a letter `letter` into the right strict binary search tree `self`.

INPUT:

- `letter` – any object comparable with the labels of `self`

OUTPUT:

The right strict binary search tree `self` with `letter` inserted into it according to the binary search insertion algorithm.

Note: `self` is supposed to be a binary search tree. This is not being checked!

A right strict binary search tree is defined to be a labelled binary tree such that for each node n with label x, every descendant of the left child of n has a label $\leq x$, and every descendant of the right child of n has a label $> x$. (Here, only nodes count as descendants, and every node counts as its own descendant too.) Leaves are assumed to have no labels.

Given a right strict binary search tree t and a letter i, the result of inserting i into t (denoted $Ins(i, t)$ in the following) is defined recursively as follows:

- If t is empty, then $Ins(i, t)$ is the tree with one node only, and this node is labelled with i.
- Otherwise, let j be the label of the root of t. If $i > j$, then $Ins(i, t)$ is obtained by replacing the right child of t by $Ins(i, r)$ in t, where r denotes the right child of t. If $i \leq j$, then $Ins(i, t)$ is obtained by replacing the left child of t by $Ins(i, l)$ in t, where l denotes the left child of t.

See, for example, [HNT05] for properties of this algorithm.

Warning: If t is nonempty, then inserting i into t does not change the root label of t. Hence, as opposed to algorithms like Robinson-Schensted-Knuth, binary search tree insertion involves no bumping.

EXAMPLES:

The example from Fig. 2 of [HNT05]:

```
sage: LBT = LabelledBinaryTree
sage: x = LBT(None)
sage: x
.
sage: x = x.binary_search_insert("b"); x
b[., .]
sage: x = x.binary_search_insert("d"); x
b[., d[., .]]
sage: x = x.binary_search_insert("e"); x
b[., d[., e[., .]]]
sage: x = x.binary_search_insert("a"); x
b[a[., .], d[., e[., .]]]
sage: x = x.binary_search_insert("b"); x
b[a[., b[., .]], d[., e[., .]]]
sage: x = x.binary_search_insert("d"); x
b[a[., b[., .]], d[d[., .], e[., .]]]
sage: x = x.binary_search_insert("a"); x
b[a[a[., .], b[., .]], d[d[., .], e[., .]]]
sage: x = x.binary_search_insert("c"); x
b[a[a[., .], b[., .]], d[d[c[., .], .], e[., .]]]
```

Other examples:

```
sage: LBT = LabelledBinaryTree
sage: LBT(None).binary_search_insert(3)
3[., .]
sage: LBT([], label = 1).binary_search_insert(3)
1[., 3[., .]]
sage: LBT([], label = 3).binary_search_insert(1)
3[1[., .], .]
sage: res = LBT(None)
sage: for i in [3,1,5,2,4,6]:
....:     res = res.binary_search_insert(i)
sage: res
3[1[., 2[., .]], 5[4[., .], 6[., .]]]
```

heap_insert(*l*)

Return the result of inserting a letter `l` into the binary heap (tree) `self`.

A binary heap is a labelled complete binary tree such that for each node, the label at the node is greater or equal to the label of each of its child nodes. (More precisely, this is called a max-heap.)

For example:

```
|      _7_    |
|     /   \   |
|    5     6  |
|   / \       |
|  3   4      |
```

is a binary heap.

See Wikipedia article Binary_heap#Insert for a description of how to insert a letter into a binary heap. The result is another binary heap.

INPUT:

- `letter` – any object comparable with the labels of `self`

Note: `self` is assumed to be a binary heap (tree). No check is performed.

TESTS:

```
sage: h = LabelledBinaryTree(None)
sage: h = h.heap_insert(3); ascii_art([h])
[ 3 ]
sage: h = h.heap_insert(4); ascii_art([h])
[   4 ]
[  /  ]
[ 3   ]
sage: h = h.heap_insert(6); ascii_art([h])
[   6   ]
[  / \  ]
[ 3   4 ]
sage: h = h.heap_insert(2); ascii_art([h])
[     6   ]
[    / \  ]
[   3   4 ]
[  /      ]
[ 2       ]
sage: ascii_art([h.heap_insert(5)])
[     _6_   ]
```

```
[    /   \  ]
[   5     4 ]
[  / \      ]
[ 2   3     ]
```

left_rotate()

Return the result of left rotation applied to the labelled binary tree `self`.

Left rotation on labelled binary trees is defined as follows: Let T be a labelled binary tree such that the right child of the root of T is a node. Let A be the left child of the root of T, and let B and C be the left and right children of the right child of the root of T. (Keep in mind that nodes of trees are identified with the subtrees consisting of their descendants.) Furthermore, let x be the label at the root of T, and y be the label at the right child of the root of T. Then, the left rotation of T is the labelled binary tree in which the root is labelled y, the right child of the root is C, whereas the left child of the root is a node labelled x whose left and right children are A and B. In pictures:

```
|     y                    x     |
|    / \                  / \    |
|   x   C <-left-rotate- A   y   |
|  / \                      / \  |
| A   B                    B   C |
```

Left rotation is the inverse operation to right rotation (`right_rotate()`).

TESTS:

```
sage: LB = LabelledBinaryTree
sage: b = LB([LB([LB([],"A"), LB([],"B")],"x"),LB([],"C")], "y"); b
y[x[A[., .], B[., .]], C[., .]]
sage: b == b.right_rotate().left_rotate()
True
```

right_rotate()

Return the result of right rotation applied to the labelled binary tree `self`.

Right rotation on labelled binary trees is defined as follows: Let T be a labelled binary tree such that the left child of the root of T is a node. Let C be the right child of the root of T, and let A and B be the left and right children of the left child of the root of T. (Keep in mind that nodes of trees are identified with the subtrees consisting of their descendants.) Furthermore, let y be the label at the root of T, and x be the label at the left child of the root of T. Then, the right rotation of T is the labelled binary tree in which the root is labelled x, the left child of the root is A, whereas the right child of the root is a node labelled y whose left and right children are B and C. In pictures:

```
|     y                     x     |
|    / \                   / \    |
|   x   C -right-rotate->  A   y   |
|  / \                        / \  |
| A   B                      B   C |
```

Right rotation is the inverse operation to left rotation (`left_rotate()`).

TESTS:

```
sage: LB = LabelledBinaryTree
sage: b = LB([LB([LB([],"A"), LB([],"B")],"x"),LB([],"C")], "y"); b
y[x[A[., .], B[., .]], C[., .]]
sage: b.right_rotate()
x[A[., .], y[B[., .], C[., .]]]
```

semistandard_insert(*letter*)

Return the result of inserting a letter `letter` into the semistandard tree `self` using the bumping algorithm.

INPUT:

- `letter` – any object comparable with the labels of `self`

OUTPUT:

The semistandard tree `self` with `letter` inserted into it according to the bumping algorithm.

Note: `self` is supposed to be a semistandard tree. This is not being checked!

A semistandard tree is defined to be a labelled binary tree such that for each node n with label x, every descendant of the left child of n has a label $> x$, and every descendant of the right child of n has a label $\geq x$. (Here, only nodes count as descendants, and every node counts as its own descendant too.) Leaves are assumed to have no labels.

Given a semistandard tree t and a letter i, the result of inserting i into t (denoted $Ins(i, t)$ in the following) is defined recursively as follows:

- If t is empty, then $Ins(i, t)$ is the tree with one node only, and this node is labelled with i.
- Otherwise, let j be the label of the root of t. If $i \geq j$, then $Ins(i, t)$ is obtained by replacing the right child of t by $Ins(i, r)$ in t, where r denotes the right child of t. If $i < j$, then $Ins(i, t)$ is obtained by replacing the label at the root of t by i, and replacing the left child of t by $Ins(j, l)$ in t, where l denotes the left child of t.

This algorithm is similar to the Robinson-Schensted-Knuth insertion algorithm for semistandard Young tableaux.

AUTHORS:

- Darij Grinberg (10 Nov 2013).

EXAMPLES:

```
sage: LBT = LabelledBinaryTree
sage: x = LBT(None)
sage: x
.
sage: x = x.semistandard_insert("b"); x
b[., .]
sage: x = x.semistandard_insert("d"); x
b[., d[., .]]
sage: x = x.semistandard_insert("e"); x
b[., d[., e[., .]]]
sage: x = x.semistandard_insert("a"); x
a[b[., .], d[., e[., .]]]
sage: x = x.semistandard_insert("b"); x
a[b[., .], b[d[., .], e[., .]]]
sage: x = x.semistandard_insert("d"); x
a[b[., .], b[d[., .], d[e[., .], .]]]
sage: x = x.semistandard_insert("a"); x
a[b[., .], a[b[d[., .], .], d[e[., .], .]]]
sage: x = x.semistandard_insert("c"); x
a[b[., .], a[b[d[., .], .], c[d[e[., .], .], .]]]
```

Other examples:

```
sage: LBT = LabelledBinaryTree
sage: LBT(None).semistandard_insert(3)
3[., .]
sage: LBT([], label = 1).semistandard_insert(3)
1[., 3[., .]]
sage: LBT([], label = 3).semistandard_insert(1)
1[3[., .], .]
sage: res = LBT(None)
sage: for i in [3,1,5,2,4,6]:
....:     res = res.semistandard_insert(i)
sage: res
1[3[., .], 2[5[., .], 4[., 6[., .]]]]
```

class sage.combinat.binary_tree.**LabelledBinaryTrees**(*category=None*)
Bases: sage.combinat.ordered_tree.LabelledOrderedTrees

This is a parent stub to serve as a factory class for trees with various labels constraints.

Element
alias of LabelledBinaryTree

labelled_trees()
Return the set of labelled trees associated to self.

EXAMPLES:

```
sage: LabelledBinaryTrees().labelled_trees()
Labelled binary trees
```

unlabelled_trees()
Return the set of unlabelled trees associated to self.

EXAMPLES:

```
sage: LabelledBinaryTrees().unlabelled_trees()
Binary trees
```

This is used to compute the shape:

```
sage: t = LabelledBinaryTrees().an_element().shape(); t
[[[., .], [., .]], [[., .], [., .]]]
sage: t.parent()
Binary trees
```

TESTS:

```
sage: t = LabelledBinaryTrees().an_element()
sage: t.canonical_labelling()
4[2[1[., .], 3[., .]], 6[5[., .], 7[., .]]]
```

5.1.11 Cartesian Products

sage.combinat.cartesian_product.**CartesianProduct**(**iters*)
This is deprecated. Use cartesian_product instead.

EXAMPLES:

```
sage: cp = CartesianProduct([1,2], [3,4]); cp
doctest:...: DeprecationWarning: CartesianProduct is deprecated. Use
cartesian_product instead
```

```
See http://trac.sagemath.org/18411 for details.
The Cartesian product of ({1, 2}, {3, 4})
sage: cp.list()
[(1, 3), (1, 4), (2, 3), (2, 4)]
```

Note that you must not use a generator-type object that is returned by a function (using "yield"). They cannot be copied or rewound (you cannot jump back to the beginning), but this is necessary to construct the Cartesian product:

```
sage: def a(n): yield 1*n; yield 2*n
sage: def b(): yield 'a'; yield 'b'
sage: CartesianProduct(a(3), b()).list()
Traceback (most recent call last):
...
ValueError: generators are not allowed, see the
documentation (type "CartesianProduct?") for a workaround
```

The usage of iterable is also deprecated, so the following will no longer be supported:

```
sage: from sage.combinat.misc import IterableFunctionCall
sage: C = CartesianProduct(IterableFunctionCall(a, 3), IterableFunctionCall(b))
doctest:...: DeprecationWarning: Usage of IterableFunctionCall in
CartesianProduct is deprecated. You can use EnumeratedSetFromIterator
(in sage.sets.set_from_iterator) instead.
See http://trac.sagemath.org/18411 for details.
sage: list(C)
doctest:...: UserWarning: Sage is not able to determine whether the
factors of this Cartesian product are finite. The lexicographic ordering
might not go through all elements.
[(3, 'a'), (3, 'b'), (6, 'a'), (6, 'b')]
```

You might use `EnumeratedSetFromIterator` for that purpose.:

```
sage: from sage.sets.set_from_iterator import EnumeratedSetFromIterator
sage: A = EnumeratedSetFromIterator(a, (3,), category=FiniteEnumeratedSets())
sage: B = EnumeratedSetFromIterator(b, category=FiniteEnumeratedSets())
sage: C = cartesian_product([A, B])
sage: C.list()
[(3, 'a'), (3, 'b'), (6, 'a'), (6, 'b')]
```

class sage.combinat.cartesian_product.**CartesianProduct_iters**(**iters*)

Bases: sage.combinat.combinat.CombinatorialClass

Cartesian product of finite sets.

This class will soon be deprecated (see trac ticket #18411 and trac ticket #19195). One should instead use the functorial construction `cartesian_product`. The main differences in behavior are:

- construction: `CartesianProduct` takes as many argument as there are factors whereas `cartesian_product` takes a single list (or iterable) of factors;
- representation of elements: elements are represented by plain Python list for `CartesianProduct` versus a custom element class for `cartesian_product`;
- membership testing: because of the above, plain Python lists are not considered as elements of a `cartesian_product`.

All of these is illustrated in the examples below.

EXAMPLES:

```
sage: F1 = ['a', 'b']
sage: F2 = [1, 2, 3, 4]
sage: F3 = Permutations(3)
sage: from sage.combinat.cartesian_product import CartesianProduct_iters
sage: C = CartesianProduct_iters(F1, F2, F3)
sage: c = cartesian_product([F1, F2, F3])

sage: type(C.an_element())
<type 'list'>
sage: type(c.an_element())
<class 'sage.sets.cartesian_product.CartesianProduct_with_category.element_class'>

sage: l = ['a', 1, Permutation([3,2,1])]
sage: l in C
True
sage: l in c
False
sage: elt = c(l)
sage: elt
('a', 1, [3, 2, 1])
sage: elt in c
True
sage: elt.parent() is c
True
```

cardinality()

Returns the number of elements in the Cartesian product of everything in *iters.

EXAMPLES:

```
sage: from sage.combinat.cartesian_product import CartesianProduct_iters
sage: CartesianProduct_iters(range(2), range(3)).cardinality()
6
sage: CartesianProduct_iters(range(2), xrange(3)).cardinality()
6
sage: CartesianProduct_iters(range(2), xrange(3), xrange(4)).cardinality()
24
```

This works correctly for infinite objects:

```
sage: CartesianProduct_iters(ZZ, QQ).cardinality()
+Infinity
sage: CartesianProduct_iters(ZZ, []).cardinality()
0
```

is_finite()

The Cartesian product is finite if all of its inputs are finite, or if any input is empty.

EXAMPLES:

```
sage: from sage.combinat.cartesian_product import CartesianProduct_iters
sage: CartesianProduct_iters(ZZ, []).is_finite()
True
sage: CartesianProduct_iters(4,4).is_finite()
Traceback (most recent call last):
...
ValueError: Unable to determine whether this product is finite
```

list()

Returns

EXAMPLES:

```
sage: from sage.combinat.cartesian_product import CartesianProduct_iters
sage: CartesianProduct_iters(range(3), range(3)).list()
[[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1], [2, 2]]
sage: CartesianProduct_iters('dog', 'cat').list()
[['d', 'c'],
 ['d', 'a'],
 ['d', 't'],
 ['o', 'c'],
 ['o', 'a'],
 ['o', 't'],
 ['g', 'c'],
 ['g', 'a'],
 ['g', 't']]
```

random_element()

Returns a random element from the Cartesian product of *iters.

EXAMPLES:

```
sage: from sage.combinat.cartesian_product import CartesianProduct_iters
sage: CartesianProduct_iters('dog', 'cat').random_element()
['d', 'a']
```

unrank(*x*)

For finite Cartesian products, we can reduce unrank to the constituent iterators.

EXAMPLES:

```
sage: from sage.combinat.cartesian_product import CartesianProduct_iters
sage: C = CartesianProduct_iters(xrange(1000), xrange(1000), xrange(1000))
sage: C[238792368]
[238, 792, 368]
```

Check for trac ticket #15919:

```
sage: FF = IntegerModRing(29)
sage: C = CartesianProduct_iters(FF, FF, FF)
sage: C.unrank(0)
[0, 0, 0]
```

sage.combinat.cartesian_product.**isgenerator**(*obj*)

5.1.12 Enumerated sets of partitions, tableaux, ...

Quickref

Catalog

- *Integer partitions*
- *Tableaux*
- *Partition tuples*
- *TableauTuples*

- *Skew Partitions*
- *Skew Tableaux*
- *Ribbons*
- *Ribbon Tableaux*
- *Cores*
- *Strong and weak tableaux*
- *Robinson-Schensted-Knuth correspondence*

5.1.13 Cluster Algebras and Quivers

- A compendium on the cluster algebra and quiver package in Sage arXiv:1102.4844 [math.CO]
- *Quiver mutation types*
- *Quiver*
- *ClusterSeed*

5.1.14 Cluster algebra and quivers features that are imported by default in the interpreter namespace

5.1.15 ClusterSeed

A *cluster seed* is a pair $(B, \mathbf{x})$ with B being a *skew-symmetrizable* $(n + m \times n)$ *-matrix* and with $\mathbf{x}$ being an n-tuple of *independent elements* in the field of rational functions in n variables.

For the compendium on the cluster algebra and quiver package see Arxiv 1102.4844.

AUTHORS:

- Gregg Musiker: Initial Version
- Christian Stump: Initial Version
- Aram Dermenjian (2015-07-01): Updating ability to not rely solely on clusters
- Jesse Levitt (2015-07-01): Updating ability to not rely solely on clusters

REFERENCES:

See also:

For mutation types of cluster seeds, see `sage.combinat.cluster_algebra_quiver.quiver_mutation_type.QuiverM`
Cluster seeds are closely related to `sage.combinat.cluster_algebra_quiver.quiver.ClusterQuiver()`.

class `sage.combinat.cluster_algebra_quiver.cluster_seed.`**`ClusterSeed`**(*data*, *frozen=None*, *is_principal=False*, *user_labels=None*, *user_labels_prefix='x'*)

Bases: `sage.structure.sage_object.SageObject`

The *cluster seed* associated to an *exchange matrix*.

INPUT:

- `data` – can be any of the following:

```
* QuiverMutationType
* str - a string representing a QuiverMutationType or a common quiver type (see Examples)
* ClusterQuiver
* Matrix - a skew-symmetrizable matrix
* DiGraph - must be the input data for a quiver
* List of edges - must be the edge list of a digraph for a quiver
```

EXAMPLES:

```
sage: S = ClusterSeed(['A',5]); S
A seed for a cluster algebra of rank 5 of type ['A', 5]

sage: S = ClusterSeed(['A',[2,5],1]); S
A seed for a cluster algebra of rank 7 of type ['A', [2, 5], 1]

sage: T = ClusterSeed( S ); T
A seed for a cluster algebra of rank 7 of type ['A', [2, 5], 1]

sage: T = ClusterSeed( S._M ); T
A seed for a cluster algebra of rank 7

sage: T = ClusterSeed( S.quiver()._digraph ); T
A seed for a cluster algebra of rank 7

sage: T = ClusterSeed( S.quiver()._digraph.edges() ); T
A seed for a cluster algebra of rank 7

sage: S = ClusterSeed(['B',2]); S
A seed for a cluster algebra of rank 2 of type ['B', 2]

sage: S = ClusterSeed(['C',2]); S
A seed for a cluster algebra of rank 2 of type ['B', 2]

sage: S = ClusterSeed(['A', [5,0],1]); S
A seed for a cluster algebra of rank 5 of type ['D', 5]

sage: S = ClusterSeed(['GR',[3,7]]); S
A seed for a cluster algebra of rank 6 of type ['E', 6]

sage: S = ClusterSeed(['F', 4, [2,1]]); S
A seed for a cluster algebra of rank 6 of type ['F', 4, [1, 2]]

sage: S = ClusterSeed(['A',4]); S._use_fpolys
True

sage: S._use_d_vec
True

sage: S._use_g_vec
True

sage: S._use_c_vec
True

sage: S = ClusterSeed(['A',4]); S.use_fpolys(False); S._use_fpolys
False
```

b_matrix()
Returns the B *-matrix* of `self`.

EXAMPLES:

```
sage: ClusterSeed(['A',4]).b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

sage: ClusterSeed(['B',4]).b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -2  0]

sage: ClusterSeed(['D',4]).b_matrix()
[ 0  1  0  0]
[-1  0 -1 -1]
[ 0  1  0  0]
[ 0  1  0  0]

sage: ClusterSeed(QuiverMutationType([['A',2],['B',2]])).b_matrix()
[ 0  1  0  0]
[-1  0  0  0]
[ 0  0  0  1]
[ 0  0 -2  0]
```

b_matrix_class(*depth=+Infinity*, *up_to_equivalence=True*)
Returns all B-matrices in the mutation class of `self`.

INPUT:

- `depth` – (default:infinity) integer or infinity, only seeds with distance at most depth from self are returned
- `up_to_equivalence` – (default: True) if True, only 'B'-matrices up to equivalence are considered.

EXAMPLES:

- for examples see `b_matrix_class_iter()`

TESTS:

```
sage: A = ClusterSeed(['A',3]).b_matrix_class()
sage: A = ClusterSeed(['A',[2,1],1]).b_matrix_class()
```

b_matrix_class_iter(*depth=+Infinity*, *up_to_equivalence=True*)
Returns an iterator through all B-matrices in the mutation class of `self`.

INPUT:

- `depth` – (default:infinity) integer or infinity, only seeds with distance at most depth from self are returned
- `up_to_equivalence` – (default: True) if True, only 'B'-matrices up to equivalence are considered.

EXAMPLES:

A standard finite type example:

```
sage: S = ClusterSeed(['A',4])
sage: it = S.b_matrix_class_iter()
sage: for T in it: print T
[ 0  0  0  1]
[ 0  0  1  1]
[ 0 -1  0  0]
[-1 -1  0  0]
[ 0  0  0  1]
[ 0  0  1  0]
[ 0 -1  0  1]
[-1  0 -1  0]
[ 0  0  1  1]
[ 0  0  0 -1]
[-1  0  0  0]
[-1  1  0  0]
[ 0  0  0  1]
[ 0  0 -1  1]
[ 0  1  0 -1]
[-1 -1  1  0]
[ 0  0  0  1]
[ 0  0 -1  0]
[ 0  1  0 -1]
[-1  0  1  0]
[ 0  0  0 -1]
[ 0  0 -1  1]
[ 0  1  0 -1]
[ 1 -1  1  0]
```

A finite type example with given depth:

```
sage: it = S.b_matrix_class_iter(depth=1)
sage: for T in it: print T
[ 0  0  0  1]
[ 0  0  1  1]
[ 0 -1  0  0]
[-1 -1  0  0]
[ 0  0  0  1]
[ 0  0  1  0]
[ 0 -1  0  1]
[-1  0 -1  0]
[ 0  0  1  1]
[ 0  0  0 -1]
[-1  0  0  0]
[-1  1  0  0]
```

Finite type example not considered up to equivalence:

```
sage: S = ClusterSeed(['A',3])
sage: it = S.b_matrix_class_iter(up_to_equivalence=False)
sage: for T in it: print T
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
[ 0  1  0]
[-1  0  1]
[ 0 -1  0]
[ 0 -1  0]
[ 1  0  1]
[ 0 -1  0]
```

```
[ 0 -1  0]
[ 1  0 -1]
[ 0  1  0]
[ 0 -1  1]
[ 1  0 -1]
[-1  1  0]
[ 0  1 -1]
[-1  0  1]
[ 1 -1  0]
[ 0  0  1]
[ 0  0 -1]
[-1  1  0]
[ 0 -1  1]
[ 1  0  0]
[-1  0  0]
[ 0  0 -1]
[ 0  0  1]
[ 1 -1  0]
[ 0  1 -1]
[-1  0  0]
[ 1  0  0]
[ 0  1  1]
[-1  0  0]
[-1  0  0]
[ 0 -1 -1]
[ 1  0  0]
[ 1  0  0]
[ 0  0 -1]
[ 0  0 -1]
[ 1  1  0]
[ 0  0  1]
[ 0  0  1]
[-1 -1  0]
```

Infinite (but finite mutation) type example:

```
sage: S = ClusterSeed(['A',[1,2],1])
sage: it = S.b_matrix_class_iter()
sage: for T in it: print T
[ 0  1  1]
[-1  0  1]
[-1 -1  0]
[ 0 -2  1]
[ 2  0 -1]
[-1  1  0]
```

Infinite mutation type example:

```
sage: S = ClusterSeed(['E',10])
sage: it = S.b_matrix_class_iter(depth=3)
sage: len ( [T for T in it] )
266
```

c_matrix(*show_warnings=True*)

Returns all *c-vectors* of `self`.

Warning: this method assumes the sign-coherence conjecture and that the input seed is sign-coherent (has an exchange matrix with columns of like signs). Otherwise, computational errors might arise.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
sage: S.c_matrix()
[ 1  0  0]
[ 0  0 -1]
[ 0 -1  0]

sage: S = ClusterSeed(['A',4]);
sage: S.use_g_vectors(False); S.use_fpolys(False);  S.use_c_vectors(False);  S.use_d_vectors
sage: S.c_matrix()
Traceback (most recent call last):
...
ValueError: Unable to calculate c-vectors. Need to use c vectors.
```

c_vector(*k*)

Returns the k-th *c-vector* of self. It is obtained as the k-th column vector of the bottom part of the B-matrix of self.

Warning: this method assumes the sign-coherence conjecture and that the input seed is sign-coherent (has an exchange matrix with columns of like signs). Otherwise, computational errors might arise.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
sage: [ S.c_vector(k) for k in range(3) ]
[(1, 0, 0), (0, 0, -1), (0, -1, 0)]

sage: S = ClusterSeed(Matrix([[0,1],[-1,0],[1,0],[-1,1]])); S
A seed for a cluster algebra of rank 2 with 2 frozen variables
sage: S.c_vector(0)
(1, 0)

sage: S = ClusterSeed(Matrix([[0,1],[-1,0],[1,0],[-1,1]])); S.use_c_vectors(bot_is_c=True);
A seed for a cluster algebra of rank 2 with 2 frozen variables
sage: S.c_vector(0)
(1, -1)
```

cluster()

Returns a copy of the *cluster* of self.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.cluster()
[x0, x1, x2]

sage: S.mutate(1)
sage: S.cluster()
[x0, (x0*x2 + 1)/x1, x2]

sage: S.mutate(2)
sage: S.cluster()
[x0, (x0*x2 + 1)/x1, (x0*x2 + x1 + 1)/(x1*x2)]

sage: S.mutate([2,1])
sage: S.cluster()
[x0, x1, x2]
```

cluster_class(*depth=+Infinity*, *show_depth=False*, *up_to_equivalence=True*)
Returns the cluster class of `self` with respect to certain constraints.

INPUT:

- `depth` – (default: infinity) integer, only seeds with distance at most depth from self are returned
- `return_depth` – (default False) - if True, ignored if depth is set; returns the depth of the mutation class, i.e., the maximal distance from self of an element in the mutation class
- `up_to_equivalence` – (default: True) if True, only clusters up to equivalence are considered.

EXAMPLES:

- for examples see `cluster_class_iter()`

TESTS:

```
sage: A = ClusterSeed(['A',3]).cluster_class()
```

cluster_class_iter(*depth=+Infinity*, *show_depth=False*, *up_to_equivalence=True*)
Returns an iterator through all clusters in the mutation class of `self`.

INPUT:

- `depth` – (default: infinity) integer or infinity, only seeds with distance at most depth from self are returned
- `show_depth` – (default False) - if True, ignored if depth is set; returns the depth of the mutation class, i.e., the maximal distance from self of an element in the mutation class
- `up_to_equivalence` – (default: True) if True, only clusters up to equivalence are considered.

EXAMPLES:

A standard finite type example:

```
sage: S = ClusterSeed(['A',3])
sage: it = S.cluster_class_iter()
sage: for T in it: print T
[x0, x1, x2]
[x0, x1, (x1 + 1)/x2]
[x0, (x0*x2 + 1)/x1, x2]
[(x1 + 1)/x0, x1, x2]
[x0, (x0*x2 + x1 + 1)/(x1*x2), (x1 + 1)/x2]
[(x1 + 1)/x0, x1, (x1 + 1)/x2]
[(x1 + 1)/x0, (x0*x2 + x1 + 1)/(x0*x1), x2]
[x0, (x0*x2 + 1)/x1, (x0*x2 + x1 + 1)/(x1*x2)]
[(x0*x2 + x1 + 1)/(x0*x1), (x0*x2 + 1)/x1, x2]
[(x1 + 1)/x0, (x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2), (x1 + 1)/x2]
[(x1 + 1)/x0, (x0*x2 + x1 + 1)/(x0*x1), (x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2)]
[(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2), (x0*x2 + x1 + 1)/(x1*x2), (x1 + 1)/x2]
[(x0*x2 + x1 + 1)/(x0*x1), (x0*x2 + 1)/x1, (x0*x2 + x1 + 1)/(x1*x2)]
[(x0*x2 + x1 + 1)/(x1*x2), (x0*x2 + x1 + 1)/(x0*x1), (x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2)]
```

A finite type example with given depth:

```
sage: it = S.cluster_class_iter(depth=1)
sage: for T in it: print T
[x0, x1, x2]
[x0, x1, (x1 + 1)/x2]
[x0, (x0*x2 + 1)/x1, x2]
[(x1 + 1)/x0, x1, x2]
```

A finite type example where the depth is returned while computing:

```
sage: it = S.cluster_class_iter(show_depth=True)
sage: for T in it: print T
[x0, x1, x2]
Depth: 0     found: 1          Time: ... s
[x0, x1, (x1 + 1)/x2]
[x0, (x0*x2 + 1)/x1, x2]
[(x1 + 1)/x0, x1, x2]
Depth: 1     found: 4          Time: ... s
[x0, (x0*x2 + x1 + 1)/(x1*x2), (x1 + 1)/x2]
[(x1 + 1)/x0, x1, (x1 + 1)/x2]
[(x1 + 1)/x0, (x0*x2 + x1 + 1)/(x0*x1), x2]
[x0, (x0*x2 + 1)/x1, (x0*x2 + x1 + 1)/(x1*x2)]
[(x0*x2 + x1 + 1)/(x0*x1), (x0*x2 + 1)/x1, x2]
Depth: 2     found: 9          Time: ... s
[(x1 + 1)/x0, (x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2), (x1 + 1)/x2]
[(x1 + 1)/x0, (x0*x2 + x1 + 1)/(x0*x1), (x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2)]
[(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2), (x0*x2 + x1 + 1)/(x1*x2), (x1 + 1)/x2]
[(x0*x2 + x1 + 1)/(x0*x1), (x0*x2 + 1)/x1, (x0*x2 + x1 + 1)/(x1*x2)]
Depth: 3     found: 13         Time: ... s
[(x0*x2 + x1 + 1)/(x1*x2), (x0*x2 + x1 + 1)/(x0*x1), (x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2)]
Depth: 4     found: 14         Time: ... s
```

Finite type examples not considered up to equivalence:

```
sage: it = S.cluster_class_iter(up_to_equivalence=False)
sage: len( [ T for T in it ] )
84

sage: it = ClusterSeed(['A',2]).cluster_class_iter(up_to_equivalence=False)
sage: for T in it: print T
[x0, x1]
[x0, (x0 + 1)/x1]
[(x1 + 1)/x0, x1]
[(x1 + 1)/x0, (x0 + x1 + 1)/(x0*x1)]
[(x0 + x1 + 1)/(x0*x1), (x0 + 1)/x1]
[(x0 + x1 + 1)/(x0*x1), (x1 + 1)/x0]
[(x0 + 1)/x1, (x0 + x1 + 1)/(x0*x1)]
[x1, (x1 + 1)/x0]
[(x0 + 1)/x1, x0]
[x1, x0]
```

Infinite type examples:

```
sage: S = ClusterSeed(['A',[1,1],1])
sage: it = S.cluster_class_iter()
sage: next(it)
[x0, x1]
sage: next(it)
[x0, (x0^2 + 1)/x1]
sage: next(it)
[(x1^2 + 1)/x0, x1]
sage: next(it)
[(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2), (x0^2 + 1)/x1]
sage: next(it)
[(x1^2 + 1)/x0, (x1^4 + x0^2 + 2*x1^2 + 1)/(x0^2*x1)]

sage: it = S.cluster_class_iter(depth=3)
sage: for T in it: print T
```

```
[x0, x1]
[x0, (x0^2 + 1)/x1]
[(x1^2 + 1)/x0, x1]
[(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2), (x0^2 + 1)/x1]
[(x1^2 + 1)/x0, (x1^4 + x0^2 + 2*x1^2 + 1)/(x0^2*x1)]
[(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2), (x0^6 + 3*x0^4 + 2*x0^2*x1^2 + x1^4 + 3*x0^2 + 2*x1^2
[(x1^6 + x0^4 + 2*x0^2*x1^2 + 3*x1^4 + 2*x0^2 + 3*x1^2 + 1)/(x0^3*x1^2), (x1^4 + x0^2 + 2*x1
```

cluster_index(*cluster_str*)

Returns the index of a cluster if use_fpolys is on

INPUT:

- `cluster_str` – The string to look for in the cluster

OUTPUT:

Returns an integer or None if the string is not a cluster variable

EXAMPLES:

```
sage: S = ClusterSeed(['A',4],user_labels=['x','y','z','w']); S.mutate('x')
sage: S.cluster_index('x')
sage: S.cluster_index('(y+1)/x')
0
```

cluster_variable(*k*)

Generates a cluster variable using F-polynomials

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.mutate([0,1])
sage: S.cluster_variable(0)
(x1 + 1)/x0
sage: S.cluster_variable(1)
(x0*x2 + x1 + 1)/(x0*x1)
```

coefficient(*k*)

Returns the *coefficient* of `self` at index `k`.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
sage: [ S.coefficient(k) for k in range(3) ]
[y0, 1/y2, 1/y1]
```

coefficients()

Returns all *coefficients* of `self`.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
sage: S.coefficients()
[y0, 1/y2, 1/y1]
```

d_matrix(*show_warnings=True*)

Returns the matrix of *d-vectors* of `self`.

EXAMPLES:: sage: S = ClusterSeed(['A',4]); S.d_matrix() [-1 0 0 0] [0 -1 0 0] [0 0 -1 0] [0 0 0 -1]
sage: S.mutate([1,2,1,0,1,3]); S.d_matrix() [1 1 0 1] [1 1 1 1] [1 0 1 1] [0 0 0 1]

d_vector(*k*)

Returns the k-th *d-vector* of self. This is the exponent vector of the denominator of the k-th cluster variable.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.mutate([2,1,2])
sage: [ S.d_vector(k) for k in range(3) ]
[(-1, 0, 0), (0, 1, 1), (0, 1, 0)]
```

exchangeable_part()

Returns the restriction to the principal part (i.e. the exchangeable variables) of self.

EXAMPLES:

```
sage: S = ClusterSeed(['A',4])
sage: T = ClusterSeed( S.quiver().digraph().edges(), frozen=1 )
sage: T.quiver().digraph().edges()
[(0, 1, (1, -1)), (2, 1, (1, -1)), (2, 3, (1, -1))]

sage: T.exchangeable_part().quiver().digraph().edges()
[(0, 1, (1, -1)), (2, 1, (1, -1))]
```

f_polynomial(*k*)

Returns the k-th *F-polynomial* of self. It is obtained from the k-th cluster variable by setting all x_i to 1.

Warning: this method assumes the sign-coherence conjecture and that the input seed is sign-coherent (has an exchange matrix with columns of like signs). Otherwise, computational errors might arise.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
sage: [S.f_polynomial(k) for k in range(3)]
[1, y1*y2 + y2 + 1, y1 + 1]

sage: S = ClusterSeed(Matrix([[0,1],[-1,0],[1,0],[-1,1]])); S.use_c_vectors(bot_is_c=True);
A seed for a cluster algebra of rank 2 with 2 frozen variables
sage: T = ClusterSeed(Matrix([[0,1],[-1,0]])).principal_extension(); T
A seed for a cluster algebra of rank 2 with principal coefficients
sage: S.mutate(0)
sage: T.mutate(0)
sage: S.f_polynomials()
[y0 + y1, 1]
sage: T.f_polynomials()
[y0 + 1, 1]
```

f_polynomials()

Returns all *F-polynomials* of self. These are obtained from the cluster variables by setting all x_i's to 1.

Warning: this method assumes the sign-coherence conjecture and that the input seed is sign-coherent (has an exchange matrix with columns of like signs). Otherwise, computational errors might arise.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
```

```
sage: S.f_polynomials()
[1, y1*y2 + y2 + 1, y1 + 1]
```

first_green_vertex()
Return the first green vertex of `self`.

A vertex is defined to be green if its c-vector has all non-positive entries. More information on green vertices can be found at [BDP2013]

EXAMPLES:

```
sage: ClusterSeed(['A',3]).principal_extension().first_green_vertex()
0

sage: ClusterSeed(['A',[3,3],1]).principal_extension().first_green_vertex()
0
```

first_red_vertex()
Return the first red vertex of `self`.

A vertex is defined to be red if its c-vector has all non-negative entries. More information on red vertices can be found at [BDP2013].

EXAMPLES:

```
sage: ClusterSeed(['A',3]).principal_extension().first_red_vertex()

sage: ClusterSeed(['A',[3,3],1]).principal_extension().first_red_vertex()

sage: Q = ClusterSeed(['A',[3,3],1]).principal_extension();
sage: Q.mutate(1);
sage: Q.first_red_vertex()
1
```

first_urban_renewal()
Return the first urban renewal vertex.

An urban renewal vertex is one in which there are two arrows pointing toward the vertex and two arrows pointing away.

EXAMPLES:

```
sage: G = ClusterSeed(['GR',[4,9]]); G.first_urban_renewal()
5
```

g_matrix(*show_warnings=True*)
Returns the matrix of all *g-vectors* of `self`. These are the degree vectors of the cluster variables after setting all y_i's to 0.

Warning: this method assumes the sign-coherence conjecture and that the input seed is sign-coherent (has an exchange matrix with columns of like signs). Otherwise, computational errors might arise.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
sage: S.g_matrix()
[ 1  0  0]
[ 0  0 -1]
[ 0 -1  0]

sage: S = ClusterSeed(['A',3])
```

```
sage: S.mutate([0,1])
sage: S.g_matrix()
[-1 -1  0]
[ 1  0  0]
[ 0  0  1]

sage: S = ClusterSeed(['A',4]); S.use_g_vectors(False); S.use_fpolys(False); S.g_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

sage: S = ClusterSeed(['A',4])
sage: S.use_g_vectors(False); S.use_c_vectors(False); S.use_fpolys(False); S.track_mutations
Traceback (most recent call last):
...
ValueError: Unable to calculate g-vectors. Need to use g vectors.
```

g_vector(*k*)

Returns the k-th *g-vector* of `self`. This is the degree vector of the k-th cluster variable after setting all y_i's to 0.

Warning: this method assumes the sign-coherence conjecture and that the input seed is sign-coherent (has an exchange matrix with columns of like signs). Otherwise, computational errors might arise.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1,2])
sage: [ S.g_vector(k) for k in range(3) ]
[(1, 0, 0), (0, 0, -1), (0, -1, 0)]
```

greedy(*a1*, *a2*, *algorithm='by_recursion'*)

Returns the greedy element $x[a_1, a_2]$ assuming that self is rank two.

The third input can be 'by_recursion', 'by_combinatorics', or 'just_numbers' to specify if the user wants the element computed by the recurrence, combinatorial formula, or wants to set x_1 and x_2 to be one.

See [LeeLiZe] for more details.

EXAMPLES:

```
sage: S = ClusterSeed(['R2', [3, 3]])
sage: S.greedy(4, 4)
(x0^12 + x1^12 + 4*x0^9 + 4*x1^9 + 6*x0^6 + 4*x0^3*x1^3 + 6*x1^6 + 4*x0^3 + 4*x1^3 + 1)/(x0^
sage: S.greedy(4, 4, 'by_combinatorics')
(x0^12 + x1^12 + 4*x0^9 + 4*x1^9 + 6*x0^6 + 4*x0^3*x1^3 + 6*x1^6 + 4*x0^3 + 4*x1^3 + 1)/(x0^
sage: S.greedy(4, 4, 'just_numbers')
35
sage: S = ClusterSeed(['R2', [2, 2]])
sage: S.greedy(1, 2)
(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2)
sage: S.greedy(1, 2, 'by_combinatorics')
(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2)
```

REFERENCES:

green_vertices()

Return the list of green vertices of `self`.

A vertex is defined to be green if its c-vector has all non-positive entries. More information on green vertices can be found at [BDP2013]

OUTPUT:

The green vertices as a list of integers.

EXAMPLES:

```
sage: ClusterSeed(['A',3]).principal_extension().green_vertices()
[0, 1, 2]

sage: ClusterSeed(['A',[3,3],1]).principal_extension().green_vertices()
[0, 1, 2, 3, 4, 5]
```

ground_field()

Returns the *ground field* of the cluster of `self`.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.ground_field()
Multivariate Polynomial Ring in x0, x1, x2, y0, y1, y2 over Rational Field
```

highest_degree_denominator(*filter=None*)

Return the vertex of the cluster polynomial with highest degree in the denominator.

INPUT:

- `filter` - Filter should be a list or iterable

OUTPUT:

Returns an integer.

EXAMPLES:

```
sage: B = matrix([[0,-1,0,-1,1,1],[1,0,1,0,-1,-1],[0,-1,0,-1,1,1],[1,0,1,0,-1,-1],[-1,1,-1,1
sage: C = ClusterSeed(B).principal_extension(); C.mutate([0,1,2,4,3,2,5,4,3])
sage: C.highest_degree_denominator()
5
```

interact(*fig_size=1*, *circular=True*)

Only in *notebook mode*. Starts an interactive window for cluster seed mutations.

INPUT:

- `fig_size` – (default: 1) factor by which the size of the plot is multiplied.
- `circular` – (default: True) if True, the circular plot is chosen, otherwise >>spring<< is used.

TESTS:

```
sage: S = ClusterSeed(['A',4])
sage: S.interact() # long time
'The interactive mode only runs in the Sage notebook.'
```

is_acyclic()

Returns True iff self is acyclic (i.e., if the underlying quiver is acyclic).

EXAMPLES:

```
sage: ClusterSeed(['A',4]).is_acyclic()
True
```

```
sage: ClusterSeed(['A',[2,1],1]).is_acyclic()
True

sage: ClusterSeed([[0,1],[1,2],[2,0]]).is_acyclic()
False
```

is_bipartite(*return_bipartition=False*)

Returns True iff self is bipartite (i.e., if the underlying quiver is bipartite).

INPUT:

- return_bipartition – (default:False) if True, the bipartition is returned in the case of `self` being bipartite.

EXAMPLES:

```
sage: ClusterSeed(['A',[3,3],1]).is_bipartite()
True

sage: ClusterSeed(['A',[4,3],1]).is_bipartite()
False
```

is_finite()

Returns True if `self` is of finite type.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.is_finite()
True

sage: S = ClusterSeed(['A',[2,2],1])
sage: S.is_finite()
False
```

is_mutation_finite(*nr_of_checks=None*, *return_path=False*)

Returns True if `self` is of finite mutation type.

INPUT:

- `nr_of_checks` – (default: None) number of mutations applied. Standard is 500*(number of vertices of self).
- `return_path` – (default: False) if True, in case of self not being mutation finite, a path from self to a quiver with an edge label (a,-b) and a*b > 4 is returned.

ALGORITHM:

- A cluster seed is mutation infinite if and only if every $b_{ij} * b_{ji} > -4$. Thus, we apply random mutations in random directions

WARNING:

- Uses a non-deterministic method by random mutations in various directions.
- In theory, it can return a wrong True.

EXAMPLES:

```
sage: S = ClusterSeed(['A',10])
sage: S._mutation_type = None
sage: S.is_mutation_finite()
True
```

```
sage: S = ClusterSeed([(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(2,9)])
sage: S.is_mutation_finite()
False
```

m()

Returns the number of *frozen variables* of `self`.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.n()
3

sage: S.m()
0

sage: S = S.principal_extension()
sage: S.m()
3
```

most_decreased_denominator_after_mutation()

Return the vertex that will produce the most decrease in denominator degrees after mutation

EXAMPLES:

```
sage: S = ClusterSeed(['A',5])
sage: S.mutate([0,2,3,1,2,3,1,2,0,2,3])
sage: S.most_decreased_denominator_after_mutation()
2
```

most_decreased_edge_after_mutation()

Return the vertex that will produce the least degrees after mutation

EXAMPLES:

```
sage: S = ClusterSeed(['A',5])
sage: S.mutate([0,2,3,1,2,3,1,2,0,2,3])
sage: S.most_decreased_edge_after_mutation()
2
```

mutate(*sequence*, *inplace=True*)

Mutates `self` at a vertex or a sequence of vertices.

INPUT:

- `sequence` – a vertex of `self`, an iterator of vertices of `self`, a function which takes in the ClusterSeed and returns a vertex or an iterator of vertices, or a string representing a type of vertices to mutate.
- `inplace` – (default: True) if False, the result is returned, otherwise `self` is modified.

Possible values for vertex types in `sequence` are:

- `"first_source"`: mutates at first found source vertex,
- `"sources"`: mutates at all sources,
- `"first_sink"`: mutates at first sink,
- `"sinks"`: mutates at all sink vertices,
- `"green"`: mutates at the first green vertex,

- "red": mutates at the first red vertex,
- "urban_renewal" or "urban": mutates at first urban renewal vertex,
- "all_urban_renewals" or "all_urban": mutates at all urban renewal vertices.

EXAMPLES:

```
sage: S = ClusterSeed(['A',4]); S.b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

sage: S.mutate(0); S.b_matrix()
[ 0 -1  0  0]
[ 1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

sage: T = S.mutate(0, inplace=False); T
A seed for a cluster algebra of rank 4 of type ['A', 4]

sage: S.mutate(0)
sage: S == T
True

sage: S.mutate([0,1,0])
sage: S.b_matrix()
[ 0 -1  1  0]
[ 1  0  0  0]
[-1  0  0  1]
[ 0  0 -1  0]

sage: S = ClusterSeed(QuiverMutationType([['A',1],['A',3]]))
sage: S.b_matrix()
[ 0  0  0  0]
[ 0  0  1  0]
[ 0 -1  0 -1]
[ 0  0  1  0]

sage: T = S.mutate(0,inplace=False)
sage: S == T
False

sage: Q = ClusterSeed(['A',3]);Q.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]

sage: Q.mutate('first_sink');Q.b_matrix()
[ 0 -1  0]
[ 1  0  1]
[ 0 -1  0]

sage: def last_vertex(self): return self._n - 1
sage: Q.mutate(last_vertex); Q.b_matrix()
[ 0 -1  0]
[ 1  0 -1]
[ 0  1  0]
```

```
sage: S = ClusterSeed(['A',4], user_labels=['a','b','c','d']);
sage: S.mutate('a'); S.mutate('(b+1)/a')
sage: S.cluster()
[a, b, c, d]

sage: S = ClusterSeed(['A',4], user_labels=['a','b','c']);
Traceback (most recent call last):
...
ValueError: The number of user-defined labels is not the number of exchangeable and frozen v

sage: S = ClusterSeed(['A',4],user_labels=['x','y','w','z'])
sage: S.mutate('x')
sage: S.cluster()
[(y + 1)/x, y, w, z]
sage: S.mutate('(y+1)/x')
sage: S.cluster()
[x, y, w, z]
sage: S.mutate('y')
sage: S.cluster()
[x, (x*w + 1)/y, w, z]
sage: S.mutate('(x*w+1)/y')
sage: S.cluster()
[x, y, w, z]

sage: S = ClusterSeed(['A',4], user_labels=[[1,2],[2,3],[4,5],[5,6]]);
sage: S.cluster()
[x_1_2, x_2_3, x_4_5, x_5_6]
sage: S.mutate('[1,2]');
sage: S.cluster()
[(x_2_3 + 1)/x_1_2, x_2_3, x_4_5, x_5_6]

sage: S = ClusterSeed(['A',4], user_labels=[[1,2],[2,3],[4,5],[5,6]],user_labels_prefix='P')
sage: S.cluster()
[P_1_2, P_2_3, P_4_5, P_5_6]
sage: S.mutate('[1,2]')
sage: S.cluster()
[(P_2_3 + 1)/P_1_2, P_2_3, P_4_5, P_5_6]
sage: S.mutate('P_4_5')
sage: S.cluster()
[(P_2_3 + 1)/P_1_2, P_2_3, (P_2_3*P_5_6 + 1)/P_4_5, P_5_6]

sage: S = ClusterSeed(['A',4])
sage: S.mutate([0,1,0,1,0,2,1])
sage: T = ClusterSeed(S)
sage: S.use_fpolys(False)
sage: S.use_g_vectors(False)
sage: S.use_c_vectors(False)
sage: S._C
sage: S._G
sage: S._F
sage: S.g_matrix()
[ 0 -1  0  0]
[ 1  1  1  0]
[ 0  0 -1  0]
[ 0  0  1  1]
sage: S.c_matrix()
[ 1 -1  0  0]
[ 1  0  0  0]
```

```
[ 1  0 -1  1]
[ 0  0  0  1]
sage: S.f_polynomials() == T.f_polynomials()
True

sage: S.cluster() == T.cluster()
True

sage: S._mut_path
[0, 1, 0, 1, 0, 2, 1]
```

mutation_analysis(*options=['all'], filter=None*)

Runs an analysis of all potential mutation options. Note that this might take a long time on large seeds.

Notes: Edges are only returned if we have a non-valued quiver. Green and red vertices are only returned if the cluster is principal.

INPUT:

- `options` – (default: ['all']) a list of mutation options.
- `filter` – (default: None) A vertex or interval of vertices to limit our search to

Possible options are:

- `"all"` - All options below
- `"edges"` - Number of edges (works with skew-symmetric quivers)
- `"edge_diff"` - Edges added/deleted (works with skew-symmetric quivers)
- `"green_vertices"` - List of green vertices (works with principals)
- `"green_vertices_diff"` - Green vertices added/removed (works with principals)
- `"red_vertices"` - List of red vertices (works with principals)
- `"red_vertices_diff"` - Red vertices added/removed (works with principals)
- `"urban_renewals"` - List of urban renewal vertices
- `"urban_renewals_diff"` - Urban renewal vertices added/removed
- `"sources"` - List of source vertices
- `"sources_diff"` - Source vertices added/removed
- `"sinks"` - List of sink vertices
- `"sinks_diff"` - Sink vertices added/removed
- `"denominators"` - List of all denominators of the cluster variables

OUTPUT:

Outputs a dictionary indexed by the vertex numbers. Each vertex will itself also be a dictionary with each desired option included as a key in the dictionary. As an example you would get something similar to: {0: {'edges': 1}, 1: {'edges': 2}}. This represents that if you were to do a mutation at the current seed then mutating at vertex 0 would result in a quiver with 1 edge and mutating at vertex 0 would result in a quiver with 2 edges.

EXAMPLES:

```
sage: B = [[0, 4, 0, -1],[-4,0, 3, 0],[0, -3, 0, 1],[1, 0, -1, 0]]
sage: S = ClusterSeed(matrix(B)); S.mutate([2,3,1,2,1,3,0,2])
sage: S.mutation_analysis()
```

```
{0: {'d_matrix': [ 0  0  1  0]
  [ 0 -1  0  0]
  [ 0  0  0 -1]
  [-1  0  0  0],
  'denominators': [1, 1, x0, 1],
  'edge_diff': 6,
  'edges': 13,
  'green_vertices': [0, 1, 3],
  'green_vertices_diff': {'added': [0], 'removed': []},
  'red_vertices': [2],
  'red_vertices_diff': {'added': [], 'removed': [0]},
  'sinks': [],
  'sinks_diff': {'added': [], 'removed': [2]},
  'sources': [],
  'sources_diff': {'added': [], 'removed': []},
  'urban_renewals': [],
  'urban_renewals_diff': {'added': [], 'removed': []}},
 1: {'d_matrix': [ 1  4  1  0]
  [ 0  1  0  0]
  [ 0  0  0 -1]
  [ 1  4  0  0],
  'denominators': [x0*x3, x0^4*x1*x3^4, x0, 1],
  'edge_diff': 2,
  'edges': 9,
  'green_vertices': [0, 3],
  'green_vertices_diff': {'added': [0], 'removed': [1]},
  'red_vertices': [1, 2],
  'red_vertices_diff': {'added': [1], 'removed': [0]},
  'sinks': [2],
  'sinks_diff': {'added': [], 'removed': []},
  'sources': [],
  'sources_diff': {'added': [], 'removed': []},
  'urban_renewals': [],
  'urban_renewals_diff': {'added': [], 'removed': []}},
 2: {'d_matrix': [ 1  0  0  0]
  [ 0 -1  0  0]
  [ 0  0  0 -1]
  [ 1  0  1  0],
  'denominators': [x0*x3, 1, x3, 1],
  'edge_diff': 0,
  'edges': 7,
  'green_vertices': [1, 2, 3],
  'green_vertices_diff': {'added': [2], 'removed': []},
  'red_vertices': [0],
  'red_vertices_diff': {'added': [], 'removed': [2]},
  'sinks': [],
  'sinks_diff': {'added': [], 'removed': [2]},
  'sources': [2],
  'sources_diff': {'added': [2], 'removed': []},
  'urban_renewals': [],
  'urban_renewals_diff': {'added': [], 'removed': []}},
 3: {'d_matrix': [ 1  0  1  1]
  [ 0 -1  0  0]
  [ 0  0  0  1]
  [ 1  0  0  1],
  'denominators': [x0*x3, 1, x0, x0*x2*x3],
  'edge_diff': -1,
  'edges': 6,
```

```
  'green_vertices': [1],
  'green_vertices_diff': {'added': [], 'removed': [3]},
  'red_vertices': [0, 2, 3],
  'red_vertices_diff': {'added': [3], 'removed': []},
  'sinks': [2],
  'sinks_diff': {'added': [], 'removed': []},
  'sources': [1],
  'sources_diff': {'added': [1], 'removed': []},
  'urban_renewals': [],
  'urban_renewals_diff': {'added': [], 'removed': []}}}

sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutation_analysis()
{0: {'d_matrix': [ 1  0  0]
  [ 0 -1  0]
  [ 0  0 -1],
  'denominators': [x0, 1, 1],
  'green_vertices': [1, 2],
  'green_vertices_diff': {'added': [], 'removed': [0]},
  'red_vertices': [0],
  'red_vertices_diff': {'added': [0], 'removed': []},
  'sinks': [],
  'sinks_diff': {'added': [], 'removed': [1]},
  'sources': [4, 5],
  'sources_diff': {'added': [], 'removed': [3]},
  'urban_renewals': [],
  'urban_renewals_diff': {'added': [], 'removed': []}},
 1: {'d_matrix': [-1  0  0]
  [ 0  1  0]
  [ 0  0 -1],
  'denominators': [1, x1, 1],
  'green_vertices': [0, 2],
  'green_vertices_diff': {'added': [], 'removed': [1]},
  'red_vertices': [1],
  'red_vertices_diff': {'added': [1], 'removed': []},
  'sinks': [0, 2, 4],
  'sinks_diff': {'added': [0, 2, 4], 'removed': [1]},
  'sources': [1, 3, 5],
  'sources_diff': {'added': [1], 'removed': [4]},
  'urban_renewals': [],
  'urban_renewals_diff': {'added': [], 'removed': []}},
 2: {'d_matrix': [-1  0  0]
  [ 0 -1  0]
  [ 0  0  1],
  'denominators': [1, 1, x2],
  'green_vertices': [0, 1],
  'green_vertices_diff': {'added': [], 'removed': [2]},
  'red_vertices': [2],
  'red_vertices_diff': {'added': [2], 'removed': []},
  'sinks': [],
  'sinks_diff': {'added': [], 'removed': [1]},
  'sources': [3, 4],
  'sources_diff': {'added': [], 'removed': [5]},
  'urban_renewals': [],
  'urban_renewals_diff': {'added': [], 'removed': []}}}
```

mutation_class(*depth=+Infinity*, *show_depth=False*, *return_paths=False*, *up_to_equivalence=True*, *only_sink_source=False*)

Returns the mutation class of `self` with respect to certain constraints.

INPUT:

- `depth` – (default: infinity) integer, only seeds with distance at most depth from self are returned.
- `show_depth` – (default: False) if True, the actual depth of the mutation is shown.
- `return_paths` – (default: False) if True, a shortest path of mutation sequences from self to the given quiver is returned as well.
- `up_to_equivalence` – (default: True) if True, only seeds up to equivalence are considered.
- `sink_source` – (default: False) if True, only mutations at sinks and sources are applied.

EXAMPLES:

- for examples see `mutation_class_iter()`

TESTS:

```
sage: A = ClusterSeed(['A',3]).mutation_class()
```

mutation_class_iter(*depth=+Infinity*, *show_depth=False*, *return_paths=False*, *up_to_equivalence=True*, *only_sink_source=False*)

Returns an iterator for the mutation class of `self` with respect to certain constrains.

INPUT:

- `depth` – (default: infinity) integer or infinity, only seeds with distance at most `depth` from `self` are returned.
- `show_depth` – (default: False) if True, the current depth of the mutation is shown while computing.
- `return_paths` – (default: False) if True, a shortest path of mutations from `self` to the given quiver is returned as well.
- `up_to_equivalence` – (default: True) if True, only one seed up to simultaneous permutation of rows and columns of the exchange matrix is recorded.
- `sink_source` – (default: False) if True, only mutations at sinks and sources are applied.

EXAMPLES:

A standard finite type example:

```
sage: S = ClusterSeed(['A',3])
sage: it = S.mutation_class_iter()
sage: for T in it: print T
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
```

A finite type example with given depth:

```
sage: it = S.mutation_class_iter(depth=1)
sage: for T in it: print T
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
A seed for a cluster algebra of rank 3 of type ['A', 3]
```

A finite type example where the depth is shown while computing:

```
sage: it = S.mutation_class_iter(show_depth=True)
sage: for T in it: pass
Depth: 0     found: 1          Time: ... s
Depth: 1     found: 4          Time: ... s
Depth: 2     found: 9          Time: ... s
Depth: 3     found: 13         Time: ... s
Depth: 4     found: 14         Time: ... s
```

A finite type example with shortest paths returned:

```
sage: it = S.mutation_class_iter(return_paths=True)
sage: for T in it: print T
(A seed for a cluster algebra of rank 3 of type ['A', 3], [])
(A seed for a cluster algebra of rank 3 of type ['A', 3], [2])
(A seed for a cluster algebra of rank 3 of type ['A', 3], [1])
(A seed for a cluster algebra of rank 3 of type ['A', 3], [0])
(A seed for a cluster algebra of rank 3 of type ['A', 3], [2, 1])
(A seed for a cluster algebra of rank 3 of type ['A', 3], [0, 2])
(A seed for a cluster algebra of rank 3 of type ['A', 3], [0, 1])
(A seed for a cluster algebra of rank 3 of type ['A', 3], [1, 2])
(A seed for a cluster algebra of rank 3 of type ['A', 3], [1, 0])
(A seed for a cluster algebra of rank 3 of type ['A', 3], [0, 2, 1])
(A seed for a cluster algebra of rank 3 of type ['A', 3], [0, 1, 2])
(A seed for a cluster algebra of rank 3 of type ['A', 3], [2, 1, 0])
(A seed for a cluster algebra of rank 3 of type ['A', 3], [1, 0, 2])
(A seed for a cluster algebra of rank 3 of type ['A', 3], [0, 1, 2, 0])
```

Finite type examples not considered up to equivalence:

```
sage: it = S.mutation_class_iter(up_to_equivalence=False)
sage: len( [ T for T in it ] )
84

sage: it = ClusterSeed(['A',2]).mutation_class_iter(return_paths=True,up_to_equivalence=Fals
sage: for T in it: print T
(A seed for a cluster algebra of rank 2 of type ['A', 2], [])
(A seed for a cluster algebra of rank 2 of type ['A', 2], [1])
(A seed for a cluster algebra of rank 2 of type ['A', 2], [0])
(A seed for a cluster algebra of rank 2 of type ['A', 2], [0, 1])
(A seed for a cluster algebra of rank 2 of type ['A', 2], [1, 0])
(A seed for a cluster algebra of rank 2 of type ['A', 2], [1, 0, 1])
(A seed for a cluster algebra of rank 2 of type ['A', 2], [0, 1, 0])
(A seed for a cluster algebra of rank 2 of type ['A', 2], [1, 0, 1, 0])
(A seed for a cluster algebra of rank 2 of type ['A', 2], [0, 1, 0, 1])
(A seed for a cluster algebra of rank 2 of type ['A', 2], [1, 0, 1, 0, 1])
```

Check that trac ticket #14638 is fixed:

```
sage: S = ClusterSeed(['E',6])
sage: MC = S.mutation_class(depth=7); len(MC)
534
```

Infinite type examples:

```
sage: S = ClusterSeed(['A',[1,1],1])
sage: it = S.mutation_class_iter()
sage: next(it)
A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1]
sage: next(it)
A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1]
sage: next(it)
A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1]
sage: next(it)
A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1]

sage: it = S.mutation_class_iter(depth=3, return_paths=True)
sage: for T in it: print T
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [1])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [0])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [1, 0])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [0, 1])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [1, 0, 1])
(A seed for a cluster algebra of rank 2 of type ['A', [1, 1], 1], [0, 1, 0])
```

mutation_sequence(*sequence*, *show_sequence=False*, *fig_size=1.2*, *return_output='seed'*)

Returns the seeds obtained by mutating `self` at all vertices in `sequence`.

INPUT:

- `sequence` – an iterable of vertices of self.
- `show_sequence` – (default: False) if True, a png containing the associated quivers is shown.
- `fig_size` – (default: 1.2) factor by which the size of the plot is multiplied.
- `return_output` – (default: 'seed') determines what output is to be returned:

```
* if 'seed', outputs all the cluster seeds obtained by the ``sequence`` of mutations.
* if 'matrix', outputs a list of exchange matrices.
* if 'var', outputs a list of new cluster variables obtained at each step.
```

EXAMPLES:

```
sage: S = ClusterSeed(['A',2])
sage: for T in S.mutation_sequence([0,1,0]):
...       print T.b_matrix()
[ 0 -1]
[ 1  0]
[ 0  1]
[-1  0]
[ 0 -1]
[ 1  0]

sage: S=ClusterSeed(['A',2])
sage: S.mutation_sequence([0,1,0,1],return_output='var')
[(x1 + 1)/x0, (x0 + x1 + 1)/(x0*x1), (x0 + 1)/x1, x0]
```

mutation_type()

Returns the mutation_type of each connected component of `self`, if it can be determined. Otherwise, the

mutation type of this component is set to be unknown.

The mutation types of the components are ordered by vertex labels.

WARNING:

- All finite types can be detected,
- All affine types can be detected, EXCEPT affine type D (the algorithm is not yet implemented)
- All exceptional types can be detected.
- Might fail to work if it is used within different Sage processes simultaneously (that happened in the doctesting).

EXAMPLES:

- finite types:

```
sage: S = ClusterSeed(['A',5])
sage: S._mutation_type = S._quiver._mutation_type = None
sage: S.mutation_type()
['A', 5]

sage: S = ClusterSeed([(0,1),(1,2),(2,3),(3,4)])
sage: S.mutation_type()
['A', 5]
```

- affine types:

```
sage: S = ClusterSeed(['E',8,[1,1]]); S
A seed for a cluster algebra of rank 10 of type ['E', 8, [1, 1]]
sage: S._mutation_type = S._quiver._mutation_type = None; S
A seed for a cluster algebra of rank 10
sage: S.mutation_type() # long time
['E', 8, [1, 1]]
```

- the not yet working affine type D:

```
sage: S = ClusterSeed(['D',4,1])
sage: S._mutation_type = S._quiver._mutation_type = None
sage: S.mutation_type() # todo: not implemented
['D', 4, 1]
```

- the exceptional types:

```
sage: S = ClusterSeed(['X',6])
sage: S._mutation_type = S._quiver._mutation_type = None
sage: S.mutation_type() # long time
['X', 6]
```

- infinite types:

```
sage: S = ClusterSeed(['GR',[4,9]])
sage: S._mutation_type = S._quiver._mutation_type = None
sage: S.mutation_type()
'undetermined infinite mutation type'
```

mutations()

Returns the list of mutations `self` has undergone if they are being tracked.

Examples:

```
sage: S = ClusterSeed(['A',3])
sage: S.mutations()
[]

sage: S.mutate([0,1,0,2])
sage: S.mutations()
[0, 1, 0, 2]

sage: S.track_mutations(False)
sage: S.mutations()
Traceback (most recent call last):
...
ValueError: Not recording mutation sequence.  Need to track mutations.
```

n()

Returns the number of *exchangeable variables* of `self`.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.n()
3
```

oriented_exchange_graph()

Return the oriented exchange graph of `self` as a directed graph.

The seed must be a cluster seed for a cluster algebra of finite type with principal coefficients (the corresponding quiver must have mutable vertices 0,1,...,n-1).

EXAMPLES:

```
sage: S = ClusterSeed(['A', 2]).principal_extension()
sage: G = S.oriented_exchange_graph(); G
Digraph on 5 vertices
sage: G.out_degree_sequence()
[2, 1, 1, 1, 0]

sage: S = ClusterSeed(['B', 2]).principal_extension()
sage: G = S.oriented_exchange_graph(); G
Digraph on 6 vertices
sage: G.out_degree_sequence()
[2, 1, 1, 1, 1, 0]
```

TESTS:

```
sage: S = ClusterSeed(['A',[2,2],1])
sage: S.oriented_exchange_graph()
Traceback (most recent call last):
...
TypeError: only works for finite mutation type
```

```
sage: S = ClusterSeed(['A', 2])
sage: S.oriented_exchange_graph()
Traceback (most recent call last):
...
TypeError: only works for principal coefficients
```

plot(*circular=False*, *mark=None*, *save_pos=False*, *force_c=False*, *with_greens=False*, *add_labels=False*)

Returns the plot of the quiver of `self`.

INPUT:

- `circular` – (default:False) if True, the circular plot is chosen, otherwise >>spring<< is used.
- `mark` – (default: None) if set to i, the vertex i is highlighted.
- `save_pos` – (default:False) if True, the positions of the vertices are saved.
- `force_c` – (default:False) if True, will show the frozen vertices even if they were never initialized
- `with_greens` – (default:False) if True, will display the green vertices in green
- `add_labels` – (default:False) if True, will use the initial variables as labels

EXAMPLES:

```
sage: S = ClusterSeed(['A',5])
sage: pl = S.plot()
sage: pl = S.plot(circular=True)
```

principal_extension()

Returns the principal extension of self, yielding a 2n-by-n matrix. Raises an error if the input seed has a non-square exchange matrix. In this case, the method instead adds n frozen variables to any previously frozen variables. I.e., the seed obtained by adding a frozen variable to every exchangeable variable of `self`.

EXAMPLES:

```
sage: S = ClusterSeed([[0,1],[1,2],[2,3],[2,4]]); S
A seed for a cluster algebra of rank 5

sage: T = S.principal_extension(); T
A seed for a cluster algebra of rank 5 with principal coefficients

sage: T.b_matrix()
[ 0  1  0  0  0]
[-1  0  1  0  0]
[ 0 -1  0  1  1]
[ 0  0 -1  0  0]
[ 0  0 -1  0  0]
[ 1  0  0  0  0]
[ 0  1  0  0  0]
[ 0  0  1  0  0]
[ 0  0  0  1  0]
[ 0  0  0  0  1]

sage: S = ClusterSeed(['A',4],user_labels=['a','b','c','d'])
sage: T= S.principal_extension()
sage: T.cluster()
[a, b, c, d]
sage: T.coefficients()
```

```
[y0, y1, y2, y3]
sage: S2 = ClusterSeed(['A',4],user_labels={0:'a',1:'b',2:'c',3:'d'})
sage: S2 == S
True
sage: T2 = S2.principal_extension()
sage: T2 == T
True
```

quiver()

Returns the *quiver* associated to `self`.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.quiver()
Quiver on 3 vertices of type ['A', 3]
```

red_vertices()

Return the list of red vertices of `self`.

A vertex is defined to be red if its c-vector has all non-negative entries. More information on red vertices can be found at [BDP2013].

OUTPUT:

The red vertices as a list of integers.

EXAMPLES:

```
sage: ClusterSeed(['A',3]).principal_extension().red_vertices()
[]

sage: ClusterSeed(['A',[3,3],1]).principal_extension().red_vertices()
[]

sage: Q = ClusterSeed(['A',[3,3],1]).principal_extension();
sage: Q.mutate(1);
sage: Q.red_vertices()
[1]
```

reorient(*data*)

Reorients `self` with respect to the given total order, or with respect to an iterator of ordered pairs.

WARNING:

- This operation might change the mutation type of `self`.
- Ignores ordered pairs (i, j) for which neither (i, j) nor (j, i) is an edge of `self`.

INPUT:

- `data` – an iterator defining a total order on `self.vertices()`, or an iterator of ordered pairs in `self` defining the new orientation of these edges.

EXAMPLES:

```
sage: S = ClusterSeed(['A',[2,3],1])
sage: S.mutation_type()
['A', [2, 3], 1]

sage: S.reorient([(0,1),(2,3)])
sage: S.mutation_type()
['D', 5]
```

```
sage: S.reorient([(1,0),(2,3)])
sage: S.mutation_type()
['A', [1, 4], 1]

sage: S.reorient([0,1,2,3,4])
sage: S.mutation_type()
['A', [1, 4], 1]
```

reset_cluster()

Resets the cluster of `self` to the initial cluster.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.mutate([1,2,1])
sage: S.cluster()
[x0, (x1 + 1)/x2, (x0*x2 + x1 + 1)/(x1*x2)]

sage: S.reset_cluster()
sage: S.cluster()
[x0, x1, x2]

sage: T = S.principal_extension()
sage: T.cluster()
[x0, x1, x2]
sage: T.mutate([1,2,1])
sage: T.cluster()
[x0, (x1*y2 + x0)/x2, (x1*y1*y2 + x0*y1 + x2)/(x1*x2)]

sage: T.reset_cluster()
sage: T.cluster()
[x0, x1, x2]

sage: S = ClusterSeed(['B',3],user_labels=[[1,2],[2,3],[3,4]],user_labels_prefix='p')
sage: S.mutate([0,1])
sage: S.cluster()
[(p_2_3 + 1)/p_1_2, (p_1_2*p_3_4^2 + p_2_3 + 1)/(p_1_2*p_2_3), p_3_4]

sage: S.reset_cluster()
sage: S.cluster()
[p_1_2, p_2_3, p_3_4]
sage: S.g_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
sage: S.f_polynomials()
[1, 1, 1]
```

reset_coefficients()

Resets the coefficients of `self` to the frozen variables but keeps the current cluster. Raises an error if the number of frozen variables is different than the number of exchangeable variables.

WARNING: This command to be phased out since 'use_c_vectors() does this more effectively.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.b_matrix()
[ 0  1  0]
```

```
[-1  0 -1]
[ 0  1  0]
[ 1  0  0]
[ 0  1  0]
[ 0  0  1]
sage: S.mutate([1,2,1])
sage: S.b_matrix()
[ 0  1 -1]
[-1  0  1]
[ 1 -1  0]
[ 1  0  0]
[ 0  1 -1]
[ 0  0 -1]
sage: S.reset_coefficients()
sage: S.b_matrix()
[ 0  1 -1]
[-1  0  1]
[ 1 -1  0]
[ 1  0  0]
[ 0  1  0]
[ 0  0  1]
```

save_image(*filename*, *circular=False*, *mark=None*, *save_pos=False*)

Saves the plot of the underlying digraph of the quiver of `self`.

INPUT:

- `filename` – the filename the image is saved to.
- `circular` – (default: False) if True, the circular plot is chosen, otherwise >>spring<< is used.
- `mark` – (default: None) if set to i, the vertex i is highlighted.
- `save_pos` – (default:False) if True, the positions of the vertices are saved.

EXAMPLES:

```
sage: S = ClusterSeed(['F',4,[1,2]])
sage: S.save_image(os.path.join(SAGE_TMP, 'sage.png'))
```

set_c_matrix(*data*)

Will force set the c matrix according to a matrix, a quiver, or a seed.

INPUT:

- `data` – The matrix to set the c matrix to. Also allowed to be a quiver or cluster seed, in which case the b_matrix is used.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]);
sage: X = matrix([[0,0,1],[0,1,0],[1,0,0]])
sage: S.set_c_matrix(X)
sage: S.c_matrix()
[0 0 1]
[0 1 0]
[1 0 0]

sage: Y = matrix([[-1,0,1],[0,1,0],[1,0,0]])
sage: S.set_c_matrix(Y)
C matrix does not look to be valid - there exists a column containing positive and negative
Continuing...
```

```
sage: Z = matrix([[1,0,1],[0,1,0],[2,0,2]])
sage: S.set_c_matrix(Z)
C matrix does not look to be valid - not a linearly independent set.
Continuing...
```

set_cluster(*cluster*, *force=False*)

Sets the cluster for `self` to `cluster`.

Warning: Initialization may lead to inconsistent data.

INPUT:

- `cluster` – an iterable defining a cluster for `self`.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: cluster = S.cluster()
sage: S.mutate([1,2,1])
sage: S.cluster()
[x0, (x1 + 1)/x2, (x0*x2 + x1 + 1)/(x1*x2)]
sage: cluster2 = S.cluster()

sage: S.set_cluster(cluster)
Warning: using set_cluster at this point could lead to inconsistent seed data.

sage: S.set_cluster(cluster, force=True)
sage: S.cluster()
[x0, x1, x2]
sage: S.set_cluster(cluster2, force=True)
sage: S.cluster()
[x0, (x1 + 1)/x2, (x0*x2 + x1 + 1)/(x1*x2)]

sage: S = ClusterSeed(['A',3]); S.use_fpolys(False)
sage: S.set_cluster([1,1,1])
Warning: clusters not being tracked so this command is ignored.
```

show(*fig_size=1*, *circular=False*, *mark=None*, *save_pos=False*, *force_c=False*, *with_greens=False*, *add_labels=False*)

Shows the plot of the quiver of `self`.

INPUT:

- `fig_size` – (default: 1) factor by which the size of the plot is multiplied.
- `circular` – (default: False) if True, the circular plot is chosen, otherwise >>spring<< is used.
- `mark` – (default: None) if set to i, the vertex i is highlighted.
- `save_pos` – (default:False) if True, the positions of the vertices are saved.
- `force_c` – (default:False) if True, will show the frozen vertices even if they were never initialized
- `with_greens` – (default:False) if True, will display the green vertices in green
- `add_labels` – (default:False) if True, will use the initial variables as labels

TESTS:

```
sage: S = ClusterSeed(['A',5])
sage: S.show() # long time
```

smallest_c_vector()
Return the vertex with the smallest c vector

OUTPUT:

Returns an integer.

EXAMPLES:: sage: B = matrix([[0,2],[-2,0]]) sage: C = ClusterSeed(B).principal_extension(); sage: C.mutate(0) sage: C.smallest_c_vector() 0

track_mutations(*use=True*)
Begins tracking the mutation path.

Warning: May initialize all other data to ensure that all c, d, and g vectors agree on the start of mutations.

INPUT:

- use – (default:True) If True, will begin filling the mutation path

EXAMPLES:

```
sage: S = ClusterSeed(['A',4]); S.track_mutations(False)
sage: S.mutate(0)
sage: S.mutations()
Traceback (most recent call last):
...
ValueError: Not recording mutation sequence.  Need to track mutations.
sage: S.track_mutations(True)
sage: S.g_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

sage: S.mutate([0,1])
sage: S.mutations()
[0, 1]
```

universal_extension()
Returns the universal extension of self.

This is the initial seed of the associated cluster algebra with universal coefficients, as defined in section 12 of Arxiv math/0602259.

This method works only if self is a bipartite, finite-type seed.

Due to some limitations in the current implementation of CartanType, we need to construct the set of almost positive coroots by hand. As a consequence their ordering is not the standard one (the rows of the bottom part of the exchange matrix might be a shuffling of those you would expect).

EXAMPLES:

```
sage: S = ClusterSeed(['A',2])
sage: T = S.universal_extension()
sage: T.b_matrix()
[ 0  1]
[-1  0]
[-1  0]
[ 1  0]
[ 1 -1]
[ 0  1]
[ 0 -1]
```

```
sage: S = ClusterSeed(['A',3])
sage: T = S.universal_extension()
sage: T.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
[-1  0  0]
[ 1  0  0]
[ 1 -1  0]
[ 1 -1  1]
[ 0  1  0]
[ 0 -1  0]
[ 0 -1  1]
[ 0  0 -1]
[ 0  0  1]

sage: S = ClusterSeed(['B',2])
sage: T = S.universal_extension()
sage: T.b_matrix()
[ 0  1]
[-2  0]
[-1  0]
[ 1  0]
[ 1 -1]
[ 2 -1]
[ 0  1]
[ 0 -1]
```

urban_renewals(*return_first=False*)

Return the list of the urban renewal vertices of `self`.

An urban renewal vertex is one in which there are two arrows pointing toward the vertex and two arrows pointing away.

INPUT:

- `return_first` – (default:False) if True, will return the first urban renewal

OUTPUT:

A list of vertices (as integers)

EXAMPLES:

```
sage: G = ClusterSeed(['GR',[4,9]]); G.urban_renewals()
[5, 6]
```

use_c_vectors(*use=True*, *bot_is_c=False*, *force=False*)

Reconstruct c vectors from other data or initialize if no usable data exists.

Warning: Initialization may lead to inconsistent data.

INPUT:

- `use` – (default:True) If True, will use c vectors
- `bot_is_c` – (default:False) If True and ClusterSeed self has self._m == self._n, then will assume bottom half of the extended exchange matrix is the c-matrix. If true, lets the ClusterSeed know c-vectors can be calculated.

EXAMPLES:

```
sage: S = ClusterSeed(['A',4]);
sage: S.use_c_vectors(False); S.use_g_vectors(False); S.use_fpolys(False); S.track_mutations
sage: S.use_c_vectors(True)
Warning: Initializing c-vectors at this point could lead to inconsistent seed data.

sage: S.use_c_vectors(True, force=True)
sage: S.c_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

sage: S = ClusterSeed(['A',4]);
sage: S.use_c_vectors(False); S.use_g_vectors(False); S.use_fpolys(False); S.track_mutations
sage: S.mutate(1);
sage: S.use_c_vectors(True, force=True)
sage: S.c_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

use_d_vectors(*use=True*, *force=False*)

Reconstruct d vectors from other data or initialize if no usable data exists.

Warning: Initialization may lead to inconsistent data.

INPUT:

- use – (default:True) If True, will use d vectors

EXAMPLES:

```
sage: S = ClusterSeed(['A',4]);
sage: S.use_d_vectors(True)
sage: S.d_matrix()
[-1  0  0  0]
[ 0 -1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]

sage: S = ClusterSeed(['A',4]); S.use_d_vectors(False); S.track_mutations(False); S.mutate(1
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]
sage: S.use_fpolys(False)
sage: S.d_matrix()
Traceback (most recent call last):
...
ValueError: Unable to calculate d-vectors. Need to use d vectors.

sage: S = ClusterSeed(['A',4]); S.use_d_vectors(False); S.track_mutations(False); S.mutate(1
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]
sage: S.use_fpolys(False)
sage: S.use_d_vectors(True)
Warning: Initializing d-vectors at this point could lead to inconsistent seed data.
```

```
sage: S.use_d_vectors(True, force=True)
sage: S.d_matrix()
[-1  0  0  0]
[ 0 -1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]

sage: S = ClusterSeed(['A',4]); S.mutate(1); S.d_matrix()
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]
sage: S = ClusterSeed(['A',4]); S.use_d_vectors(True); S.mutate(1); S.d_matrix()
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]
```

use_fpolys(*use=True*, *user_labels=None*, *user_labels_prefix=None*)

Use F-polynomials in our Cluster Seed

Note: This will automatically try to recompute the cluster variables if possible

INPUT:

- `use` – (default:True) If True, will use F-polynomials
- `user_labels` – (default:None) If set will overwrite the default cluster variable labels
- `user_labels_prefix` – (default:None) If set will overwrite the default

EXAMPLES:

```
sage: S = ClusterSeed(['A',4]); S.use_fpolys(False); S._cluster
sage: S.use_fpolys(True)
sage: S.cluster()
[x0, x1, x2, x3]

sage: S = ClusterSeed(['A',4]); S.use_fpolys(False); S.track_mutations(False); S.mutate(1)
sage: S.use_fpolys(True)
Traceback (most recent call last):
...
ValueError: F-polynomials and Cluster Variables cannot be reconstructed from given data.
sage: S.cluster()
Traceback (most recent call last):
...
ValueError: Clusters not being tracked
```

use_g_vectors(*use=True*, *force=False*)

Reconstruct g vectors from other data or initialize if no usable data exists.

Warning: Initialization may lead to inconsistent data.

INPUT:

- `use` – (default:True) If True, will use g vectors

EXAMPLES:

```
sage: S = ClusterSeed(['A',4]);
sage: S.use_g_vectors(False); S.use_fpolys(False)
sage: S.use_g_vectors(True)
```

```
sage: S.g_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

sage: S = ClusterSeed(['A',4]);
sage: S.use_g_vectors(False); S.use_fpolys(False)
sage: S.mutate(1);
sage: S.use_g_vectors(True)
sage: S.g_matrix()
[ 1  0  0  0]
[ 0 -1  0  0]
[ 0  0  1  0]
[ 0  0  0  1]

sage: S = ClusterSeed(['A',4]);
sage: S.use_g_vectors(False); S.use_fpolys(False); S.track_mutations(False)
sage: S.mutate(1);
sage: S.use_c_vectors(False)
sage: S.g_matrix()
Traceback (most recent call last):
...
ValueError: Unable to calculate g-vectors. Need to use g vectors.

sage: S = ClusterSeed(['A',4]);
sage: S.use_g_vectors(False); S.use_fpolys(False); S.track_mutations(False)
sage: S.mutate(1);
sage: S.use_c_vectors(False)
sage: S.use_g_vectors(True)
Warning: Initializing g-vectors at this point could lead to inconsistent seed data.

sage: S.use_g_vectors(True, force=True)
sage: S.g_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

variable_class(*depth=+Infinity*, *ignore_bipartite_belt=False*)

Returns all cluster variables in the mutation class of `self`.

INPUT:

- `depth` – (default:infinity) integer, only seeds with distance at most depth from self are returned
- `ignore_bipartite_belt` – (default:False) if True, the algorithms does not use the bipartite belt

EXAMPLES:

- for examples see `variable_class_iter()`

TESTS:

```
sage: A = ClusterSeed(['A',3]).variable_class()
```

variable_class_iter(*depth=+Infinity*, *ignore_bipartite_belt=False*)

Returns an iterator for all cluster variables in the mutation class of `self`.

INPUT:

- depth – (default:infinity) integer, only seeds with distance at most depth from self are returned
- ignore_bipartite_belt – (default:False) if True, the algorithms does not use the bipartite belt

EXAMPLES:

A standard finite type example:

```
sage: S = ClusterSeed(['A',3])
sage: it = S.variable_class_iter()
sage: for T in it: print T
x0
x1
x2
(x1 + 1)/x0
(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2)
(x1 + 1)/x2
(x0*x2 + x1 + 1)/(x0*x1)
(x0*x2 + 1)/x1
(x0*x2 + x1 + 1)/(x1*x2)
```

Finite type examples with given depth:

```
sage: it = S.variable_class_iter(depth=1)
sage: for T in it: print T
Found a bipartite seed - restarting the depth counter at zero and constructing the variable
x0
x1
x2
(x1 + 1)/x0
(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2)
(x1 + 1)/x2
(x0*x2 + x1 + 1)/(x0*x1)
(x0*x2 + 1)/x1
(x0*x2 + x1 + 1)/(x1*x2)
```

Note that the notion of *depth* depends on whether a bipartite seed is found or not, or if it is manually ignored:

```
sage: it = S.variable_class_iter(depth=1,ignore_bipartite_belt=True)
sage: for T in it: print T
x0
x1
x2
(x1 + 1)/x2
(x0*x2 + 1)/x1
(x1 + 1)/x0

sage: S.mutate([0,1])
sage: it2 = S.variable_class_iter(depth=1)
sage: for T in it2: print T
(x1 + 1)/x0
(x0*x2 + x1 + 1)/(x0*x1)
x2
(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2)
x1
(x0*x2 + 1)/x1
```

Infinite type examples:

```
sage: S = ClusterSeed(['A',[1,1],1])
sage: it = S.variable_class_iter(depth=2)
sage: for T in it: print T
Found a bipartite seed - restarting the depth counter at zero and constructing the variable
x0
x1
(x1^2 + 1)/x0
(x1^4 + x0^2 + 2*x1^2 + 1)/(x0^2*x1)
(x0^4 + 2*x0^2 + x1^2 + 1)/(x0*x1^2)
(x0^2 + 1)/x1
(x1^6 + x0^4 + 2*x0^2*x1^2 + 3*x1^4 + 2*x0^2 + 3*x1^2 + 1)/(x0^3*x1^2)
(x1^8 + x0^6 + 2*x0^4*x1^2 + 3*x0^2*x1^4 + 4*x1^6 + 3*x0^4 + 6*x0^2*x1^2 + 6*x1^4 + 3*x0^2 +
(x0^8 + 4*x0^6 + 3*x0^4*x1^2 + 2*x0^2*x1^4 + x1^6 + 6*x0^4 + 6*x0^2*x1^2 + 3*x1^4 + 4*x0^2 +
(x0^6 + 3*x0^4 + 2*x0^2*x1^2 + x1^4 + 3*x0^2 + 2*x1^2 + 1)/(x0^2*x1^3)
```

x (k)

Returns the k *-th initial cluster variable* for the associated cluster seed.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: S.mutate([2,1])
sage: S.x(0)
x0

sage: S.x(1)
x1

sage: S.x(2)
x2
```

y (k)

Returns the k *-th initial coefficient (frozen variable)* for the associated cluster seed.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3]).principal_extension()
sage: S.mutate([2,1])
sage: S.y(0)
y0

sage: S.y(1)
y1

sage: S.y(2)
y2
```

class sage.combinat.cluster_algebra_quiver.cluster_seed.**ClusterVariable**(*parent*, *numerator*, *denominator*, *coerce=True*, *reduce=True*, *mutation_type=None*, *variable_type=None*, *xdim=0*)

Bases: sage.rings.fraction_field_element.FractionFieldElement

This class is a thin wrapper for cluster variables in cluster seeds.

It provides the extra feature to store if a variable is frozen or not.

•the associated positive root:

```
sage: S = ClusterSeed(['A',3])
sage: for T in S.variable_class_iter(): print T, T.almost_positive_root()
x0 -alpha[1]
x1 -alpha[2]
x2 -alpha[3]
(x1 + 1)/x0 alpha[1]
(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2) alpha[1] + alpha[2] + alpha[3]
(x1 + 1)/x2 alpha[3]
(x0*x2 + x1 + 1)/(x0*x1) alpha[1] + alpha[2]
(x0*x2 + 1)/x1 alpha[2]
(x0*x2 + x1 + 1)/(x1*x2) alpha[2] + alpha[3]
```

almost_positive_root()

Returns the *almost positive root* associated to `self` if `self` is of finite type.

EXAMPLES:

```
sage: S = ClusterSeed(['A',3])
sage: for T in S.variable_class_iter(): print T, T.almost_positive_root()
x0 -alpha[1]
x1 -alpha[2]
x2 -alpha[3]
(x1 + 1)/x0 alpha[1]
(x1^2 + x0*x2 + 2*x1 + 1)/(x0*x1*x2) alpha[1] + alpha[2] + alpha[3]
(x1 + 1)/x2 alpha[3]
(x0*x2 + x1 + 1)/(x0*x1) alpha[1] + alpha[2]
(x0*x2 + 1)/x1 alpha[2]
(x0*x2 + x1 + 1)/(x1*x2) alpha[2] + alpha[3]
```

sage.combinat.cluster_algebra_quiver.cluster_seed.**PathSubset**(*n*, *m*)

Encodes a *maximal* Dyck path from (0,0) to (n,m) (for n >= m >= 0) as a subset of {0,1,2,..., 2n-1}. The encoding is given by indexing horizontal edges by odd numbers and vertical edges by evens.

The horizontal between (i,j) and (i+1,j) is indexed by the odd number 2*i+1. The vertical between (i,j) and

(i,j+1) is indexed by the even number 2*j.

EXAMPLES:

```
sage: from sage.combinat.cluster_algebra_quiver.cluster_seed import PathSubset
sage: PathSubset(4,0)
{1, 3, 5, 7}
sage: PathSubset(4,1)
{1, 3, 5, 6, 7}
sage: PathSubset(4,2)
{1, 2, 3, 5, 6, 7}
sage: PathSubset(4,3)
{1, 2, 3, 4, 5, 6, 7}
sage: PathSubset(4,4)
{0, 1, 2, 3, 4, 5, 6, 7}
```

sage.combinat.cluster_algebra_quiver.cluster_seed.**SetToPath**(*T*)

Rearranges the encoding for a *maximal* Dyck path (as a set) so that it is a list in the proper order of the edges.

EXAMPLES:

```
sage: from sage.combinat.cluster_algebra_quiver.cluster_seed import PathSubset
sage: from sage.combinat.cluster_algebra_quiver.cluster_seed import SetToPath
sage: SetToPath(PathSubset(4,0))
[1, 3, 5, 7]
sage: SetToPath(PathSubset(4,1))
[1, 3, 5, 7, 6]
sage: SetToPath(PathSubset(4,2))
[1, 3, 2, 5, 7, 6]
sage: SetToPath(PathSubset(4,3))
[1, 3, 2, 5, 4, 7, 6]
sage: SetToPath(PathSubset(4,4))
[1, 0, 3, 2, 5, 4, 7, 6]
```

sage.combinat.cluster_algebra_quiver.cluster_seed.**coeff_recurs**(*p*, *q*, *a1*, *a2*, *b*, *c*)

Coefficients in Laurent expansion of greedy element, as defined by recursion.

EXAMPLES:

```
sage: from sage.combinat.cluster_algebra_quiver.cluster_seed import coeff_recurs
sage: coeff_recurs(1, 1, 5, 5, 3, 3)
10
```

sage.combinat.cluster_algebra_quiver.cluster_seed.**get_green_vertices**(*C*)

Get the green vertices from a matrix. Will go through each clumn and return the ones where no entry is greater than 0.

INPUT:

- C – The C matrix to check

EXAMPLES:

```
sage: from sage.combinat.cluster_algebra_quiver.cluster_seed import get_green_vertices
sage: S = ClusterSeed(['A',4]); S.mutate([1,2,3,2,0,1,2,0,3])
sage: get_green_vertices(S.c_matrix())
[0, 3]
```

sage.combinat.cluster_algebra_quiver.cluster_seed.**get_red_vertices**(*C*)

Get the red vertices from a matrix. Will go through each clumn and return the ones where no entry is less than 0.

INPUT:

- C – The C matrix to check

EXAMPLES:: sage: from sage.combinat.cluster_algebra_quiver.cluster_seed import get_red_vertices sage: S = ClusterSeed(['A',4]); S.mutate([1,2,3,2,0,1,2,0,3]) sage: get_red_vertices(S.c_matrix()) [1, 2]

sage.combinat.cluster_algebra_quiver.cluster_seed.**is_LeeLiZel_allowable**(*T*, *n*, *m*, *b*, *c*)

Check if the subset T contributes to the computation of the greedy element x[m,n] in the rank two (b,c)-cluster algebra.

This uses the conditions of Lee-Li-Zelevinsky's paper [LeeLiZe].

EXAMPLES:

```
sage: from sage.combinat.cluster_algebra_quiver.cluster_seed import is_LeeLiZel_allowable
sage: is_LeeLiZel_allowable({1,3,2,5,7,6},4,2,6,6)
False
sage: is_LeeLiZel_allowable({1,2,5},3,3,1,1)
True
```

5.1.16 mutation_class

This file contains helper functions for compute the mutation class of a cluster algebra or quiver.

For the compendium on the cluster algebra and quiver package see

http://arxiv.org/abs/1102.4844

AUTHORS:

- Gregg Musiker
- Christian Stump

5.1.17 This file contains helper functions for detecting the mutation type of

a cluster algebra or quiver.

For the compendium on the cluster algebra and quiver package see

Arxiv 1102.4844

AUTHORS:

- Gregg Musiker
- Christian Stump

sage.combinat.cluster_algebra_quiver.mutation_type.**is_mutation_finite**(*M*, *nr_of_checks=None*)

Use a non-deterministic method by random mutations in various directions. Can result in a wrong answer.

INPUT:

- nr_of_checks – (default: None) number of mutations applied. Standard is 500*(number of vertices of self).

ALGORITHM:

A quiver is mutation infinite if and only if every edge label (a,-b) satisfy a*b > 4. Thus, we apply random mutations in random directions

EXAMPLES:

```
sage: from sage.combinat.cluster_algebra_quiver.mutation_type import is_mutation_finite

sage: Q = ClusterQuiver(['A',10])
sage: M = Q.b_matrix()
sage: is_mutation_finite(M)
(True, None)

sage: Q = ClusterQuiver([(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(2,9)])
sage: M = Q.b_matrix()
sage: is_mutation_finite(M) # random
(False, [9, 6, 9, 8, 9, 4, 0, 4, 5, 2, 1, 0, 1, 0, 7, 1, 9, 2, 5, 7, 8, 6, 3, 0, 2, 5, 4, 2, 6,
```

Check that trac ticket #19495 is fixed:

```
sage: dg = DiGraph(); dg.add_vertex(0); S = ClusterSeed(dg); S
A seed for a cluster algebra of rank 1
sage: S.is_mutation_finite()
True
```

sage.combinat.cluster_algebra_quiver.mutation_type.**load_data**(*n*)

Load a dict with keys being tuples representing exceptional QuiverMutationTypes, and with values being lists or sets containing all mutation equivalent quivers as dig6 data.

We check

- if the data is stored by the user, and if this is not the case
- if the data is stored by the optional package install

EXAMPLES:

```
sage: from sage.combinat.cluster_algebra_quiver.mutation_type import load_data
sage: load_data(2) # random
{('G', 2): [('AO', (((0, 1), (1, -3)),)), ('AO', (((0, 1), (3, -1)),))]}
```

5.1.18 Quiver

A *quiver* is an oriented graph without loops, two-cycles, or multiple edges. The edges are labelled by pairs $(i, -j)$ (with i and j being positive integers) such that the matrix $M = (m_{ab})$ with $m_{ab} = i, m_{ba} = -j$ for an edge $(i, -j)$ between vertices a and b is skew-symmetrizable.

For the compendium on the cluster algebra and quiver package see

http://arxiv.org/abs/1102.4844.

AUTHORS:

- Gregg Musiker
- Christian Stump

See also:

For mutation types of combinatorial quivers, see `QuiverMutationType()`. Cluster seeds are closely related to `ClusterSeed()`.

class sage.combinat.cluster_algebra_quiver.quiver.**ClusterQuiver**(*data*, *frozen=None*)

Bases: sage.structure.sage_object.SageObject

The *quiver* associated to an *exchange matrix*.

INPUT:

- data – can be any of the following:

```
* QuiverMutationType
* str - a string representing a QuiverMutationType or a common quiver type (see Examples)
* ClusterQuiver
* Matrix - a skew-symmetrizable matrix
* DiGraph - must be the input data for a quiver
* List of edges - must be the edge list of a digraph for a quiver
```

- frozen – (default:None) sets the number of frozen variables if the input type is a DiGraph, it is ignored otherwise.

EXAMPLES:

from a QuiverMutationType:

```
sage: Q = ClusterQuiver(['A',5]); Q
Quiver on 5 vertices of type ['A', 5]

sage: Q = ClusterQuiver(['B',2]); Q
Quiver on 2 vertices of type ['B', 2]
sage: Q2 = ClusterQuiver(['C',2]); Q2
Quiver on 2 vertices of type ['B', 2]
sage: MT = Q.mutation_type(); MT.standard_quiver() == Q
True
sage: MT = Q2.mutation_type(); MT.standard_quiver() == Q2
False

sage: Q = ClusterQuiver(['A',[2,5],1]); Q
Quiver on 7 vertices of type ['A', [2, 5], 1]

sage: Q = ClusterQuiver(['A', [5,0],1]); Q
Quiver on 5 vertices of type ['D', 5]
sage: Q.is_finite()
True
sage: Q.is_acyclic()
False

sage: Q = ClusterQuiver(['F', 4, [2,1]]); Q
Quiver on 6 vertices of type ['F', 4, [1, 2]]
sage: MT = Q.mutation_type(); MT.standard_quiver() == Q
False
sage: dg = Q.digraph(); Q.mutate([2,1,4,0,5,3])
sage: dg2 = Q.digraph(); dg2.is_isomorphic(dg,edge_labels=True)
False
sage: dg2.is_isomorphic(MT.standard_quiver().digraph(),edge_labels=True)
True

sage: Q = ClusterQuiver(['G',2, (3,1)]); Q
Quiver on 4 vertices of type ['G', 2, [1, 3]]
sage: MT = Q.mutation_type(); MT.standard_quiver() == Q
```

```
False

sage: Q = ClusterQuiver(['GR',[3,6]]); Q
Quiver on 4 vertices of type ['D', 4]
sage: MT = Q.mutation_type(); MT.standard_quiver() == Q
False

sage: Q = ClusterQuiver(['GR',[3,7]]); Q
Quiver on 6 vertices of type ['E', 6]

sage: Q = ClusterQuiver(['TR',2]); Q
Quiver on 3 vertices of type ['A', 3]
sage: MT = Q.mutation_type(); MT.standard_quiver() == Q
False
sage: Q.mutate([1,0]); MT.standard_quiver() == Q
True

sage: Q = ClusterQuiver(['TR',3]); Q
Quiver on 6 vertices of type ['D', 6]
sage: MT = Q.mutation_type(); MT.standard_quiver() == Q
False
```

from a ClusterQuiver:

```
sage: Q = ClusterQuiver(['A',[2,5],1]); Q
Quiver on 7 vertices of type ['A', [2, 5], 1]
sage: T = ClusterQuiver( Q ); T
Quiver on 7 vertices of type ['A', [2, 5], 1]
```

from a Matrix:

```
sage: Q = ClusterQuiver(['A',[2,5],1]); Q
Quiver on 7 vertices of type ['A', [2, 5], 1]
sage: T = ClusterQuiver( Q._M ); T
Quiver on 7 vertices

sage: Q = ClusterQuiver( matrix([[0,1,-1],[-1,0,1],[1,-1,0],[1,2,3]]) ); Q
Quiver on 4 vertices with 1 frozen vertex

sage: Q = ClusterQuiver( matrix([]) ); Q
Quiver without vertices
```

from a DiGraph:

```
sage: Q = ClusterQuiver(['A',[2,5],1]); Q
Quiver on 7 vertices of type ['A', [2, 5], 1]
sage: T = ClusterQuiver( Q._digraph ); T
Quiver on 7 vertices

sage: Q = ClusterQuiver( DiGraph([[1,2],[2,3],[3,4],[4,1]]) ); Q
Quiver on 4 vertices
```

from a List of edges:

```
sage: Q = ClusterQuiver(['A',[2,5],1]); Q
Quiver on 7 vertices of type ['A', [2, 5], 1]
sage: T = ClusterQuiver( Q._digraph.edges() ); T
Quiver on 7 vertices

sage: Q = ClusterQuiver( [[1,2],[2,3],[3,4],[4,1]] ); Q
Quiver on 4 vertices
```

TESTS:

```
sage: Q = ClusterQuiver(DiGraph([[1,1]]))
Traceback (most recent call last):
...
ValueError: The input DiGraph contains a loop

sage: Q = ClusterQuiver([[1,1]])
Traceback (most recent call last):
...
ValueError: The input DiGraph contains a loop

sage: Q = ClusterQuiver(DiGraph([[1, 0],[0,1]]))
Traceback (most recent call last):
...
ValueError: The input DiGraph contains two-cycles

sage: Q = ClusterQuiver('whatever')
Traceback (most recent call last):
...
ValueError: The input data was not recognized.
```

b_matrix()

Returns the b-matrix of `self`.

EXAMPLES:

```
sage: ClusterQuiver(['A',4]).b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

sage: ClusterQuiver(['B',4]).b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -2  0]

sage: ClusterQuiver(['D',4]).b_matrix()
[ 0  1  0  0]
[-1  0 -1 -1]
[ 0  1  0  0]
[ 0  1  0  0]

sage: ClusterQuiver(QuiverMutationType([['A',2],['B',2]])).b_matrix()
[ 0  1  0  0]
[-1  0  0  0]
[ 0  0  0  1]
[ 0  0 -2  0]
```

canonical_label(*certify=False*)

Returns the canonical labelling of `self`, see `sage.graphs.graph.GenericGraph.canonical_label()`.

INPUT:

- `certify` – (default: False) if True, the dictionary from `self.vertices()` to the vertices of the returned quiver is returned as well.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',4]); Q.digraph().edges()
[(0, 1, (1, -1)), (2, 1, (1, -1)), (2, 3, (1, -1))]

sage: T = Q.canonical_label(); T.digraph().edges()
[(0, 3, (1, -1)), (1, 2, (1, -1)), (1, 3, (1, -1))]

sage: T,iso = Q.canonical_label(certify=True); T.digraph().edges(); iso
[(0, 3, (1, -1)), (1, 2, (1, -1)), (1, 3, (1, -1))]
{0: 0, 1: 3, 2: 1, 3: 2}

sage: Q = ClusterQuiver(QuiverMutationType([['B',2],['A',1]])); Q
Quiver on 3 vertices of type [ ['B', 2], ['A', 1] ]

sage: Q.canonical_label()
Quiver on 3 vertices of type [ ['A', 1], ['B', 2] ]

sage: Q.canonical_label(certify=True)
(Quiver on 3 vertices of type [ ['A', 1], ['B', 2] ], {0: 1, 1: 2, 2: 0})
```

d_vector_fan()

Return the d-vector fan associated with the quiver.

It is the fan whose maximal cones are generated by the d-matrices of the clusters.

This is a complete simplicial fan (and even smooth when the initial quiver is acyclic). It only makes sense for quivers of finite type.

EXAMPLES:

```
sage: Fd = ClusterQuiver([[1,2]]).d_vector_fan(); Fd
Rational polyhedral fan in 2-d lattice N
sage: Fd.ngenerating_cones()
5

sage: Fd = ClusterQuiver([[1,2],[2,3]]).d_vector_fan(); Fd
Rational polyhedral fan in 3-d lattice N
sage: Fd.ngenerating_cones()
14
sage: Fd.is_smooth()
True

sage: Fd = ClusterQuiver([[1,2],[2,3],[3,1]]).d_vector_fan(); Fd
Rational polyhedral fan in 3-d lattice N
sage: Fd.ngenerating_cones()
14
sage: Fd.is_smooth()
False
```

TESTS:

```
sage: ClusterQuiver(['A',[2,2],1]).d_vector_fan()
Traceback (most recent call last):
```

```
...
ValueError: only makes sense for quivers of finite type
```

digraph()

Returns the underlying digraph of `self`.

EXAMPLES:

```
sage: ClusterQuiver(['A',1]).digraph()
Digraph on 1 vertex
sage: ClusterQuiver(['A',1]).digraph().vertices()
[0]
sage: ClusterQuiver(['A',1]).digraph().edges()
[]

sage: ClusterQuiver(['A',4]).digraph()
Digraph on 4 vertices
sage: ClusterQuiver(['A',4]).digraph().edges()
[(0, 1, (1, -1)), (2, 1, (1, -1)), (2, 3, (1, -1))]

sage: ClusterQuiver(['B',4]).digraph()
Digraph on 4 vertices
sage: ClusterQuiver(['A',4]).digraph().edges()
[(0, 1, (1, -1)), (2, 1, (1, -1)), (2, 3, (1, -1))]

sage: ClusterQuiver(QuiverMutationType([['A',2],['B',2]])).digraph()
Digraph on 4 vertices

sage: ClusterQuiver(QuiverMutationType([['A',2],['B',2]])).digraph().edges()
[(0, 1, (1, -1)), (2, 3, (1, -2))]
```

exchangeable_part()

Returns the restriction to the principal part (i.e. exchangeable part) of `self`, the subquiver obtained by deleting the frozen vertices of `self`.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',4])
sage: T = ClusterQuiver( Q.digraph().edges(), frozen=1 )
sage: T.digraph().edges()
[(0, 1, (1, -1)), (2, 1, (1, -1)), (2, 3, (1, -1))]

sage: T.exchangeable_part().digraph().edges()
[(0, 1, (1, -1)), (2, 1, (1, -1))]

sage: Q2 = Q.principal_extension()
sage: Q3 = Q2.principal_extension()
sage: Q2.exchangeable_part() == Q3.exchangeable_part()
True
```

first_sink()

Return the first vertex of `self` that is a sink

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',5]);
sage: Q.mutate([1,2,4,3,2]);
sage: Q.first_sink()
0
```

first_source()
Return the first vertex of `self` that is a source

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',5])
sage: Q.mutate([2,1,3,4,2])
sage: Q.first_source()
1
```

g_vector_fan()
Return the g-vector fan associated with the quiver.

It is the fan whose maximal cones are generated by the g-matrices of the clusters.

This is a complete simplicial fan. It is only supported for quivers of finite type.

EXAMPLES:

```
sage: Fg = ClusterQuiver([[1,2]]).g_vector_fan(); Fg
Rational polyhedral fan in 2-d lattice N
sage: Fg.ngenerating_cones()
5

sage: Fg = ClusterQuiver([[1,2],[2,3]]).g_vector_fan(); Fg
Rational polyhedral fan in 3-d lattice N
sage: Fg.ngenerating_cones()
14
sage: Fg.is_smooth()
True

sage: Fg = ClusterQuiver([[1,2],[2,3],[3,1]]).g_vector_fan(); Fg
Rational polyhedral fan in 3-d lattice N
sage: Fg.ngenerating_cones()
14
sage: Fg.is_smooth()
True
```

TESTS:

```
sage: ClusterQuiver(['A',[2,2],1]).g_vector_fan()
Traceback (most recent call last):
...
ValueError: only supported for quivers of finite type
```

interact(*fig_size=1*, *circular=True*)
Only in notebook mode. Starts an interactive window for cluster seed mutations.

INPUT:

- `fig_size` – (default: 1) factor by which the size of the plot is multiplied.
- `circular` – (default: False) if True, the circular plot is chosen, otherwise >>spring<< is used.

TESTS:

```
sage: Q = ClusterQuiver(['A',4])
sage: Q.interact() # long time
'The interactive mode only runs in the Sage notebook.'
```

is_acyclic()
Returns true if `self` is acyclic.

EXAMPLES:

```
sage: ClusterQuiver(['A',4]).is_acyclic()
True

sage: ClusterQuiver(['A',[2,1],1]).is_acyclic()
True

sage: ClusterQuiver([[0,1],[1,2],[2,0]]).is_acyclic()
False
```

is_bipartite(*return_bipartition=False*)

Returns true if `self` is bipartite.

EXAMPLES:

```
sage: ClusterQuiver(['A',[3,3],1]).is_bipartite()
True

sage: ClusterQuiver(['A',[4,3],1]).is_bipartite()
False
```

is_finite()

Returns True if self is of finite type.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',3])
sage: Q.is_finite()
True
sage: Q = ClusterQuiver(['A',[2,2],1])
sage: Q.is_finite()
False
sage: Q = ClusterQuiver([['A',3],['B',3]])
sage: Q.is_finite()
True
sage: Q = ClusterQuiver(['T',[4,4,4]])
sage: Q.is_finite()
False
sage: Q = ClusterQuiver([['A',3],['T',[4,4,4]]])
sage: Q.is_finite()
False
sage: Q = ClusterQuiver([['A',3],['T',[2,2,3]]])
sage: Q.is_finite()
True
sage: Q = ClusterQuiver([['A',3],['D',5]])
sage: Q.is_finite()
True
sage: Q = ClusterQuiver([['A',3],['D',5,1]])
sage: Q.is_finite()
False

sage: Q = ClusterQuiver([[0,1,2],[1,2,2],[2,0,2]])
sage: Q.is_finite()
False

sage: Q = ClusterQuiver([[0,1,2],[1,2,2],[2,0,2],[3,4,1],[4,5,1]])
sage: Q.is_finite()
False
```

is_mutation_finite(*nr_of_checks=None*, *return_path=False*)

Uses a non-deterministic method by random mutations in various directions. Can result in a wrong answer.

INPUT:

- `nr_of_checks` – (default: None) number of mutations applied. Standard is 500*(number of vertices of self).
- `return_path` – (default: False) if True, in case of self not being mutation finite, a path from self to a quiver with an edge label (a,-b) and a*b > 4 is returned.

ALGORITHM:

A quiver is mutation infinite if and only if every edge label (a,-b) satisfy a*b > 4. Thus, we apply random mutations in random directions

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',10])
sage: Q._mutation_type = None
sage: Q.is_mutation_finite()
True

sage: Q = ClusterQuiver([(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(2,9)])
sage: Q.is_mutation_finite()
False
```

m()

Returns the number of frozen vertices of `self`.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',4])
sage: Q.m()
0

sage: T = ClusterQuiver( Q.digraph().edges(), frozen=1 )
sage: T.n()
3
sage: T.m()
1
```

mutate(*data*, *inplace=True*)

Mutates `self` at a sequence of vertices.

INPUT:

- `sequence` – a vertex of `self`, an iterator of vertices of `self`, a function which takes in the ClusterQuiver and returns a vertex or an iterator of vertices, or a string of the parameter wanting to be called on ClusterQuiver that will return a vertex or an iterator of vertices.
- `inplace` – (default: True) if False, the result is returned, otherwise `self` is modified.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',4]); Q.b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  1  0  1]
[ 0  0 -1  0]

sage: Q.mutate(0); Q.b_matrix()
[ 0 -1  0  0]
[ 1  0 -1  0]
[ 0  1  0  1]
```

```
[ 0  0 -1  0]

sage: T = Q.mutate(0, inplace=False); T
Quiver on 4 vertices of type ['A', 4]

sage: Q.mutate(0)
sage: Q == T
True

sage: Q.mutate([0,1,0])
sage: Q.b_matrix()
[ 0 -1  1  0]
[ 1  0  0  0]
[-1  0  0  1]
[ 0  0 -1  0]

sage: Q = ClusterQuiver(QuiverMutationType([['A',1],['A',3]]))
sage: Q.b_matrix()
[ 0  0  0  0]
[ 0  0  1  0]
[ 0 -1  0 -1]
[ 0  0  1  0]

sage: T = Q.mutate(0,inplace=False)
sage: Q == T
True

sage: Q = ClusterQuiver(['A',3]); Q.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
sage: Q.mutate('first_sink'); Q.b_matrix()
[ 0 -1  0]
[ 1  0  1]
[ 0 -1  0]
sage: Q.mutate('first_source'); Q.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
```

TESTS:

```
sage: Q = ClusterQuiver(['A',4]); Q.mutate(0,1)
Traceback (most recent call last):
...
ValueError: The second parameter must be boolean.  To mutate at a sequence of length 2, inpu

sage: Q = ClusterQuiver(['A',4]); Q.mutate(0,0)
Traceback (most recent call last):
...
ValueError: The second parameter must be boolean.  To mutate at a sequence of length 2, inpu
```

mutation_class(*depth=+Infinity*, *show_depth=False*, *return_paths=False*, *data_type='quiver'*, *up_to_equivalence=True*, *sink_source=False*)

Returns the mutation class of self together with certain constrains.

INPUT:

- `depth` – (default: infinity) integer, only seeds with distance at most depth from self are returned.

- `show_depth` – (default: False) if True, the actual depth of the mutation is shown.
- `return_paths` – (default: False) if True, a shortest path of mutation sequences from self to the given quiver is returned as well.
- `data_type` – (default: "quiver") can be one of the following:

```
* "quiver" -- the quiver is returned
* "dig6" -- the dig6-data is returned
* "path" -- shortest paths of mutation sequences from self are returned
```

- `sink_source` – (default: False) if True, only mutations at sinks and sources are applied.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',3])
sage: Ts = Q.mutation_class()
sage: for T in Ts: print T
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

sage: Ts = Q.mutation_class(depth=1)
sage: for T in Ts: print T
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

sage: Ts = Q.mutation_class(show_depth=True)
Depth: 0     found: 1          Time: ... s
Depth: 1     found: 3          Time: ... s
Depth: 2     found: 4          Time: ... s

sage: Ts = Q.mutation_class(return_paths=True)
sage: for T in Ts: print T
(Quiver on 3 vertices of type ['A', 3], [])
(Quiver on 3 vertices of type ['A', 3], [1])
(Quiver on 3 vertices of type ['A', 3], [0])
(Quiver on 3 vertices of type ['A', 3], [0, 1])

sage: Ts = Q.mutation_class(up_to_equivalence=False)
sage: for T in Ts: print T
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

sage: Ts = Q.mutation_class(return_paths=True,up_to_equivalence=False)
```

```
sage: for T in Ts: print T
(Quiver on 3 vertices of type ['A', 3], [])
(Quiver on 3 vertices of type ['A', 3], [2])
(Quiver on 3 vertices of type ['A', 3], [1])
(Quiver on 3 vertices of type ['A', 3], [0])
(Quiver on 3 vertices of type ['A', 3], [2, 1])
(Quiver on 3 vertices of type ['A', 3], [0, 1])
(Quiver on 3 vertices of type ['A', 3], [0, 1, 2])
(Quiver on 3 vertices of type ['A', 3], [0, 1, 0])
(Quiver on 3 vertices of type ['A', 3], [2, 1, 2])
(Quiver on 3 vertices of type ['A', 3], [2, 1, 0])
(Quiver on 3 vertices of type ['A', 3], [2, 1, 0, 2])
(Quiver on 3 vertices of type ['A', 3], [2, 1, 0, 1])
(Quiver on 3 vertices of type ['A', 3], [2, 1, 2, 1])
(Quiver on 3 vertices of type ['A', 3], [2, 1, 2, 0])

sage: Ts = Q.mutation_class(show_depth=True)
Depth: 0     found: 1          Time: ... s
Depth: 1     found: 3          Time: ... s
Depth: 2     found: 4          Time: ... s

sage: Ts = Q.mutation_class(show_depth=True, up_to_equivalence=False)
Depth: 0     found: 1          Time: ... s
Depth: 1     found: 4          Time: ... s
Depth: 2     found: 6          Time: ... s
Depth: 3     found: 10        Time: ... s
Depth: 4     found: 14        Time: ... s
```

TESTS:

```
sage: all( len(ClusterQuiver(['A',n]).mutation_class()) == ClusterQuiver(['A',n]).mutation_t
True

sage: all( len(ClusterQuiver(['B',n]).mutation_class()) == ClusterQuiver(['B',n]).mutation_t
True
```

mutation_class_iter(*depth=+Infinity*, *show_depth=False*, *return_paths=False*, *data_type='quiver'*, *up_to_equivalence=True*, *sink_source=False*)

Returns an iterator for the mutation class of self together with certain constrains.

INPUT:

- `depth` – (default: infinity) integer, only quivers with distance at most depth from self are returned.
- `show_depth` – (default: False) if True, the actual depth of the mutation is shown.
- `return_paths` – (default: False) if True, a shortest path of mutation sequences from self to the given quiver is returned as well.
- `data_type` – (default: "quiver") can be one of the following:

  * "quiver"
  * "matrix"
  * "digraph"
  * "dig6"
  * "path"

- `up_to_equivalence` – (default: True) if True, only one quiver for each graph-isomorphism class is recorded.

- `sink_source` – (default: False) if True, only mutations at sinks and sources are applied.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',3])
sage: it = Q.mutation_class_iter()
sage: for T in it: print T
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

sage: it = Q.mutation_class_iter(depth=1)
sage: for T in it: print T
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

sage: it = Q.mutation_class_iter(show_depth=True)
sage: for T in it: pass
Depth: 0     found: 1          Time: ... s
Depth: 1     found: 3          Time: ... s
Depth: 2     found: 4          Time: ... s

sage: it = Q.mutation_class_iter(return_paths=True)
sage: for T in it: print T
(Quiver on 3 vertices of type ['A', 3], [])
(Quiver on 3 vertices of type ['A', 3], [1])
(Quiver on 3 vertices of type ['A', 3], [0])
(Quiver on 3 vertices of type ['A', 3], [0, 1])

sage: it = Q.mutation_class_iter(up_to_equivalence=False)
sage: for T in it: print T
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]
Quiver on 3 vertices of type ['A', 3]

sage: it = Q.mutation_class_iter(return_paths=True,up_to_equivalence=False)
sage: for T in it: print T
(Quiver on 3 vertices of type ['A', 3], [])
(Quiver on 3 vertices of type ['A', 3], [2])
(Quiver on 3 vertices of type ['A', 3], [1])
(Quiver on 3 vertices of type ['A', 3], [0])
(Quiver on 3 vertices of type ['A', 3], [2, 1])
(Quiver on 3 vertices of type ['A', 3], [0, 1])
(Quiver on 3 vertices of type ['A', 3], [0, 1, 2])
(Quiver on 3 vertices of type ['A', 3], [0, 1, 0])
(Quiver on 3 vertices of type ['A', 3], [2, 1, 2])
(Quiver on 3 vertices of type ['A', 3], [2, 1, 0])
```

```
(Quiver on 3 vertices of type ['A', 3], [2, 1, 0, 2])
(Quiver on 3 vertices of type ['A', 3], [2, 1, 0, 1])
(Quiver on 3 vertices of type ['A', 3], [2, 1, 2, 1])
(Quiver on 3 vertices of type ['A', 3], [2, 1, 2, 0])

sage: Q = ClusterQuiver(['A',3])
sage: it = Q.mutation_class_iter(data_type='path')
sage: for T in it: print T
[]
[1]
[0]
[0, 1]

sage: Q = ClusterQuiver(['A',3])
sage: it = Q.mutation_class_iter(return_paths=True,data_type='matrix')
sage: next(it)
(
[ 0  0  1]
[ 0  0  1]
[-1 -1  0], []
)
```

mutation_sequence(*sequence*, *show_sequence=False*, *fig_size=1.2*)

Returns a list containing the sequence of quivers obtained from `self` by a sequence of mutations on vertices.

INPUT:

- `sequence` – a list or tuple of vertices of `self`.
- `show_sequence` – (default: False) if True, a png containing the mutation sequence is shown.
- `fig_size` – (default: 1.2) factor by which the size of the sequence is expanded.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',4])
sage: seq = Q.mutation_sequence([0,1]); seq
[Quiver on 4 vertices of type ['A', 4], Quiver on 4 vertices of type ['A', 4], Quiver on 4 v
sage: [T.b_matrix() for T in seq]
[
[ 0  1  0  0]  [ 0 -1  0  0]  [ 0  1 -1  0]
[-1  0 -1  0]  [ 1  0 -1  0]  [-1  0  1  0]
[ 0  1  0  1]  [ 0  1  0  1]  [ 1 -1  0  1]
[ 0  0 -1  0], [ 0  0 -1  0], [ 0  0 -1  0]
]
```

mutation_type()

Returns the mutation type of `self`.

Returns the mutation_type of each connected component of self if it can be determined, otherwise, the mutation type of this component is set to be unknown.

The mutation types of the components are ordered by vertex labels.

If you do many type recognitions, you should consider to save exceptional mutation types using *meth*:sage.combinat.cluster_algebra_quiver.quiver_mutation_type.save_exceptional_data

WARNING:

- All finite types can be detected,

- All affine types can be detected, EXCEPT affine type D (the algorithm is not yet implemented)
- All exceptional types can be detected.

EXAMPLES:

```
sage: ClusterQuiver(['A',4]).mutation_type()
['A', 4]
sage: ClusterQuiver(['A',(3,1),1]).mutation_type()
['A', [1, 3], 1]
sage: ClusterQuiver(['C',2]).mutation_type()
['B', 2]
sage: ClusterQuiver(['B',4,1]).mutation_type()
['BD', 4, 1]
```

finite types:

```
sage: Q = ClusterQuiver(['A',5])
sage: Q._mutation_type = None
sage: Q.mutation_type()
['A', 5]

sage: Q = ClusterQuiver([(0,1),(1,2),(2,3),(3,4)])
sage: Q.mutation_type()
['A', 5]
```

affine types:

```
sage: Q = ClusterQuiver(['E',8,[1,1]]); Q
Quiver on 10 vertices of type ['E', 8, [1, 1]]
sage: Q._mutation_type = None; Q
Quiver on 10 vertices
sage: Q.mutation_type() # long time
['E', 8, [1, 1]]
```

the not yet working affine type D (unless user has saved small classical quiver data):

```
sage: Q = ClusterQuiver(['D',4,1])
sage: Q._mutation_type = None
sage: Q.mutation_type() # todo: not implemented
['D', 4, 1]
```

the exceptional types:

```
sage: Q = ClusterQuiver(['X',6])
sage: Q._mutation_type = None
sage: Q.mutation_type() # long time
['X', 6]
```

examples from page 8 of Keller's article "Cluster algebras, quiver representations and triangulated categories" (arXiv:0807.1960):

```
sage: dg = DiGraph(); dg.add_edges([(9,0),(9,4),(4,6),(6,7),(7,8),(8,3),(3,5),(5,6),(8,1),(2
sage: ClusterQuiver( dg ).mutation_type() # long time
['E', 8, [1, 1]]

sage: dg = DiGraph( { 0:[3], 1:[0,4], 2:[0,6], 3:[1,2,7], 4:[3,8], 5:[2], 6:[3,5], 7:[4,6],
sage: ClusterQuiver( dg ).mutation_type() # long time
['E', 8, 1]

sage: dg = DiGraph( { 0:[3,9], 1:[0,4], 2:[0,6], 3:[1,2,7], 4:[3,8], 5:[2], 6:[3,5], 7:[4,6]
sage: ClusterQuiver( dg ).mutation_type() # long time
```

```
['E', 8, [1, 1]]
```

infinite types:

```
sage: Q = ClusterQuiver(['GR',[4,9]])
sage: Q._mutation_type = None
sage: Q.mutation_type()
'undetermined infinite mutation type'
```

reducible types:

```
sage: Q = ClusterQuiver([['A', 3], ['B', 3]])
sage: Q._mutation_type = None
sage: Q.mutation_type()
[ ['A', 3], ['B', 3] ]

sage: Q = ClusterQuiver([['A', 3], ['T', [4,4,4]]])
sage: Q._mutation_type = None
sage: Q.mutation_type()
[['A', 3], 'undetermined infinite mutation type']

sage: Q = ClusterQuiver([['A', 3], ['B', 3], ['T', [4,4,4]]])
sage: Q._mutation_type = None
sage: Q.mutation_type()
[['A', 3], ['B', 3], 'undetermined infinite mutation type']

sage: Q = ClusterQuiver([[0,1,2],[1,2,2],[2,0,2],[3,4,1],[4,5,1]])
sage: Q.mutation_type()
['undetermined finite mutation type', ['A', 3]]
```

TESTS:

```
sage: Q = ClusterQuiver(matrix([[0, 3], [-1, 0], [1, 0], [0, 1]]))
sage: Q.mutation_type()
['G', 2]
sage: Q = ClusterQuiver(matrix([[0, -1, -1, 1, 0], [1, 0, 1, 0, 1], [1, -1, 0, -1, 0], [-1,
sage: Q.mutation_type()
'undetermined infinite mutation type'
```

n()

Returns the number of free vertices of `self`.

EXAMPLES:

```
sage: ClusterQuiver(['A',4]).n()
4
sage: ClusterQuiver(['A',(3,1),1]).n()
4
sage: ClusterQuiver(['B',4]).n()
4
sage: ClusterQuiver(['B',4,1]).n()
5
```

number_of_edges()

Return the total number of edges on the quiver

Note: This only works with non-valued quivers. If used on a non-valued quiver then the positive value is taken to be the number of edges added

OUTPUT:

Returns an integer of the number of edges

EXAMPLES:

```
sage: S = ClusterQuiver(['A',4]); S.number_of_edges()
3

sage: S = ClusterQuiver(['B',4]); S.number_of_edges()
3
```

plot(*circular=True*, *center=(0, 0)*, *directed=True*, *mark=None*, *save_pos=False*, *greens=[]*)

Return the plot of the underlying digraph of `self`.

INPUT:

- `circular` – (default: `True`) if `True`, the circular plot is chosen, otherwise >>spring<< is used.
- `center` – (default:(0,0)) sets the center of the circular plot, otherwise it is ignored.
- `directed` – (default: `True`) if `True`, the directed version is shown, otherwise the undirected.
- `mark` – (default: `None`) if set to i, the vertex i is highlighted.
- `save_pos` – (default: `False`) if `True`, the positions of the vertices are saved.
- `greens` – (default: []) if set to a list, will display the green vertices as green

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',5])
sage: pl = Q.plot()
sage: pl = Q.plot(circular=True)
```

principal_extension(*inplace=False*)

Returns the principal extension of `self`, adding n frozen vertices to any previously frozen vertices. I.e., the quiver obtained by adding an outgoing edge to every mutable vertex of `self`.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',2]); Q
Quiver on 2 vertices of type ['A', 2]
sage: T = Q.principal_extension(); T
Quiver on 4 vertices of type ['A', 2] with 2 frozen vertices
sage: T2 = T.principal_extension(); T2
Quiver on 6 vertices of type ['A', 2] with 4 frozen vertices
sage: Q.digraph().edges()
[(0, 1, (1, -1))]
sage: T.digraph().edges()
[(0, 1, (1, -1)), (2, 0, (1, -1)), (3, 1, (1, -1))]
sage: T2.digraph().edges()
[(0, 1, (1, -1)), (2, 0, (1, -1)), (3, 1, (1, -1)), (4, 0, (1, -1)), (5, 1, (1, -1))]
```

qmu_save(*filename=None*)

Saves a .qmu file of `self` that can then be opened in Bernhard Keller's Quiver Applet.

INPUT:

- `filename` – the filename the image is saved to.

If a filename is not specified, the default name is from_sage.qmu in the current sage directory.

EXAMPLES:

```
sage: Q = ClusterQuiver(['F',4,[1,2]])
sage: Q.qmu_save(os.path.join(SAGE_TMP, 'sage.qmu'))
```

Make sure we can save quivers with $m! = n$ frozen variables, see trac ticket #14851:

```
sage: S=ClusterSeed(['A',3])
sage: T1=S.principal_extension()
sage: Q=T1.quiver()
sage: Q.qmu_save(os.path.join(SAGE_TMP, 'sage.qmu'))
```

relabel (*relabelling*, *inplace=True*)

Returns the quiver after doing a relabelling

Will relabel the vertices of the quiver

INPUT:

- `relabelling` – Dictionary of labels to move around
- `inplace` – (default:True) if True, will return a duplicate of the quiver

EXAMPLES:

```
sage: S = ClusterQuiver(['A',4]).relabel({1:'5',2:'go'})
```

reorient (*data*)

Reorients `self` with respect to the given total order, or with respect to an iterator of edges in `self` to be reverted.

WARNING:

```
This operation might change the mutation type of ``self``.
```

INPUT:

- `data` – an iterator defining a total order on `self.vertices()`, or an iterator of edges in `self` to be reoriented.

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',(2,3),1])
sage: Q.mutation_type()
['A', [2, 3], 1]

sage: Q.reorient([(0,1),(1,2),(2,3),(3,4)])
sage: Q.mutation_type()
['D', 5]

sage: Q.reorient([0,1,2,3,4])
sage: Q.mutation_type()
['A', [1, 4], 1]
```

save_image (*filename*, *circular=False*)

Saves the plot of the underlying digraph of `self`.

INPUT:

- `filename` – the filename the image is saved to.
- `circular` – (default: False) if True, the circular plot is chosen, otherwise >>spring<< is used.

EXAMPLES:

```
sage: Q = ClusterQuiver(['F',4,[1,2]])
sage: Q.save_image(os.path.join(SAGE_TMP, 'sage.png'))
```

show(*fig_size=1*, *circular=False*, *directed=True*, *mark=None*, *save_pos=False*, *greens=[]*)
Show the plot of the underlying digraph of `self`.

INPUT:

- `fig_size` – (default: 1) factor by which the size of the plot is multiplied.
- `circular` – (default: False) if True, the circular plot is chosen, otherwise >>spring<< is used.
- `directed` – (default: True) if True, the directed version is shown, otherwise the undirected.
- `mark` – (default: None) if set to i, the vertex i is highlighted.
- `save_pos` – (default:False) if True, the positions of the vertices are saved.
- `greens` – (default:[]) if set to a list, will display the green vertices as green

TESTS:

```
sage: Q = ClusterQuiver(['A',5])
sage: Q.show() # long time
```

sinks()
Return all vertices of `self` that are sinks

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',5]);
sage: Q.mutate([1,2,4,3,2]);
sage: Q.sinks()
[0, 2]

sage: Q = ClusterQuiver(['A',5])
sage: Q.mutate([2,1,3,4,2])
sage: Q.sinks()
[3]
```

sources()
Returns all vertices of `self` that are sources

EXAMPLES:

```
sage: Q = ClusterQuiver(['A',5]);
sage: Q.mutate([1,2,4,3,2]);
sage: Q.sources()
[]

sage: Q = ClusterQuiver(['A',5])
sage: Q.mutate([2,1,3,4,2])
sage: Q.sources()
[1]
```

5.1.19 Quiver mutation types

AUTHORS:

- Gregg Musiker (2012, initial version)
- Christian Stump (2012, initial version)
- Hugh Thomas (2012, initial version)

`sage.combinat.cluster_algebra_quiver.quiver_mutation_type.`**QuiverMutationType**(**args*)

***Quiver mutation types* can be seen as a slight generalization of** *generalized Cartan types*.

Background on generalized Cartan types can be found at

Wikipedia article Generalized_Cartan_matrix

For the compendium on the cluster algebra and quiver package in Sage see

Arxiv 1102.4844

A B-matrix is a skew-symmetrizable $(n \times n)$-matrix M. I.e., there exists an invertible diagonal matrix D such that DM is skew-symmetric. M can be encoded as a *quiver* by having a directed edge from vertex i to vertex j with label (a, b) if $a = M_{i,j} > 0$ and $b = M_{j,i} < 0$. We consider quivers up to *mutation equivalence*.

To a quiver mutation type we can associate a *generalized Cartan type* by sending M to the generalized Cartan matrix $C(M)$ obtained by replacing all positive entries by their negatives and adding 2's on the main diagonal.

`QuiverMutationType` constructs a quiver mutation type object. For more detail on the possible different types, please see the compendium.

INPUT:

The input consists either of a quiver mutation type, or of a `letter` (a string), a `rank` (one integer or a list/tuple of integers), and an optional `twist` (an integer or a list of integers). There are several different naming conventions for quiver mutation types.

•Finite type – `letter` is a Dynkin type (A-G), and `rank` is the rank.

•Affine type – there is more than one convention for naming affine types.

–Kac's notation: `letter` is a Dynkin type, `rank` is the rank of the associated finite Dynkin diagram, and `twist` is the twist, which could be 1, 2, or 3. In the special case of affine type A, there is more than one quiver mutation type associated to the Cartan type. In this case only, `rank` is a pair of integers (i,j), giving the number of edges pointing clockwise and the number of edges pointing counter-clockwise. The total number of vertices is given by i+j in this case.

–Naive notation: `letter` is one of 'BB', 'BC', 'BD', 'CC', 'CD'. The name specifies the two ends of the diagram, which are joined by a path. The total number of vertices is given by `rank +1` (to match the indexing people expect because these are affine types). In general, `rank` must be large enough for the picture to make sense, but we accept `letter` is `BC` and `rank=1`.

–Macdonald notation: for the dual of an untwisted affine type (such as ['C', 6,1]), we accept a twist of -1 (i.e., ['C',6,-1]).

•Elliptic type – `letter` is a Dynkin type, `rank` is the rank of the finite Dynkin diagram, and `twist` is a tuple of two integers. We follow Saito's notation.

•Other shapes:

–Rank 2: `letter` is 'R2', and `rank` is a pair of integers specifying the label on the unique edge.

–Triangle: `letter` is `TR`, and `rank` is the number of vertices along a side.

–T: This defines a quiver shaped like a T. `letter` is 'T', and the `rank` is a triple, whose entries specify the number of vertices along each path from the branch point (counting the branch point).

–Grassmannian: This defines the cluster algebra (without coefficients) corresponding to the cluster algebra with coefficients which is the co-ordinate ring of a Grassmannian. `letter` is 'GR'. `rank` is a pair of integers (k, n) with 'k' < 'n' specifying the Grassmannian of k-planes in n-space. This defines a quiver given by a (k-1) x (n-k-1) grid where each square is cyclically oriented.

–Exceptional mutation finite quivers: The two exceptional mutation finite quivers, found by Derksen-Owen, have `letter` as 'X' and `rank` 6 or 7, equal to the number of vertices.

– AE, BE, CE, DE: Quivers are built of one end which looks like type (affine A), B, C, or D, and the other end which looks like type E (i.e., it consists of two antennae, one of length one, and one of length two). `letter` is 'AE', 'BE', 'CE', or 'DE', and `rank` is the total number of vertices. Note that 'AE' is of a slightly different form and requires `rank` to be a pair of integers (i,j) just as in the case of affine type A. See Exercise 4.3 in Kac's book Infinite Dimensional Lie Algebras for more details.

– Infinite type E: It is also possible to obtain infinite-type E quivers by specifying `letter` as 'E' and `rank` as the number of vertices.

REFERENCES:

- A good reference for finite and affine Dynkin diagrams, including Kac's notation, is the Wikipedia article Dynkin_diagram.
- A good reference for the skew-symmetrizable elliptic diagrams is "Cluster algebras of finite mutation type via unfolding" by A. Felikson, M. Shapiro, and P. Tumarkin, Arxiv 1006.4276v4.

EXAMPLES:

Finite types:

```
sage: QuiverMutationType('A',1)
['A', 1]
sage: QuiverMutationType('A',5)
['A', 5]

sage: QuiverMutationType('B',2)
['B', 2]
sage: QuiverMutationType('B',5)
['B', 5]

sage: QuiverMutationType('C',2)
['B', 2]
sage: QuiverMutationType('C',5)
['C', 5]

sage: QuiverMutationType('D',2)
[ ['A', 1], ['A', 1] ]
sage: QuiverMutationType('D',3)
['A', 3]
sage: QuiverMutationType('D',4)
['D', 4]

sage: QuiverMutationType('E',6)
['E', 6]

sage: QuiverMutationType('G',2)
['G', 2]

sage: QuiverMutationType('A',(1,0),1)
['A', 1]

sage: QuiverMutationType('A',(2,0),1)
[ ['A', 1], ['A', 1] ]

sage: QuiverMutationType('A',(7,0),1)
['D', 7]
```

Affine types:

```
sage: QuiverMutationType('A',(1,1),1)
['A', [1, 1], 1]
sage: QuiverMutationType('A',(2,4),1)
['A', [2, 4], 1]

sage: QuiverMutationType('BB',2,1)
['BB', 2, 1]
sage: QuiverMutationType('BB',4,1)
['BB', 4, 1]

sage: QuiverMutationType('CC',2,1)
['CC', 2, 1]
sage: QuiverMutationType('CC',4,1)
['CC', 4, 1]

sage: QuiverMutationType('BC',1,1)
['BC', 1, 1]
sage: QuiverMutationType('BC',5,1)
['BC', 5, 1]

sage: QuiverMutationType('BD',3,1)
['BD', 3, 1]
sage: QuiverMutationType('BD',5,1)
['BD', 5, 1]

sage: QuiverMutationType('CD',3,1)
['CD', 3, 1]
sage: QuiverMutationType('CD',5,1)
['CD', 5, 1]

sage: QuiverMutationType('D',4,1)
['D', 4, 1]
sage: QuiverMutationType('D',6,1)
['D', 6, 1]

sage: QuiverMutationType('E',6,1)
['E', 6, 1]
sage: QuiverMutationType('E',7,1)
['E', 7, 1]
sage: QuiverMutationType('E',8,1)
['E', 8, 1]

sage: QuiverMutationType('F',4,1)
['F', 4, 1]
sage: QuiverMutationType('F',4,-1)
['F', 4, -1]

sage: QuiverMutationType('G',2,1)
['G', 2, 1]
sage: QuiverMutationType('G',2,-1)
['G', 2, -1]
sage: QuiverMutationType('A',3,2) == QuiverMutationType('D',3,2)
True
```

Affine types using Kac's Notation:

```
sage: QuiverMutationType('A',1,1)
['A', [1, 1], 1]
sage: QuiverMutationType('B',5,1)
```

```
['BD', 5, 1]
sage: QuiverMutationType('C',5,1)
['CC', 5, 1]
sage: QuiverMutationType('A',2,2)
['BC', 1, 1]
sage: QuiverMutationType('A',7,2)
['CD', 4, 1]
sage: QuiverMutationType('A',8,2)
['BC', 4, 1]
sage: QuiverMutationType('D',6,2)
['BB', 5, 1]
sage: QuiverMutationType('E',6,2)
['F', 4, -1]
sage: QuiverMutationType('D',4,3)
['G', 2, -1]
```

Elliptic types:

```
sage: QuiverMutationType('E',6,[1,1])
['E', 6, [1, 1]]
sage: QuiverMutationType('F',4,[2,1])
['F', 4, [1, 2]]
sage: QuiverMutationType('G',2,[3,3])
['G', 2, [3, 3]]
```

Mutation finite types:

rank 2 cases:

```
sage: QuiverMutationType('R2',(1,1))
['A', 2]
sage: QuiverMutationType('R2',(1,2))
['B', 2]
sage: QuiverMutationType('R2',(1,3))
['G', 2]
sage: QuiverMutationType('R2',(1,4))
['BC', 1, 1]
sage: QuiverMutationType('R2',(1,5))
['R2', [1, 5]]
sage: QuiverMutationType('R2',(2,2))
['A', [1, 1], 1]
sage: QuiverMutationType('R2',(3,5))
['R2', [3, 5]]
```

Exceptional Derksen-Owen quivers:

```
sage: QuiverMutationType('X',6)
['X', 6]
```

(Mainly) mutation infinite types:

Infinite type E:

```
sage: QuiverMutationType('E',9)
['E', 8, 1]
sage: QuiverMutationType('E',10)
```

```
['E', 10]
sage: QuiverMutationType('E',12)
['E', 12]

sage: QuiverMutationType('AE',(2,3))
['AE', [2, 3]]
sage: QuiverMutationType('BE',5)
['BE', 5]
sage: QuiverMutationType('CE',5)
['CE', 5]
sage: QuiverMutationType('DE',6)
['DE', 6]
```

Grassmannian types:

```
sage: QuiverMutationType('GR',(2,4))
['A', 1]
sage: QuiverMutationType('GR',(2,6))
['A', 3]
sage: QuiverMutationType('GR',(3,6))
['D', 4]
sage: QuiverMutationType('GR',(3,7))
['E', 6]
sage: QuiverMutationType('GR',(3,8))
['E', 8]
sage: QuiverMutationType('GR',(3,10))
['GR', [3, 10]]
```

Triangular types:

```
sage: QuiverMutationType('TR',2)
['A', 3]
sage: QuiverMutationType('TR',3)
['D', 6]
sage: QuiverMutationType('TR',4)
['E', 8, [1, 1]]
sage: QuiverMutationType('TR',5)
['TR', 5]
```

T types:

```
sage: QuiverMutationType('T',(1,1,1))
['A', 1]
sage: QuiverMutationType('T',(1,1,4))
['A', 4]
sage: QuiverMutationType('T',(1,4,4))
['A', 7]
sage: QuiverMutationType('T',(2,2,2))
['D', 4]
sage: QuiverMutationType('T',(2,2,4))
['D', 6]
sage: QuiverMutationType('T',(2,3,3))
['E', 6]
sage: QuiverMutationType('T',(2,3,4))
```

```
['E', 7]
sage: QuiverMutationType('T',(2,3,5))
['E', 8]
sage: QuiverMutationType('T',(2,3,6))
['E', 8, 1]
sage: QuiverMutationType('T',(2,3,7))
['E', 10]
sage: QuiverMutationType('T',(3,3,3))
['E', 6, 1]
sage: QuiverMutationType('T',(3,3,4))
['T', [3, 3, 4]]
```

Reducible types:

```
sage: QuiverMutationType(['A',3],['B',4])
[ ['A', 3], ['B', 4] ]
```

class sage.combinat.cluster_algebra_quiver.quiver_mutation_type.**QuiverMutationTypeFactory**

Bases: sage.structure.sage_object.SageObject

samples(*finite=None*, *affine=None*, *elliptic=None*, *mutation_finite=None*)

Return a sample of the available quiver mutations types.

INPUT:

- finite
- affine
- elliptic
- mutation_finite

All four input keywords default values are None. If set to True or False, only these samples are returned.

EXAMPLES:

```
sage: QuiverMutationType.samples()
[['A', 1], ['A', 5], ['B', 2], ['B', 5], ['C', 3],
 ['C', 5], [ ['A', 1], ['A', 1] ], ['D', 5], ['E', 6],
 ['E', 7], ['E', 8], ['F', 4], ['G', 2],
 ['A', [1, 1], 1], ['A', [4, 5], 1], ['D', 4, 1],
 ['BB', 5, 1], ['E', 6, [1, 1]], ['E', 7, [1, 1]],
 ['R2', [1, 5]], ['R2', [3, 5]], ['E', 10], ['BE', 5],
 ['GR', [3, 10]], ['T', [3, 3, 4]]]

sage: QuiverMutationType.samples(finite=True)
[['A', 1], ['A', 5], ['B', 2], ['B', 5], ['C', 3],
 ['C', 5], [ ['A', 1], ['A', 1] ], ['D', 5], ['E', 6],
 ['E', 7], ['E', 8], ['F', 4], ['G', 2]]

sage: QuiverMutationType.samples(affine=True)
[['A', [1, 1], 1], ['A', [4, 5], 1], ['D', 4, 1], ['BB', 5, 1]]

sage: QuiverMutationType.samples(elliptic=True)
[['E', 6, [1, 1]], ['E', 7, [1, 1]]]

sage: QuiverMutationType.samples(mutation_finite=False)
[['R2', [1, 5]], ['R2', [3, 5]], ['E', 10], ['BE', 5],
 ['GR', [3, 10]], ['T', [3, 3, 4]]]
```

class sage.combinat.cluster_algebra_quiver.quiver_mutation_type.**QuiverMutationType_Irreducibl**

Bases: sage.combinat.cluster_algebra_quiver.quiver_mutation_type.QuiverMutationType_abst

The mutation type for a cluster algebra or a quiver. Should not be called directly, but through QuiverMutationType.

class_size()

If it is known, the size of the mutation class of all quivers which are mutation equivalent to the standard quiver of `self` (up to isomorphism) is returned.

Otherwise, `NotImplemented` is returned.

Formula for finite type A is taken from Torkildsen - Counting cluster-tilted algebras of type A_n. Formulas for affine type A and finite type D are taken from Bastian, Prellberg, Rubey, Stump - Counting the number of elements in the mutation classes of $\widetilde{A}_n$ quivers. Formulas for finite and affine types B and C are proven but not yet published. Conjectural formulas for several other non-simply-laced affine types are implemented. Exceptional Types (finite, affine, and elliptic) E, F, G, and X are hardcoded.

EXAMPLES:

```
sage: mut_type = QuiverMutationType( ['A',5] ); mut_type
['A', 5]
sage: mut_type.class_size()
19

sage: mut_type = QuiverMutationType( ['A',[10,3],1] ); mut_type
['A', [3, 10], 1]
sage: mut_type.class_size()
142120

sage: mut_type = QuiverMutationType( ['B',6] ); mut_type
['B', 6]
sage: mut_type.class_size()
132

sage: mut_type = QuiverMutationType( ['BD',6,1] ); mut_type
['BD', 6, 1]
sage: mut_type.class_size()
Warning: This method uses a formula which has not been proved correct.
504
```

Check that trac ticket #14048 is fixed:

```
sage: mut_type = QuiverMutationType( ['F',4,(2,1)] )
sage: mut_type.class_size()
90
```

dual()

Return the QuiverMutationType which is dual to `self`.

EXAMPLES:

```
sage: mut_type = QuiverMutationType('A',5); mut_type
['A', 5]
sage: mut_type.dual()
['A', 5]

sage: mut_type = QuiverMutationType('B',5); mut_type
```

```
['B', 5]
sage: mut_type.dual()
['C', 5]
sage: mut_type.dual().dual()
['B', 5]
sage: mut_type.dual().dual() == mut_type
True
```

irreducible_components()

Return a list of all irreducible components of `self`.

EXAMPLES:

```
sage: mut_type = QuiverMutationType('A',3); mut_type
['A', 3]
sage: mut_type.irreducible_components()
(['A', 3],)
```

class sage.combinat.cluster_algebra_quiver.quiver_mutation_type.**QuiverMutationType_Reducible**(

Bases: sage.combinat.cluster_algebra_quiver.quiver_mutation_type.QuiverMutationType_abst

The mutation type for a cluster algebra or a quiver. Should not be called directly, but through QuiverMutationType. Inherits from QuiverMutationType_abstract.

class_size()

If it is known, the size of the mutation class of all quivers which are mutation equivalent to the standard quiver of `self` (up to isomorphism) is returned.

Otherwise, `NotImplemented` is returned.

EXAMPLES:

```
sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
[ ['A', 3], ['B', 3] ]
sage: mut_type.class_size()
20

sage: mut_type = QuiverMutationType(['A',3],['B',3],['X',6])
sage: mut_type
[ ['A', 3], ['B', 3], ['X', 6] ]
sage: mut_type.class_size()
100
```

dual()

Return the QuiverMutationType which is dual to `self`.

EXAMPLES:

```
sage: mut_type = QuiverMutationType(['A',5],['B',6],['C',5],['D',4]); mut_type
[ ['A', 5], ['B', 6], ['C', 5], ['D', 4] ]
sage: mut_type.dual()
[ ['A', 5], ['C', 6], ['B', 5], ['D', 4] ]
```

irreducible_components()

Return a list of all irreducible components of `self`.

EXAMPLES:

```
sage: mut_type = QuiverMutationType('A',3); mut_type
['A', 3]
sage: mut_type.irreducible_components()
(['A', 3],)
```

```
sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
[ ['A', 3], ['B', 3] ]
sage: mut_type.irreducible_components()
(['A', 3], ['B', 3])

sage: mut_type = QuiverMutationType(['A',3],['B',3],['X',6])
sage: mut_type
[ ['A', 3], ['B', 3], ['X', 6] ]
sage: mut_type.irreducible_components()
(['A', 3], ['B', 3], ['X', 6])
```

class sage.combinat.cluster_algebra_quiver.quiver_mutation_type.**QuiverMutationType_abstract**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.sage_object.SageObject

b_matrix()

Return the B-matrix of the standard quiver of `self`.

The conventions for B-matrices agree with Fomin-Zelevinsky (up to a reordering of the simple roots).

EXAMPLES:

```
sage: mut_type = QuiverMutationType( ['A',5] ); mut_type
['A', 5]
sage: mut_type.b_matrix()
[ 0  1  0  0  0]
[-1  0 -1  0  0]
[ 0  1  0  1  0]
[ 0  0 -1  0 -1]
[ 0  0  0  1  0]

sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
[ ['A', 3], ['B', 3] ]
sage: mut_type.b_matrix()
[ 0  1  0  0  0  0]
[-1  0 -1  0  0  0]
[ 0  1  0  0  0  0]
[ 0  0  0  0  1  0]
[ 0  0  0 -1  0 -1]
[ 0  0  0  0  2  0]
```

cartan_matrix()

Return the Cartan matrix of `self`.

Note that (up to a reordering of the simple roots) the convention for the definition of Cartan matrix, used here and elsewhere in Sage, agrees with the conventions of Kac, Fulton-Harris, and Fomin-Zelevinsky, but disagrees with the convention of Bourbaki. The (i, j) entry is $2(\alpha_i, \alpha_j)/(\alpha_i, \alpha_i)$.

EXAMPLES:

```
sage: mut_type = QuiverMutationType(['A',5]); mut_type
['A', 5]
sage: mut_type.cartan_matrix()
[ 2 -1  0  0  0]
[-1  2 -1  0  0]
[ 0 -1  2 -1  0]
[ 0  0 -1  2 -1]
[ 0  0  0 -1  2]

sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
```

```
[ ['A', 3], ['B', 3] ]
sage: mut_type.cartan_matrix()
[ 2 -1  0  0  0  0]
[-1  2 -1  0  0  0]
[ 0 -1  2  0  0  0]
[ 0  0  0  2 -1  0]
[ 0  0  0 -1  2 -1]
[ 0  0  0  0 -2  2]
```

is_affine()

Return `True` if `self` is of affine type.

EXAMPLES:

```
sage: mt = QuiverMutationType(['A',2])
sage: mt.is_affine()
False

sage: mt = QuiverMutationType(['A',[4,2],1])
sage: mt.is_affine()
True
```

is_elliptic()

Return `True` if `self` is of elliptic type.

EXAMPLES:

```
sage: mt = QuiverMutationType(['A',2])
sage: mt.is_elliptic()
False

sage: mt = QuiverMutationType(['E',6,[1,1]])
sage: mt.is_elliptic()
True
```

is_finite()

Return `True` if `self` is of finite type.

This means that the cluster algebra associated to `self` has only a finite number of cluster variables.

EXAMPLES:

```
sage: mt = QuiverMutationType(['A',2])
sage: mt.is_finite()
True

sage: mt = QuiverMutationType(['A',[4,2],1])
sage: mt.is_finite()
False
```

is_irreducible()

Return `True` if `self` is irreducible.

EXAMPLES:

```
sage: mt = QuiverMutationType(['A',2])
sage: mt.is_irreducible()
True
```

is_mutation_finite()

Return `True` if `self` is of finite mutation type.

This means that its mutation class has only finitely many different B-matrices.

EXAMPLES:

```
sage: mt = QuiverMutationType(['D',5,1])
sage: mt.is_mutation_finite()
True
```

is_simply_laced()

Return `True` if `self` is simply laced.

This means that the only arrows that appear in the quiver of `self` are single unlabelled arrows.

EXAMPLES:

```
sage: mt = QuiverMutationType(['A',2])
sage: mt.is_simply_laced()
True

sage: mt = QuiverMutationType(['B',2])
sage: mt.is_simply_laced()
False

sage: mt = QuiverMutationType(['A',(1,1),1])
sage: mt.is_simply_laced()
False
```

is_skew_symmetric()

Return `True` if the B-matrix of `self` is skew-symmetric.

EXAMPLES:

```
sage: mt = QuiverMutationType(['A',2])
sage: mt.is_skew_symmetric()
True

sage: mt = QuiverMutationType(['B',2])
sage: mt.is_skew_symmetric()
False

sage: mt = QuiverMutationType(['A',(1,1),1])
sage: mt.is_skew_symmetric()
True
```

letter()

Return the classification letter of `self`.

EXAMPLES:

```
sage: mut_type = QuiverMutationType( ['A',5] ); mut_type
['A', 5]
sage: mut_type.letter()
'A'

sage: mut_type = QuiverMutationType( ['BC',5,1] ); mut_type
['BC', 5, 1]
sage: mut_type.letter()
'BC'

sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
[ ['A', 3], ['B', 3] ]
sage: mut_type.letter()
```

```
'A x B'

sage: mut_type = QuiverMutationType(['A',3],['B',3],['X',6]); mut_type
[ ['A', 3], ['B', 3], ['X', 6] ]
sage: mut_type.letter()
'A x B x X'
```

plot(*circular=False*, *directed=True*)

Return the plot of the underlying graph or digraph of `self`.

INPUT:

- `circular` – (default:`False`) if `True`, the circular plot is chosen, otherwise >>spring<< is used.
- `directed` – (default: `True`) if `True`, the directed version is shown, otherwise the undirected.

EXAMPLES:

```
sage: QMT = QuiverMutationType(['A',5])
sage: pl = QMT.plot()
sage: pl = QMT.plot(circular=True)
```

properties()

Print a scheme of all properties of `self`.

Most properties have natural definitions for either irreducible or reducible types. `affine` and `elliptic` are only defined for irreducible types.

EXAMPLES:

```
sage: mut_type = QuiverMutationType(['A',3]); mut_type
['A', 3]
sage: mut_type.properties()
['A', 3] has rank 3 and the following properties:
    - irreducible:       True
    - mutation finite:   True
    - simply-laced:      True
    - skew-symmetric:    True
    - finite:            True
    - affine:            False
    - elliptic:          False

sage: mut_type = QuiverMutationType(['B',3]); mut_type
['B', 3]
sage: mut_type.properties()
['B', 3] has rank 3 and the following properties:
    - irreducible:       True
    - mutation finite:   True
    - simply-laced:      False
    - skew-symmetric:    False
    - finite:            True
    - affine:            False
    - elliptic:          False

sage: mut_type = QuiverMutationType(['B',3,1]); mut_type
['BD', 3, 1]
sage: mut_type.properties()
['BD', 3, 1] has rank 4 and the following properties:
    - irreducible:       True
    - mutation finite:   True
    - simply-laced:      False
```

```
    - skew-symmetric:    False
    - finite:            False
    - affine:            True
    - elliptic:          False

sage: mut_type = QuiverMutationType(['E',6,[1,1]]); mut_type
['E', 6, [1, 1]]
sage: mut_type.properties()
['E', 6, [1, 1]] has rank 8 and the following properties:
    - irreducible:       True
    - mutation finite:   True
    - simply-laced:      False
    - skew-symmetric:    True
    - finite:            False
    - affine:            False
    - elliptic:          True

sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
[ ['A', 3], ['B', 3] ]
sage: mut_type.properties()
[ ['A', 3], ['B', 3] ] has rank 6 and the following properties:
    - irreducible:       False
    - mutation finite:   True
    - simply-laced:      False
    - skew-symmetric:    False
    - finite:            True

sage: mut_type = QuiverMutationType('GR',[4,9]); mut_type
['GR', [4, 9]]
sage: mut_type.properties()
['GR', [4, 9]] has rank 12 and the following properties:
    - irreducible:       True
    - mutation finite:   False
    - simply-laced:      True
    - skew-symmetric:    True
    - finite:            False
    - affine:            False
    - elliptic:          False
```

rank()

Return the rank in the standard quiver of `self`.

The rank is the number of vertices.

EXAMPLES:

```
sage: mut_type = QuiverMutationType( ['A',5] ); mut_type
['A', 5]
sage: mut_type.rank()
5

sage: mut_type = QuiverMutationType( ['A',[4,5],1] ); mut_type
['A', [4, 5], 1]
sage: mut_type.rank()
9

sage: mut_type = QuiverMutationType( ['BC',5,1] ); mut_type
['BC', 5, 1]
sage: mut_type.rank()
```

```
6

sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
[ ['A', 3], ['B', 3] ]
sage: mut_type.rank()
6

sage: mut_type = QuiverMutationType(['A',3],['B',3],['X',6]); mut_type
[ ['A', 3], ['B', 3], ['X', 6] ]
sage: mut_type.rank()
12
```

show(*circular=False*, *directed=True*)

Show the plot of the underlying digraph of `self`.

INPUT:

- `circular` – (default:`False`) if `True`, the circular plot is chosen, otherwise >>spring<< is used.
- `directed` – (default: `True`) if `True`, the directed version is shown, otherwise the undirected.

TESTS:

```
sage: QMT = QuiverMutationType(['A',5])
sage: QMT.show() # long time
```

standard_quiver()

Return the standard quiver of `self`.

EXAMPLES:

```
sage: mut_type = QuiverMutationType( ['A',5] ); mut_type
['A', 5]
sage: mut_type.standard_quiver()
Quiver on 5 vertices of type ['A', 5]

sage: mut_type = QuiverMutationType( ['A',[5,3],1] ); mut_type
['A', [3, 5], 1]
sage: mut_type.standard_quiver()
Quiver on 8 vertices of type ['A', [3, 5], 1]

sage: mut_type = QuiverMutationType(['A',3],['B',3]); mut_type
[ ['A', 3], ['B', 3] ]
sage: mut_type.standard_quiver()
Quiver on 6 vertices of type [ ['A', 3], ['B', 3] ]

sage: mut_type = QuiverMutationType(['A',3],['B',3],['X',6]); mut_type
[ ['A', 3], ['B', 3], ['X', 6] ]
sage: mut_type.standard_quiver()
Quiver on 12 vertices of type [ ['A', 3], ['B', 3], ['X', 6] ]
```

sage.combinat.cluster_algebra_quiver.quiver_mutation_type.**save_quiver_data**(*n*, *up_to=True*, *types='ClassicalExcep*
ver-
bose=True)

Save mutation classes of certain quivers of ranks up to and equal to `n` or equal to `n` to `DOT_SAGE/cluster_algebra_quiver/mutation_classes_n.dig6`.

This data will then be used to determine quiver mutation types.

INPUT:

- n: the rank (or the upper limit on the rank) of the mutation classes that are being saved.
- up_to – (default:True) if True, saves data for ranks smaller than or equal to n. If False, saves data for rank exactly n.
- types – (default:'ClassicalExceptional') if all, saves data for both exceptional mutation-finite quivers and for classical quiver. The input 'Exceptional' or 'Classical' is also allowed to save only part of this data.

TESTS:

```
sage: from sage.combinat.cluster_algebra_quiver.quiver_mutation_type import save_quiver_data
sage: save_quiver_data(2)

The following types are saved to file ... and will now be used to determine quiver mutation type
[('A', 1)]

The following types are saved to file ... and will now be used to determine quiver mutation type
[('A', (1, 1), 1), ('A', 2), ('B', 2), ('BC', 1, 1), ('G', 2)]

sage: save_quiver_data(2,up_to=False)

The following types are saved to file ... and will now be used to determine quiver mutation type
[('A', (1, 1), 1), ('A', 2), ('B', 2), ('BC', 1, 1), ('G', 2)]

sage: save_quiver_data(2,up_to=False, types='Classical')

The following types are saved to file ... and will now be used to determine quiver mutation type
[('A', (1, 1), 1), ('A', 2), ('B', 2), ('BC', 1, 1)]

sage: save_quiver_data(2,up_to=False, types='Exceptional')

The following types are saved to file ... and will now be used to determine quiver mutation type
[('G', 2)]

sage: save_quiver_data(2,up_to=False, verbose=False)
```

5.1.20 Cluster complex (or generalized dual associahedron)

EXAMPLES:

A first example of a cluster complex:

```
sage: C = ClusterComplex(['A', 2]); C
Cluster complex of type ['A', 2] with 5 vertices and 5 facets
```

Its vertices, facets, and minimal non-faces:

```
sage: C.vertices()
(0, 1, 2, 3, 4)

sage: C.facets()
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]

sage: for F in C.facets(): F.cluster()
[(-1, 0), (0, -1)]
[(-1, 0), (0, 1)]
[(0, -1), (1, 0)]
```

```
[(1, 0), (1, 1)]
[(1, 1), (0, 1)]

sage: C.minimal_nonfaces()
[[0, 2], [0, 3], [1, 3], [1, 4], [2, 4]]
```

We can do everything we can do on simplicial complexes, e.g. computing its homology:

```
sage: C.homology()
{0: 0, 1: Z}
```

AUTHORS:

- Christian Stump (2011) Initial version

class sage.combinat.cluster_complex.**ClusterComplex**(*W*, *k*, *coxeter_element*, *algorithm*)

Bases: sage.combinat.subword_complex.SubwordComplex

A cluster complex (or generalized dual associahedron).

The cluster complex (or generalized dual associahedron) is a simplicial complex constructed from a cluster algebra. Its vertices are the cluster variables and its facets are the clusters, i.e., maximal subsets of compatible cluster variables.

The cluster complex of type A_n is the simplicial complex with vertices being (proper) diagonals in a convex $(n+3)$-gon and with facets being triangulations.

The implementation of the cluster complex depends on its connection to subword complexes, see [CLS]. Let c be a Coxeter element with reduced word $\mathbf{c}$ in a finite Coxeter group W, and let $\mathbf{w}_\circ$ be the c-sorting word for the longest element $w_\circ \in W$.

The multi-cluster complex $\Delta(W, k)$ has vertices in one-to-one correspondence with letters in the word $Q = \mathbf{c}^{\mathbf{k}}\mathbf{w}_\circ$ and with facets being complements in Q of reduced expressions for $w_\circ$.

For $k = 1$, the multi-cluster complex is isomorphic to the cluster complex as defined above.

EXAMPLES:

A first example of a cluster complex:

```
sage: C = ClusterComplex(['A', 2]); C
Cluster complex of type ['A', 2] with 5 vertices and 5 facets
```

Its vertices, facets, and minimal non-faces:

```
sage: C.vertices()
(0, 1, 2, 3, 4)

sage: C.facets()
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]

sage: C.minimal_nonfaces()
[[0, 2], [0, 3], [1, 3], [1, 4], [2, 4]]
```

We can do everything we can do on simplicial complexes, e.g. computing its homology:

```
sage: C.homology()
{0: 0, 1: Z}
```

We can also create a multi-cluster complex:

```
sage: ClusterComplex(['A', 2], k=2)
Multi-cluster complex of type ['A', 2] with 7 vertices and 14 facets
```

REFERENCES:

Element
alias of ClusterComplexFacet

cyclic_rotation()
Return the operation on the facets of self obtained by the cyclic rotation as defined in _[CLS].

EXAMPLES:

```
sage: ClusterComplex(['A', 2]).cyclic_rotation()
<function act at ...>
```

k()
Return the index k of self.

EXAMPLES:

```
sage: ClusterComplex(['A', 2]).k()
1
```

minimal_nonfaces()
Return the minimal non-faces of self.

EXAMPLES:

```
sage: ClusterComplex(['A', 2]).minimal_nonfaces()
[[0, 2], [0, 3], [1, 3], [1, 4], [2, 4]]
```

class sage.combinat.cluster_complex.**ClusterComplexFacet**(*parent*, *positions*, *facet_test=True*)
Bases: sage.combinat.subword_complex.SubwordComplexFacet

A cluster (i.e., a facet) of a cluster complex.

cluster()
Return this cluster as a set of almost positive roots.

EXAMPLES:

```
sage: C = ClusterComplex(['A', 2])
sage: F = C((0, 1))
sage: F.cluster()
[(-1, 0), (0, -1)]
```

product_of_upper_cluster()
Return the product of the upper cluster in reversed order.

EXAMPLES:

```
sage: C = ClusterComplex(['A', 2])
sage: for F in C: F.product_of_upper_cluster().reduced_word()
[]
[2]
[1]
[1, 2]
[1, 2]
```

upper_cluster()
Return the part of the cluster that contains positive roots

EXAMPLES:

```
sage: C = ClusterComplex(['A', 2])
sage: F = C((0, 1))
sage: F.upper_cluster()
[]
```

5.1.21 Colored Permutations

Todo

Much of the colored permutations (and element) class can be generalized to $G \wr S_n$

class sage.combinat.colored_permutations.**ColoredPermutation**(*parent*, *colors*, *perm*)

Bases: sage.structure.element.MultiplicativeGroupElement

A colored permutation.

colors()

Return the colors of self.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: s1,s2,t = C.gens()
sage: x = s1*s2*t
sage: x.colors()
[1, 0, 0]
```

inverse()

Return the inverse of self.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: s1,s2,t = C.gens()
sage: ~t
[[0, 0, 3], [1, 2, 3]]
sage: all(x * ~x == C.one() for x in C.gens())
True
```

one_line_form()

Return the one line form of self.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: s1,s2,t = C.gens()
sage: x = s1*s2*t
sage: x
[[1, 0, 0], [3, 1, 2]]
sage: x.one_line_form()
[(1, 3), (0, 1), (0, 2)]
```

permutation()

Return the permutation of self.

This is obtained by forgetting the colors.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: s1,s2,t = C.gens()
sage: x = s1*s2*t
sage: x.permutation()
[3, 1, 2]
```

to_matrix()

Return a matrix of `self`.

The colors are mapped to roots of unity.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: s1,s2,t = C.gens()
sage: x = s1*s2*t*s2; x.one_line_form()
[(1, 2), (0, 1), (0, 3)]
sage: M = x.to_matrix(); M
[    0     1     0]
[zeta4     0     0]
[    0     0     1]
```

The matrix multiplication is in the *opposite* order:

```
sage: M == s2.to_matrix()*t.to_matrix()*s2.to_matrix()*s1.to_matrix()
True
```

class sage.combinat.colored_permutations.**ColoredPermutations**(*m*, *n*, *category=None*)

Bases: sage.structure.parent.Parent, sage.structure.unique_representation.UniqueRepresenta

The group of m-colored permutations on $\{1, 2, \ldots, n\}$.

Let S_n be the symmetric group on n letters and C_m be the cyclic group of order m. The m-colored permutation group on n letters is given by $P_n^m = C_m \wr S_n$. This is also the complex reflection group $G(m, 1, n)$.

We define our multiplication by

$$((s_1, \ldots s_n), \sigma) \cdot ((t_1, \ldots, t_n), \tau) = ((s_1 t_{\sigma(1)}, \ldots, s_n t_{\sigma(n)}), \tau\sigma).$$

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3); C
4-colored permutations of size 3
sage: s1,s2,t = C.gens()
sage: (s1, s2, t)
([[0, 0, 0], [2, 1, 3]], [[0, 0, 0], [1, 3, 2]], [[0, 0, 1], [1, 2, 3]])
sage: s1*s2
[[0, 0, 0], [3, 1, 2]]
sage: s1*s2*s1 == s2*s1*s2
True
sage: t^4 == C.one()
True
sage: s2*t*s2
[[0, 1, 0], [1, 2, 3]]
```

We can also create a colored permutation by passing either a list of tuples consisting of `(color, element)`:

```
sage: x = C([(2,1), (3,3), (3,2)]); x
[[2, 3, 3], [1, 3, 2]]
```

or a list of colors and a permutation:

```
sage: C([[3,3,1], [1,3,2]])
[[3, 3, 1], [1, 3, 2]]
```

There is also the natural lift from permutations:

```
sage: P = Permutations(3)
sage: C(P.an_element())
[[0, 0, 0], [3, 1, 2]]
```

REFERENCES:

- Wikipedia article Generalized_symmetric_group
- Wikipedia article Complex_reflection_group

Element
alias of ColoredPermutation

cardinality()
Return the cardinality of self.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.cardinality()
384
sage: C.cardinality() == 4**3 * factorial(3)
True
```

codegrees()
Return the codegrees of self.

Let G be a complex reflection group. The codegrees $d_1^* \leq d_2^* \leq \cdots \leq d_\ell^*$ of G can be defined by:

$$\prod_{i=1}^{\ell}(q - d_i^* - 1) = \sum_{g \in G} \det(g) q^{\dim(V^g)},$$

where V is the natural complex vector space that G acts on and ℓ is the rank().

If $m = 1$, then we are in the special case of the symmetric group and the codegrees are $(n-2, n-3, \ldots 1, 0)$. Otherwise the degrees are $((n-1)m, (n-2)m, \ldots, m, 0)$.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.codegrees()
[8, 4, 0]
sage: S = ColoredPermutations(1, 3)
sage: S.codegrees()
[1, 0]
```

TESTS:

We check the polynomial identity:

```
sage: R.<q> = ZZ[]
sage: C = ColoredPermutations(3, 2)
sage: f = prod(q - ds - 1 for ds in C.codegrees())
sage: d = lambda x: sum(1 for e in x.to_matrix().eigenvalues() if e == 1)
sage: g = sum(det(x.to_matrix()) * q**d(x) for x in C)
sage: f == g
True
```

degrees()

Return the degrees of `self`.

The degrees of a complex reflection group are the degrees of the fundamental invariants of the ring of polynomial invariants.

If $m = 1$, then we are in the special case of the symmetric group and the degrees are $(2, 3, \ldots, n, n+1)$. Otherwise the degrees are $(m, 2m, \ldots, nm)$.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.degrees()
[4, 8, 12]
sage: S = ColoredPermutations(1, 3)
sage: S.degrees()
[2, 3]
```

We now check that the product of the degrees is equal to the cardinality of `self`:

```
sage: prod(C.degrees()) == C.cardinality()
True
sage: prod(S.degrees()) == S.cardinality()
True
```

fixed_point_polynomial(*q=None*)

The fixed point polynomial of `self`.

The fixed point polynomial f_G of a complex reflection group G is counting the dimensions of fixed points subspaces:

$$f_G(q) = \sum_{w \in W} q^{\dim V^w}.$$

Furthermore, let $d_1, d_2, \ldots, d_\ell$ be the degrees of G, where ℓ is the `rank()`. Then the fixed point polynomial is given by

$$f_G(q) = \prod_{i=1}^{\ell} (q + d_i - 1).$$

INPUT:

- `q` – (default: the generator of `ZZ['q']`) the parameter q

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.fixed_point_polynomial()
q^3 + 21*q^2 + 131*q + 231

sage: S = ColoredPermutations(1, 3)
sage: S.fixed_point_polynomial()
q^2 + 3*q + 2
```

TESTS:

We check the against the degrees and codegrees:

```
sage: R.<q> = ZZ[]
sage: C = ColoredPermutations(4, 3)
sage: C.fixed_point_polynomial(q) == prod(q + d - 1 for d in C.degrees())
True
```

gens()
Return the generators of `self`.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.gens()
([[0, 0, 0], [2, 1, 3]],
 [[0, 0, 0], [1, 3, 2]],
 [[0, 0, 1], [1, 2, 3]])
```

is_well_generated()
Return if `self` is a well-generated complex reflection group.

A complex reflection group G is well-generated if it is generated by ℓ reflections. Equivalently, G is well-generated if $d_i + d_i^* = d_\ell$ for all $1 \leq i \leq \ell$.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.is_well_generated()
True
sage: C = ColoredPermutations(2, 8)
sage: C.is_well_generated()
True
sage: C = ColoredPermutations(1, 4)
sage: C.is_well_generated()
True
```

matrix_group()
Return the matrix group corresponding to `self`.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.matrix_group()
Matrix group over Cyclotomic Field of order 4 and degree 2 with 3 generators (
[0 1 0]  [1 0 0]  [    1     0     0]
[1 0 0]  [0 0 1]  [    0     1     0]
[0 0 1], [0 1 0], [    0     0 zeta4]
)
```

number_of_reflection_hyperplanes()
Return the number of reflection hyperplanes of `self`.

The number of reflection hyperplanes of a complex reflection group is equal to the sum of the codegrees plus the rank.

EXAMPLES:

```
sage: C = ColoredPermutations(1, 2)
sage: C.number_of_reflection_hyperplanes()
1
sage: C = ColoredPermutations(1, 3)
sage: C.number_of_reflection_hyperplanes()
3
sage: C = ColoredPermutations(4, 12)
sage: C.number_of_reflection_hyperplanes()
276
```

one()
Return the identity element of `self`.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 3)
sage: C.one()
[[0, 0, 0], [1, 2, 3]]
```

rank()

Return the rank of `self`.

The rank of a complex reflection group is equal to the dimension of the complex vector space the group acts on.

EXAMPLES:

```
sage: C = ColoredPermutations(4, 12)
sage: C.rank()
12
sage: C = ColoredPermutations(7, 4)
sage: C.rank()
4
sage: C = ColoredPermutations(1, 4)
sage: C.rank()
3
```

class sage.combinat.colored_permutations.**SignedPermutation**(*parent*, *colors*, *perm*)

Bases: sage.combinat.colored_permutations.ColoredPermutation

A signed permutation.

has_left_descent(*i*)

Return `True` if `i` is a left descent of `self`.

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: s1,s2,s3,s4 = S.gens()
sage: x = s4*s1*s2*s3*s4
sage: [x.has_left_descent(i) for i in S.index_set()]
[True, False, False, True]
```

inverse()

Return the inverse of `self`.

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: s1,s2,s3,s4 = S.gens()
sage: x = s4*s1*s2*s3*s4
sage: ~x
[2, 3, -4, -1]
sage: x * ~x == S.one()
True
```

to_matrix()

Return a matrix of `self`.

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: s1,s2,s3,s4 = S.gens()
sage: x = s4*s1*s2*s3*s4
sage: M = x.to_matrix(); M
[ 0  1  0  0]
```

```
[ 0  0  1  0]
[ 0  0  0 -1]
[-1  0  0  0]
```

The matrix multiplication is in the *opposite* order:

```
sage: m1,m2,m3,m4 = [g.to_matrix() for g in S.gens()]
sage: M == m4 * m3 * m2 * m1 * m4
True
```

class sage.combinat.colored_permutations.**SignedPermutations**(*n*)

Bases: sage.combinat.colored_permutations.ColoredPermutations

Group of signed permutations.

The group of signed permutations is also known as the hyperoctahedral group, the Coxeter group of type B_n, and the 2-colored permutation group. Thus it can be constructed as the wreath product $S_2 \wr S_n$.

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: s1,s2,s3,s4 = S.group_generators()
sage: x = s4*s1*s2*s3*s4; x
[-4, 1, 2, -3]
sage: x^4 == S.one()
True
```

This is a finite Coxeter group of type B_n:

```
sage: S.canonical_representation()
Finite Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[1 3 2 2]
[3 1 3 2]
[2 3 1 4]
[2 2 4 1]
sage: S.long_element()
[-4, -3, -2, -1]
sage: S.long_element().reduced_word()
[4, 3, 4, 2, 3, 4, 1, 2, 3, 4]
```

We can also go between the 2-colored permutation group:

```
sage: C = ColoredPermutations(2, 3)
sage: S = SignedPermutations(3)
sage: S.an_element()
[-3, 1, 2]
sage: C(S.an_element())
[[1, 0, 0], [3, 1, 2]]
sage: S(C(S.an_element())) == S.an_element()
True
sage: S(C.an_element())
[1, 2, 3]
```

There is also the natural lift from permutations:

```
sage: P = Permutations(3)
sage: x = S(P.an_element()); x
[3, 1, 2]
sage: x.parent()
Signed permutations of 3
```

REFERENCES:

- Wikipedia article Hyperoctahedral_group

Element
alias of SignedPermutation

coxeter_matrix()
Return the Coxeter matrix of self.

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: S.coxeter_matrix()
[1 3 2 2]
[3 1 3 2]
[2 3 1 4]
[2 2 4 1]
```

gens()
Return the generators of self.

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: S.gens()
([2, 1, 3, 4], [1, 3, 2, 4], [1, 2, 4, 3], [1, 2, 3, -4])
```

index_set()
Return the index set of self.

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: S.index_set()
(1, 2, 3, 4)
```

long_element(*index_set=None*)
Return the longest element of self, or of the parabolic subgroup corresponding to the given index_set.

INPUT:

- index_set – (optional) a subset (as a list or iterable) of the nodes of the indexing set

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: S.long_element()
[-4, -3, -2, -1]
```

one()
Return the identity element of self.

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: S.one()
[1, 2, 3, 4]
```

simple_reflection(*i*)
Return the i-th simple reflection of self.

EXAMPLES:

```
sage: S = SignedPermutations(4)
sage: S.simple_reflection(1)
[2, 1, 3, 4]
sage: S.simple_reflection(4)
[1, 2, 3, -4]
```

5.1.22 Combinatorial Functions

This module implements some combinatorial functions, as listed below. For a more detailed description, see the relevant docstrings.

Sequences:

- Bell numbers, `bell_number()`
- Catalan numbers, `catalan_number()` (not to be confused with the Catalan constant)
- Eulerian/Euler numbers, `euler_number()` (Maxima)
- Fibonacci numbers, `fibonacci()` (PARI) and `fibonacci_number()` (GAP) The PARI version is better.
- Lucas numbers, `lucas_number1()`, `lucas_number2()`.
- Stirling numbers, `stirling_number1()`, `stirling_number2()`.

Set-theoretic constructions:

- Derangements of a multiset, `derangements()` and `number_of_derangements()`.
- Tuples of a multiset, `tuples()` and `number_of_tuples()`. An ordered tuple of length k of set S is a ordered selection with repetitions of S and is represented by a sorted list of length k containing elements from S.
- Unordered tuples of a set, `unordered_tuples()` and `number_of_unordered_tuples()`. An unordered tuple of length k of set S is an unordered selection with repetitions of S and is represented by a sorted list of length k containing elements from S.

Warning: The following function is deprecated and will soon be removed.

- Permutations of a multiset, `permutations()`, `permutations_iterator()`, `number_of_permutations()`. A permutation is a list that contains exactly the same elements but possibly in different order.

Related functions:

- Bernoulli polynomials, `bernoulli_polynomial()`

Implemented in other modules (listed for completeness):

The `sage.arith.all` module contains the following combinatorial functions:

- binomial the binomial coefficient (wrapped from PARI)
- factorial (wrapped from PARI)
- partition (from the Python Cookbook) Generator of the list of all the partitions of the integer n.
- `number_of_partitions()` (wrapped from PARI) the *number* of partitions:
- `falling_factorial()` Definition: for integer $a \geq 0$ we have $x(x-1)\cdots(x-a+1)$. In all other cases we use the GAMMA-function: $\frac{\Gamma(x+1)}{\Gamma(x-a+1)}$.

- `rising_factorial()` Definition: for integer $a \geq 0$ we have $x(x+1)\cdots(x+a-1)$. In all other cases we use the GAMMA-function: $\frac{\Gamma(x+a)}{\Gamma(x)}$.
- gaussian_binomial the gaussian binomial

$$\binom{n}{k}_q = \frac{(1-q^m)(1-q^{m-1})\cdots(1-q^{m-r+1})}{(1-q)(1-q^2)\cdots(1-q^r)}.$$

The `sage.groups.perm_gps.permgroup_elements` contains the following combinatorial functions:

- matrix method of PermutationGroupElement yielding the permutation matrix of the group element.

Todo

GUAVA commands:

- VandermondeMat
- GrayMat returns a list of all different vectors of length n over the field F, using Gray ordering.

Not in GAP:

- Rencontres numbers http://en.wikipedia.org/wiki/Rencontres_number

---

REFERENCES:

- http://en.wikipedia.org/wiki/Twelvefold_way (general reference)

AUTHORS:

- David Joyner (2006-07): initial implementation.
- William Stein (2006-07): editing of docs and code; many optimizations, refinements, and bug fixes in corner cases
- David Joyner (2006-09): bug fix for combinations, added permutations_iterator, combinations_iterator from Python Cookbook, edited docs.
- David Joyner (2007-11): changed permutations, added hadamard_matrix
- Florent Hivert (2009-02): combinatorial class cleanup
- Fredrik Johansson (2010-07): fast implementation of `stirling_number2`
- Punarbasu Purkayastha (2012-12): deprecate arrangements, combinations, combinations_iterator, and clean up very old deprecated methods.

Functions and classes

class `sage.combinat.combinat.`**`CombinatorialClass`**(*category=None*)
Bases: `sage.structure.parent.Parent`

This class is deprecated, and will disappear as soon as all derived classes in Sage's library will have been fixed. Please derive directly from Parent and use the category `EnumeratedSets`, `FiniteEnumeratedSets`, or `InfiniteEnumeratedSets`, as appropriate.

For examples, see:

```
sage: FiniteEnumeratedSets().example()
An example of a finite enumerated set: {1,2,3}
sage: InfiniteEnumeratedSets().example()
An example of an infinite enumerated set: the non negative integers
```

Element
alias of CombinatorialObject

cardinality()
Default implementation of cardinality which just goes through the iterator of the combinatorial class to count the number of objects.

EXAMPLES:
```
sage: class C(CombinatorialClass):
...     def __iter__(self):
...         return iter([1,2,3])
...
sage: C().cardinality() #indirect doctest
3
```

element_class()
This function is a temporary helper so that a CombinatorialClass behaves as a parent for creating elements. This will disappear when combinatorial classes will be turned into actual parents (in the category EnumeratedSets).

TESTS:
```
sage: P5 = Partitions(5)
sage: P5.element_class
<class 'sage.combinat.partition.Partitions_n_with_category.element_class'>
```

filter(*f*, *name=None*)
Returns the combinatorial subclass of f which consists of the elements x of self such that f(x) is True.

EXAMPLES:
```
sage: from sage.combinat.combinat import Permutations_CC
sage: P = Permutations_CC(3).filter(lambda x: x.avoids([1,2]))
sage: P.list()
[[3, 2, 1]]
```

first()
Default implementation for first which uses iterator.

EXAMPLES:
```
sage: C = CombinatorialClass()
sage: C.list = lambda: [1,2,3]
sage: C.first() # indirect doctest
1
```

is_finite()
Returns whether self is finite or not.

EXAMPLES:
```
sage: Partitions(5).is_finite()
True
sage: Permutations().is_finite()
False
```

last()
Default implementation for first which uses iterator.

EXAMPLES:

```
sage: C = CombinatorialClass()
sage: C.list = lambda: [1,2,3]
sage: C.last() # indirect doctest
3
```

list()

The default implementation of list which builds the list from the iterator.

EXAMPLES:

```
sage: class C(CombinatorialClass):
...       def __iter__(self):
...           return iter([1,2,3])
...
sage: C().list() #indirect doctest
[1, 2, 3]
```

map(*f*, *name=None*)

Returns the image $\{f(x)|x \in \text{self}\}$ of this combinatorial class by f, as a combinatorial class.

f is supposed to be injective.

EXAMPLES:

```
sage: R = Permutations(3).map(attrcall('reduced_word')); R
Image of Standard permutations of 3 by *.reduced_word()
sage: R.cardinality()
6
sage: R.list()
[[], [2], [1], [1, 2], [2, 1], [2, 1, 2]]
sage: [ r for r in R]
[[], [2], [1], [1, 2], [2, 1], [2, 1, 2]]

If the function is not injective, then there may be repeated elements:
sage: P = Partitions(4)
sage: P.list()
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
sage: P.map(len).list()
[1, 2, 2, 3, 4]
```

TESTS:

```
sage: R = Permutations(3).map(attrcall('reduced_word'))
sage: R == loads(dumps(R))
True
```

next(*obj*)

Default implementation for next which uses iterator.

EXAMPLES:

```
sage: C = CombinatorialClass()
sage: C.list = lambda: [1,2,3]
sage: C.next(2) # indirect doctest
3
```

previous(*obj*)

Default implementation for next which uses iterator.

EXAMPLES:

```
sage: C = CombinatorialClass()
sage: C.list = lambda: [1,2,3]
sage: C.previous(2) # indirect doctest
1
```

random_element()
Default implementation of random which uses unrank.

EXAMPLES:

```
sage: C = CombinatorialClass()
sage: C.list = lambda: [1,2,3]
sage: C.random_element()        # indirect doctest
1
```

rank(*obj*)
Default implementation of rank which uses iterator.

EXAMPLES:

```
sage: C = CombinatorialClass()
sage: C.list = lambda: [1,2,3]
sage: C.rank(3) # indirect doctest
2
```

union(*right_cc*, *name=None*)
Returns the combinatorial class representing the union of self and right_cc.

EXAMPLES:

```
sage: from sage.combinat.combinat import Permutations_CC
sage: P = Permutations_CC(2).union(Permutations_CC(1))
sage: P.list()
[[1, 2], [2, 1], [1]]
```

unrank(*r*)
Default implementation of unrank which goes through the iterator.

EXAMPLES:

```
sage: C = CombinatorialClass()
sage: C.list = lambda: [1,2,3]
sage: C.unrank(1) # indirect doctest
2
```

class sage.combinat.combinat.**CombinatorialElement**(*parent*, **args*, ***kwds*)
Bases: sage.combinat.combinat.CombinatorialObject, sage.structure.element.Element

CombinatorialElement is both a CombinatorialObject and an Element. So it represents a list which is an element of some parent.

A CombinatorialElement subclass also automatically supports the __classcall__ mechanism.

Warning: This class is slowly being deprecated. Use ClonableList instead.

INPUT:

- parent – the Parent class for this element.
- lst – a list or any object that can be converted to a list by calling list().

EXAMPLES:

```
sage: from sage.combinat.combinat import CombinatorialElement
sage: e = CombinatorialElement(Partitions(6), [3,2,1])
sage: e == loads(dumps(e))
True
sage: parent(e)
Partitions of the integer 6
sage: list(e)
[3, 2, 1]
```

Check classcalls:

```
sage: class Foo(CombinatorialElement):
....:     @staticmethod
....:     def __classcall__(cls, x):
....:         return x
sage: Foo(17)
17
```

class sage.combinat.combinat.**CombinatorialObject**(*l*, *copy=True*)

Bases: sage.structure.sage_object.SageObject

CombinatorialObject provides a thin wrapper around a list. The main differences are that __setitem__ is disabled so that CombinatorialObjects are shallowly immutable, and the intention is that they are semantically immutable.

Because of this, CombinatorialObjects provide a __hash__ function which computes the hash of the string representation of a list and the hash of its parent's class. Thus, each CombinatorialObject should have a unique string representation.

See also:

CombinatorialElement if you want a combinatorial object which is an element of a parent.

Warning: This class is slowly being deprecated. Use ClonableList instead.

INPUT:

- l – a list or any object that can be converted to a list by calling list().
- copy – (boolean, default True) if False, then l must be a list, which is assigned to self._list without copying.

EXAMPLES:

```
sage: c = CombinatorialObject([1,2,3])
sage: c == loads(dumps(c))
True
sage: c._list
[1, 2, 3]
sage: c._hash is None
True
```

For efficiency, you can specify copy=False if you know what you are doing:

```
sage: from sage.combinat.combinat import CombinatorialObject
sage: x = [3, 2, 1]
sage: C = CombinatorialObject(x, copy=False)
sage: C
[3, 2, 1]
sage: x[0] = 5
```

```
sage: C
[5, 2, 1]
```

TESTS:

Test indirectly that we copy the input (see trac ticket #18184):

```
sage: L = IntegerListsLex(element_class=Partition)
sage: x = [3, 2, 1]
sage: P = L(x)
sage: x[0] = 5
sage: list(P)
[3, 2, 1]
```

index(*key*)

EXAMPLES:

```
sage: c = CombinatorialObject([1,2,3])
sage: c.index(1)
0
sage: c.index(3)
2
```

class sage.combinat.combinat.**FilteredCombinatorialClass**(*combinatorial_class*, *f*, *name=None*)

Bases: sage.combinat.combinat.CombinatorialClass

A filtered combinatorial class F is a subset of another combinatorial class C specified by a function f that takes in an element c of C and returns True if and only if c is in F.

TESTS:

```
sage: from sage.combinat.combinat import Permutations_CC
sage: Permutations_CC(3).filter(lambda x: x.avoids([1,2]))
Filtered subclass of Standard permutations of 3
```

cardinality()

EXAMPLES:

```
sage: from sage.combinat.combinat import Permutations_CC
sage: P = Permutations_CC(3).filter(lambda x: x.avoids([1,2]))
sage: P.cardinality()
1
```

class sage.combinat.combinat.**InfiniteAbstractCombinatorialClass**(*category=None*)

Bases: sage.combinat.combinat.CombinatorialClass

This is an internal class that should not be used directly. A class which inherits from InfiniteAbstractCombinatorialClass inherits the standard methods list and count.

If self._infinite_cclass_slice exists then self.__iter__ returns an iterator for self, otherwise raise NotImplementedError. The method self._infinite_cclass_slice is supposed to accept any integer as an argument and return something which is iterable.

cardinality()

Counts the elements of the combinatorial class.

EXAMPLES:

```
sage: R = InfiniteAbstractCombinatorialClass()
sage: R.cardinality()
+Infinity
```

list()
Returns an error since self is an infinite combinatorial class.

EXAMPLES:

```
sage: R = InfiniteAbstractCombinatorialClass()
sage: R.list()
Traceback (most recent call last):
...
NotImplementedError: infinite list
```

class sage.combinat.combinat.**MapCombinatorialClass**(*cc*, *f*, *name=None*)
Bases: sage.combinat.combinat.CombinatorialClass

A MapCombinatorialClass models the image of a combinatorial class through a function which is assumed to be injective

See CombinatorialClass.map for examples

an_element()
Returns an element of this combinatorial class

EXAMPLES:

```
sage: R = SymmetricGroup(10).map(attrcall('reduced_word'))
sage: R.an_element()
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

cardinality()
Returns the cardinality of this combinatorial class

EXAMPLES:

```
sage: R = Permutations(10).map(attrcall('reduced_word'))
sage: R.cardinality()
3628800
```

class sage.combinat.combinat.**Permutations_CC**(*n*)
Bases: sage.combinat.combinat.CombinatorialClass

A testing class for CombinatorialClass since Permutations no longer inherits from CombinatorialClass in trac ticket #14772.

class sage.combinat.combinat.**UnionCombinatorialClass**(*left_cc*, *right_cc*, *name=None*)
Bases: sage.combinat.combinat.CombinatorialClass

A UnionCombinatorialClass is a union of two other combinatorial classes.

TESTS:

```
sage: from sage.combinat.combinat import Permutations_CC
sage: P = Permutations_CC(3).union(Permutations_CC(2))
sage: P == loads(dumps(P))
True
```

cardinality()
EXAMPLES:

```
sage: from sage.combinat.combinat import Permutations_CC
sage: P = Permutations_CC(3).union(Permutations_CC(2))
sage: P.cardinality()
8
```

first()

EXAMPLES:

```
sage: from sage.combinat.combinat import Permutations_CC
sage: P = Permutations_CC(3).union(Permutations_CC(2))
sage: P.first()
[1, 2, 3]
```

last()

EXAMPLES:

```
sage: from sage.combinat.combinat import Permutations_CC
sage: P = Permutations_CC(3).union(Permutations_CC(2))
sage: P.last()
[2, 1]
```

list()

EXAMPLES:

```
sage: from sage.combinat.combinat import Permutations_CC
sage: P = Permutations_CC(3).union(Permutations_CC(2))
sage: P.list()
[[1, 2, 3],
 [1, 3, 2],
 [2, 1, 3],
 [2, 3, 1],
 [3, 1, 2],
 [3, 2, 1],
 [1, 2],
 [2, 1]]
```

rank(*x*)

EXAMPLES:

```
sage: from sage.combinat.combinat import Permutations_CC
sage: P = Permutations_CC(3).union(Permutations_CC(2))
sage: P.rank(Permutation([2,1]))
7
sage: P.rank(Permutation([1,2,3]))
0
```

unrank(*x*)

EXAMPLES:

```
sage: from sage.combinat.combinat import Permutations_CC
sage: P = Permutations_CC(3).union(Permutations_CC(2))
sage: P.unrank(7)
[2, 1]
sage: P.unrank(0)
[1, 2, 3]
```

sage.combinat.combinat.**bell_number**(*n*, *algorithm='flint'*, ***options*)

Return the n-th Bell number (the number of ways to partition a set of n elements into pairwise disjoint nonempty subsets).

INPUT:

- n – a positive integer

- `algorithm` – (Default: `'flint'`) any one of the following:
  - `'dobinski'` – Use Dobinski's formula implemented in Sage
  - `'flint'` – Wrap FLINT's `arith_bell_number`
  - `'gap'` – Wrap libGAP's `Bell`
  - `'mpmath'` – Wrap mpmath's `bell`

Warning: When using the mpmath algorithm to compute Bell numbers and you specify `prec`, it can return incorrect results due to low precision. See the examples section.

Let B_n denote the n-th Bell number. Dobinski's formula is:

$$B_n = e^{-1} \sum_{k=0}^{\infty} \frac{k^n}{k!}.$$

To show our implementation of Dobinski's method works, suppose that $n \geq 5$ and let k_0 be the smallest positive integer such that $\frac{k_0^n}{k_0!} < 1$. Note that $k_0 > n$ and $k_0 \leq 2n$ because we can prove that $\frac{(2n)^n}{(2n)!} < 1$ by Stirling.

If $k > k_0$, then we have $\frac{k^n}{k!} < \frac{1}{2^{k-k_0}}$. We show this by induction: let $c_k = \frac{k^n}{k!}$, if $k > n$ then

$$\frac{c_{k+1}}{c_k} = \frac{(1+k^{-1})^n}{k+1} < \frac{(1+n^{-1})^n}{n} < \frac{1}{2}.$$

The last inequality can easily be checked numerically for $n \geq 5$.

Using this, we can see that $\frac{c_k}{c_{k_0}} < \frac{1}{2^{k-k_0}}$ for $k > k_0 > n$. So summing this it gives that $\sum_{k=k_0+1}^{\infty} \frac{k^n}{k!} < 1$, and hence

$$B_n = e^{-1} \left(\sum_{k=0}^{k_0} \frac{k^n}{k!} + E_1 \right) = e^{-1} \sum_{k=0}^{k_0} \frac{k^n}{k!} + E_2,$$

where $0 < E_1 < 1$ and $0 < E_2 < e^{-1}$. Next we have for any $q > 0$

$$\sum_{k=0}^{k_0} \frac{k^n}{k!} = \frac{1}{q} \sum_{k=0}^{k_0} \left\lfloor \frac{qk^n}{k!} \right\rfloor + \frac{E_3}{q}$$

where $0 \leq E_3 \leq k_0 + 1 \leq 2n + 1$. Let $E_4 = \frac{E_3}{q}$ and let $q = 2n + 1$. We find $0 \leq E_4 \leq 1$. These two bounds give:

$$\begin{aligned} B_n &= \frac{e^{-1}}{q} \sum_{k=0}^{k_0} \left\lfloor \frac{qk^n}{k!} \right\rfloor + e^{-1} E_4 + E_2 \\ &= \frac{e^{-1}}{q} \sum_{k=0}^{k_0} \left\lfloor \frac{qk^n}{k!} \right\rfloor + E_5 \end{aligned}$$

where

$$0 < E_5 = e^{-1} E_4 + E_2 \leq e^{-1} + e^{-1} < \frac{3}{4}.$$

It follows that

$$B_n = \left\lceil \frac{e^{-1}}{q} \sum_{k=0}^{k_0} \left\lfloor \frac{qk^n}{k!} \right\rfloor \right\rceil.$$

Now define

$$b = \sum_{k=0}^{k_0} \left\lfloor \frac{qk^n}{k!} \right\rfloor.$$

This b can be computed exactly using integer arithmetic. To avoid the costly integer division by $k!$, we collect more terms and do only one division, for example with 3 terms:

$$\frac{k^n}{k!} + \frac{(k+1)^n}{(k+1)!} + \frac{(k+2)^n}{(k+2)!} = \frac{k^n(k+1)(k+2) + (k+1)^n(k+2) + (k+2)^n}{(k+2)!}$$

In the implementation, we collect $\sqrt{n}/2$ terms.

To actually compute B_n from b, we let $p = \lfloor \log_2(b) \rfloor + 1$ such that $b < 2^p$ and we compute with p bits of precision. This implies that b (and $q < b$) can be represented exactly.

We compute $\frac{e^{-1}}{q} b$, rounding down, and we must have an absolute error of at most $1/4$ (given that $E_5 < 3/4$). This means that we need a relative error of at most

$$\frac{eq}{4b} > \frac{(eq)/4}{2^p} > \frac{7}{2^p}$$

(assuming $n \geq 5$). With a precision of p bits and rounding down, every rounding has a relative error of at most $2^{1-p} = 2/2^p$. Since we do 3 roundings (b and q do not require rounding), we get a relative error of at most $6/2^p$. All this implies that the precision of p bits is sufficient.

EXAMPLES:

```
sage: bell_number(10)
115975
sage: bell_number(2)
2
sage: bell_number(-10)
Traceback (most recent call last):
...
ArithmeticError: Bell numbers not defined for negative indices
sage: bell_number(1)
1
sage: bell_number(1/3)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

When using the mpmath algorithm, we are required have mpmath's precision set to at least $\log_2(B_n)$ bits. If upon computing the Bell number the first time, we deem the precision too low, we use our guess to (temporarily) raise mpmath's precision and the Bell number is recomputed.

```
sage: k = bell_number(30, 'mpmath'); k
846749014511809332450147
sage: k == bell_number(30)
True
```

If you knows what precision is necessary before computing the Bell number, you can use the `prec` option:

```
sage: k2 = bell_number(30, 'mpmath', prec=30); k2
846749014511809332450147
sage: k == k2
True
```

Warning: Running mpmath with the precision set too low can result in incorrect results:

```
sage: k = bell_number(30, 'mpmath', prec=15); k
846749014511809388871680
sage: k == bell_number(30)
False
```

TESTS:

```
sage: all([bell_number(n) == bell_number(n,'dobinski') for n in range(200)])
True
sage: all([bell_number(n) == bell_number(n,'gap') for n in range(200)])
True
sage: all([bell_number(n) == bell_number(n,'mpmath', prec=500) for n in range(200, 220)])
True
```

AUTHORS:

- Robert Gerbicz
- Jeroen Demeyer: improved implementation of Dobinski formula with more accurate error estimates (trac ticket #17157)

REFERENCES:

- Wikipedia article Bell_number
- http://fredrik-j.blogspot.com/2009/03/computing-generalized-bell-numbers.html
- http://mathworld.wolfram.com/DobinskisFormula.html

`sage.combinat.combinat.`**`bell_polynomial`**`(n, k)`

Return the Bell Polynomial

$$B_{n,k}(x_0, x_1, \ldots, x_{n-k}) = \sum_{\sum j_i = k, \sum (i+1) j_i = n} \frac{n!}{j_0! j_1! \cdots j_{n-k}!} \left(\frac{x_0}{(0+1)!}\right)^{j_0} \left(\frac{x_1}{(1+1)!}\right)^{j_1} \cdots \left(\frac{x_{n-k}}{(n-k+1)!}\right)^{j_{n-k}}.$$

INPUT:

- n – integer
- k – integer

OUTPUT:

- a polynomial in $n - k + 1$ variables over $\mathbf{Z}$

EXAMPLES:

```
sage: bell_polynomial(6,2)
10*x2^2 + 15*x1*x3 + 6*x0*x4
sage: bell_polynomial(6,3)
15*x1^3 + 60*x0*x1*x2 + 15*x0^2*x3
```

TESTS:

Check that trac ticket #18338 is fixed:

```
sage: bell_polynomial(0,0).parent()
Multivariate Polynomial Ring in x over Integer Ring

sage: for n in (0..4):
```

```
....:     print [bell_polynomial(n,k).coefficients() for k in (0..n)]
[[1]]
[[], [1]]
[[], [1], [1]]
[[], [1], [3], [1]]
[[], [1], [3, 4], [6], [1]]
```

REFERENCES:

•E.T. Bell, "Partition Polynomials"

AUTHORS:

•Blair Sutton (2009-01-26)

•Thierry Monteil (2015-09-29): the result must always be a polynomial.

sage.combinat.combinat.**bernoulli_polynomial**(*x*, *n*)

Return the n-th Bernoulli polynomial evaluated at x.

The generating function for the Bernoulli polynomials is

$$\frac{te^{xt}}{e^t - 1} = \sum_{n=0}^{\infty} B_n(x)\frac{t^n}{n!},$$

and they are given directly by

$$B_n(x) = \sum_{i=0}^{n} \binom{n}{i} B_{n-i}x^i.$$

One has $B_n(x) = -n\zeta(1-n, x)$, where $\zeta(s, x)$ is the Hurwitz zeta function. Thus, in a certain sense, the Hurwitz zeta function generalizes the Bernoulli polynomials to non-integer values of n.

EXAMPLES:

```
sage: y = QQ['y'].0
sage: bernoulli_polynomial(y, 5)
y^5 - 5/2*y^4 + 5/3*y^3 - 1/6*y
sage: bernoulli_polynomial(y, 5)(12)
199870
sage: bernoulli_polynomial(12, 5)
199870
sage: bernoulli_polynomial(y^2 + 1, 5)
y^10 + 5/2*y^8 + 5/3*y^6 - 1/6*y^2
sage: P.<t> = ZZ[]
sage: p = bernoulli_polynomial(t, 6)
sage: p.parent()
Univariate Polynomial Ring in t over Rational Field
```

We verify an instance of the formula which is the origin of the Bernoulli polynomials (and numbers):

```
sage: power_sum = sum(k^4 for k in range(10))
sage: 5*power_sum == bernoulli_polynomial(10, 5) - bernoulli(5)
True
```

REFERENCES:

•Wikipedia article Bernoulli_polynomials

sage.combinat.combinat.**catalan_number**(*n*)

Return the n-th Catalan number.

Catalan numbers: The n-th Catalan number is given directly in terms of binomial coefficients by

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} \qquad \text{for} \quad n \geq 0.$$

Consider the set $S = \{1, ..., n\}$. A noncrossing partition of S is a partition in which no two blocks "cross" each other, i.e., if a and b belong to one block and x and y to another, they are not arranged in the order $axby$. C_n is the number of noncrossing partitions of the set S. There are many other interpretations (see REFERENCES).

When $n = -1$, this function raises a ZeroDivisionError; for other $n < 0$ it returns 0.

INPUT:

- n - integer

OUTPUT: integer

EXAMPLES:

```
sage: [catalan_number(i) for i in range(7)]
[1, 1, 2, 5, 14, 42, 132]
sage: taylor((-1/2)*sqrt(1 - 4*x^2), x, 0, 15)
132*x^14 + 42*x^12 + 14*x^10 + 5*x^8 + 2*x^6 + x^4 + x^2 - 1/2
sage: [catalan_number(i) for i in range(-7,7) if i != -1]
[0, 0, 0, 0, 0, 0, 1, 1, 2, 5, 14, 42, 132]
sage: catalan_number(-1)
Traceback (most recent call last):
...
ZeroDivisionError
sage: [catalan_number(n).mod(2) for n in range(16)]
[1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1]
```

REFERENCES:

- http://en.wikipedia.org/wiki/Catalan_number
- http://www-history.mcs.st-andrews.ac.uk/~history/Miscellaneous/CatalanNumbers/catalan.html

sage.combinat.combinat.**euler_number**(*n*)

Return the n-th Euler number.

IMPLEMENTATION: Wraps Maxima's euler.

EXAMPLES:

```
sage: [euler_number(i) for i in range(10)]
[1, 0, -1, 0, 5, 0, -61, 0, 1385, 0]
sage: maxima.eval("taylor (2/(exp(x)+exp(-x)), x, 0, 10)")
'1-x^2/2+5*x^4/24-61*x^6/720+277*x^8/8064-50521*x^10/3628800'
sage: [euler_number(i)/factorial(i) for i in range(11)]
[1, 0, -1/2, 0, 5/24, 0, -61/720, 0, 277/8064, 0, -50521/3628800]
sage: euler_number(-1)
Traceback (most recent call last):
...
ValueError: n (=-1) must be a nonnegative integer
```

REFERENCES:

- http://en.wikipedia.org/wiki/Euler_number

sage.combinat.combinat.**fibonacci**(*n*, *algorithm='pari'*)

Return the n-th Fibonacci number.

The Fibonacci sequence F_n is defined by the initial conditions $F_1 = F_2 = 1$ and the recurrence relation $F_{n+2} = F_{n+1} + F_n$. For negative n we define $F_n = (-1)^{n+1}F_{-n}$, which is consistent with the recurrence relation.

INPUT:

- `algorithm` – a string:
  - `"pari"` - (default) use the PARI C library's fibo function
  - `"gap"` - use GAP's Fibonacci function

Note: PARI is tens to hundreds of times faster than GAP here; moreover, PARI works for every large input whereas GAP doesn't.

EXAMPLES:

```
sage: fibonacci(10)
55
sage: fibonacci(10, algorithm='gap')
55

sage: fibonacci(-100)
-354224848179261915075
sage: fibonacci(100)
354224848179261915075

sage: fibonacci(0)
0
sage: fibonacci(1/2)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

sage.combinat.combinat.**fibonacci_sequence**(*start*, *stop=None*, *algorithm=None*)

Return an iterator over the Fibonacci sequence, for all fibonacci numbers f_n from `n = start` up to (but not including) `n = stop`

INPUT:

- `start` – starting value
- `stop` – stopping value
- `algorithm` – (default: `None`) passed on to fibonacci function (or not passed on if None, i.e., use the default)

EXAMPLES:

```
sage: fibs = [i for i in fibonacci_sequence(10, 20)]
sage: fibs
[55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]

sage: sum([i for i in fibonacci_sequence(100, 110)])
6991937692307530873001 3
```

See also:

`fibonacci_xrange()`

AUTHORS:

- Bobby Moretti

sage.combinat.combinat.**fibonacci_xrange**(*start*, *stop=None*, *algorithm='pari'*)

Return an iterator over all of the Fibonacci numbers in the given range, including `f_n = start` up to, but not including, `f_n = stop`.

EXAMPLES:

```
sage: fibs_in_some_range =  [i for i in fibonacci_xrange(10^7, 10^8)]
sage: len(fibs_in_some_range)
4
sage: fibs_in_some_range
[14930352, 24157817, 39088169, 63245986]

sage: fibs = [i for i in fibonacci_xrange(10, 100)]
sage: fibs
[13, 21, 34, 55, 89]

sage: list(fibonacci_xrange(13, 34))
[13, 21]
```

A solution to the second Project Euler problem:

```
sage: sum([i for i in fibonacci_xrange(10^6) if is_even(i)])
1089154
```

See also:

`fibonacci_sequence()`

AUTHORS:

- Bobby Moretti

sage.combinat.combinat.**lucas_number1**(*n*, *P*, *Q*)

Return the n-th Lucas number "of the first kind" (this is not standard terminology). The Lucas sequence $L_n^{(1)}$ is defined by the initial conditions $L_1^{(1)} = 0$, $L_2^{(1)} = 1$ and the recurrence relation $L_{n+2}^{(1)} = P \cdot L_{n+1}^{(1)} - Q \cdot L_n^{(1)}$.

Wraps GAP's `Lucas(...)[1]`.

$P = 1, Q = -1$ gives the Fibonacci sequence.

INPUT:

- `n` – integer
- `P, Q` – integer or rational numbers

OUTPUT: integer or rational number

EXAMPLES:

```
sage: lucas_number1(5,1,-1)
5
sage: lucas_number1(6,1,-1)
8
sage: lucas_number1(7,1,-1)
13
sage: lucas_number1(7,1,-2)
43
sage: lucas_number1(5,2,3/5)
229/25
sage: lucas_number1(5,2,1.5)
1/4
```

There was a conjecture that the sequence L_n defined by $L_{n+2} = L_{n+1} + L_n$, $L_1 = 1$, $L_2 = 3$, has the property that n prime implies that L_n is prime.

```
sage: lucas = lambda n : Integer((5/2)*lucas_number1(n,1,-1)+(1/2)*lucas_number2(n,1,-1))
sage: [[lucas(n),is_prime(lucas(n)),n+1,is_prime(n+1)] for n in range(15)]
[[1, False, 1, False],
 [3, True, 2, True],
 [4, False, 3, True],
 [7, True, 4, False],
 [11, True, 5, True],
 [18, False, 6, False],
 [29, True, 7, True],
 [47, True, 8, False],
 [76, False, 9, False],
 [123, False, 10, False],
 [199, True, 11, True],
 [322, False, 12, False],
 [521, True, 13, True],
 [843, False, 14, False],
 [1364, False, 15, False]]
```

Can you use Sage to find a counterexample to the conjecture?

sage.combinat.combinat.**lucas_number2**(n, P, Q)

Return the n-th Lucas number "of the second kind" (this is not standard terminology). The Lucas sequence $L_n^{(2)}$ is defined by the initial conditions $L_1^{(2)} = 2$, $L_2^{(2)} = P$ and the recurrence relation $L_{n+2}^{(2)} = P \cdot L_{n+1}^{(2)} - Q \cdot L_n^{(2)}$.

Wraps GAP's Lucas(...)[2].

INPUT:

- `n` - integer
- `P, Q` - integer or rational numbers

OUTPUT: integer or rational number

EXAMPLES:

```
sage: [lucas_number2(i,1,-1) for i in range(10)]
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
sage: [fibonacci(i-1)+fibonacci(i+1) for i in range(10)]
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]

sage: n = lucas_number2(5,2,3); n
2
sage: type(n)
<type 'sage.rings.integer.Integer'>
sage: n = lucas_number2(5,2,-3/9); n
418/9
sage: type(n)
<type 'sage.rings.rational.Rational'>
```

The case $P = 1$, $Q = -1$ is the Lucas sequence in Brualdi's Introductory Combinatorics, 4th ed., Prentice-Hall, 2004:

```
sage: [lucas_number2(n,1,-1) for n in range(10)]
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
```

sage.combinat.combinat.**number_of_tuples**(*S*, *k*, *algorithm='naive'*)

Return the size of `tuples(S, k)` when S is a set. More generally, return the size of `tuples(set(S), k)`. (So, unlike `tuples()`, this method removes redundant entries from S.)

INPUT:

- `S` – the base set
- `k` – the length of the tuples
- `algorithm` – can be one of the following:
  - `'naive'` - (default) use the naive counting $|S|^k$
  - `'gap'` - wraps GAP's `NrTuples`

Warning: When using `algorithm='gap'`, `S` must be a list of objects that have string representations that can be interpreted by the GAP interpreter. If `S` consists of at all complicated Sage objects, this function might *not* do what you expect.

EXAMPLES:

```
sage: S = [1,2,3,4,5]
sage: number_of_tuples(S,2)
25
sage: number_of_tuples(S,2, algorithm="gap")
25
sage: S = [1,1,2,3,4,5]
sage: number_of_tuples(S,2)
25
sage: number_of_tuples(S,2, algorithm="gap")
25
sage: number_of_tuples(S,0)
1
sage: number_of_tuples(S,0, algorithm="gap")
1
```

`sage.combinat.combinat.`**`number_of_unordered_tuples`**(*S*, *k*, *algorithm='naive'*)

Return the size of `unordered_tuples(S, k)` when S is a set.

INPUT:

- `S` – the base set
- `k` – the length of the tuples
- `algorithm` – can be one of the following:
  - `'naive'` - (default) use the naive counting $\binom{|S|+k-1}{k}$
  - `'gap'` - wraps GAP's `NrUnorderedTuples`

Warning: When using `algorithm='gap'`, `S` must be a list of objects that have string representations that can be interpreted by the GAP interpreter. If `S` consists of at all complicated Sage objects, this function might *not* do what you expect.

EXAMPLES:

```
sage: S = [1,2,3,4,5]
sage: number_of_unordered_tuples(S,2)
15
sage: number_of_unordered_tuples(S,2, algorithm="gap")
15
sage: S = [1,1,2,3,4,5]
sage: number_of_unordered_tuples(S,2)
15
```

```
sage: number_of_unordered_tuples(S,2, algorithm="gap")
15
sage: number_of_unordered_tuples(S,0)
1
sage: number_of_unordered_tuples(S,0, algorithm="gap")
1
```

sage.combinat.combinat.**stirling_number1**(n, k)

Return the n-th Stirling number $S_1(n,k)$ of the first kind.

This is the number of permutations of n points with k cycles.

This wraps GAP's Stirling1.

EXAMPLES:

```
sage: stirling_number1(3,2)
3
sage: stirling_number1(5,2)
50
sage: 9*stirling_number1(9,5)+stirling_number1(9,4)
269325
sage: stirling_number1(10,5)
269325
```

Indeed, $S_1(n,k) = S_1(n-1,k-1) + (n-1)S_1(n-1,k)$.

sage.combinat.combinat.**stirling_number2**(n, k, *algorithm=None*)

Return the n-th Stirling number $S_2(n,k)$ of the second kind (the number of ways to partition a set of n elements into k pairwise disjoint nonempty subsets). (The n-th Bell number is the sum of the $S_2(n,k)$'s, $k = 0, ..., n$.)

INPUT:

- `n` - nonnegative machine-size integer
- `k` - nonnegative machine-size integer
- `algorithm`:
  - None (default) - use native implementation
  - `"maxima"` - use Maxima's stirling2 function
  - `"gap"` - use GAP's Stirling2 function

EXAMPLES:

Print a table of the first several Stirling numbers of the second kind:

```
sage: for n in range(10):
...       for k in range(10):
...           print str(stirling_number2(n,k)).rjust(k and 6),
...       print
...
1      0      0      0      0      0      0      0      0      0
0      1      0      0      0      0      0      0      0      0
0      1      1      0      0      0      0      0      0      0
0      1      3      1      0      0      0      0      0      0
0      1      7      6      1      0      0      0      0      0
0      1     15     25     10      1      0      0      0      0
0      1     31     90     65     15      1      0      0      0
0      1     63    301    350    140     21      1      0      0
```

```
0      1    127    966   1701   1050    266     28      1      0
0      1    255   3025   7770   6951   2646    462     36      1
```

Stirling numbers satisfy $S_2(n,k) = S_2(n-1,k-1) + kS_2(n-1,k)$:

```
sage: 5*stirling_number2(9,5) + stirling_number2(9,4)
42525
sage: stirling_number2(10,5)
42525
```

TESTS:

```
sage: stirling_number2(500,501)
0
sage: stirling_number2(500,500)
1
sage: stirling_number2(500,499)
124750
sage: stirling_number2(500,498)
7739801875
sage: stirling_number2(500,497)
318420320812125
sage: stirling_number2(500,0)
0
sage: stirling_number2(500,1)
1
sage: stirling_number2(500,2)
1636695303948070935006594848413799576108321023021532394741645684048066898202337277441635046164
sage: stirling_number2(500,3)
6060048632644989473730877846590553186337230837666937173391005972096766698597315914033083073800
sage: stirling_number2(500,30)
1370776714124945492944910842432843284500132747909971303787683275932391813484053722973762401895
sage: stirling_number2(500,31)
5832088795102666690960147007601603328246123996896731854823915012140005028360632199516298102440
sage: n = stirling_number2(20,11)
sage: n
1900842429486
sage: type(n)
<type 'sage.rings.integer.Integer'>
sage: n = stirling_number2(20,11,algorithm='gap')
sage: n
1900842429486
sage: type(n)
<type 'sage.rings.integer.Integer'>
sage: n = stirling_number2(20,11,algorithm='maxima')
sage: n
1900842429486
sage: type(n)
<type 'sage.rings.integer.Integer'>
```

Sage's implementation splitting the computation of the Stirling numbers of the second kind in two cases according to `n`, let us check the result it gives agree with both maxima and gap.

For `n<200`::

```
sage: for n in Subsets(range(100,200), 5).random_element():
...       for k in Subsets(range(n), 5).random_element():
...           s_sage = stirling_number2(n,k)
```

```
        ...          s_maxima = stirling_number2(n,k, algorithm = "maxima")
        ...          s_gap = stirling_number2(n,k, algorithm = "gap")
        ...          if not (s_sage == s_maxima and s_sage == s_gap):
        ...              print "Error with n<200"

    For `n\geq 200`::

        sage: for n in Subsets(range(200,300), 5).random_element():
        ...       for k in Subsets(range(n), 5).random_element():
        ...          s_sage = stirling_number2(n,k)
        ...          s_maxima = stirling_number2(n,k, algorithm = "maxima")
        ...          s_gap = stirling_number2(n,k, algorithm = "gap")
        ...          if not (s_sage == s_maxima and s_sage == s_gap):
        ...              print "Error with n<200"


    TESTS:

    Checking an exception is raised whenever a wrong value is given
    for ``algorithm``::

        sage: s_sage = stirling_number2(50,3, algorithm = "CloudReading")
        Traceback (most recent call last):
        ...
        ValueError: unknown algorithm: CloudReading
```

sage.combinat.combinat.**tuples**(*S*, *k*, *algorithm='itertools'*)

Return a list of all k-tuples of elements of a given set `S`.

This function accepts the set `S` in the form of any iterable (list, tuple or iterator), and returns a list of k-tuples. If `S` contains duplicate entries, then you should expect the method to return tuples multiple times!

Recall that k-tuples are ordered (in the sense that two k-tuples differing in the order of their entries count as different) and can have repeated entries (even if `S` is a list with no repetition).

INPUT:

- `S` – the base set
- `k` – the length of the tuples
- `algorithm` – can be one of the following:
  - `'itertools'` - (default) use python's itertools
  - `'native'` - use a native Sage implementation

Note: The ordering of the list of tuples differs for the algorithms.

EXAMPLES:

```
sage: S = [1,2]
sage: tuples(S,3)
[(1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 2, 2),
 (2, 1, 1), (2, 1, 2), (2, 2, 1), (2, 2, 2)]
sage: mset = ["s","t","e","i","n"]
sage: tuples(mset, 2)
[('s', 's'), ('s', 't'), ('s', 'e'), ('s', 'i'), ('s', 'n'),
 ('t', 's'), ('t', 't'), ('t', 'e'), ('t', 'i'), ('t', 'n'),
 ('e', 's'), ('e', 't'), ('e', 'e'), ('e', 'i'), ('e', 'n'),
```

```
 ('i', 's'), ('i', 't'), ('i', 'e'), ('i', 'i'), ('i', 'n'),
 ('n', 's'), ('n', 't'), ('n', 'e'), ('n', 'i'), ('n', 'n')]

sage: K.<a> = GF(4, 'a')
sage: mset = [x for x in K if x != 0]
sage: tuples(mset, 2)
[(a, a), (a, a + 1), (a, 1), (a + 1, a), (a + 1, a + 1),
 (a + 1, 1), (1, a), (1, a + 1), (1, 1)]
```

We check that the implementations agree (up to ordering):

```
sage: tuples(S, 3, 'native')
[(1, 1, 1), (2, 1, 1), (1, 2, 1), (2, 2, 1),
 (1, 1, 2), (2, 1, 2), (1, 2, 2), (2, 2, 2)]
```

Lastly we check on a multiset:

```
sage: S = [1,1,2]
sage: sorted(tuples(S, 3)) == sorted(tuples(S, 3, 'native'))
True
```

AUTHORS:

- Jon Hanke (2006-08)

sage.combinat.combinat.**unordered_tuples**(*S*, *k*, *algorithm='itertools'*)

Return a list of all unordered tuples of length `k` of the set `S`.

An unordered tuple of length k of set S is a unordered selection with repetitions of S and is represented by a sorted list of length k containing elements from S.

Unlike `tuples()`, the result of this method does not depend on how often an element appears in S; only the *set* S is being used. For example, `unordered_tuples([1, 1, 1], 2)` will return `[(1, 1)]`. If you want it to return `[(1, 1), (1, 1), (1, 1)]`, use Python's `itertools.combinations_with_replacement` instead.

INPUT:

- `S` – the base set
- `k` – the length of the tuples
- `algorithm` – can be one of the following:
  - `'itertools'` - (default) use python's itertools
  - `'gap'` - wraps GAP's `UnorderedTuples`

Warning: When using `algorithm='gap'`, `S` must be a list of objects that have string representations that can be interpreted by the GAP interpreter. If `S` consists of at all complicated Sage objects, this function might *not* do what you expect.

EXAMPLES:

```
sage: S = [1,2]
sage: unordered_tuples(S, 3)
[(1, 1, 1), (1, 1, 2), (1, 2, 2), (2, 2, 2)]
```

We check that this agrees with GAP:

```
sage: unordered_tuples(S, 3, algorithm='gap')
[(1, 1, 1), (1, 1, 2), (1, 2, 2), (2, 2, 2)]
```

We check the result on strings:

```
sage: S = ["a","b","c"]
sage: unordered_tuples(S, 2)
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('b', 'b'), ('b', 'c'), ('c', 'c')]
sage: unordered_tuples(S, 2, algorithm='gap')
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('b', 'b'), ('b', 'c'), ('c', 'c')]
```

Lastly we check on a multiset:

```
sage: S = [1,1,2]
sage: unordered_tuples(S, 3) == unordered_tuples(S, 3, 'gap')
True
sage: unordered_tuples(S, 3)
[(1, 1, 1), (1, 1, 2), (1, 2, 2), (2, 2, 2)]
```

sage.combinat.combinat.**unshuffle_iterator**(*a*, *one=1*)

Iterate over the unshuffles of a list (or tuple) `a`, also yielding the signs of the respective permutations.

If n and k are integers satisfying $0 \leq k \leq n$, then a $(k, n-k)$-*unshuffle* means a permutation $\pi \in S_n$ such that $\pi(1) < \pi(2) < \cdots < \pi(k)$ and $\pi(k+1) < \pi(k+2) < \cdots < \pi(n)$. This method provides, for a list $a = (a_1, a_2, \ldots, a_n)$ of length n, an iterator yielding all pairs:

$$\left(\left((a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(k)}), (a_{\pi(k+1)}, a_{\pi(k+2)}, \ldots, a_{\pi(n)})\right), (-1)^{\pi}\right)$$

for all $k \in \{0, 1, \ldots, n\}$ and all $(k, n-k)$-unshuffles π. The optional variable `one` can be set to a different value which results in the $(-1)^{\pi}$ component being multiplied by said value.

The iterator does not yield these in order of increasing k.

EXAMPLES:

```
sage: from sage.combinat.combinat import unshuffle_iterator
sage: list(unshuffle_iterator([1, 3, 4]))
[((), (1, 3, 4)), 1), (((1,), (3, 4)), 1), (((3,), (1, 4)), -1),
 (((1, 3), (4,)), 1), (((4,), (1, 3)), 1), (((1, 4), (3,)), -1),
 (((3, 4), (1,)), 1), (((1, 3, 4), ()), 1)]
sage: list(unshuffle_iterator([3, 1]))
[((), (3, 1)), 1), (((3,), (1,)), 1), (((1,), (3,)), -1),
 (((3, 1), ()), 1)]
sage: list(unshuffle_iterator([8]))
[((), (8,)), 1), (((8,), ()), 1)]
sage: list(unshuffle_iterator([]))
[((), ()), 1)]
sage: list(unshuffle_iterator([3, 1], 3/2))
[((), (3, 1)), 3/2), (((3,), (1,)), 3/2), (((1,), (3,)), -3/2),
 (((3, 1), ()), 3/2)]
```

5.1.23 Fast computation of combinatorial functions (Cython + mpz).

Currently implemented: - Stirling numbers of the second kind

AUTHORS: - Fredrik Johansson (2010-10): Stirling numbers of second kind

5.1.24 Combinations

AUTHORS:

- Mike Hansen (2007): initial implementation
- Vincent Delecroix (2011): cleaning, bug corrections, doctests

class sage.combinat.combination.**ChooseNK**(*mset*, *k*)
Bases: sage.combinat.combination.Combinations_setk

TESTS:

```
sage: C = Combinations([1,2,3],2)
sage: C == loads(dumps(C))
True
```

sage.combinat.combination.**Combinations**(*mset*, *k=None*)
Return the combinatorial class of combinations of the multiset `mset`. If `k` is specified, then it returns the combinatorial class of combinations of `mset` of size `k`.

A *combination* of a multiset M is an unordered selection of k objects of M, where every object can appear at most as many times as it appears in M.

The combinatorial classes correctly handle the cases where mset has duplicate elements.

EXAMPLES:

```
sage: C = Combinations(range(4)); C
Combinations of [0, 1, 2, 3]
sage: C.list()
[[],
 [0],
 [1],
 [2],
 [3],
 [0, 1],
 [0, 2],
 [0, 3],
 [1, 2],
 [1, 3],
 [2, 3],
 [0, 1, 2],
 [0, 1, 3],
 [0, 2, 3],
 [1, 2, 3],
 [0, 1, 2, 3]]
 sage: C.cardinality()
 16

sage: C2 = Combinations(range(4),2); C2
Combinations of [0, 1, 2, 3] of length 2
sage: C2.list()
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
sage: C2.cardinality()
6

sage: Combinations([1,2,2,3]).list()
[[],
 [1],
 [2],
 [3],
 [1, 2],
 [1, 3],
 [2, 2],
 [2, 3],
```

```
 [1, 2, 2],
 [1, 2, 3],
 [2, 2, 3],
 [1, 2, 2, 3]]

sage: Combinations([1,2,3], 2).list()
[[1, 2], [1, 3], [2, 3]]

sage: mset = [1,1,2,3,4,4,5]
sage: Combinations(mset,2).list()
[[1, 1],
 [1, 2],
 [1, 3],
 [1, 4],
 [1, 5],
 [2, 3],
 [2, 4],
 [2, 5],
 [3, 4],
 [3, 5],
 [4, 4],
 [4, 5]]

sage: mset = ["d","a","v","i","d"]
sage: Combinations(mset,3).list()
[['d', 'd', 'a'],
 ['d', 'd', 'v'],
 ['d', 'd', 'i'],
 ['d', 'a', 'v'],
 ['d', 'a', 'i'],
 ['d', 'v', 'i'],
 ['a', 'v', 'i']]

sage: X = Combinations([1,2,3,4,5],3)
sage: [x for x in X]
[[1, 2, 3],
 [1, 2, 4],
 [1, 2, 5],
 [1, 3, 4],
 [1, 3, 5],
 [1, 4, 5],
 [2, 3, 4],
 [2, 3, 5],
 [2, 4, 5],
 [3, 4, 5]]
```

It is possible to take combinations of Sage objects:

```
sage: Combinations([vector([1,1]), vector([2,2]), vector([3,3])], 2).list()
[[(1, 1), (2, 2)], [(1, 1), (3, 3)], [(2, 2), (3, 3)]]
```

TESTS:

We check that the code works even for non mutable objects:

```
sage: l = [vector((0,0)), vector((0,1))]
sage: Combinations(l).list()
[[], [(0, 0)], [(0, 1)], [(0, 0), (0, 1)]]
```

class sage.combinat.combination.**Combinations_mset**(*mset*)
Bases: sage.combinat.combinat.CombinatorialClass

TESTS:

```
sage: C = Combinations(range(4))
sage: C == loads(dumps(C))
True
```

cardinality()
TESTS:

```
sage: Combinations([1,2,3]).cardinality()
8
sage: Combinations(['a','a','b']).cardinality()
6
```

class sage.combinat.combination.**Combinations_msetk**(*mset*, *k*)
Bases: sage.combinat.combinat.CombinatorialClass

TESTS:

```
sage: C = Combinations([1,2,3],2)
sage: C == loads(dumps(C))
True
```

cardinality()
Returns the size of combinations(mset,k). IMPLEMENTATION: Wraps GAP's NrCombinations.

EXAMPLES:

```
sage: mset = [1,1,2,3,4,4,5]
sage: Combinations(mset,2).cardinality()
12
```

class sage.combinat.combination.**Combinations_set**(*mset*)
Bases: sage.combinat.combination.Combinations_mset

TESTS:

```
sage: C = Combinations(range(4))
sage: C == loads(dumps(C))
True
```

rank(*x*)
EXAMPLES:

```
sage: c = Combinations([1,2,3])
sage: range(c.cardinality()) == map(c.rank, c)
True
```

unrank(*r*)
EXAMPLES:

```
sage: c = Combinations([1,2,3])
sage: c.list() == map(c.unrank, range(c.cardinality()))
True
```

class sage.combinat.combination.**Combinations_setk**(*mset*, *k*)
Bases: sage.combinat.combination.Combinations_msetk

TESTS:

```
sage: C = Combinations([1,2,3],2)
sage: C == loads(dumps(C))
True
```

list()

EXAMPLES:

```
sage: Combinations([1,2,3,4,5],3).list()
[[1, 2, 3],
 [1, 2, 4],
 [1, 2, 5],
 [1, 3, 4],
 [1, 3, 5],
 [1, 4, 5],
 [2, 3, 4],
 [2, 3, 5],
 [2, 4, 5],
 [3, 4, 5]]
```

rank(*x*)

EXAMPLES:

```
sage: c = Combinations([1,2,3], 2)
sage: range(c.cardinality()) == map(c.rank, c.list())
True
```

unrank(*r*)

EXAMPLES:

```
sage: c = Combinations([1,2,3], 2)
sage: c.list() == map(c.unrank, range(c.cardinality()))
True
```

sage.combinat.combination.**from_rank**(*r*, *n*, *k*)

Returns the combination of rank `r` in the subsets of `range(n)` of size `k` when listed in lexicographic order.

The algorithm used is based on combinadics and James McCaffrey's MSDN article. See: Wikipedia article Combinadic

EXAMPLES:

```
sage: import sage.combinat.combination as combination
sage: combination.from_rank(0,3,0)
()
sage: combination.from_rank(0,3,1)
(0,)
sage: combination.from_rank(1,3,1)
(1,)
sage: combination.from_rank(2,3,1)
(2,)
sage: combination.from_rank(0,3,2)
(0, 1)
sage: combination.from_rank(1,3,2)
(0, 2)
sage: combination.from_rank(2,3,2)
(1, 2)
sage: combination.from_rank(0,3,3)
(0, 1, 2)
```

sage.combinat.combination.**rank**(*comb*, *n*, *check=True*)

Return the rank of `comb` in the subsets of `range(n)` of size `k` where `k` is the length of `comb`.

The algorithm used is based on combinadics and James McCaffrey's MSDN article. See: Wikipedia article Combinadic.

EXAMPLES:

```
sage: import sage.combinat.combination as combination
sage: combination.rank((), 3)
0
sage: combination.rank((0,), 3)
0
sage: combination.rank((1,), 3)
1
sage: combination.rank((2,), 3)
2
sage: combination.rank((0,1), 3)
0
sage: combination.rank((0,2), 3)
1
sage: combination.rank((1,2), 3)
2
sage: combination.rank((0,1,2), 3)
0

sage: combination.rank((0,1,2,3), 3)
Traceback (most recent call last):
...
ValueError: len(comb) must be <= n
sage: combination.rank((0,0), 2)
Traceback (most recent call last):
...
ValueError: comb must be a subword of (0,1,...,n)

sage: combination.rank([1,2], 3)
2
sage: combination.rank([0,1,2], 3)
0
```

5.1.25 Combinatorial Algebras

A combinatorial algebra is an algebra whose basis elements are indexed by a combinatorial class. Some examples of combinatorial algebras are the symmetric group algebra of order n (indexed by permutations of size n) and the algebra of symmetric functions (indexed by integer partitions).

The CombinatorialAlgebra base class makes it easy to define and work with new combinatorial algebras in Sage. For example, the following code constructs an algebra which models the power-sum symmetric functions.

```
sage: class PowerSums(CombinatorialAlgebra):
...       def __init__(self, R):
...           self._one = Partition([])
...           self._name = 'Power-sum symmetric functions'
...           CombinatorialAlgebra.__init__(self, R, Partitions())
...           self.print_options(prefix='p')
...
...       def _multiply_basis(self, a, b):
...           l = list(a)+list(b)
...           l.sort(reverse=True)
```

```
...         return Partition(l)
...

sage: ps = PowerSums(QQ); ps
Power-sum symmetric functions over Rational Field
sage: ps([2,1])^2
p[2, 2, 1, 1]
sage: ps([2,1])+2*ps([1,1,1])
2*p[1, 1, 1] + p[2, 1]
sage: ps(2)
2*p[]
```

The important things to define are ._indices which specifies the combinatorial class that indexes the basis elements, ._one which specifies the identity element in the algebra, ._name which specifies the name of the algebra, .print_options is used to set the print options for the elements, and finally a _multiply or _multiply_basis method that defines the multiplication in the algebra.

class sage.combinat.combinatorial_algebra.**CombinatorialAlgebra**(*R*, *cc=None*, *element_class=None*, *category=None*)

Bases: sage.combinat.free_module.CombinatorialFreeModule

Deprecated! Don't use!

one_basis()

TESTS:

```
sage: s = sage.combinat.combinatorial_algebra.TestAlgebra(QQ)
sage: s.one_basis()
[]
```

product(*left*, *right*)

Returns left*right where left and right are elements of self. product() uses either _multiply or _multiply basis to carry out the actual multiplication.

EXAMPLES:

```
sage: s = sage.combinat.combinatorial_algebra.TestAlgebra(QQ)
sage: a = s([2])
sage: s.product(a,a)
s[2, 2] + s[3, 1] + s[4]
sage: ZS3 = SymmetricGroupAlgebra(ZZ, 3)
sage: a = 2 + ZS3([2,1,3])
sage: a*a
5*[1, 2, 3] + 4*[2, 1, 3]
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ,3)
sage: j2 = H3.jucys_murphy(2)
sage: j2*j2
(q^3-q^2+q)*T[1, 2, 3] + (q^3-q^2+q-1)*T[2, 1, 3]
sage: X = SchubertPolynomialRing(ZZ)
sage: X([1,2,3])*X([2,1,3])
X[2, 1]
```

class sage.combinat.combinatorial_algebra.**CombinatorialAlgebraElementOld**(*M*, *x*)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.combinatorial_algebra.**TestAlgebra**(*R*)

Bases: sage.combinat.combinatorial_algebra.CombinatorialAlgebra

TESTS:

```
sage: s = sage.combinat.combinatorial_algebra.TestAlgebra(QQ)
sage: TestSuite(s).run()
```

prefix()

TESTS:

```
sage: sa = sage.combinat.combinatorial_algebra.TestAlgebra(QQ)
sage: x = sa([2]); x  # indirect doctest
s[2]
```

5.1.26 Combinatorial maps

This module provides a decorator that can be used to add semantic to a Python method by marking it as implementing a *combinatorial map*, that is a map between two enumerated sets:

```
sage: from sage.combinat.combinatorial_map import combinatorial_map
sage: class MyPermutation(object):
....:
....:     @combinatorial_map()
....:     def reverse(self):
....:         '''
....:         Reverse the permutation
....:         '''
....:         # ... code ...
```

By default, this decorator is a no-op: it returns the decorated method as is:

```
sage: MyPermutation.reverse
<unbound method MyPermutation.reverse>
```

See combinatorial_map_wrapper() for the various options this decorator can take.

Projects built on top of Sage are welcome to customize locally this hook to instrument the Sage code and exploit this semantic information. Typically, the decorator could be used to populate a database of maps. For a real-life application, see the project $FindStat < http://findstat.org/ >$. As a basic example, a variant of the decorator is provided as combinatorial_map_wrapper(); it wraps the decorated method, so that one can later use combinatorial_maps_in_class() to query an object, or class thereof, for all the combinatorial maps that apply to it.

Note: Since decorators are evaluated upon loading Python modules, customizing combinatorial map needs to be done before the modules using it are loaded. In the examples below, where we illustrate the customized combinatorial_map decorator on the sage.combinat.permutation module, we resort to force a reload of this module after dynamically changing sage.combinat.combinatorial_map.combinatorial_map. This is good enough for those doctests, but remains fragile.

For real use cases it's probably best to just edit this source file statically (see below).

---

class sage.combinat.combinatorial_map.**CombinatorialMap**(*f*, *order=None*, *name=None*)

Bases: object

This is a wrapper class for methods that are *combinatorial maps*.

For further details and doctests, see *Combinatorial maps* and combinatorial_map_wrapper().

name()

Returns the name of a combinatorial map. This is used for the string representation of self.

EXAMPLES:

```
sage: from sage.combinat.combinatorial_map import combinatorial_map
sage: class CombinatorialClass:
....:     @combinatorial_map(name='map1')
....:     def to_self_1(): pass
....:     @combinatorial_map()
....:     def to_self_2(): pass
sage: CombinatorialClass.to_self_1.name()
'map1'
sage: CombinatorialClass.to_self_2.name()
'to_self_2'
```

order()

Returns the order of self, or None if the order is not known.

EXAMPLES:

```
sage: from sage.combinat.combinatorial_map import combinatorial_map
sage: class CombinatorialClass:
....:     @combinatorial_map(order=2)
....:     def to_self_1(): pass
....:     @combinatorial_map()
....:     def to_self_2(): pass
sage: CombinatorialClass.to_self_1.order()
2
sage: CombinatorialClass.to_self_2.order() is None
True
```

unbounded_map()

Return the unbounded version of self.

You can use this method to return a function which takes as input an element in the domain of the combinatorial map. See the example below.

EXAMPLES:

```
sage: sage.combinat.combinatorial_map.combinatorial_map = sage.combinat.combinatorial_map.co
sage: import imp
sage: _ = imp.reload(sage.combinat.permutation);
sage: from sage.combinat.permutation import Permutation
sage: pi = Permutation([1,3,2])
sage: f = pi.reverse
sage: F = f.unbounded_map()
sage: F(pi)
[2, 3, 1]
```

sage.combinat.combinatorial_map.**combinatorial_map**(*f=None*, *order=None*, *name=None*)

Combinatorial map decorator

See *Combinatorial maps* for a description of this decorator and its purpose. This default implementation does nothing.

INPUT:

- `f` – (default: `None`, if combinatorial_map is used as a decorator) a function
- `name` – (default: `None`) the name for nicer outputs on combinatorial maps
- `order` – (default: `None`) the order of the combinatorial map, if it is known. Is not used, but might be helpful later

OUTPUT:

- `f` unchanged

EXAMPLES:

```
sage: from sage.combinat.combinatorial_map import combinatorial_map_trivial as combinatorial_map
sage: class MyPermutation(object):
....:
....:     @combinatorial_map
....:     def reverse(self):
....:         '''
....:         Reverse the permutation
....:         '''
....:         # ... code ...
....:
....:     @combinatorial_map(name='descent set of permutation')
....:     def descent_set(self):
....:         '''
....:         The descent set of the permutation
....:         '''
....:         # ... code ...

sage: MyPermutation.reverse
<unbound method MyPermutation.reverse>
sage: MyPermutation.descent_set
<unbound method MyPermutation.descent_set>
```

sage.combinat.combinatorial_map.**combinatorial_map_trivial**(*f=None*, *order=None*, *name=None*)

Combinatorial map decorator

See *Combinatorial maps* for a description of this decorator and its purpose. This default implementation does nothing.

INPUT:

- `f` – (default: `None`, if combinatorial_map is used as a decorator) a function
- `name` – (default: `None`) the name for nicer outputs on combinatorial maps
- `order` – (default: `None`) the order of the combinatorial map, if it is known. Is not used, but might be helpful later

OUTPUT:

- `f` unchanged

EXAMPLES:

```
sage: from sage.combinat.combinatorial_map import combinatorial_map_trivial as combinatorial_map
sage: class MyPermutation(object):
....:
```

```
....:     @combinatorial_map
....:     def reverse(self):
....:         '''
....:         Reverse the permutation
....:         '''
....:         # ... code ...
....:
....:     @combinatorial_map(name='descent set of permutation')
....:     def descent_set(self):
....:         '''
....:         The descent set of the permutation
....:         '''
....:         # ... code ...

sage: MyPermutation.reverse
<unbound method MyPermutation.reverse>
sage: MyPermutation.descent_set
<unbound method MyPermutation.descent_set>
```

sage.combinat.combinatorial_map.**combinatorial_map_wrapper**(*f=None*, *order=None*, *name=None*)

Combinatorial map decorator (basic example).

See *Combinatorial maps* for a description of the combinatorial_map decorator and its purpose. This implementation, together with combinatorial_maps_in_class() illustrates how to use this decorator as a hook to instrument the Sage code.

INPUT:

- f – (default: None, if combinatorial_map is used as a decorator) a function
- name – (default: None) the name for nicer outputs on combinatorial maps
- order – (default: None) the order of the combinatorial map, if it is known. Is not used, but might be helpful later

OUTPUT:

- A combinatorial map. This is an instance of the CombinatorialMap.

EXAMPLES:

We define a class illustrating the use of this implementation of the combinatorial_map decorator with its various arguments:

```
sage: from sage.combinat.combinatorial_map import combinatorial_map_wrapper as combinatorial_map
sage: class MyPermutation(object):
....:
....:     @combinatorial_map()
....:     def reverse(self):
....:         '''
....:         Reverse the permutation
....:         '''
....:         pass
....:
....:     @combinatorial_map(order=2)
....:     def inverse(self):
....:         '''
....:         The inverse of the permutation
....:         '''
....:         pass
```

```
....:
....:     @combinatorial_map(name='descent set of permutation')
....:     def descent_set(self):
....:         '''
....:         The descent set of the permutation
....:         '''
....:         pass
....:
....:     def major_index(self):
....:         '''
....:         The major index of the permutation
....:         '''
....:         pass
sage: MyPermutation.reverse
Combinatorial map: reverse
sage: MyPermutation.descent_set
Combinatorial map: descent set of permutation
sage: MyPermutation.inverse
Combinatorial map: inverse
```

One can now determine all the combinatorial maps associated with a given object as follows:

```
sage: from sage.combinat.combinatorial_map import combinatorial_maps_in_class
sage: X = combinatorial_maps_in_class(MyPermutation); X # random
[Combinatorial map: reverse,
 Combinatorial map: descent set of permutation,
 Combinatorial map: inverse]
```

The method `major_index` defined about is not a combinatorial map:

```
sage: MyPermutation.major_index
<unbound method MyPermutation.major_index>
```

But one can define a function that turns `major_index` into a combinatorial map:

```
sage: def major_index(p):
....:     return p.major_index()
....:
sage: major_index
<function major_index at ...>
sage: combinatorial_map(major_index)
Combinatorial map: major_index
```

sage.combinat.combinatorial_map.**combinatorial_maps_in_class**(*cls*)

Return the combinatorial maps of the class as a list of combinatorial maps.

For further details and doctests, see *Combinatorial maps* and `combinatorial_map_wrapper()`.

EXAMPLES:

```
sage: sage.combinat.combinatorial_map.combinatorial_map = sage.combinat.combinatorial_map.combin
sage: import imp
sage: _ = imp.reload(sage.combinat.permutation);
sage: from sage.combinat.combinatorial_map import combinatorial_maps_in_class
sage: p = Permutation([1,3,2,4])
sage: cmaps = combinatorial_maps_in_class(p)
sage: cmaps # random
[Combinatorial map: Robinson-Schensted insertion tableau,
 Combinatorial map: Robinson-Schensted recording tableau,
 Combinatorial map: Robinson-Schensted tableau shape,
```

```
 Combinatorial map: complement,
 Combinatorial map: descent composition,
 Combinatorial map: inverse, ...]
sage: p.left_tableau in cmaps
True
sage: p.right_tableau in cmaps
True
sage: p.complement in cmaps
True
```

5.1.27 Integer compositions

A composition c of a nonnegative integer n is a list of positive integers (the *parts* of the composition) with total sum n.

This module provides tools for manipulating compositions and enumerated sets of compositions.

EXAMPLES:

```
sage: Composition([5, 3, 1, 3])
[5, 3, 1, 3]
sage: list(Compositions(4))
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1], [4]]
```

AUTHORS:

- Mike Hansen, Nicolas M. Thiery
- MuPAD-Combinat developers (algorithms and design inspiration)
- Travis Scrimshaw (2013-02-03): Removed `CombinatorialClass`

class `sage.combinat.composition.`**`Composition`**(*parent*, **args*, ***kwds*)

Bases: `sage.combinat.combinat.CombinatorialElement`

Integer compositions

A composition of a nonnegative integer n is a list $(i_1, \ldots, i_k)$ of positive integers with total sum n.

EXAMPLES:

The simplest way to create a composition is by specifying its entries as a list, tuple (or other iterable):

```
sage: Composition([3,1,2])
[3, 1, 2]
sage: Composition((3,1,2))
[3, 1, 2]
sage: Composition(i for i in range(2,5))
[2, 3, 4]
```

You can also create a composition from its code. The *code* of a composition $(i_1, i_2, \ldots, i_k)$ of n is a list of length n that consists of a 1 followed by $i_1 - 1$ zeros, then a 1 followed by $i_2 - 1$ zeros, and so on.

```
sage: Composition([4,1,2,3,5]).to_code()
[1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
sage: Composition(code=_)
[4, 1, 2, 3, 5]
sage: Composition([3,1,2,3,5]).to_code()
[1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
sage: Composition(code=_)
[3, 1, 2, 3, 5]
```

You can also create the composition of n corresponding to a subset of $\{1, 2, \ldots, n-1\}$ under the bijection that maps the composition $(i_1, i_2, \ldots, i_k)$ of n to the subset $\{i_1, i_1 + i_2, i_1 + i_2 + i_3, \ldots, i_1 + \cdots + i_{k-1}\}$ (see `to_subset()`):

```
sage: Composition(from_subset=({1, 2, 4}, 5))
[1, 1, 2, 1]
sage: Composition([1, 1, 2, 1]).to_subset()
{1, 2, 4}
```

The following notation equivalently specifies the composition from the set $\{i_1 - 1, i_1 + i_2 - 1, i_1 + i_2 + i_3 - 1, \ldots, i_1 + \cdots + i_{k-1} - 1, n-1\}$ or $\{i_1 - 1, i_1 + i_2 - 1, i_1 + i_2 + i_3 - 1, \ldots, i_1 + \cdots + i_{k-1} - 1\}$ and n. This provides compatibility with Python's 0-indexing.

```
sage: Composition(descents=[1,0,4,8,11])
[1, 1, 3, 4, 3]
sage: Composition(descents=[0,1,3,4])
[1, 1, 2, 1]
sage: Composition(descents=([0,1,3],5))
[1, 1, 2, 1]
sage: Composition(descents=({0,1,3},5))
[1, 1, 2, 1]
```

EXAMPLES:

```
sage: C = Composition([3,1,2])
sage: TestSuite(C).run()
```

complement()

Return the complement of the composition `self`.

The complement of a composition I is defined as follows:

If I is the empty composition, then the complement is the empty composition as well. Otherwise, let S be the descent set of I (that is, the subset $\{i_1, i_1 + i_2, \ldots, i_1 + i_2 + \cdots + i_{k-1}\}$ of $\{1, 2, \ldots, |I| - 1\}$, where I is written as $(i_1, i_2, \ldots, i_k)$). Then, the complement of I is defined as the composition of size $|I|$ whose descent set is $\{1, 2, \ldots, |I| - 1\} \setminus S$.

The complement of a composition I also is the reverse composition (`reversed()`) of the conjugate (`conjugate()`) of I.

EXAMPLES:

```
sage: Composition([1, 1, 3, 1, 2, 1, 3]).conjugate()
[1, 1, 3, 3, 1, 3]
sage: Composition([1, 1, 3, 1, 2, 1, 3]).complement()
[3, 1, 3, 3, 1, 1]
```

conjugate()

Return the conjugate of the composition `self`.

The conjugate of a composition I is defined as the complement (see `complement()`) of the reverse composition (see `reversed()`) of I.

An equivalent definition of the conjugate goes by saying that the ribbon shape of the conjugate of a composition I is the conjugate of the ribbon shape of I. (The ribbon shape of a composition is returned by `to_skew_partition()`.)

This implementation uses the algorithm from mupad-combinat.

EXAMPLES:

```
sage: Composition([1, 1, 3, 1, 2, 1, 3]).conjugate()
[1, 1, 3, 3, 1, 3]
```

The ribbon shape of the conjugate of I is the conjugate of the ribbon shape of I:

```
sage: all( I.conjugate().to_skew_partition()
....:      == I.to_skew_partition().conjugate()
....:      for I in Compositions(4) )
True
```

TESTS:

```
sage: parent(list(Compositions(1))[0].conjugate())
Compositions of 1
sage: parent(list(Compositions(0))[0].conjugate())
Compositions of 0
```

descents (*final_descent=False*)

This gives one fewer than the partial sums of the composition.

This is here to maintain some sort of backward compatibility, even through the original implementation was broken (it gave the wrong answer). The same information can be found in `partial_sums()`.

See also:

`partial_sums()`

INPUT:

- `final_descent` – (Default: `False`) a boolean integer

OUTPUT:

- the list of partial sums of `self` with each part decremented by 1. This includes the sum of all entries when `final_descent` is `True`.

EXAMPLES:

```
sage: c = Composition([2,1,3,2])
sage: c.descents()
[1, 2, 5]
sage: c.descents(final_descent=True)
[1, 2, 5, 7]
```

fatten (*grouping*)

Return the composition fatter than `self`, obtained by grouping together consecutive parts according to `grouping`.

INPUT:

- `grouping` – a composition whose sum is the length of `self`

EXAMPLES:

Let us start with the composition:

```
sage: c = Composition([4,5,2,7,1])
```

With `grouping` equal to $(1,\ldots,1)$, c is left unchanged:

```
sage: c.fatten(Composition([1,1,1,1,1]))
[4, 5, 2, 7, 1]
```

With `grouping` equal to (ℓ) where ℓ is the length of c, this yields the coarsest composition above c:

```
sage: c.fatten(Composition([5]))
[19]
```

Other values for `grouping` yield (all the) other compositions coarser than c:

```
sage: c.fatten(Composition([2,1,2]))
[9, 2, 8]
sage: c.fatten(Composition([3,1,1]))
[11, 7, 1]
```

TESTS:

```
sage: Composition([]).fatten(Composition([]))
[]
sage: c.fatten(Composition([3,1,1]))).__class__ == c.__class__
True
```

fatter()

Return the set of compositions which are fatter than `self`.

Complexity for generation: $O(|c|)$ memory, $O(|r|)$ time where $|c|$ is the size of `self` and r is the result.

EXAMPLES:

```
sage: C = Composition([4,5,2]).fatter()
sage: C.cardinality()
4
sage: list(C)
[[4, 5, 2], [4, 7], [9, 2], [11]]
```

Some extreme cases:

```
sage: list(Composition([5]).fatter())
[[5]]
sage: list(Composition([]).fatter())
[[]]
sage: list(Composition([1,1,1,1]).fatter()) == list(Compositions(4))
True
```

finer()

Return the set of compositions which are finer than `self`.

EXAMPLES:

```
sage: C = Composition([3,2]).finer()
sage: C.cardinality()
8
sage: C.list()
[[1, 1, 1, 1, 1], [1, 1, 1, 2], [1, 2, 1, 1], [1, 2, 2], [2, 1, 1, 1], [2, 1, 2], [3, 1, 1],

sage: Composition([]).finer()
{[]}
```

inf(*other*, *check=True*)

Return the meet of `self` with a composition `other` of the same size.

The meet of two compositions I and J of size n is the finest composition of n which is coarser than each of I and J. It can be described as the composition whose descent set is the intersection of the descent sets of I and J.

INPUT:

- other – composition of same size as self
- check – (default: True) a Boolean determining whether to check the input compositions for having the same size

OUTPUT:

- the meet of the compositions self and other

EXAMPLES:

```
sage: Composition([3, 1, 1, 3, 1]).meet([4, 3, 2])
[4, 5]
sage: Composition([9, 6]).meet([1, 3, 6, 3, 2])
[15]
sage: Composition([9, 6]).meet([1, 3, 5, 1, 3, 2])
[9, 6]
sage: Composition([1, 1, 1, 1, 1]).meet([3, 2])
[3, 2]
sage: Composition([4, 2]).meet([3, 3])
[6]
sage: Composition([]).meet([])
[]
sage: Composition([1]).meet([1])
[1]
```

Let us verify on small examples that the meet of I and J is coarser than both of I and J:

```
sage: all( all( I.is_finer(I.meet(J)) and
....:           J.is_finer(I.meet(J))
....:           for J in Compositions(4) )
....:      for I in Compositions(4) )
True
```

and is the finest composition to do so:

```
sage: all( all( all( I.meet(J).is_finer(K)
....:                for K in I.fatter()
....:                if J.is_finer(K) )
....:           for J in Compositions(3) )
....:      for I in Compositions(3) )
True
```

The descent set of the meet of I and J is the intersection of the descent sets of I and J:

```
sage: def test_meet(n):
....:     return all( all( I.to_subset().intersection(J.to_subset())
....:                      == I.meet(J).to_subset()
....:                      for J in Compositions(n) )
....:                 for I in Compositions(n) )
sage: all( test_meet(n) for n in range(1, 5) )
True
sage: all( test_meet(n) for n in range(5, 9) )  # long time
True
```

TESTS:

```
sage: Composition([3, 1, 1, 3, 1]).meet([4, 3, 1])
Traceback (most recent call last):
...
ValueError: [3, 1, 1, 3, 1] is not the same size as [4, 3, 1]
```

See also:

join()

AUTHORS:

•Darij Grinberg (2013-09-05)

is_finer(*co2*)

Return True if the composition self is finer than the composition co2; otherwise, return False.

EXAMPLES:

```
sage: Composition([4,1,2]).is_finer([3,1,3])
False
sage: Composition([3,1,3]).is_finer([4,1,2])
False
sage: Composition([1,2,2,1,1,2]).is_finer([5,1,3])
True
sage: Composition([2,2,2]).is_finer([4,2])
True
```

join(*other*, *check=True*)

Return the join of self with a composition other of the same size.

The join of two compositions I and J of size n is the coarsest composition of n which refines each of I and J. It can be described as the composition whose descent set is the union of the descent sets of I and J. It is also the concatenation of $I_1, I_2, \cdots, I_m$, where $I = I_1 \bullet I_2 \bullet \ldots \bullet I_m$ is the ribbon decomposition of I with respect to J (see ribbon_decomposition()).

INPUT:

•other – composition of same size as self

•check – (default: True) a Boolean determining whether to check the input compositions for having the same size

OUTPUT:

•the join of the compositions self and other

EXAMPLES:

```
sage: Composition([3, 1, 1, 3, 1]).join([4, 3, 2])
[3, 1, 1, 2, 1, 1]
sage: Composition([9, 6]).join([1, 3, 6, 3, 2])
[1, 3, 5, 1, 3, 2]
sage: Composition([9, 6]).join([1, 3, 5, 1, 3, 2])
[1, 3, 5, 1, 3, 2]
sage: Composition([1, 1, 1, 1, 1]).join([3, 2])
[1, 1, 1, 1, 1]
sage: Composition([4, 2]).join([3, 3])
[3, 1, 2]
sage: Composition([]).join([])
[]
```

Let us verify on small examples that the join of I and J refines both of I and J:

```
sage: all( all( I.join(J).is_finer(I) and
....:           I.join(J).is_finer(J)
....:           for J in Compositions(4) )
....:      for I in Compositions(4) )
True
```

and is the coarsest composition to do so:

```
sage: all( all( all( K.is_finer(I.join(J))
....:                for K in I.finer()
....:                if K.is_finer(J) )
....:           for J in Compositions(3) )
....:      for I in Compositions(3) )
True
```

Let us check that the join of I and J is indeed the conctenation of $I_1, I_2, \cdots, I_m$, where $I = I_1 \bullet I_2 \bullet \ldots \bullet I_m$ is the ribbon decomposition of I with respect to J:

```
sage: all( all( Composition.sum(I.ribbon_decomposition(J)[0])
....:           == I.join(J) for J in Compositions(4) )
....:      for I in Compositions(4) )
True
```

Also, the descent set of the join of I and J is the union of the descent sets of I and J:

```
sage: all( all( I.to_subset().union(J.to_subset())
....:           == I.join(J).to_subset()
....:           for J in Compositions(4) )
....:      for I in Compositions(4) )
True
```

TESTS:

```
sage: Composition([3, 1, 1, 3, 1]).join([4, 3, 1])
Traceback (most recent call last):
...
ValueError: [3, 1, 1, 3, 1] is not the same size as [4, 3, 1]
```

See also:

`meet()`, `ribbon_decomposition()`

AUTHORS:

- Darij Grinberg (2013-09-05)

major_index()

Return the major index of `self`. The major index is defined as the sum of the descents.

EXAMPLES:

```
sage: Composition([1, 1, 3, 1, 2, 1, 3]).major_index()
31
```

meet(*other*, *check=True*)

Return the meet of `self` with a composition `other` of the same size.

The meet of two compositions I and J of size n is the finest composition of n which is coarser than each of I and J. It can be described as the composition whose descent set is the intersection of the descent sets of I and J.

INPUT:

- `other` – composition of same size as `self`
- `check` – (default: `True`) a Boolean determining whether to check the input compositions for having the same size

OUTPUT:

•the meet of the compositions `self` and `other`

EXAMPLES:

```
sage: Composition([3, 1, 1, 3, 1]).meet([4, 3, 2])
[4, 5]
sage: Composition([9, 6]).meet([1, 3, 6, 3, 2])
[15]
sage: Composition([9, 6]).meet([1, 3, 5, 1, 3, 2])
[9, 6]
sage: Composition([1, 1, 1, 1, 1]).meet([3, 2])
[3, 2]
sage: Composition([4, 2]).meet([3, 3])
[6]
sage: Composition([]).meet([])
[]
sage: Composition([1]).meet([1])
[1]
```

Let us verify on small examples that the meet of I and J is coarser than both of I and J:

```
sage: all( all( I.is_finer(I.meet(J)) and
....:           J.is_finer(I.meet(J))
....:           for J in Compositions(4) )
....:      for I in Compositions(4) )
True
```

and is the finest composition to do so:

```
sage: all( all( all( I.meet(J).is_finer(K)
....:                for K in I.fatter()
....:                if J.is_finer(K) )
....:           for J in Compositions(3) )
....:      for I in Compositions(3) )
True
```

The descent set of the meet of I and J is the intersection of the descent sets of I and J:

```
sage: def test_meet(n):
....:     return all( all( I.to_subset().intersection(J.to_subset())
....:                      == I.meet(J).to_subset()
....:                      for J in Compositions(n) )
....:                 for I in Compositions(n) )
sage: all( test_meet(n) for n in range(1, 5) )
True
sage: all( test_meet(n) for n in range(5, 9) )  # long time
True
```

TESTS:

```
sage: Composition([3, 1, 1, 3, 1]).meet([4, 3, 1])
Traceback (most recent call last):
...
ValueError: [3, 1, 1, 3, 1] is not the same size as [4, 3, 1]
```

See also:

`join()`

AUTHORS:

•Darij Grinberg (2013-09-05)

near_concatenation(*other*)

Return the near-concatenation of two nonempty compositions `self` and `other`.

The near-concatenation $I \odot J$ of two nonempty compositions I and J is defined as the composition $(i_1, i_2, \ldots, i_{n-1}, i_n + j_1, j_2, j_3, \ldots, j_m)$, where $(i_1, i_2, \ldots, i_n) = I$ and $(j_1, j_2, \ldots, j_m) = J$.

This method returns `None` if one of the two input compositions is empty.

EXAMPLES:

```
sage: Composition([1, 1, 3]).near_concatenation(Composition([4, 1, 2]))
[1, 1, 7, 1, 2]
sage: Composition([6]).near_concatenation(Composition([1, 5]))
[7, 5]
sage: Composition([1, 5]).near_concatenation(Composition([6]))
[1, 11]
```

TESTS:

```
sage: Composition([]).near_concatenation(Composition([]))

sage: Composition([]).near_concatenation(Composition([2, 1]))

sage: Composition([3, 2]).near_concatenation(Composition([]))
```

partial_sums(*final=True*)

The partial sums of the sequence defined by the entries of the composition.

If $I = (i_1, \ldots, i_m)$ is a composition, then the partial sums of the entries of the composition are $[i_1, i_1 + i_2, \ldots, i_1 + i_2 + \cdots + i_m]$.

INPUT:

- `final` – (default: `True`) whether or not to include the final partial sum, which is always the size of the composition.

See also:

`to_subset()`

EXAMPLES:

```
sage: Composition([1,1,3,1,2,1,3]).partial_sums()
[1, 2, 5, 6, 8, 9, 12]
```

With `final = False`, the last partial sum is not included:

```
sage: Composition([1,1,3,1,2,1,3]).partial_sums(final=False)
[1, 2, 5, 6, 8, 9]
```

peaks()

Return a list of the peaks of the composition `self`. The peaks of a composition are the descents which do not immediately follow another descent.

EXAMPLES:

```
sage: Composition([1, 1, 3, 1, 2, 1, 3]).peaks()
[4, 7]
```

refinement_splitting(*J*)

Return the refinement splitting of `self` according to `J`.

INPUT:

- J – A composition such that self is finer than J

OUTPUT:

- the unique list of compositions $(I^{(p)})_{p=1,\dots,m}$, obtained by splitting I, such that $|I^{(p)}| = J_p$ for all $p = 1, \dots, m$.

See also:

refinement_splitting_lengths()

EXAMPLES:

```
sage: Composition([1,2,2,1,1,2]).refinement_splitting([5,1,3])
[[1, 2, 2], [1], [1, 2]]
sage: Composition([]).refinement_splitting([])
[]
sage: Composition([3]).refinement_splitting([2])
Traceback (most recent call last):
...
ValueError: compositions self (= [3]) and J (= [2]) must be of the same size
sage: Composition([2,1]).refinement_splitting([1,2])
Traceback (most recent call last):
...
ValueError: composition J (= [2, 1]) does not refine self (= [1, 2])
```

refinement_splitting_lengths(*J*)

Return the lengths of the compositions in the refinement splitting of self according to J.

See also:

refinement_splitting() for the definition of refinement splitting

EXAMPLES:

```
sage: Composition([1,2,2,1,1,2]).refinement_splitting_lengths([5,1,3])
[3, 1, 2]
sage: Composition([]).refinement_splitting_lengths([])
[]
sage: Composition([3]).refinement_splitting_lengths([2])
Traceback (most recent call last):
...
ValueError: compositions self (= [3]) and J (= [2]) must be of the same size
sage: Composition([2,1]).refinement_splitting_lengths([1,2])
Traceback (most recent call last):
...
ValueError: composition J (= [2, 1]) does not refine self (= [1, 2])
```

reversed()

Return the reverse composition of self.

The reverse composition of a composition $(i_1, i_2, \dots, i_k)$ is defined as the composition $(i_k, i_{k-1}, \dots, i_1)$.

EXAMPLES:

```
sage: Composition([1, 1, 3, 1, 2, 1, 3]).reversed()
[3, 1, 2, 1, 3, 1, 1]
```

ribbon_decomposition(*other*, *check=True*)

Return a pair describing the ribbon decomposition of a composition self with respect to a composition other of the same size.

If I and J are two compositions of the same nonzero size, then the ribbon decomposition of I with respect to J is defined as follows: Write I and J as $I = (i_1, i_2, \ldots, i_n)$ and $J = (j_1, j_2, \ldots, j_m)$. Then, the equality $I = I_1 \bullet I_2 \bullet \ldots \bullet I_m$ holds for a unique m-tuple $(I_1, I_2, \ldots, I_m)$ of compositions such that each I_k has size j_k and for a unique choice of $m-1$ signs $\bullet$ each of which is either the concatenation sign $\cdot$ or the near-concatenation sign $\odot$ (see `__add__()` and `near_concatenation()` for the definitions of these two signs). This m-tuple and this choice of signs together are said to form the ribbon decomposition of I with respect to J. If I and J are empty, then the same definition applies, except that there are 0 rather than $m-1$ signs.

See Section 4.8 of [NCSF1].

INPUT:

- `other` – composition of same size as `self`
- `check` – (default: `True`) a Boolean determining whether to check the input compositions for having the same size

OUTPUT:

- a pair `(u, v)`, where `u` is a tuple of compositions (corresponding to the m-tuple $(I_1, I_2, \ldots, I_m)$ in the above definition), and `v` is a tuple of 0'$sand$'1'$s(encodingthechoiceofsigns$' in the above definition, with a 0 standing for $\cdot$ and a 1 standing for $\odot$).

EXAMPLES:

```
sage: Composition([3, 1, 1, 3, 1]).ribbon_decomposition([4, 3, 2])
(([3, 1], [1, 2], [1, 1]), (0, 1))
sage: Composition([9, 6]).ribbon_decomposition([1, 3, 6, 3, 2])
(([1], [3], [5, 1], [3], [2]), (1, 1, 1, 1))
sage: Composition([9, 6]).ribbon_decomposition([1, 3, 5, 1, 3, 2])
(([1], [3], [5], [1], [3], [2]), (1, 1, 0, 1, 1))
sage: Composition([1, 1, 1, 1, 1]).ribbon_decomposition([3, 2])
(([1, 1, 1], [1, 1]), (0,))
sage: Composition([4, 2]).ribbon_decomposition([6])
(([4, 2],), ())
sage: Composition([]).ribbon_decomposition([])
((), ())
```

Let us check that the defining property $I = I_1 \bullet I_2 \bullet \ldots \bullet I_m$ is satisfied:

```
sage: def compose_back(u, v):
....:     comp = u[0]
....:     r = len(v)
....:     if len(u) != r + 1:
....:         raise ValueError("something is wrong")
....:     for i in range(r):
....:         if v[i] == 0:
....:             comp += u[i + 1]
....:         else:
....:             comp = comp.near_concatenation(u[i + 1])
....:     return comp
sage: all( all( all( compose_back(*(I.ribbon_decomposition(J))) == I
....:                for J in Compositions(n) )
....:           for I in Compositions(n) )
....:      for n in range(1, 5) )
True
```

TESTS:

```
sage: Composition([3, 1, 1, 3, 1]).ribbon_decomposition([4, 3, 1])
Traceback (most recent call last):
```

```
...
ValueError: [3, 1, 1, 3, 1] is not the same size as [4, 3, 1]
```

AUTHORS:

- Darij Grinberg (2013-08-29)

shuffle_product (*other*, *overlap=False*)

The (overlapping) shuffles of `self` and `other`.

Suppose $I = (i_1, \ldots, i_k)$ and $J = (j_1, \ldots, j_l)$ are two compositions. A *shuffle* of I and J is a composition of length $k + l$ that contains both I and J as subsequences.

More generally, an *overlapping shuffle* of I and J is obtained by distributing the elements of I and J (preserving the relative ordering of these elements) among the positions of an empty list; an element of I and an element of J are permitted to share the same position, in which case they are replaced by their sum. In particular, a shuffle of I and J is an overlapping shuffle of I and J.

INPUT:

- `other` – composition
- `overlap` – boolean (default: `False`); if `True`, the overlapping shuffle product is returned.

OUTPUT:

An enumerated set (allowing for mutliplicities)

EXAMPLES:

The shuffle product of $[2, 2]$ and $[1, 1, 3]$:

```
sage: alph = Composition([2,2])
sage: beta = Composition([1,1,3])
sage: S = alph.shuffle_product(beta); S
Shuffle product of [2, 2] and [1, 1, 3]
sage: S.list()
[[2, 2, 1, 1, 3], [2, 1, 2, 1, 3], [2, 1, 1, 2, 3], [2, 1, 1, 3, 2], [1, 2, 2, 1, 3], [1, 2,
```

The *overlapping* shuffle product of $[2, 2]$ and $[1, 1, 3]$:

```
sage: alph = Composition([2,2])
sage: beta = Composition([1,1,3])
sage: O = alph.shuffle_product(beta, overlap=True); O
Overlapping shuffle product of [2, 2] and [1, 1, 3]
sage: O.list()
[[2, 2, 1, 1, 3], [2, 1, 2, 1, 3], [2, 1, 1, 2, 3], [2, 1, 1, 3, 2], [1, 2, 2, 1, 3], [1, 2,
```

Note that the shuffle product of two compositions can include the same composition more than once since a composition can be a shuffle of two compositions in several ways. For example:

```
sage: S = Composition([1]).shuffle_product([1]); S
Shuffle product of [1] and [1]
sage: S.list()
[[1, 1], [1, 1]]
sage: O = Composition([1]).shuffle_product([1], overlap=True); O
Overlapping shuffle product of [1] and [1]
sage: O.list()
[[1, 1], [1, 1], [2]]
```

TESTS:

```
sage: Composition([]).shuffle_product([]).list()
[[]]
```

size()

Return the size of `self`, that is the sum of its parts.

EXAMPLES:

```
sage: Composition([7,1,3]).size()
11
```

static sum(*compositions*)

Return the concatenation of the given compositions.

INPUT:

- `compositions` – a list (or iterable) of compositions

EXAMPLES:

```
sage: Composition.sum([Composition([1, 1, 3]), Composition([4, 1, 2]), Composition([3,1])])
[1, 1, 3, 4, 1, 2, 3, 1]
```

Any iterable can be provided as input:

```
sage: Composition.sum([Composition([i,i]) for i in [4,1,3]])
[4, 4, 1, 1, 3, 3]
```

Empty inputs are handled gracefully:

```
sage: Composition.sum([]) == Composition([])
True
```

sup(*other*, *check=True*)

Return the join of `self` with a composition `other` of the same size.

The join of two compositions I and J of size n is the coarsest composition of n which refines each of I and J. It can be described as the composition whose descent set is the union of the descent sets of I and J. It is also the concatenation of $I_1, I_2, \cdots, I_m$, where $I = I_1 \bullet I_2 \bullet \ldots \bullet I_m$ is the ribbon decomposition of I with respect to J (see `ribbon_decomposition()`).

INPUT:

- `other` – composition of same size as `self`
- `check` – (default: `True`) a Boolean determining whether to check the input compositions for having the same size

OUTPUT:

- the join of the compositions `self` and `other`

EXAMPLES:

```
sage: Composition([3, 1, 1, 3, 1]).join([4, 3, 2])
[3, 1, 1, 2, 1, 1]
sage: Composition([9, 6]).join([1, 3, 6, 3, 2])
[1, 3, 5, 1, 3, 2]
sage: Composition([9, 6]).join([1, 3, 5, 1, 3, 2])
[1, 3, 5, 1, 3, 2]
sage: Composition([1, 1, 1, 1, 1]).join([3, 2])
[1, 1, 1, 1, 1]
sage: Composition([4, 2]).join([3, 3])
```

```
[3, 1, 2]
sage: Composition([]).join([])
[]
```

Let us verify on small examples that the join of I and J refines both of I and J:

```
sage: all( all( I.join(J).is_finer(I) and
....:           I.join(J).is_finer(J)
....:           for J in Compositions(4) )
....:      for I in Compositions(4) )
True
```

and is the coarsest composition to do so:

```
sage: all( all( all( K.is_finer(I.join(J))
....:                for K in I.finer()
....:                if K.is_finer(J) )
....:           for J in Compositions(3) )
....:      for I in Compositions(3) )
True
```

Let us check that the join of I and J is indeed the conctenation of $I_1, I_2, \cdots, I_m$, where $I = I_1 \bullet I_2 \bullet \ldots \bullet I_m$ is the ribbon decomposition of I with respect to J:

```
sage: all( all( Composition.sum(I.ribbon_decomposition(J)[0])
....:           == I.join(J) for J in Compositions(4) )
....:      for I in Compositions(4) )
True
```

Also, the descent set of the join of I and J is the union of the descent sets of I and J:

```
sage: all( all( I.to_subset().union(J.to_subset())
....:           == I.join(J).to_subset()
....:           for J in Compositions(4) )
....:      for I in Compositions(4) )
True
```

TESTS:

```
sage: Composition([3, 1, 1, 3, 1]).join([4, 3, 1])
Traceback (most recent call last):
...
ValueError: [3, 1, 1, 3, 1] is not the same size as [4, 3, 1]
```

See also:

meet(), ribbon_decomposition()

AUTHORS:

- Darij Grinberg (2013-09-05)

to_code()

Return the code of the composition `self`. The code of a composition I is a list of length $\mathrm{size}(I)$ of 1s and 0s such that there is a 1 wherever a new part starts. (Exceptional case: When the composition is empty, the code is `[0]`.)

EXAMPLES:

```
sage: Composition([4,1,2,3,5]).to_code()
[1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
```

to_partition()
Return the partition obtained by sorting `self` into decreasing order.

EXAMPLES:

```
sage: Composition([2,1,3]).to_partition()
[3, 2, 1]
sage: Composition([4,2,2]).to_partition()
[4, 2, 2]
sage: Composition([]).to_partition()
[]
```

to_skew_partition(*overlap=1*)
Return the skew partition obtained from `self`. This is a skew partition whose rows have the entries of `self` as their length, taken in reverse order (so the first entry of `self` is the length of the lowermost row, etc.). The parameter `overlap` indicates the number of cells on each row that are directly below cells of the previous row. When it is set to 1 (its default value), the result is the ribbon shape of `self`.

EXAMPLES:

```
sage: Composition([3,4,1]).to_skew_partition()
[6, 6, 3] / [5, 2]
sage: Composition([3,4,1]).to_skew_partition(overlap=0)
[8, 7, 3] / [7, 3]
sage: Composition([]).to_skew_partition()
[] / []
sage: Composition([1,2]).to_skew_partition()
[2, 1] / []
sage: Composition([2,1]).to_skew_partition()
[2, 2] / [1]
```

to_subset(*final=False*)
The subset corresponding to `self` under the bijection (see below) between compositions of n and subsets of $\{1, 2, \ldots, n-1\}$.

The bijection maps a composition $(i_1, \ldots, i_k)$ of n to $\{i_1, i_1 + i_2, i_1 + i_2 + i_3, \ldots, i_1 + \cdots + i_{k-1}\}$.

INPUT:

- `final` – (default: `False`) whether or not to include the final partial sum, which is always the size of the composition.

See also:

`partial_sums()`

EXAMPLES:

```
sage: Composition([1,1,3,1,2,1,3]).to_subset()
{1, 2, 5, 6, 8, 9}
sage: for I in Compositions(3): print I.to_subset()
{1, 2}
{1}
{2}
{}
```

With `final=True`, the sum of all the elements of the composition is included in the subset:

```
sage: Composition([1,1,3,1,2,1,3]).to_subset(final=True)
{1, 2, 5, 6, 8, 9, 12}
```

TESTS:

We verify that `to_subset` is indeed a bijection for compositions of size $n = 8$:

```
sage: n = 8
sage: all(Composition(from_subset=(S, n)).to_subset() == S \
...       for S in Subsets(n-1))
True
sage: all(Composition(from_subset=(I.to_subset(), n)) == I \
...       for I in Compositions(n))
True
```

wll_gt(*co2*)

Return `True` if the composition `self` is greater than the composition `co2` with respect to the wll-ordering; otherwise, return `False`.

The wll-ordering is a total order on the set of all compositions defined as follows: A composition I is greater than a composition J if and only if one of the following conditions holds:

- The size of I is greater than the size of J.
- The size of I equals the size of J, but the length of I is greater than the length of J.
- The size of I equals the size of J, and the length of I equals the length of J, but I is lexicographically greater than J.

("wll-ordering" is short for "weight, length, lexicographic ordering".)

EXAMPLES:

```
sage: Composition([4,1,2]).wll_gt([3,1,3])
True
sage: Composition([7]).wll_gt([4,1,2])
False
sage: Composition([8]).wll_gt([4,1,2])
True
sage: Composition([3,2,2,2]).wll_gt([5,2])
True
sage: Composition([]).wll_gt([3])
False
sage: Composition([2,1]).wll_gt([2,1])
False
sage: Composition([2,2,2]).wll_gt([4,2])
True
sage: Composition([4,2]).wll_gt([2,2,2])
False
sage: Composition([1,1,2]).wll_gt([2,2])
True
sage: Composition([2,2]).wll_gt([1,3])
True
sage: Composition([2,1,2]).wll_gt([])
True
```

class sage.combinat.composition.**Compositions**(*is_infinite=False*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Set of integer compositions.

A composition c of a nonnegative integer n is a list of positive integers with total sum n.

See also:

- Composition

- Partitions
- IntegerVectors

EXAMPLES:

There are 8 compositions of 4:

```
sage: Compositions(4).cardinality()
8
```

Here is the list of them:

```
sage: Compositions(4).list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1], [4]]
```

You can use the `.first()` method to get the 'first' composition of a number:

```
sage: Compositions(4).first()
[1, 1, 1, 1]
```

You can also calculate the 'next' composition given the current one:

```
sage: Compositions(4).next([1,1,2])
[1, 2, 1]
```

If n is not specified, this returns the combinatorial class of all (non-negative) integer compositions:

```
sage: Compositions()
Compositions of non-negative integers
sage: [] in Compositions()
True
sage: [2,3,1] in Compositions()
True
sage: [-2,3,1] in Compositions()
False
```

If n is specified, it returns the class of compositions of n:

```
sage: Compositions(3)
Compositions of 3
sage: list(Compositions(3))
[[1, 1, 1], [1, 2], [2, 1], [3]]
sage: Compositions(3).cardinality()
4
```

The following examples show how to test whether or not an object is a composition:

```
sage: [3,4] in Compositions()
True
sage: [3,4] in Compositions(7)
True
sage: [3,4] in Compositions(5)
False
```

Similarly, one can check whether or not an object is a composition which satisfies further constraints:

```
sage: [4,2] in Compositions(6, inner=[2,2])
True
sage: [4,2] in Compositions(6, inner=[2,3])
False
sage: [4,1] in Compositions(5, inner=[2,1], max_slope = 0)
True
```

An example with incompatible constraints:

```
sage: [4,2] in Compositions(6, inner=[2,2], min_part=3)
False
```

The options `length`, `min_length`, and `max_length` can be used to set length constraints on the compositions. For example, the compositions of 4 of length equal to, at least, and at most 2 are given by:

```
sage: Compositions(4, length=2).list()
[[3, 1], [2, 2], [1, 3]]
sage: Compositions(4, min_length=2).list()
[[3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
sage: Compositions(4, max_length=2).list()
[[4], [3, 1], [2, 2], [1, 3]]
```

Setting both `min_length` and `max_length` to the same value is equivalent to setting `length` to this value:

```
sage: Compositions(4, min_length=2, max_length=2).list()
[[3, 1], [2, 2], [1, 3]]
```

The options `inner` and `outer` can be used to set part-by-part containment constraints. The list of compositions of 4 bounded above by `[3,1,2]` is given by:

```
sage: list(Compositions(4, outer=[3,1,2]))
[[3, 1], [2, 1, 1], [1, 1, 2]]
```

`outer` sets `max_length` to the length of its argument. Moreover, the parts of `outer` may be infinite to clear the constraint on specific parts. This is the list of compositions of 4 of length at most 3 such that the first and third parts are at most 1:

```
sage: Compositions(4, outer=[1,oo,1]).list()
[[1, 3], [1, 2, 1]]
```

This is the list of compositions of 4 bounded below by `[1,1,1]`:

```
sage: Compositions(4, inner=[1,1,1]).list()
[[2, 1, 1], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

The options `min_slope` and `max_slope` can be used to set constraints on the slope, that is the difference `p[i+1]-p[i]` of two consecutive parts. The following is the list of weakly increasing compositions of 4:

```
sage: Compositions(4, min_slope=0).list()
[[4], [2, 2], [1, 3], [1, 1, 2], [1, 1, 1, 1]]
```

Here are the weakly decreasing ones:

```
sage: Compositions(4, max_slope=0).list()
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

The following is the list of compositions of 4 such that two consecutive parts differ by at most one:

```
sage: Compositions(4, min_slope=-1, max_slope=1).list()
[[4], [2, 2], [2, 1, 1], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

The constraints can be combined together in all reasonable ways. This is the list of compositions of 5 of length between 2 and 4 such that the difference between consecutive parts is between -2 and 1:

```
sage: Compositions(5, max_slope=1, min_slope=-2, min_length=2, max_length=4).list()
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2], [2, 1, 1, 1], [1, 2, 2], [1, 2, 1, 1], [1, 1,
```

We can do the same thing with an outer constraint:

```
sage: Compositions(5, max_slope=1, min_slope=-2, min_length=2, max_length=4, outer=[2,5,2]).list
[[2, 3], [2, 2, 1], [2, 1, 2], [1, 2, 2]]
```

However, providing incoherent constraints may yield strange results. It is up to the user to ensure that the inner and outer compositions themselves satisfy the parts and slope constraints.

Note that if you specify `min_part=0`, then the objects produced may have parts equal to zero. This violates the internal assumptions that the composition class makes. Use at your own risk, or preferably consider using `IntegerVectors` instead:

```
sage: Compositions(2, length=3, min_part=0).list()
doctest:...: RuntimeWarning: Currently, setting min_part=0 produces Composition objects which vi
[[2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]

sage: list(IntegerVectors(2, 3))
[[2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]
```

The generation algorithm is constant amortized time, and handled by the generic tool `IntegerListsLex`.

TESTS:

```
sage: C = Compositions(4, length=2)
sage: C == loads(dumps(C))
True

sage: Compositions(6, min_part=2, length=3)
Compositions of the integer 6 satisfying constraints length=3, min_part=2

sage: [2, 1] in Compositions(3, length=2)
True
sage: [2,1,2] in Compositions(5, min_part=1)
True
sage: [2,1,2] in Compositions(5, min_part=2)
False

sage: Compositions(4, length=2).cardinality()
3
sage: Compositions(4, min_length=2).cardinality()
7
sage: Compositions(4, max_length=2).cardinality()
4
sage: Compositions(4, max_part=2).cardinality()
5
sage: Compositions(4, min_part=2).cardinality()
2
sage: Compositions(4, outer=[3,1,2]).cardinality()
3

sage: Compositions(4, length=2).list()
[[3, 1], [2, 2], [1, 3]]
sage: Compositions(4, min_length=2).list()
[[3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
sage: Compositions(4, max_length=2).list()
[[4], [3, 1], [2, 2], [1, 3]]
sage: Compositions(4, max_part=2).list()
[[2, 2], [2, 1, 1], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
sage: Compositions(4, min_part=2).list()
[[4], [2, 2]]
sage: Compositions(4, outer=[3,1,2]).list()
```

```
[[3, 1], [2, 1, 1], [1, 1, 2]]
sage: Compositions(3, outer = Composition([3,2])).list()
[[3], [2, 1], [1, 2]]
sage: Compositions(4, outer=[1,oo,1]).list()
[[1, 3], [1, 2, 1]]
sage: Compositions(4, inner=[1,1,1]).list()
[[2, 1, 1], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
sage: Compositions(4, inner=Composition([1,2])).list()
[[2, 2], [1, 3], [1, 2, 1]]
sage: Compositions(4, min_slope=0).list()
[[4], [2, 2], [1, 3], [1, 1, 2], [1, 1, 1, 1]]
sage: Compositions(4, min_slope=-1, max_slope=1).list()
[[4], [2, 2], [2, 1, 1], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
sage: Compositions(5, max_slope=1, min_slope=-2, min_length=2, max_length=4).list()
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2], [2, 1, 1, 1], [1, 2, 2], [1, 2, 1, 1], [1, 1,
sage: Compositions(5, max_slope=1, min_slope=-2, min_length=2, max_length=4, outer=[2,5,2]).list
[[2, 3], [2, 2, 1], [2, 1, 2], [1, 2, 2]]
```

Element
alias of Composition

from_code(*code*)
Return the composition from its code. The code of a composition I is a list of length $\mathrm{size}(I)$ consisting of 1s and 0s such that there is a 1 wherever a new part starts. (Exceptional case: When the composition is empty, the code is `[0]`.)

EXAMPLES:

```
sage: Composition([4,1,2,3,5]).to_code()
[1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
sage: Compositions().from_code(_)
[4, 1, 2, 3, 5]
sage: Composition([3,1,2,3,5]).to_code()
[1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
sage: Compositions().from_code(_)
[3, 1, 2, 3, 5]
```

from_descents(*descents*, *nps=None*)
Return a composition from the list of descents.

INPUT:

- `descents` – an iterable
- `nps` – (default: `None`) an integer or `None`

OUTPUT:

- The composition of `nps` whose descents are listed in `descents`, assuming that `nps` is not `None` (otherwise, the last element of `descents` is removed from `descents`, and `nps` is set to be this last element plus 1).

EXAMPLES:

```
sage: [x-1 for x in Composition([1, 1, 3, 4, 3]).to_subset()]
[0, 1, 4, 8]
sage: Compositions().from_descents([1,0,4,8],12)
[1, 1, 3, 4, 3]
sage: Compositions().from_descents([1,0,4,8,11])
[1, 1, 3, 4, 3]
```

from_subset(*S*, *n*)

The composition of n corresponding to the subset `S` of $\{1, 2, \ldots, n-1\}$ under the bijection that maps the composition $(i_1, i_2, \ldots, i_k)$ of n to the subset $\{i_1, i_1 + i_2, i_1 + i_2 + i_3, \ldots, i_1 + \cdots + i_{k-1}\}$ (see `Composition.to_subset()`).

INPUT:

- `S` – an iterable, a subset of $\{1, 2, \ldots, n-1\}$
- `n` – an integer

EXAMPLES:

```
sage: Compositions().from_subset([2,1,5,9], 12)
[1, 1, 3, 4, 3]
sage: Compositions().from_subset({2,1,5,9}, 12)
[1, 1, 3, 4, 3]

sage: Compositions().from_subset([], 12)
[12]
sage: Compositions().from_subset([], 0)
[]
```

TESTS:

```
sage: Compositions().from_subset([2,1,5,9],9)
Traceback (most recent call last):
...
ValueError: S (=[1, 2, 5, 9]) is not a subset of {1, ..., 8}
```

class sage.combinat.composition.**Compositions_all**

Bases: sage.combinat.composition.Compositions

Class of all compositions.

subset(*size=None*)

Return the set of compositions of the given size.

EXAMPLES:

```
sage: C = Compositions()
sage: C.subset(4)
Compositions of 4
sage: C.subset(size=3)
Compositions of 3
```

class sage.combinat.composition.**Compositions_constraints**(**args*, ***kwds*)

Bases: sage.combinat.integer_lists.invlex.IntegerListsLex

Initialize `self`.

TESTS:

```
sage: from sage.combinat.integer_lists import IntegerLists
sage: C = IntegerLists(2, length=3)
sage: C == loads(dumps(C))
True
```

class sage.combinat.composition.**Compositions_n**(*n*)

Bases: sage.combinat.composition.Compositions

Class of compositions of a fixed n.

cardinality()
Return the number of compositions of n.

TESTS:

```
sage: Compositions(3).cardinality()
4
sage: Compositions(0).cardinality()
1
```

random_element()
Return a random Composition with uniform probability.

This method generates a random binary word starting with a 1 and then uses the bijection between compositions and their code.

EXAMPLES:

```
sage: Compositions(5).random_element() # random
[2, 1, 1, 1]
sage: Compositions(0).random_element()
[]
sage: Compositions(1).random_element()
[1]
```

TESTS:

```
sage: all([Compositions(10).random_element() in Compositions(10) for i in range(20)])
True
```

sage.combinat.composition.**composition_iterator_fast**(n)
Iterator over compositions of n yielded as lists.

TESTS:

```
sage: from sage.combinat.composition import composition_iterator_fast
sage: L = list(composition_iterator_fast(4)); L
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1], [4]]
sage: type(L[0])
<type 'list'>
```

5.1.28 Signed Compositions

class sage.combinat.composition_signed.**SignedCompositions**(n)
Bases: sage.combinat.composition.Compositions_n

The class of signed compositions of n.

EXAMPLES:

```
sage: SC3 = SignedCompositions(3); SC3
Signed compositions of 3
sage: SC3.cardinality()
18
sage: len(SC3.list())
18
sage: SC3.first()
[1, 1, 1]
sage: SC3.last()
[-3]
sage: SC3.random_element() # random
```

```
[1, -1, 1]
sage: SC3.list()
[[1, 1, 1],
 [1, 1, -1],
 [1, -1, 1],
 [1, -1, -1],
 [-1, 1, 1],
 [-1, 1, -1],
 [-1, -1, 1],
 [-1, -1, -1],
 [1, 2],
 [1, -2],
 [-1, 2],
 [-1, -2],
 [2, 1],
 [2, -1],
 [-2, 1],
 [-2, -1],
 [3],
 [-3]]
```

TESTS:

```
sage: SC = SignedCompositions(3)
sage: TestSuite(SC).run()
```

cardinality()

Return the number of elements in `self`.

The number of signed compositions of n is equal to

$$\sum_{i=1}^{n+1} \binom{n-1}{i-1} 2^i$$

TESTS:

```
sage: SC4 = SignedCompositions(4)
sage: SC4.cardinality() == len(SC4.list())
True
sage: SignedCompositions(3).cardinality()
18
```

5.1.29 Composition Tableaux

AUTHORS:

- Chris Berg, Jeff Ferreira (2012-9): Initial version

class sage.combinat.composition_tableau.**CompositionTableau**(*parent*, *t*)

Bases: sage.combinat.combinat.CombinatorialElement

A composition tableau.

A *composition tableau* t of shape $I = (I_1, \ldots, I_\ell)$ is an array of boxes in rows, I_i boxes in row i, filled with positive integers such that:

1.the entries in the rows of t weakly decrease from left to right,

2.the left-most column of t strictly increase from top to bottom.

3. Add zero entries to the rows of t until the resulting array is rectangular of shape $\ell \times m$. For $1 \leq i < j \leq \ell, 2 \leq k \leq m$ and $(t(j,k) \neq 0$, and also if $t(j,k) \geq t(i,k))$ implies $t(j,k) > t(i,k-1)$.

INPUT:

- `t` – A list of lists

EXAMPLES:

```
sage: CompositionTableau([[1],[2,2]])
[[1], [2, 2]]
sage: CompositionTableau([[1],[3,2],[4,4]])
[[1], [3, 2], [4, 4]]
sage: CompositionTableau([])
[]
```

descent_composition()

Return the composition corresponding to the set of all i that do not have $i+1$ appearing strictly to the left of i in `self`.

EXAMPLES:

```
sage: CompositionTableau([[1],[3,2],[4,4]]).descent_composition()
[1, 2, 2]
```

descent_set()

Return the set of all i that do *not* have $i+1$ appearing strictly to the left of i in `self`.

EXAMPLES:

```
sage: CompositionTableau([[1],[3,2],[4,4]]).descent_set()
[1, 3]
```

is_standard()

Return `True` if `self` is a standard composition tableau and `False` otherwise.

EXAMPLES:

```
sage: CompositionTableau([[1,1],[3,2],[4,4,3]]).is_standard()
False
sage: CompositionTableau([[2,1],[3],[4]]).is_standard()
True
```

pp()

Return a pretty print string of `self`.

EXAMPLES:

```
sage: CompositionTableau([[1],[3,2],[4,4]]).pp()
1
3  2
4  4
```

shape_composition()

Return a Composition object which is the shape of `self`.

EXAMPLES:

```
sage: CompositionTableau([[1,1],[3,2],[4,4,3]]).shape_composition()
[2, 2, 3]
sage: CompositionTableau([[2,1],[3],[4]]).shape_composition()
[2, 1, 1]
```

shape_partition()
Return a partition which is the shape of `self` sorted into weakly decreasing order.

EXAMPLES:

```
sage: CompositionTableau([[1,1],[3,2],[4,4,3]]).shape_partition()
[3, 2, 2]
sage: CompositionTableau([[2,1],[3],[4]]).shape_partition()
[2, 1, 1]
```

size()
Return the number of boxes in `self`.

EXAMPLES:

```
sage: CompositionTableau([[1],[3,2],[4,4]]).size()
5
```

weight()
Return a composition where entry i is the number of times that i appears in `self`.

EXAMPLES:

```
sage: CompositionTableau([[1],[3,2],[4,4]]).weight()
[1, 1, 1, 2, 0]
```

class sage.combinat.composition_tableau.**CompositionTableaux**(***kwds*)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Composition tableaux.

INPUT:

Keyword arguments:

- `size` – the size of the composition tableaux
- `shape` – the shape of the composition tableaux
- `max_entry` – the maximum entry for the composition tableaux

Positional arguments:

- The first argument is interpreted as `size` or `shape` depending on whether it is an integer or a composition.

EXAMPLES:

```
sage: CT = CompositionTableaux(3); CT
Composition Tableaux of size 3 and maximum entry 3
sage: list(CT)
[[[1], [2], [3]],
 [[1], [2, 2]],
 [[1], [3, 2]],
 [[1], [3, 3]],
 [[2], [3, 3]],
 [[1, 1], [2]],
 [[1, 1], [3]],
 [[2, 1], [3]],
 [[2, 2], [3]],
 [[1, 1, 1]],
 [[2, 1, 1]],
 [[2, 2, 1]],
 [[2, 2, 2]],
```

```
 [[3, 1, 1]],
 [[3, 2, 1]],
 [[3, 2, 2]],
 [[3, 3, 1]],
 [[3, 3, 2]],
 [[3, 3, 3]]]

sage: CT = CompositionTableaux([1,2,1]); CT
Composition tableaux of shape [1, 2, 1] and maximun entry 4
sage: list(CT)
[[[1], [2, 2], [3]],
 [[1], [2, 2], [4]],
 [[1], [3, 2], [4]],
 [[1], [3, 3], [4]],
 [[2], [3, 3], [4]]]

sage: CT = CompositionTableaux(shape=[1,2,1],max_entry=3); CT
Composition tableaux of shape [1, 2, 1] and maximun entry 3
sage: list(CT)
[[[1], [2, 2], [3]]]

sage: CT = CompositionTableaux(2,max_entry=3); CT
Composition Tableaux of size 2 and maximum entry 3
sage: list(CT)
[[[1], [2]],
 [[1], [3]],
 [[2], [3]],
 [[1, 1]],
 [[2, 1]],
 [[2, 2]],
 [[3, 1]],
 [[3, 2]],
 [[3, 3]]]

sage: CT = CompositionTableaux(0); CT
Composition Tableaux of size 0 and maximum entry 0
sage: list(CT)
[[]]
```

Element
alias of CompositionTableau

class sage.combinat.composition_tableau.**CompositionTableauxBacktracker**(*shape*, *max_entry=None*)

Bases: sage.combinat.backtrack.GenericBacktracker

A backtracker class for generating sets of composition tableaux.

get_next_pos(*ii*, *jj*)

EXAMPLES:

```
sage: from sage.combinat.composition_tableau import CompositionTableauxBacktracker
sage: T = CompositionTableau([[2,1],[5,4,3,2],[6],[7,7,6]])
sage: n = CompositionTableauxBacktracker(T.shape_composition())
sage: n.get_next_pos(1,1)
(1, 2)
```

class sage.combinat.composition_tableau.**CompositionTableaux_all**(*max_entry=None*)

Bases: sage.combinat.composition_tableau.CompositionTableaux,

sage.sets.disjoint_union_enumerated_sets.DisjointUnionEnumeratedSets

All composition tableaux.

an_element()
Return a particular element of self.

EXAMPLES:
```
sage: CT = CompositionTableaux()
sage: CT.an_element()
[[1, 1], [2]]
```

class sage.combinat.composition_tableau.**CompositionTableaux_shape**(*comp*, *max_entry=None*)
Bases: sage.combinat.composition_tableau.CompositionTableaux

Composition tableaux of a fixed shape comp with a given max entry.

INPUT:

- comp – a composition.
- max_entry – a nonnegative integer. This keyword argument defaults to the size of comp.

an_element()
Return a particular element of CompositionTableaux_shape.

EXAMPLES:
```
sage: CT = CompositionTableaux([1,2,1])
sage: CT.an_element()
[[1], [2, 2], [3]]
```

class sage.combinat.composition_tableau.**CompositionTableaux_size**(*n*, *max_entry=None*)
Bases: sage.combinat.composition_tableau.CompositionTableaux

Composition tableaux of a fixed size n.

INPUT:

- n – a nonnegative integer.
- max_entry – a nonnegative integer. This keyword argument defaults to n.

OUTUT:

- The class of composition tableaux of size n.

5.1.30 Cores

A k-core is a partition from which no rim hook of size k can be removed. Alternatively, a k-core is an integer partition such that the Ferrers diagram for the partition contains no cells with a hook of size (a multiple of) k.

Authors:

- Anne Schilling and Mike Zabrocki (2011): initial version
- Travis Scrimshaw (2012): Added latex output for Core class

class sage.combinat.core.**Core**(*parent*, *core*)
Bases: sage.combinat.combinat.CombinatorialElement

A k-core is an integer partition from which no rim hook of size k can be removed.

EXAMPLES:

```
sage: c = Core([2,1],4); c
[2, 1]
sage: c = Core([3,1],4); c
Traceback (most recent call last):
...
ValueError: [3, 1] is not a 4-core
```

affine_symmetric_group_action(*w*, *transposition=False*)

Returns the (left) action of the affine symmetric group on `self`.

INPUT:

- w is a tupe of integers $[w_1, \ldots, w_m]$ with $0 \leq w_j < k$. If transposition is set to be True, then $w = [w_0, w_1]$ is interpreted as a transposition t_{w_0,w_1} (see `_transposition_to_reduced_word()`).

The output is the (left) action of the product of the corresponding simple transpositions on `self`, that is $s_{w_1} \cdots s_{w_m}(self)$. See `affine_symmetric_group_simple_action()`.

EXAMPLES:

```
sage: c = Core([4,2],3)
sage: c.affine_symmetric_group_action([0,1,0,2,1])
[8, 6, 4, 2]
sage: c.affine_symmetric_group_action([0,2], transposition=True)
[4, 2, 1, 1]

sage: c = Core([11,8,5,5,3,3,1,1,1],4)
sage: c.affine_symmetric_group_action([2,5],transposition=True)
[11, 8, 7, 6, 5, 4, 3, 2, 1]
```

affine_symmetric_group_simple_action(*i*)

Returns the action of the simple transposition s_i of the affine symmetric group on `self`.

This gives the action of the affine symmetric group of type $A_k^{(1)}$ on the k-core `self`. If `self` has outside (resp. inside) corners of content i modulo k, then these corners are added (resp. removed). Otherwise the action is trivial.

EXAMPLES:

```
sage: c = Core([4,2],3)
sage: c.affine_symmetric_group_simple_action(0)
[3, 1]
sage: c.affine_symmetric_group_simple_action(1)
[5, 3, 1]
sage: c.affine_symmetric_group_simple_action(2)
[4, 2]
```

This action corresponds to the left action by the i-th simple reflection in the affine symmetric group:

```
sage: c = Core([4,2],3)
sage: W = c.to_grassmannian().parent()
sage: i=0
sage: c.affine_symmetric_group_simple_action(i).to_grassmannian() == W.simple_reflection(i)*
True
sage: i=1
sage: c.affine_symmetric_group_simple_action(i).to_grassmannian() == W.simple_reflection(i)*
True
```

contains(*other*)

Checks whether `self` contains `other`.

INPUT:

- other – another k-core or a list

OUTPUT: a boolean

Returns True if the Ferrers diagram of self contains the Ferrers diagram of other.

EXAMPLES:

```
sage: c = Core([4,2],3)
sage: x = Core([4,2,2,1,1],3)
sage: x.contains(c)
True
sage: c.contains(x)
False
```

k()

Returns k of the k-core self.

EXAMPLES:

```
sage: c = Core([2,1],4)
sage: c.k()
4
```

length()

Returns the length of self.

The length of a k-core is the size of the corresponding $(k-1)$-bounded partition which agrees with the length of the corresponding Grassmannian element, see to_grassmannian().

EXAMPLES:

```
sage: c = Core([4,2],3); c.length()
4
sage: c.to_grassmannian().length()
4

sage: Core([9,5,3,2,1,1], 5).length()
13
```

size()

Returns the size of self as a partition.

EXAMPLES:

```
sage: Core([2,1],4).size()
3
sage: Core([4,2],3).size()
6
```

strong_covers()

Returns a list of all elements that cover self in strong order.

EXAMPLES:

```
sage: c = Core([1],3)
sage: c.strong_covers()
[[2], [1, 1]]
sage: c = Core([4,2],3)
sage: c.strong_covers()
[[5, 3, 1], [4, 2, 1, 1]]
```

strong_down_list()
Returns a list of all elements that are covered by `self` in strong order.

EXAMPLES:
```
sage: c = Core([1],3)
sage: c.strong_down_list()
[[]]
sage: c = Core([5,3,1],3)
sage: c.strong_down_list()
[[4, 2], [3, 1, 1]]
```

strong_le(*other*)
Strong order (Bruhat) comparison on cores.

INPUT:

- `other` – another k-core

OUTPUT: a boolean

Returns whether `self` <= `other` in Bruhat (or strong) order.

EXAMPLES:
```
sage: c = Core([4,2],3)
sage: x = Core([4,2,2,1,1],3)
sage: c.strong_le(x)
True
sage: c.strong_le([4,2,2,1,1])
True

sage: x = Core([4,1],4)
sage: c.strong_le(x)
Traceback (most recent call last):
...
ValueError: The two cores do not have the same k
```

to_bounded_partition()
Bijection between k-cores and $(k-1)$-bounded partitions.

Maps the k-core `self` to the corresponding $(k-1)$-bounded partition. This bijection is achieved by deleting all cells in `self` of hook length greater than k.

EXAMPLES:
```
sage: gamma = Core([9,5,3,2,1,1], 5)
sage: gamma.to_bounded_partition()
[4, 3, 2, 2, 1, 1]
```

to_grassmannian()
Bijection between k-cores and Grassmannian elements in the affine Weyl group of type $A^{(1)}_{k-1}$.

For further details, see the documentation of the method `from_kbounded_to_reduced_word()` and `from_kbounded_to_grassmannian()`.

EXAMPLES:
```
sage: c = Core([3,1,1],3)
sage: w = c.to_grassmannian(); w
[-1  1  1]
[-2  2  1]
[-2  1  2]
sage: c.parent()
```

```
3-Cores of length 4
sage: w.parent()
Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root space)

sage: c = Core([],3)
sage: c.to_grassmannian()
[1 0 0]
[0 1 0]
[0 0 1]
```

to_partition()

Turns the core `self` into the partition identical to `self`.

EXAMPLES:

```
sage: mu = Core([2,1,1],3)
sage: mu.to_partition()
[2, 1, 1]
```

weak_covers()

Returns a list of all elements that cover `self` in weak order.

EXAMPLES:

```
sage: c = Core([1],3)
sage: c.weak_covers()
[[2], [1, 1]]

sage: c = Core([4,2],3)
sage: c.weak_covers()
[[5, 3, 1]]
```

weak_le(*other*)

Weak order comparison on cores.

INPUT:

- `other` – another k-core

OUTPUT: a boolean

Returns whether `self` <= `other` in weak order.

EXAMPLES:

```
sage: c = Core([4,2],3)
sage: x = Core([5,3,1],3)
sage: c.weak_le(x)
True
sage: c.weak_le([5,3,1])
True

sage: x = Core([4,2,2,1,1],3)
sage: c.weak_le(x)
False

sage: x = Core([5,3,1],6)
sage: c.weak_le(x)
Traceback (most recent call last):
...
ValueError: The two cores do not have the same k
```

sage.combinat.core.**Cores**(*k*, *length=None*, ***kwargs*)

A k-core is a partition from which no rim hook of size k can be removed. Alternatively, a k-core is an integer partition such that the Ferrers diagram for the partition contains no cells with a hook of size (a multiple of) k.

The k-cores generally have two notions of size which are useful for different applications. One is the number of cells in the Ferrers diagram with hook less than k, the other is the total number of cells of the Ferrers diagram. In the implementation in Sage, the first of notion is referred to as the `length` of the k-core and the second is the `size` of the k-core. The class of Cores requires that either the size or the length of the elements in the class is specified.

EXAMPLES:

We create the set of the 4-cores of length 6. Here the length of a k-core is the size of the corresponding $(k-1)$-bounded partition, see also `length()`:

```
sage: C = Cores(4, 6); C
4-Cores of length 6
sage: C.list()
[[6, 3], [5, 2, 1], [4, 1, 1, 1], [4, 2, 2], [3, 3, 1, 1], [3, 2, 1, 1, 1], [2, 2, 2, 1, 1, 1]]
sage: C.cardinality()
7
sage: C.an_element()
[6, 3]
```

We may also list the set of 4-cores of size 6, where the size is the number of boxes in the core, see also `size()`:

```
sage: C = Cores(4, size=6); C
4-Cores of size 6
sage: C.list()
[[4, 1, 1], [3, 2, 1], [3, 1, 1, 1]]
sage: C.cardinality()
3
sage: C.an_element()
[4, 1, 1]
```

class sage.combinat.core.**Cores_length**(*k*, *n*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The class of k-cores of length n.

Element

alias of `Core`

from_partition(*part*)

Converts the partition `part` into a core (as the identity map).

This is the inverse method to `to_partition()`.

EXAMPLES:

```
sage: C = Cores(3,4)
sage: c = C.from_partition([4,2]); c
[4, 2]

sage: mu = Partition([2,1,1])
sage: C = Cores(3,3)
sage: C.from_partition(mu).to_partition() == mu
True

sage: mu = Partition([])
sage: C = Cores(3,0)
```

```
sage: C.from_partition(mu).to_partition() == mu
True
```

list()
Returns the list of all k-cores of length n.

EXAMPLES:

```
sage: C = Cores(3,4)
sage: C.list()
[[4, 2], [3, 1, 1], [2, 2, 1, 1]]
```

class sage.combinat.core.**Cores_size**(k, n)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The class of k-cores of size n.

Element
alias of Core

from_partition(*part*)
Converts the partition `part` into a core (as the identity map).

This is the inverse method to `to_partition()`.

EXAMPLES:

```
sage: C = Cores(3,size=4)
sage: c = C.from_partition([2,1,1]); c
[2, 1, 1]

sage: mu = Partition([2,1,1])
sage: C = Cores(3,size=4)
sage: C.from_partition(mu).to_partition() == mu
True

sage: mu = Partition([])
sage: C = Cores(3,size=0)
sage: C.from_partition(mu).to_partition() == mu
True
```

list()
Returns the list of all k-cores of size n.

EXAMPLES:

```
sage: C = Cores(3, size = 4)
sage: C.list()
[[3, 1], [2, 1, 1]]
```

5.1.31 Counting

- *The On-Line Encyclopedia of Integer Sequences (OEIS)*
- *Functions that compute some of the sequences in Sloane's tables*
- *Compute Bell and Uppuluri-Carpenter numbers*

- *q-Analogues*, *-Bernoulli Numbers and Polynomials*
- *Binary Recurrence Sequences.*
- *C-Finite Sequences*
- *Combinatorial Functions*

Todo

Mention sage/combinat/degree_sequences?

5.1.32 Crystals

Quickref

Todo

Write it!

Introductory material

- *An introduction to crystals*
- The Lie Methods and Related Combinatorics thematic tutorial

Catalogs of crystals

- *Catalog Of Crystals*

See also

- The categories for crystals: `Crystals`, `HighestWeightCrystals`, `FiniteCrystals`, `ClassicalCrystals`, `RegularCrystals` – The categories for crystals
- *Root Systems*

5.1.33 Affine Crystals

class `sage.combinat.crystals.affine.`**`AffineCrystalFromClassical`**(*cartan_type*, *classical_crystal*, *category=None*)

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.Parent`

This abstract class can be used for affine crystals that are constructed from a classical crystal. The zero arrows can be implemented using different methods (for example using a Dynkin diagram automorphisms or virtual crystals).

This is a helper class, mostly used to implement Kirillov-Reshetikhin crystals (see: `KirillovReshetikhinCrystal()`).

For general information about crystals see `sage.combinat.crystals`.

INPUT:

- `cartan_type` – the Cartan type of the resulting affine crystal
- `classical_crystal` – instance of a classical crystal

EXAMPLES:

```
sage: n = 2
sage: C = crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A = crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: A.list()
[[[1]], [[2]], [[3]]]
sage: A.cartan_type()
['A', 2, 1]
sage: A.index_set()
(0, 1, 2)
sage: b=A(rows=[[1]])
sage: b.weight()
-Lambda[0] + Lambda[1]
sage: b.classical_weight()
(1, 0, 0)
sage: [x.s(0) for x in A.list()]
[[[3]], [[2]], [[1]]]
sage: [x.s(1) for x in A.list()]
[[[2]], [[1]], [[3]]]
```

Element
alias of `AffineCrystalFromClassicalElement`

lift(*affine_elt*)
Lift an affine crystal element to the corresponding classical crystal element.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: b=A.list()[0]
sage: A.lift(b)
[[1]]
sage: A.lift(b).parent()
The crystal of tableaux of type ['A', 2] and shape(s) [[1]]
```

list()
Return the list of all crystal elements using the underlying classical crystal.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: A.list()
[[[1]], [[2]], [[3]]]
```

retract(*classical_elt*)

Transform a classical crystal element to the corresponding affine crystal element.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: t=C(rows=[[1]])
sage: t.parent()
The crystal of tableaux of type ['A', 2] and shape(s) [[1]]
sage: A.retract(t)
[[1]]
sage: A.retract(t).parent() is A
True
```

class sage.combinat.crystals.affine.**AffineCrystalFromClassicalAndPromotion**(*cartan_type*, *classical_crystal*, *p_automorphism*, *p_inverse_automorphism*, *dynkin_node*)

Bases: sage.combinat.crystals.affine.AffineCrystalFromClassical

Crystals that are constructed from a classical crystal and a Dynkin diagram automorphism σ. In type A_n, the Dynkin diagram automorphism is $i \to i+1 \pmod{n}+1$ and the corresponding map on the crystal is the promotion operation pr on tableaux. The affine crystal operators are given by $f_0 = \mathrm{pr}^{-1} f_{\sigma(0)} \mathrm{pr}$.

INPUT:

- `cartan_type` – the Cartan type of the resulting affine crystal
- `classical_crystal` – instance of a classical crystal
- `automorphism, inverse_automorphism` – a function on the elements of the `classical_crystal`
- `dynkin_node` – an integer specifying the classical node in the image of the zero node under the automorphism sigma

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: A.list()
[[[1]], [[2]], [[3]]]
sage: A.cartan_type()
['A', 2, 1]
sage: A.index_set()
(0, 1, 2)
sage: b=A(rows=[[1]])
sage: b.weight()
-Lambda[0] + Lambda[1]
sage: b.classical_weight()
(1, 0, 0)
sage: [x.s(0) for x in A.list()]
```

```
[[[3]], [[2]], [[1]]]
sage: [x.s(1) for x in A.list()]
[[[2]], [[1]], [[3]]]
```

Element
alias of AffineCrystalFromClassicalAndPromotionElement

automorphism(*x*)
Give the analogue of the affine Dynkin diagram automorphism on the level of crystals.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: b=A.list()[0]
sage: A.automorphism(b)
[[2]]
```

inverse_automorphism(*x*)
Give the analogue of the inverse of the affine Dynkin diagram automorphism on the level of crystals.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: b=A.list()[0]
sage: A.inverse_automorphism(b)
[[3]]
```

class sage.combinat.crystals.affine.**AffineCrystalFromClassicalAndPromotionElement**
Bases: sage.combinat.crystals.affine.AffineCrystalFromClassicalElement

Elements of crystals that are constructed from a classical crystal and a Dynkin diagram automorphism. In type A, the automorphism is the promotion operation on tableaux.

This class is not instantiated directly but rather __call__-ed from AffineCrystalFromClassicalAndPromotion. The syntax of this is governed by the (classical) crystal.

Since this class inherits from AffineCrystalFromClassicalElement, the methods that need to be implemented are e0(), f0() and possibly epsilon0() and phi0() if more efficient algorithms exist.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: b=A(rows=[[1]])
sage: b._repr_()
'[[1]]'
```

e0()
Implements e_0 using the automorphism as $e_0 = \mathrm{pr}^{-1}\, e_{dynkin_node}\, \mathrm{pr}$

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: b=A(rows=[[1]])
sage: b.e0()
[[3]]
```

epsilon0()

Implements $epsilon_0$ using the automorphism.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: [x.epsilon0() for x in A.list()]
[1, 0, 0]
```

f0()

Implements f_0 using the automorphism as $f_0 = \mathrm{pr}^{-1} f_{dynkin_n ode} \mathrm{pr}$

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: b=A(rows=[[3]])
sage: b.f0()
[[1]]
```

phi0()

Implements phi_0 using the automorphism.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: [x.phi0() for x in A.list()]
[0, 0, 1]
```

class sage.combinat.crystals.affine.**AffineCrystalFromClassicalElement**

Bases: sage.structure.element_wrapper.ElementWrapper

Elements of crystals that are constructed from a classical crystal. The elements inherit many of their methods from the classical crystal using lift and retract.

This class is not instantiated directly but rather __call__-ed from AffineCrystalFromClassical. The syntax of this is governed by the (classical) crystal.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: b=A(rows=[[1]])
sage: b._repr_()
'[[1]]'
```

classical_weight()

Return the classical weight corresponding to `self`.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: b=A(rows=[[1]])
sage: b.classical_weight()
(1, 0, 0)
```

e(i)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: b=A(rows=[[1]])
sage: b.e(0)
[[3]]
sage: b.e(1)
```

e0()

Assumes that e_0 is implemented separately.

epsilon(i)

Return the maximal time the crystal operator e_i can be applied to `self`.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: [x.epsilon(0) for x in A.list()]
[1, 0, 0]
sage: [x.epsilon(1) for x in A.list()]
[0, 1, 0]
```

epsilon0()

Uses ε_0 from the super class, but should be implemented if a faster implementation exists.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: [x.epsilon0() for x in A.list()]
[1, 0, 0]
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: b=A(rows=[[3]])
sage: b.f(0)
[[1]]
sage: b.f(2)
```

f0()

Assumes that f_0 is implemented separately.

lift()

Lift an affine crystal element to the corresponding classical crystal element.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: b=A.list()[0]
sage: b.lift()
[[1]]
sage: b.lift().parent()
The crystal of tableaux of type ['A', 2] and shape(s) [[1]]
```

phi(*i*)

Returns the maximal time the crystal operator f_i can be applied to self.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: [x.phi(0) for x in A.list()]
[0, 0, 1]
sage: [x.phi(1) for x in A.list()]
[1, 0, 0]
```

phi0()

Uses φ_0 from the super class, but should be implemented if a faster implementation exists.

EXAMPLES:

```
sage: n=2
sage: C=crystals.Tableaux(['A',n],shape=[1])
sage: pr = attrcall("promotion")
sage: pr_inverse = attrcall("promotion_inverse")
sage: A=crystals.AffineFromClassicalAndPromotion(['A',n,1],C,pr,pr_inverse,1)
sage: [x.phi0() for x in A.list()]
[0, 0, 1]
```

pp()

Method for pretty printing.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',3,2],1,1)
sage: t=K(rows=[[1]])
sage: t.pp()
1
```

5.1.34 Affine factorization crystal of type A

class sage.combinat.crystals.affine_factorization.**AffineFactorizationCrystal**(*w*, *n*, *x=None*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The crystal on affine factorizations with a cut-point, as introduced by [MS14].

INPUT:

- w – an element in an (affine) Weyl group or a skew shape of k-bounded partitions (if k was specified)
- n – the number of factors in the factorization
- x – (default: None) the cut point; if not specified it is determined as the minimal missing residue in w
- k – (default: None) positive integer, specifies that w is k-bounded or a $k+1$-core when specified

EXAMPLES:

```
sage: W = WeylGroup(['A',3,1], prefix='s')
sage: w = W.from_reduced_word([2,3,2,1])
sage: B = crystals.AffineFactorization(w,3); B
Crystal on affine factorizations of type A2 associated to s2*s3*s2*s1
sage: B.list()
[(1, s2, s3*s2*s1),
 (1, s3*s2, s3*s1),
 (1, s3*s2*s1, s3),
 (s3, s2, s3*s1),
 (s3, s2*s1, s3),
 (s3*s2, s1, s3),
 (s3*s2*s1, 1, s3),
 (s3*s2*s1, s3, 1),
 (s3*s2, 1, s3*s1),
 (s3*s2, s3, s1),
 (s3*s2, s3*s1, 1),
 (s2, 1, s3*s2*s1),
 (s2, s3, s2*s1),
```

```
 (s2, s3*s2, s1),
 (s2, s3*s2*s1, 1)]
```

We can also access the crystal by specifying a skew shape in terms of k-bounded partitions:

```
sage: crystals.AffineFactorization([[3,1,1],[1]], 3, k=3)
Crystal on affine factorizations of type A2 associated to s2*s3*s2*s1
```

We can compute the highest weight elements:

```
sage: hw = [w for w in B if w.is_highest_weight()]
sage: hw
[(1, s2, s3*s2*s1)]
sage: hw[0].weight()
(3, 1, 0)
```

And show that this crystal is isomorphic to the tableau model of the same weight:

```
sage: C = crystals.Tableaux(['A',2],shape=[3,1])
sage: GC = C.digraph()
sage: GB = B.digraph()
sage: GC.is_isomorphic(GB, edge_labels=True)
True
```

The crystal operators themselves move elements between adjacent factors:

```
sage: b = hw[0];b
(1, s2, s3*s2*s1)
sage: b.f(1)
(1, s3*s2, s3*s1)
```

The cut point x is not supposed to occur in the reduced words for w:

```
sage: B = crystals.AffineFactorization([[3,2],[2]],4,x=0,k=3)
Traceback (most recent call last):
...
ValueError: x cannot be in reduced word of s0*s3*s2
```

REFERENCES:

class **Element**

Bases: `sage.structure.element_wrapper.ElementWrapper`

bracketing(i)

Removes all bracketed letters between i-th and $i+1$-th entry.

EXAMPLES:

```
sage: B = crystals.AffineFactorization([[3,1],[1]], 3, k=3, x=4)
sage: W = B.w.parent()
sage: t = B((W.one(),W.from_reduced_word([3]),W.from_reduced_word([2,1]))); t
(1, s3, s2*s1)
sage: t.bracketing(1)
[[3], [2, 1]]
```

e(i)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: B = crystals.AffineFactorization([[3,1],[1]], 4, k=3)
sage: W = B.w.parent()
```

```
sage: t = B((W.one(),W.one(),W.from_reduced_word([3]),W.from_reduced_word([2,1]))); t
(1, 1, s3, s2*s1)
sage: t.e(1)
(1, 1, 1, s3*s2*s1)
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: B = crystals.AffineFactorization([[3,1],[1]], 4, k=3)
sage: W = B.w.parent()
sage: t = B((W.one(),W.one(),W.from_reduced_word([3]),W.from_reduced_word([2,1]))); t
(1, 1, s3, s2*s1)
sage: t.f(2)
(1, s3, 1, s2*s1)
sage: t.f(1)
(1, 1, s3*s2, s1)
```

to_tableau()

Return the tableau representation of `self`.

Uses the recording tableau of a minor variation of Edelman-Greene insertion. See Theorem 4.11 in [MS14].

EXAMPLES:

```
sage: W = WeylGroup(['A',3,1], prefix='s')
sage: w = W.from_reduced_word([2,1,3,2])
sage: B = crystals.AffineFactorization(w,3)
sage: for x in B:
....:     x
....:     x.to_tableau().pp()
(1, s2*s1, s3*s2)
  1  1
  2  2
(s2, s1, s3*s2)
  1  1
  2  3
(s2, s3*s1, s2)
  1  2
  2  3
(s2*s1, 1, s3*s2)
  1  1
  3  3
(s2*s1, s3, s2)
  1  2
  3  3
(s2*s1, s3*s2, 1)
  2  2
  3  3
```

class sage.combinat.crystals.affine_factorization.**FactorizationToTableaux**(*parent*, *cartan_type=None*, *virtualization=None*, *scaling_factors=None*)

Bases: sage.categories.crystals.CrystalMorphism

Initialize self.

TESTS:

```
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: H = Hom(B, B)
sage: psi = H.an_element()
```

is_embedding()

Return True as this is an isomorphism.

EXAMPLES:

```
sage: W = WeylGroup(['A',3,1], prefix='s')
sage: w = W.from_reduced_word([2,1,3,2])
sage: B = crystals.AffineFactorization(w,3)
sage: phi = B._tableaux_isomorphism
sage: phi.is_isomorphism()
True
```

TESTS:

```
sage: W = WeylGroup(['A',4,1], prefix='s')
sage: w = W.from_reduced_word([2,1,3,2,4,3,2,1])
sage: B = crystals.AffineFactorization(w, 4)
sage: phi = B._tableaux_isomorphism
sage: all(phi(b).e(i) == phi(b.e(i)) and phi(b).f(i) == phi(b.f(i))
....:     for b in B for i in B.index_set())
True
sage: set(phi(b) for b in B) == set(phi.codomain())
True
```

is_isomorphism()

Return True as this is an isomorphism.

EXAMPLES:

```
sage: W = WeylGroup(['A',3,1], prefix='s')
sage: w = W.from_reduced_word([2,1,3,2])
sage: B = crystals.AffineFactorization(w,3)
sage: phi = B._tableaux_isomorphism
sage: phi.is_isomorphism()
True
```

TESTS:

```
sage: W = WeylGroup(['A',4,1], prefix='s')
sage: w = W.from_reduced_word([2,1,3,2,4,3,2,1])
sage: B = crystals.AffineFactorization(w, 4)
sage: phi = B._tableaux_isomorphism
```

```
sage: all(phi(b).e(i) == phi(b.e(i)) and phi(b).f(i) == phi(b.f(i))
....:     for b in B for i in B.index_set())
True
sage: set(phi(b) for b in B) == set(phi.codomain())
True
```

is_surjective()

Return `True` as this is an isomorphism.

EXAMPLES:

```
sage: W = WeylGroup(['A',3,1], prefix='s')
sage: w = W.from_reduced_word([2,1,3,2])
sage: B = crystals.AffineFactorization(w,3)
sage: phi = B._tableaux_isomorphism
sage: phi.is_isomorphism()
True
```

TESTS:

```
sage: W = WeylGroup(['A',4,1], prefix='s')
sage: w = W.from_reduced_word([2,1,3,2,4,3,2,1])
sage: B = crystals.AffineFactorization(w, 4)
sage: phi = B._tableaux_isomorphism
sage: all(phi(b).e(i) == phi(b.e(i)) and phi(b).f(i) == phi(b.f(i))
....:     for b in B for i in B.index_set())
True
sage: set(phi(b) for b in B) == set(phi.codomain())
True
```

sage.combinat.crystals.affine_factorization.**affine_factorizations**(*w*, *l*, *weight=None*)

Return all factorizations of `w` into `l` factors or of weight `weight`.

INPUT:

- `w` – an (affine) permutation or element of the (affine) Weyl group
- `l` – nonegative integer
- `weight` – (default: None) tuple of nonnegative integers specifying the length of the factors

EXAMPLES:

```
sage: W = WeylGroup(['A',3,1], prefix='s')
sage: w = W.from_reduced_word([3,2,3,1,0,1])
sage: from sage.combinat.crystals.affine_factorization import affine_factorizations
sage: affine_factorizations(w,4)
[[s2, s3, s0, s2*s1*s0],
[s2, s3, s2*s0, s1*s0],
[s2, s3, s2*s1*s0, s1],
[s2, s3*s2, s0, s1*s0],
[s2, s3*s2, s1*s0, s1],
[s2, s3*s2*s1, s0, s1],
[s3*s2, s3, s0, s1*s0],
[s3*s2, s3, s1*s0, s1],
[s3*s2, s3*s1, s0, s1],
[s3*s2*s1, s3, s0, s1]]

sage: W = WeylGroup(['A',2], prefix='s')
sage: w0 = W.long_element()
sage: affine_factorizations(w0,3)
```

```
[[1, s1, s2*s1],
[1, s2*s1, s2],
[s1, 1, s2*s1],
[s1, s2, s1],
[s1, s2*s1, 1],
[s2, s1, s2],
[s2*s1, 1, s2],
[s2*s1, s2, 1]]
sage: affine_factorizations(w0,3,(0,1,2))
[[1, s1, s2*s1]]
sage: affine_factorizations(w0,3,(1,1,1))
[[s1, s2, s1], [s2, s1, s2]]
sage: W = WeylGroup(['A',3], prefix='s')
sage: w0 = W.long_element()
sage: affine_factorizations(w0,6,(1,1,1,1,1,1))
[[s1, s2, s1, s3, s2, s1],
[s1, s2, s3, s1, s2, s1],
[s1, s2, s3, s2, s1, s2],
[s1, s3, s2, s1, s3, s2],
[s1, s3, s2, s3, s1, s2],
[s2, s1, s2, s3, s2, s1],
[s2, s1, s3, s2, s1, s3],
[s2, s1, s3, s2, s3, s1],
[s2, s3, s1, s2, s1, s3],
[s2, s3, s1, s2, s3, s1],
[s2, s3, s2, s1, s2, s3],
[s3, s1, s2, s1, s3, s2],
[s3, s1, s2, s3, s1, s2],
[s3, s2, s1, s2, s3, s2],
[s3, s2, s1, s3, s2, s3],
[s3, s2, s3, s1, s2, s3]]
sage: affine_factorizations(w0,6,(0,0,0,1,2,3))
[[1, 1, 1, s1, s2*s1, s3*s2*s1]]
```

5.1.35 Affinization Crystals

class sage.combinat.crystals.affinization.**AffinizationOfCrystal**(*B*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

An affiniziation of a crystal.

Let $\mathfrak{g}$ be a Kac-Moody algebra of affine type. The affinization of a finite $U'_q(\mathfrak{g})$-crystal B is the (infinite) $U_q(\mathfrak{g})$-crystal with underlying set:

$$B^{\mathrm{aff}} = \{b(m) \mid b \in B, m \in \mathbf{Z}\}$$

and crystal structure determined by:

$$e_i(b(m)) = \begin{cases} (e_0 b)(m+1) & i = 0, \\ (e_i b)(m) & i \neq 0, \end{cases}$$

$$f_i(b(m)) = \begin{cases} (f_0 b)(m-1) & i = 0, \\ (f_i b)(m) & i \neq 0, \end{cases}$$

$$\mathrm{wt}(b(m)) = \mathrm{wt}(b) + m\delta.$$

EXAMPLES:

We first construct a Kirillov-Reshetikhin crystal and then take it's corresponding affinization:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1], 2, 2)
sage: A = K.affinization()
```

Next we construct an affinization crystal from a tensor product of KR crystals:

```
sage: KT = crystals.TensorProductOfKirillovReshetikhinTableaux(['C',2,1], [[1,2],[2,1]])
sage: A = crystals.AffinizationOf(KT)
```

REFERENCES:

- [HK02] Chapter 10

class **Element** (*parent*, *b*, *m*)

Bases: `sage.structure.element.Element`

An element in an affinization crystal.

e (*i*)

Return the action of e_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: A = crystals.KirillovReshetikhin(['A',2,1], 2,2).affinization()
sage: mg = A.module_generators[0]
sage: mg.e(0)
[[1, 2], [2, 3]](1)
sage: mg.e(1)
sage: mg.e(0).e(1)
[[1, 1], [2, 3]](1)
```

epsilon (*i*)

Return ε_i of `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: A = crystals.KirillovReshetikhin(['A',2,1], 2,2).affinization()
sage: mg = A.module_generators[0]
sage: mg.epsilon(0)
2
sage: mg.epsilon(1)
0
```

f (*i*)

Return the action of f_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: A = crystals.KirillovReshetikhin(['A',2,1], 2,2).affinization()
sage: mg = A.module_generators[0]
sage: mg.f(2)
[[1, 1], [2, 3]](0)
sage: mg.f(2).f(2).f(0)
sage: mg.f_string([2,1,1])
sage: mg.f_string([2,1])
```

```
[[1, 2], [2, 3]](0)
sage: mg.f_string([2,1,0])
[[1, 1], [2, 2]](-1)
```

phi(*i*)

Return φ_i of `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: A = crystals.KirillovReshetikhin(['A',2,1], 2,2).affinization()
sage: mg = A.module_generators[0]
sage: mg.phi(0)
0
sage: mg.phi(2)
2
```

weight()

Return the weight of `self`.

The weight wt of an element is:

$$\mathrm{wt}\big(b(m)\big) = \mathrm{wt}(b) + m\delta,$$

where δ is the null root.

EXAMPLES:

```
sage: A = crystals.KirillovReshetikhin(['A',2,1], 2,2).affinization()
sage: mg = A.module_generators[0]
sage: mg.weight()
-2*Lambda[0] + 2*Lambda[2]
sage: mg.e(0).weight()
-Lambda[1] + Lambda[2] + delta
sage: mg.e(0).e(0).weight()
2*Lambda[0] - 2*Lambda[1] + 2*delta
```

AffinizationOfCrystal.**digraph**(*subset=None*, *index_set=None*)

Return the DiGraph associated with `self`. See `digraph()` for more information.

EXAMPLES:

```
sage: A = crystals.KirillovReshetikhin(['A',2,1], 2, 2).affinization()
sage: S = A.subcrystal(max_depth=3)
sage: G = A.digraph(subset=S)
```

AffinizationOfCrystal.**weight_lattice_realization**()

Return the weight lattice realization of `self`.

EXAMPLES:

```
sage: A = crystals.KirillovReshetikhin(['A',2,1], 1, 1).affinization()
sage: A.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]
```

5.1.36 Alcove paths

AUTHORS:

- Brant Jones (2008): initial version

- Arthur Lubovsky (2013-03-07): rewritten to implement affine type

Special thanks to: Nicolas Borie, Anne Schilling, Travis Scrimshaw, and Nicolas Thiery.

class sage.combinat.crystals.alcove_path.**CrystalOfAlcovePaths**(*starting_weight*, *highest_weight_crystal*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Crystal of alcove paths generated from a "straight-line" path to the negative of a given dominant weight.

INPUT:

- `cartan_type` – Cartan type of a finite or affine untwisted root system.
- `weight` – Dominant weight as a list of (integral) coefficients of the fundamental weights.
- `highest_weight_crystal` – (Default: `True`) If `True` returns the highest weight crystal. If `False` returns an object which is close to being isomorphic to the tensor product of Kirillov-Reshetikhin crystals of column shape in the following sense: We get all the vertices, but only some of the edges. We'll call the included edges pseudo-Demazure. They are all non-zero edges and the 0-edges not at the end of a 0-string of edges, i.e. not those with $f_0(b) = b'$ with $\varphi_0(b) = 1$. (Whereas Demazure 0-edges are those that are not at the beginning of a zero string.) In this case the weight $[c_1, c_2, \ldots, c_k]$ represents $\sum_{i=1}^{k} c_i \omega_i$.

Note: If `highest_weight_crystal` = `False`, since we do not get the full crystal, `TestSuite` will fail on the Stembridge axioms.

See also:

- `Crystals`

EXAMPLES:

The following example appears in Figure 2 of [LP2008]:

```
sage: C = crystals.AlcovePaths(['G',2],[0,1])
sage: G = C.digraph()
sage: GG = DiGraph({
....:     ()         : {(0)          : 2 },
....:     (0)        : {(0,8)        : 1 },
....:     (0,1)      : {(0,1,7)      : 2 },
....:     (0,1,2)    : {(0,1,2,9)    : 1 },
....:     (0,1,2,3)  : {(0,1,2,3,4)  : 2 },
....:     (0,1,2,6)  : {(0,1,2,3)    : 1 },
....:     (0,1,2,9)  : {(0,1,2,6)    : 1 },
....:     (0,1,7)    : {(0,1,2)      : 2 },
....:     (0,1,7,9)  : {(0,1,2,9)    : 2 },
....:     (0,5)      : {(0,1)        : 1, (0,5,7) : 2 },
....:     (0,5,7)    : {(0,5,7,9)    : 1 },
....:     (0,5,7,9)  : {(0,1,7,9)    : 1 },
....:     (0,8)      : {(0,5)        : 1 },
....:     })
sage: G.is_isomorphic(GG)
True
sage: for (u,v,i) in G.edges(): print (u.integer_sequence() , v.integer_sequence(), i)
([], [0], 2)
([0], [0, 8], 1)
([0, 1], [0, 1, 7], 2)
([0, 1, 2], [0, 1, 2, 9], 1)
([0, 1, 2, 3], [0, 1, 2, 3, 4], 2)
```

```
([0, 1, 2, 6], [0, 1, 2, 3], 1)
([0, 1, 2, 9], [0, 1, 2, 6], 1)
([0, 1, 7], [0, 1, 2], 2)
([0, 1, 7, 9], [0, 1, 2, 9], 2)
([0, 5], [0, 1], 1)
([0, 5], [0, 5, 7], 2)
([0, 5, 7], [0, 5, 7, 9], 1)
([0, 5, 7, 9], [0, 1, 7, 9], 1)
([0, 8], [0, 5], 1)
```

Alcove path crystals are a discrete version of Littelmann paths. We verify that the alcove path crystal is isomorphic to the LS path crystal:

```
sage: C1 = crystals.AlcovePaths(['C',3],[2,1,0])
sage: g1 = C1.digraph() #long time
sage: C2 = crystals.LSPaths(['C',3],[2,1,0])
sage: g2 = C2.digraph() #long time
sage: g1.is_isomorphic(g2, edge_labels=True) #long time
True
```

The preferred initialization method is via explicit weights rather than a Cartan type and the coefficients of the fundamental weights:

```
sage: R = RootSystem(['C',3])
sage: P = R.weight_lattice()
sage: La = P.fundamental_weights()
sage: C = crystals.AlcovePaths(2*La[1]+La[2]); C
Highest weight crystal of alcove paths of type ['C', 3] and weight 2*Lambda[1] + Lambda[2]
sage: C1==C
True
```

We now explain the data structure:

```
sage: C = crystals.AlcovePaths(['A',2],[2,0]) ; C
Highest weight crystal of alcove paths of type ['A', 2] and weight 2*Lambda[1]
sage: C._R.lambda_chain()
[(alpha[1], 0), (alpha[1] + alpha[2], 0), (alpha[1], 1), (alpha[1] + alpha[2], 1)]
```

The previous list gives the initial "straight line" path from the fundamental alcove A_o to its translation $A_o - \lambda$ where $\lambda = 2\omega_1$ in this example. The initial path for weight λ is called the λ-chain. This path is constructed from the ordered pairs (β, k), by crossing the hyperplane orthogonal to β at height $-k$. We can view a plot of this path as follows:

```
sage: x=C( () )
sage: x.plot() # not tested - outputs a pdf
```

An element of the crystal is given by a subset of the λ-chain. This subset indicates the hyperplanes where the initial path should be folded. The highest weight element is given by the empty subset.

```
sage: x
()
sage: x.f(1).f(2)
((alpha[1], 1), (alpha[1] + alpha[2], 1))
sage: x.f(1).f(2).integer_sequence()
[2, 3]
sage: C([2,3])
((alpha[1], 1), (alpha[1] + alpha[2], 1))
sage: C([2,3]).is_admissible() #check if a valid vertex
True
sage: C([1,3]).is_admissible() #check if a valid vertex
```

```
False
```

Alcove path crystals now works in affine type (trac ticket #14143):

```
sage: C = crystals.AlcovePaths(['A',2,1],[1,0,0]) ; C
Highest weight crystal of alcove paths of type ['A', 2, 1] and weight Lambda[0]
sage: x=C(  () )
sage: x.f(0)
((alpha[0], 0),)
sage: C.R
Root system of type ['A', 2, 1]
sage: C.weight
Lambda[0]
```

Test that the tensor products of Kirillov-Reshetikhin crystals minus non-pseudo-Demazure arrows is in bijection with alcove path construction:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],2,1)
sage: T = crystals.TensorProduct(K,K)
sage: g = T.digraph() #long time
sage: for e in g.edges(): #long time
....:     if e[0].phi(0) == 1 and e[2] == 0: #long time
....:         g.delete_edge(e)  #long time

sage: C = crystals.AlcovePaths(['B',3,1],[0,2,0], highest_weight_crystal=False)
sage: g2 = C.digraph_fast() #long time
sage: g.is_isomorphic(g2, edge_labels = True) #long time
True
```

Note: In type $C_n^{(1)}$, the Kirillov-Reshetikhin crystal is not connected when restricted to pseudo-Demazure arrows, hence the previous example will fail for type $C_n^{(1)}$ crystals.

```
sage: R = RootSystem(['B',3])
sage: P = R.weight_lattice()
sage: La = P.fundamental_weights()
sage: D = crystals.AlcovePaths(2*La[2], highest_weight_crystal=False)
sage: C == D
True
```

Warning: Weights from finite root systems index non-highest weight crystals.

REFERENCES:

Element
alias of CrystalOfAlcovePathsElement

digraph_fast (*depth=None*)
Return the crystal graph with maximum depth depth deep starting at the module generator. Significant speed up for highest_weight_crystals of affine type.

EXAMPLES:

```
sage: crystals.AlcovePaths(['A',2], [1,1]).digraph_fast(depth=3)
Digraph on 7 vertices
```

TESTS:

The following example demonstrates the speed improvement. The speedup in non-affine types is small

however:

```
sage: cartan_type = ['A',2,1] #long time
sage: weight = [1,1,0] #long time
sage: depth = 5 #long time
sage: C = crystals.AlcovePaths(cartan_type, weight) #long time
sage: %timeit C.digraph_fast(depth) # not tested
10 loops, best of 3: 171 ms per loop
sage: %timeit C.digraph(subset=C.subcrystal(max_depth=depth, direction='lower')) #not tested
1 loops, best of 3: 19.7 s per loop
sage: G1 = C.digraph_fast(depth) #long time
sage: G2 = C.digraph(subset=C.subcrystal(max_depth=depth, direction='lower')) #long time
sage: G1.is_isomorphic(G2, edge_labels=True) #long time
True
```

vertices()

Return a list of all the vertices of the crystal.

The vertices are represented as lists of integers recording the folding positions.

One can compute all vertices of the crystal by finding all the admissible subsets of the λ-chain (see method is_admissible, for definition). We use the breath first search algorithm.

Warning: This method is (currently) only useful for the case when `highest_weight_crystal = False`, where you cannot always reach all vertices of the crystal using crystal operators, starting from the highest weight vertex. This method is typically slower than generating the crystal graph using crystal operators.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['C',2],[1,0])
sage: C.vertices()
[[], [0], [0, 1], [0, 1, 2]]
sage: C = crystals.AlcovePaths(['C',2,1],[2,1],False)
sage: len(C.vertices())
80
```

The number of elements reachable using the crystal operators from the module generator:

```
sage: len(list(C))
55
```

weight_lattice_realization()

Return the weight lattice realization of `self`.

EXAMPLES:

```
sage: B = crystals.AlcovePaths(['A',2,1],[1,0,0])
sage: B.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]

sage: C = crystals.AlcovePaths("B3",[1,0,0])
sage: C.weight_lattice_realization()
Ambient space of the Root system of type ['B', 3]
```

class sage.combinat.crystals.alcove_path.**CrystalOfAlcovePathsElement**

Bases: sage.structure.element_wrapper.ElementWrapper

Crystal of alcove paths element.

INPUT:

- data – a list of folding positions in the lambda chain (indexing starts at 0) or a tuple of RootsWithHeight giving folding positions in the lambda chain.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[3,2])
sage: x = C ( () )
sage: x.f(1).f(2)
((alpha[1], 2), (alpha[1] + alpha[2], 4))
sage: x.f(1).f(2).integer_sequence()
[8, 9]
sage: C([8,9])
((alpha[1], 2), (alpha[1] + alpha[2], 4))
```

e(*i*)

Return the i-th crystal raising operator on self.

INPUT:

- i – element of the index set of the underlying root system.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[2,0]); C
Highest weight crystal of alcove paths of type ['A', 2] and weight 2*Lambda[1]
sage: x = C( () )
sage: x.e(1)
sage: x.f(1) == x.f(1).f(2).e(2)
True
```

epsilon(*i*)

Return the distance to the start of the i-string.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[1,1])
sage: [c.epsilon(1) for c in C]
[0, 1, 0, 1, 0, 0, 2, 1]
sage: [c.epsilon(2) for c in C]
[0, 0, 1, 0, 1, 2, 0, 1]
```

f(*i*)

Returns the i-th crystal lowering operator on self.

INPUT:

- i – element of the index_set of the underlying root_system.

EXAMPLES:

```
sage: C=crystals.AlcovePaths(['B',2],[1,1])
sage: x=C(  () )
sage: x.f(1)
((alpha[1], 0),)
sage: x.f(1).f(2)
((alpha[1], 0), (alpha[1] + alpha[2], 2))
```

integer_sequence()

Return a list of integers corresponding to positions in the λ-chain where it is folded.

Todo

Incorporate this method into the _repr_ for finite Cartan type.

Note: Only works for finite Cartan types and indexing starts at 0.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[3,2])
sage: x = C( () )
sage: x.f(1).f(2).integer_sequence()
[8, 9]
```

is_admissible()

Diagnostic test to check if `self` is a valid element of the crystal.

If `self.value` is given by

$$(\beta_1, i_1), (\beta_2, i_2), \ldots, (\beta_k, i_k),$$

for highest weight crystals this checks if the sequence

$$1 \to s_{\beta_1} \to s_{\beta_1} s_{\beta_2} \to \cdots \to s_{\beta_1} s_{\beta_2} \ldots s_{\beta_k}$$

is a path in the Bruhat graph. If `highest_weight_crystal=False`, then the method checks if the above sequence is a path in the quantum Bruhat graph.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[1,1]); C
Highest weight crystal of alcove paths of type ['A', 2] and weight Lambda[1] + Lambda[2]
sage: roots = sorted(list(C._R._root_lattice.positive_roots())); roots
[alpha[1], alpha[1] + alpha[2], alpha[2]]
sage: r1 = C._R(roots[0],0); r1
(alpha[1], 0)
sage: r2 = C._R(roots[2],0); r2
(alpha[2], 0)
sage: r3 = C._R(roots[1],1); r3
(alpha[1] + alpha[2], 1)
sage: x = C( ( r1,r2) )
sage: x.is_admissible()
True
sage: x = C( (r3,) ); x
((alpha[1] + alpha[2], 1),)
sage: x.is_admissible()
False
sage: C = crystals.AlcovePaths(['C',2,1],[2,1],False)
sage: C([7,8]).is_admissible()
True
sage: C = crystals.AlcovePaths(['A',2],[3,2])
sage: C([2,3]).is_admissible()
True
```

Todo

Better doctest

phi(i)

Return the distance to the end of the i-string.

This method overrides the generic implementation in the category of crystals since this computation is more efficient.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[1,1])
sage: [c.phi(1) for c in C]
[1, 0, 2, 1, 0, 1, 0, 0]
sage: [c.phi(2) for c in C]
[1, 2, 0, 0, 1, 0, 1, 0]
```

plot()

Return a plot `self`.

Note: Currently only implemented for types A_2, B_2, and C_2.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[2,0])
sage: x = C( () ).f(1).f(2)
sage: x.plot() # Not tested - creates a pdf
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: C = crystals.AlcovePaths(['A',2],[2,0])
sage: for i in C: i.weight()
(0, 2, 0)
(0, 0, 1)
(0, -2, 2)
(0, 1, -1)
(0, -1, 0)
(0, 0, -2)
sage: B = crystals.AlcovePaths(['A',2,1],[1,0,0])
sage: p = B.module_generators[0].f_string([0,1,2])
sage: p.weight()
Lambda[0] - delta
```

TESTS:

Check that crystal morphisms work (trac ticket #19481):

```
sage: C1 = crystals.AlcovePaths(['A',2],[1,0])
sage: C2 = crystals.AlcovePaths(['A',2],[2,0])
sage: phi = C1.crystal_morphism(C2.module_generators, scaling_factors={1:2, 2:2})
sage: [phi(x) for x in C1]
[(), ((alpha[1], 0),), ((alpha[1], 0), (alpha[1] + alpha[2], 0))]
```

class sage.combinat.crystals.alcove_path.**RootsWithHeight**(*weight*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Data structure of the ordered pairs (β, k), where β is a positive root and k is a non-negative integer. A total order is implemented on this set, and depends on the weight.

INPUT:

- `cartan_type` – Cartan type of a finite or affine untwisted root system
- `weight` – dominant weight as a list of (integral) coefficients of the fundamental weights

EXAMPLES:

```
sage: from sage.combinat.crystals.alcove_path import RootsWithHeight
sage: R = RootsWithHeight(['A',2],[1,1]); R
Roots with height of Cartan type ['A', 2] and dominant weight Lambda[1] + Lambda[2]

sage: r1 = R._root_lattice.from_vector(vector([1,0])); r1
alpha[1]
sage: r2 = R._root_lattice.from_vector(vector([1,1])); r2
alpha[1] + alpha[2]

sage: x = R(r1,0); x
(alpha[1], 0)
sage: y = R(r2,1); y
(alpha[1] + alpha[2], 1)
sage: x < y
True
```

Element
alias of RootsWithHeightElement

lambda_chain()
Return the unfolded λ-chain.

Note: Only works in root systems of finite type.

EXAMPLES:

```
sage: from sage.combinat.crystals.alcove_path import RootsWithHeight
sage: R = RootsWithHeight(['A',2],[1,1]); R
Roots with height of Cartan type ['A', 2] and dominant weight Lambda[1] + Lambda[2]
sage: R.lambda_chain()
[(alpha[2], 0), (alpha[1] + alpha[2], 0), (alpha[1], 0), (alpha[1] + alpha[2], 1)]
```

word()
Gives the initial alcove path (λ-chain) in terms of simple roots. Used for plotting the path.

Note: Currently only implemented for finite Cartan types.

EXAMPLES:

```
sage: from sage.combinat.crystals.alcove_path import RootsWithHeight
sage: R = RootsWithHeight(['A',2],[3,2])
sage: R.word()
[2, 1, 2, 0, 1, 2, 1, 0, 1, 2]
```

class sage.combinat.crystals.alcove_path.**RootsWithHeightElement**(*parent*, *root*, *height*)

Bases: sage.structure.element.Element

Element of RootsWithHeight.

INPUT:

- root – A positive root β in our root system
- height – Is an integer, such that $0 \leq l \leq \langle \lambda, \beta^\vee \rangle$

EXAMPLES:

```
sage: from sage.combinat.crystals.alcove_path import RootsWithHeight
sage: rl = RootSystem(['A',2]).root_lattice()
```

```
sage: x = rl.from_vector(vector([1,1])); x
alpha[1] + alpha[2]
sage: R = RootsWithHeight(['A',2],[1,1]); R
Roots with height of Cartan type ['A', 2] and dominant weight Lambda[1] + Lambda[2]
sage: y = R(x, 1); y
(alpha[1] + alpha[2], 1)
```

sage.combinat.crystals.alcove_path.**compare_graphs**(*g1*, *g2*, *node1*, *node2*)

Compare two edge-labeled `graphs` obtained from `Crystal.digraph()`, starting from the root nodes of each graph.

- `g1` – `graphs`, first digraph
- `g2` – `graphs`, second digraph
- `node1` – element of `g1`
- `node2` – element of `g2`

Traverse `g1` starting at `node1` and compare this graph with the one obtained by traversing `g2` starting with `node2`. If the graphs match (including labels) then return `True`. Return `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.crystals.alcove_path import compare_graphs
sage: G1 = crystals.Tableaux(['A',3], shape=[1,1]).digraph()
sage: C = crystals.AlcovePaths(['A',3],[0,1,0])
sage: G2 = C.digraph()
sage: compare_graphs(G1, G2, C( () ), G2.vertices()[0])
True
```

5.1.37 Crystal features that are imported by default in the interpreter namespace

5.1.38 Catalog Of Crystals

See also:

- `sage.categories.crystals`
- `sage.combinat.crystals.crystals`

Catalog

This is a catalog of crystals that are currently implemented in Sage:

- `AffineCrystalFromClassical`
- `AffineCrystalFromClassicalAndPromotion`
- `AffineFactorization`
- `AffinizationOf`
- `AlcovePaths`
- `FastRankTwo`
- `GeneralizedYoungWalls`
- `HighestWeight`

- Induced
- KirillovReshetikhin
- KyotoPathModel
- Letters
- LSPaths
- NakajimaMonomials
- ProjectedLevelZeroLSPaths
- RiggedConfigurations
- Spins
- SpinsPlus
- SpinsMinus
- Tableaux

Subcatalogs:

- *Catalog Of Crystal Models For*
- *Catalog Of Elementary Crystals*
- *Catalog Of Crystal Models For Kirillov-Reshetikhin Crystals*

Functorial constructions:

- DirectSum
- TensorProduct

5.1.39 Catalog Of Elementary Crystals

See elementary_crystals.

- Component
- Elementary or B
- R
- T

5.1.40 Catalog Of Crystal Models For $B(\infty)$

We currently have the following models:

- GeneralizedYoungWalls
- LSPaths
- NakajimaMonomials
- PolyhedralRealization
- RiggedConfigurations
- Star
- Tableaux

5.1.41 Catalog Of Crystal Models For Kirillov-Reshetikhin Crystals

We currently have the following models:

- KashiwaraNakashimaTableaux
- KirillovReshetikhinTableaux
- LSPaths
- RiggedConfigurations

5.1.42 An introduction to crystals

Informally, a crystal $\mathcal{B}$ is an oriented graph with edges colored in some set I such that, for each $i \in I$, each node x has:

- at most one i-successor, denoted $f_i x$;
- at most one i-predecessor, denoted $e_i x$.

By convention, one writes $f_i x = \emptyset$ and $e_i x = \emptyset$ when x has no successor resp. predecessor.

One may think of $\mathcal{B}$ as essentially a deterministic automaton whose dual is also deterministic; in this context, the f_i's and e_i's are respectively the transition functions of the automaton and of its dual, and $\emptyset$ is the sink.

A crystal comes further endowed with a weight function $\mathrm{wt} : \mathcal{B} \to L$ which satisfies appropriate conditions.

In combinatorial representation theory, crystals are used as combinatorial data to model representations of Lie algebra.

Axiomatic definition

Let C be a Cartan type (CartanType) with index set I, and L be a realization of the weight lattice of the type C. Let α_i and $\alpha_i^\vee$ denote the simple roots and coroots respectively.

A type C crystal is a non-empty set $\mathcal{B}$ endowed with maps $\mathrm{wt} : \mathcal{B} \to L$, $e_i, f_i : \mathcal{B} \to \mathcal{B} \cup \{\emptyset\}$, and $\varepsilon_i, \varphi_i : \mathcal{B} \to \mathbf{Z} \cup \{-\infty\}$ for $i \in I$ satisfying the following properties for all $i \in I$:

- for $b, b' \in \mathcal{B}$, we have $f_i b' = b$ if and only if $e_i b = b'$;
- if $e_i b \in \mathcal{B}$, then:
  - $\mathrm{wt}(e_i b) = \mathrm{wt}(b) + \alpha_i$,
  - $\varepsilon_i(e_i b) = \varepsilon_i(b) - 1$,
  - $\varphi_i(e_i b) = \varphi_i(b) + 1$;
- if $f_i b \in \mathcal{B}$, then:
  - $\mathrm{wt}(f_i b) = \mathrm{wt}(b) - \alpha_i$,
  - $\varepsilon_i(f_i b) = \varepsilon_i(b) + 1$,
  - $\varphi_i(f_i b) = \varphi_i(b) - 1$;
- $\varphi_i(b) = \varepsilon_i(b) + \langle \alpha_i^\vee, \mathrm{wt}(b) \rangle$,
- if $\varphi_i(b) = -\infty$ for $b \in \mathcal{B}$, then $e_i b = f_i b = \emptyset$.

Some further conditions are required to guarantee that this data indeed models a representation of a Lie algebra. For finite simply laced types a complete characterization is given by Stembridge's local axioms [St2003].

REFERENCES:

EXAMPLES:

We construct the type A_5 crystal on letters (or in representation theoretic terms, the highest weight crystal of type A_5 corresponding to the highest weight Λ_1):

```
sage: C = crystals.Letters(['A',5]); C
The crystal of letters for type ['A', 5]
```

It has a single highest weight element:

```
sage: C.highest_weight_vectors()
(1,)
```

A crystal is an enumerated set (see `EnumeratedSets`); and we can count and list its elements in the usual way:

```
sage: C.cardinality()
6
sage: C.list()
[1, 2, 3, 4, 5, 6]
```

as well as use it in for loops:

```
sage: [x for x in C]
[1, 2, 3, 4, 5, 6]
```

Here are some more elaborate crystals (see their respective documentations):

```
sage: Tens = crystals.TensorProduct(C, C)
sage: Spin = crystals.Spins(['B', 3])
sage: Tab  = crystals.Tableaux(['A', 3], shape = [2,1,1])
sage: Fast = crystals.FastRankTwo(['B', 2], shape = [3/2, 1/2])
sage: KR = crystals.KirillovReshetikhin(['A',2,1],1,1)
```

One can get (currently) crude plotting via:

```
sage: Tab.plot()
Graphics object consisting of 52 graphics primitives
```

If dot2tex is installed, one can obtain nice latex pictures via:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 1,1)
sage: view(K, pdflatex=True, tightpage=True) # optional - dot2tex graphviz, not tested (opens externa
```

or with colored edges:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 1,1)
sage: G = K.digraph()
sage: G.set_latex_options(color_by_label = {0:"black", 1:"red", 2:"blue", 3:"green"}) #optional - dot
sage: view(G, pdflatex=True, tightpage=True) # optional - dot2tex graphviz, not tested (opens externa
```

For rank two crystals, there is an alternative method of getting metapost pictures. For more information see `C.metapost?`.

See also:

The overview of crystal features in Sage

Todo

- Vocabulary and conventions:
  - For a classical crystal: connected / highest weight / irreducible
  - ...
- Layout instructions for plot() for rank 2 types
- RestrictionOfCrystal

The crystals library in Sage grew up from an initial implementation in MuPAD-Combinat (see <MuPAD-Combinat>/lib/COMBINAT/crystals.mu).

class sage.combinat.crystals.crystals.**CrystalBacktracker**(*crystal*, *index_set=None*)

Bases: sage.combinat.backtrack.GenericBacktracker

Time complexity: $O(nF)$ amortized for each produced element, where n is the size of the index set, and F is the cost of computing e and f operators.

Memory complexity: $O(D)$ where D is the depth of the crystal.

Principle of the algorithm:

Let C be a classical crystal. It's an acyclic graph where each connected component has a unique element without predecessors (the highest weight element for this component). Let's assume for simplicity that C is irreducible (i.e. connected) with highest weight element u.

One can define a natural spanning tree of C by taking u as the root of the tree, and for any other element y taking as ancestor the element x such that there is an i-arrow from x to y with i minimal. Then, a path from u to y describes the lexicographically smallest sequence $i_1, \ldots, i_k$ such that $(f_{i_k} \circ f_{i_1})(u) = y$.

Morally, the iterator implemented below just does a depth first search walk through this spanning tree. In practice, this can be achieved recursively as follows: take an element x, and consider in turn each successor $y = f_i(x)$, ignoring those such that $y = f_j(x')$ for some x' and $j < i$ (this can be tested by computing $e_j(y)$ for $j < i$).

EXAMPLES:

```
sage: from sage.combinat.crystals.crystals import CrystalBacktracker
sage: C = crystals.Tableaux(['B',3],shape=[3,2,1])
sage: CB = CrystalBacktracker(C)
sage: len(list(CB))
1617
sage: CB = CrystalBacktracker(C, [1,2])
sage: len(list(CB))
8
```

5.1.43 Direct Sum of Crystals

class sage.combinat.crystals.direct_sum.**DirectSumOfCrystals**(*crystals*, *facade*, *keep-key*, *category*, ***options*)

Bases: sage.sets.disjoint_union_enumerated_sets.DisjointUnionEnumeratedSets

Direct sum of crystals.

Given a list of crystals $B_0, \ldots, B_k$ of the same Cartan type, one can form the direct sum $B_0 \oplus \cdots \oplus B_k$.

INPUT:

- `crystals` – a list of crystals of the same Cartan type
- `keepkey` – a boolean

The option `keepkey` is by default set to `False`, assuming that the crystals are all distinct. In this case the elements of the direct sum are just represented by the elements in the crystals B_i. If the crystals are not all distinct, one should set the `keepkey` option to `True`. In this case, the elements of the direct sum are represented as tuples (i, b) where $b \in B_i$.

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: C1 = crystals.Tableaux(['A',2],shape=[1,1])
sage: B = crystals.DirectSum([C,C1])
sage: B.list()
[1, 2, 3, [[1], [2]], [[1], [3]], [[2], [3]]]
sage: [b.f(1) for b in B]
[2, None, None, None, [[2], [3]], None]
sage: B.module_generators
[1, [[1], [2]]]

sage: B = crystals.DirectSum([C,C], keepkey=True)
sage: B.list()
[(0, 1), (0, 2), (0, 3), (1, 1), (1, 2), (1, 3)]
sage: B.module_generators
[(0, 1), (1, 1)]
sage: b = B( tuple([0,C(1)]) )
sage: b
(0, 1)
sage: b.weight()
(1, 0, 0)
```

The following is required, because `DirectSumOfCrystals` takes the same arguments as `DisjointUnionEnumeratedSets` (which see for details).

TESTS:

```
sage: C = crystals.Letters(['A',2])
sage: B = crystals.DirectSum([C,C], keepkey=True)
sage: B
Direct sum of the crystals Family (The crystal of letters for type ['A', 2], The crystal of lett

sage: TestSuite(C).run()
```

class `Element`

Bases: `sage.structure.element_wrapper.ElementWrapper`

A class for elements of direct sums of crystals

`e`(i)

Returns the action of e_i on self.

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: B = crystals.DirectSum([C,C], keepkey=True)
sage: [[b, b.e(2)] for b in B]
[[(0, 1), None], [(0, 2), None], [(0, 3), (0, 2)], [(1, 1), None], [(1, 2), None], [(1,
```

`epsilon`(i)

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: B = crystals.DirectSum([C,C], keepkey=True)
sage: b = B( tuple([0,C(2)]) )
sage: b.epsilon(2)
0
```

f(*i*)

Returns the action of f_i on self.

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: B = crystals.DirectSum([C,C], keepkey=True)
sage: [[b,b.f(1)] for b in B]
[[(0, 1), (0, 2)], [(0, 2), None], [(0, 3), None], [(1, 1), (1, 2)], [(1, 2), None], [(1
```

phi(*i*)

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: B = crystals.DirectSum([C,C], keepkey=True)
sage: b = B( tuple([0,C(2)]) )
sage: b.phi(2)
1
```

weight()

Returns the weight of self.

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: B = crystals.DirectSum([C,C], keepkey=True)
sage: b = B( tuple([0,C(2)]) )
sage: b
(0, 2)
sage: b.weight()
(0, 1, 0)
```

DirectSumOfCrystals.**weight_lattice_realization**()

Return the weight lattice realization used to express weights.

The weight lattice realization is the common parent which all weight lattice realizations of the crystals of `self` coerce into.

EXAMPLES:

```
sage: LaZ = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_weights()
sage: LaQ = RootSystem(['A',2,1]).weight_space(extended=True).fundamental_weights()
sage: B = crystals.LSPaths(LaQ[1])
sage: B.weight_lattice_realization()
Extended weight space over the Rational Field of the Root system of type ['A', 2, 1]
sage: C = crystals.AlcovePaths(LaZ[1])
sage: C.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]
sage: D = crystals.DirectSum([B,C])
sage: D.weight_lattice_realization()
Extended weight space over the Rational Field of the Root system of type ['A', 2, 1]
```

5.1.44 Elementary Crystals

Let λ be a weight. The crystals T_λ, R_λ, B_i, and C are important objects in the tensor category of crystals. For example, the crystal T_0 is the neutral object in this category; i.e., $T_0 \otimes B \cong B \otimes T_0 \cong B$ for any crystal B. We list some other properties of these crystals:

- The crystal $T_\lambda \otimes B(\infty)$ is the crystal of the Verma module with highest weight λ, where λ is a dominant integral weight.
- Let u_∞ be the highest weight vector of $B(\infty)$ and λ be a dominant integral weight. There is an embedding of crystals $B(\lambda) \longrightarrow T_\lambda \otimes B(\infty)$ sending $u_\lambda \mapsto t_\lambda \otimes u_\infty$ which is not strict, but the embedding $B(\lambda) \longrightarrow C \otimes T_\lambda \otimes B(\infty)$ by $u_\lambda \mapsto c \otimes t_\lambda \otimes u_\infty$ is a strict embedding.
- For any dominant integral weight λ, there is a surjective crystal morphism $\Psi_\lambda \colon R_\lambda \otimes B(\infty) \longrightarrow B(\lambda)$. More precisely, if $B = \{r_\lambda \otimes b \in R_\lambda \otimes B(\infty) : \Psi_\lambda(r_\lambda \otimes b) \neq 0\}$, then $B \cong B(\lambda)$ as crystals.
- For all Cartan types and all weights λ, we have $R_\lambda \cong C \otimes T_\lambda$ as crystals.
- For each i, there is a strict crystal morphism $\Psi_i \colon B(\infty) \longrightarrow B_i \otimes B(\infty)$ defined by $u_\infty \mapsto b_i(0) \otimes u_\infty$, where u_∞ is the highest weight vector of $B(\infty)$.

For more information on $B(\infty)$, see `InfinityCrystalOfTableaux`.

Note: As with `TensorProductOfCrystals`, we are using the opposite of Kashiwara's convention.

AUTHORS:

- Ben Salisbury: Initial version

REFERENCES:

class `sage.combinat.crystals.elementary_crystals.`**`AbstractSingleCrystalElement`**
Bases: `sage.structure.element.Element`

Abstract base class for elements in crystals with a single element.

`e`(i)

Return e_i of `self`, which is `None` for all i.

INPUT:

- `i` – An element of the index set

EXAMPLES:

```
sage: ct = CartanType(['A',2])
sage: la = RootSystem(ct).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(ct,la[1])
sage: t = T.highest_weight_vector()
sage: t.e(1)
sage: t.e(2)
```

`f`(i)

Return f_i of `self`, which is `None` for all i.

INPUT:

- `i` – An element of the index set

EXAMPLES:

```
sage: ct = CartanType(['A',2])
sage: la = RootSystem(ct).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(ct,la[1])
```

```
sage: t = T.highest_weight_vector()
sage: t.f(1)
sage: t.f(2)
```

class sage.combinat.crystals.elementary_crystals.**ComponentCrystal**(*cartan_type*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The component crystal.

Defined in [Kashiwara93], the component crystal $C = \{c\}$ is the single element crystal whose crystal structure is defined by

$$\mathrm{wt}(c) = 0, \quad e_i c = f_i c = 0, \quad \varepsilon_i(c) = \varphi_i(c) = 0.$$

Note $C \cong B(0)$, where $B(0)$ is the highest weight crystal of highest weight 0.

INPUT:

- cartan_type – A Cartan type

class Element

Bases: sage.combinat.crystals.elementary_crystals.AbstractSingleCrystalElement

Element of a component crystal.

epsilon(*i*)

Return ε_i of self, which is 0 for all i.

INPUT:

- i – An element of the index set

EXAMPLES:

```
sage: C = crystals.elementary.Component("C5")
sage: c = C.highest_weight_vector()
sage: [c.epsilon(i) for i in C.index_set()]
[0, 0, 0, 0, 0]
```

phi(*i*)

Return φ_i of self, which is 0 for all i.

INPUT:

- i – An element of the index set

EXAMPLES:

```
sage: C = crystals.elementary.Component("C5")
sage: c = C.highest_weight_vector()
sage: [c.phi(i) for i in C.index_set()]
[0, 0, 0, 0, 0]
```

weight()

Return the weight of self, which is always 0.

EXAMPLES:

```
sage: C = crystals.elementary.Component("F4")
sage: c = C.highest_weight_vector()
sage: c.weight()
(0, 0, 0, 0)
```

ComponentCrystal.**cardinality**()

Return the cardinality of self, which is always 1.

EXAMPLES:

```
sage: C = crystals.elementary.Component("E6")
sage: c = C.highest_weight_vector()
sage: C.cardinality()
1
```

class sage.combinat.crystals.elementary_crystals.**ElementaryCrystal**(*cartan_type*, *i*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The elementary crystal B_i.

For i an element of the index set of type X, the crystal B_i of type X is the set

$$B_i = \{b_i(m) : m \in \mathbf{Z}\},$$

where the crystal stucture is given by

$$\begin{aligned}
\mathrm{wt}\big(b_i(m)\big) &= m\alpha_i \\
\varphi_j\big(b_i(m)\big) &= \begin{cases} m & \text{if } j = i, \\ -\infty & \text{if } j \neq i, \end{cases} \\
\varepsilon_j\big(b_i(m)\big) &= \begin{cases} -m & \text{if } j = i, \\ -\infty & \text{if } j \neq i, \end{cases} \\
e_j b_i(m) &= \begin{cases} b_i(m+1) & \text{if } j = i, \\ 0 & \text{if } j \neq i, \end{cases} \\
f_j b_i(m) &= \begin{cases} b_i(m-1) & \text{if } j = i, \\ 0 & \text{if } j \neq i. \end{cases}
\end{aligned}$$

The *Kashiwara embedding theorem* asserts there is a unique strict crystal embedding of crystals

$$B(\infty) \hookrightarrow B_i \otimes B(\infty),$$

satisfying certain properties (see [Kashiwara93]). The above embedding may be iterated to obtain a new embedding

$$B(\infty) \hookrightarrow B_{i_N} \otimes B_{i_{N-1}} \otimes \cdots \otimes B_{i_2} \otimes B_{i_1} \otimes B(\infty),$$

which is a foundational object in the study of *polyhedral realizations of crystals* (see, for example, [NZ97]).

class **Element**(*parent*, *m*)

Bases: sage.structure.element.Element

Element of a B_i crystal.

e(*i*)

Return the action of e_i on self.

INPUT:

- i – An element of the index set

EXAMPLES:

```
sage: B = crystals.elementary.Elementary(['E',7],1)
sage: B(3).e(1)
4
sage: B(172).e_string([1]*171)
343
sage: B(0).e(2)
```

epsilon(*i*)
Return ε_i of `self`.

INPUT:
- `i` – An element of the index set

EXAMPLES:
```
sage: B = crystals.elementary.Elementary(['F',4],3)
sage: [[B(j).epsilon(i) for i in B.index_set()] for j in range(5)]
[[-inf, -inf, 0, -inf],
 [-inf, -inf, -1, -inf],
 [-inf, -inf, -2, -inf],
 [-inf, -inf, -3, -inf],
 [-inf, -inf, -4, -inf]]
```

f(*i*)
Return the action of f_i on `self`.

INPUT:
- `i` – An element of the index set

EXAMPLES:
```
sage: B = crystals.elementary.Elementary(['E',7],1)
sage: B(3).f(1)
2
sage: B(172).f_string([1]*171)
1
sage: B(0).e(2)
```

phi(*i*)
Return φ_i of `self`.

INPUT:
- `i` – An element of the index set

EXAMPLES:
```
sage: B = crystals.elementary.Elementary(['E',8,1],4)
sage: [[B(m).phi(j) for j in B.index_set()] for m in range(44,49)]
[[-inf, -inf, -inf, -inf, 44, -inf, -inf, -inf, -inf],
 [-inf, -inf, -inf, -inf, 45, -inf, -inf, -inf, -inf],
 [-inf, -inf, -inf, -inf, 46, -inf, -inf, -inf, -inf],
 [-inf, -inf, -inf, -inf, 47, -inf, -inf, -inf, -inf],
 [-inf, -inf, -inf, -inf, 48, -inf, -inf, -inf, -inf]]
```

weight()
Return the weight of `self`.

EXAMPLES:
```
sage: B = crystals.elementary.Elementary(['C',14],12)
sage: B(-385).weight()
-385*alpha[12]
```

ElementaryCrystal.**weight_lattice_realization**()
Return a realization of the lattice containing the weights of `self`.

EXAMPLES:
```
sage: B = crystals.elementary.Elementary(['A',4, 1], 2)
sage: B.weight_lattice_realization()
Root lattice of the Root system of type ['A', 4, 1]
```

class sage.combinat.crystals.elementary_crystals.**RCrystal**(*cartan_type*, *weight*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The crystal R_λ.

For a fixed weight λ, the crystal $R_\lambda = \{r_\lambda\}$ is a single element crystal with the crystal structure defined by

$$\mathrm{wt}(r_\lambda) = \lambda, \quad e_i r_\lambda = f_i r_\lambda = 0, \quad \varepsilon_i(r_\lambda) = -\langle h_i, \lambda \rangle, \quad \varphi_i(r_\lambda) = 0,$$

where $\{h_i\}$ are the simple coroots.

Tensoring R_λ with a crystal B results in shifting the weights of the vertices in B by λ and may also cut a subset out of the original graph of B. That is, $\mathrm{wt}(r_\lambda \otimes b) = \mathrm{wt}(b) + \lambda$, where $b \in B$, provided $r_\lambda \otimes b \neq 0$. For example, the crystal graph of $B(\lambda)$ is the same as the crystal graph of $R_\lambda \otimes B(\infty)$ generated from the component $r_\lambda \otimes u_\infty$.

INPUT:

- cartan_type – A Cartan type
- weight – An element of the weight lattice of type cartan_type

EXAMPLES:

We check by tensoring R_λ with $B(\infty)$ results in a component of $B(\lambda)$:

```
sage: B = crystals.infinity.Tableaux("A2")
sage: R = crystals.elementary.R("A2", B.Lambda()[1]+B.Lambda()[2])
sage: T = crystals.TensorProduct(R, B)
sage: mg = T(R.highest_weight_vector(), B.highest_weight_vector())
sage: S = T.subcrystal(generators=[mg])
sage: for x in S: x.weight()
(2, 1, 0)
(2, 0, 1)
(1, 2, 0)
(1, 1, 1)
(1, 1, 1)
(1, 0, 2)
(0, 2, 1)
(0, 1, 2)
sage: C = crystals.Tableaux("A2", shape=[2,1])
sage: for x in C: x.weight()
(2, 1, 0)
(1, 2, 0)
(1, 1, 1)
(1, 0, 2)
(0, 1, 2)
(2, 0, 1)
(1, 1, 1)
(0, 2, 1)
sage: GT = T.digraph(subset=S)
sage: GC = C.digraph()
sage: GT.is_isomorphic(GC, edge_labels=True)
True
```

class Element

Bases: sage.combinat.crystals.elementary_crystals.AbstractSingleCrystalElement

Element of a R_λ crystal.

epsilon(*i*)

Return ε_i of self.

We have $\varepsilon_i(r_\lambda) = -\langle h_i, \lambda \rangle$ for all i, where h_i is a simple coroot.

INPUT:

- `i` – An element of the index set

EXAMPLES:

```
sage: la = RootSystem(['A',2]).weight_lattice().fundamental_weights()
sage: R = crystals.elementary.R("A2",la[1])
sage: r = R.highest_weight_vector()
sage: [r.epsilon(i) for i in R.index_set()]
[-1, 0]
```

phi(i)

Return φ_i of `self`, which is 0 for all i.

INPUT:

- `i` – An element of the index set

EXAMPLES:

```
sage: la = RootSystem("C5").weight_lattice().fundamental_weights()
sage: R = crystals.elementary.R("C5",la[4]+la[5])
sage: r = R.highest_weight_vector()
sage: [r.phi(i) for i in R.index_set()]
[0, 0, 0, 0, 0]
```

weight()

Return the weight of `self`, which is always λ.

EXAMPLES:

```
sage: ct = CartanType(['C',5])
sage: la = RootSystem(ct).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(ct,la[4]+la[5]-la[1]-la[2])
sage: t = T.highest_weight_vector()
sage: t.weight()
(0, 1, 2, 2, 1)
```

RCrystal.**cardinality**()

Return the cardinality of `self`, which is always 1.

EXAMPLES:

```
sage: La = RootSystem(['C',12]).weight_lattice().fundamental_weights()
sage: R = crystals.elementary.R(['C',12],La[9])
sage: R.cardinality()
1
```

RCrystal.**weight_lattice_realization**()

Return a realization of the lattice containing the weights of `self`.

EXAMPLES:

```
sage: La = RootSystem(['C',12]).weight_lattice().fundamental_weights()
sage: R = crystals.elementary.R(['C',12], La[9])
sage: R.weight_lattice_realization()
Ambient space of the Root system of type ['C', 12]

sage: ct = CartanMatrix([[2, -4], [-5, 2]])
sage: La = RootSystem(ct).weight_lattice().fundamental_weights()
sage: R = crystals.elementary.R(ct, La[1])
sage: R.weight_lattice_realization()
Weight space over the Rational Field of the Root system of type
[ 2 -4]
[-5  2]
```

class sage.combinat.crystals.elementary_crystals.**TCrystal**(*cartan_type*, *weight*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The crystal T_λ.

Let λ be a weight. As defined in [Kashiwara93] the crystal $T_\lambda = \{t_\lambda\}$ is a single element crystal with the crystal structure defined by

$$\mathrm{wt}(t_\lambda) = \lambda, \quad e_i t_\lambda = f_i t_\lambda = 0, \quad \varepsilon_i(t_\lambda) = \varphi_i(t_\lambda) = -\infty.$$

The crystal T_λ shifts the weights of the vertices in a crystal B by λ when tensored with B, but leaves the graph structure of B unchanged. That is to say, for all $b \in B$, we have $\mathrm{wt}(b \otimes t_\lambda) = \mathrm{wt}(b) + \lambda$.

INPUT:

- cartan_type – A Cartan type
- weight – An element of the weight lattice of type cartan_type

EXAMPLES:

```
sage: ct = CartanType(['A',2])
sage: C = crystals.Tableaux(ct, shape=[1])
sage: for x in C: x.weight()
(1, 0, 0)
(0, 1, 0)
(0, 0, 1)
sage: La = RootSystem(ct).ambient_space().fundamental_weights()
sage: TLa = crystals.elementary.T(ct, 3*(La[1] + La[2]))
sage: TP = crystals.TensorProduct(TLa, C)
sage: for x in TP: x.weight()
(7, 3, 0)
(6, 4, 0)
(6, 3, 1)
sage: G = C.digraph()
sage: H = TP.digraph()
sage: G.is_isomorphic(H,edge_labels=True)
True
```

class Element

Bases: sage.combinat.crystals.elementary_crystals.AbstractSingleCrystalElement

Element of a T_λ crystal.

epsilon(*i*)

Return ε_i of self, which is $-\infty$ for all i.

INPUT:

- i – An element of the index set

EXAMPLES:

```
sage: ct = CartanType(['C',5])
sage: la = RootSystem(ct).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(ct,la[4]+la[5]-la[1]-la[2])
sage: t = T.highest_weight_vector()
sage: [t.epsilon(i) for i in T.index_set()]
[-inf, -inf, -inf, -inf, -inf]
```

phi(*i*)

Return φ_i of self, which is $-\infty$ for all i.

INPUT:

- i – An element of the index set

EXAMPLES:

```
sage: ct = CartanType(['C',5])
sage: la = RootSystem(ct).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(ct,la[4]+la[5]-la[1]-la[2])
sage: t = T.highest_weight_vector()
sage: [t.phi(i) for i in T.index_set()]
[-inf, -inf, -inf, -inf, -inf]
```

weight()

Return the weight of self, which is always λ.

EXAMPLES:

```
sage: ct = CartanType(['C',5])
sage: la = RootSystem(ct).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(ct,la[4]+la[5]-la[1]-la[2])
sage: t = T.highest_weight_vector()
sage: t.weight()
(0, 1, 2, 2, 1)
```

TCrystal.**cardinality**()

Return the cardinality of self, which is always 1.

EXAMPLES:

```
sage: La = RootSystem(['C',12]).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(['C',12], La[9])
sage: T.cardinality()
1
```

TCrystal.**weight_lattice_realization**()

Return a realization of the lattice containing the weights of self.

EXAMPLES:

```
sage: La = RootSystem(['C',12]).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(['C',12], La[9])
sage: T.weight_lattice_realization()
Ambient space of the Root system of type ['C', 12]

sage: ct = CartanMatrix([[2, -4], [-5, 2]])
sage: La = RootSystem(ct).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(ct, La[1])
sage: T.weight_lattice_realization()
Weight space over the Rational Field of the Root system of type
[ 2 -4]
[-5  2]
```

5.1.45 Fast Rank Two Crystals

class sage.combinat.crystals.fast_crystals.**FastCrystal**(*ct*, *shape*, *format*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

An alternative implementation of rank 2 crystals. The root operators are implemented in memory by table lookup. This means that in comparison with the CrystalsOfTableaux, these crystals are slow to instantiate but faster for computation. Implemented for types A_2, B_2, and C_2.

INPUT:

- `cartan_type` – the Cartan type and must be either type A_2, B_2, or C_2
- `shape` – A shape is of the form `[l1,l2]` where `l1` and `l2` are either integers or (in type B_2) half integers such that `l1 - l2` is integral. It is assumed that `l1 >= l2 >= 0`. If `l1` and `l2` are integers, this will produce the a crystal isomorphic to the one obtained by ``crystals.Tableaux(type, shape=[l1,l2]). Furthermore `crystals.FastRankTwo(['B', 2], l1+1/2, l2+1/2)` produces a crystal isomorphic to the following crystal `T`:

```
sage: C = crystals.Tableaux(['B',2], shape=[l1,l2])               # not tested
sage: D = crystals.Spins(['B',2])                                 # not tested
sage: T = crystals.TensorProduct(C, D, C.list()[0], D.list()[0])  # not tested
```

- `format` – (default: `'string'`) the default representation of elements is in term of theBerenstein-Zelevinsky-Littelmann (BZL) strings `[a1, a2, ...]` described under metapost in `crystals`. Alternative representations may be obtained by the options `'dual_string'` or `'simple'`. In the `'simple'` format, the element is represented by and integer, and in the `'dual_string'` format, it is represented by the BZL string, but the underlying decomposition of the long Weyl group element into simple reflections is changed.

TESTS:

```
sage: C = crystals.FastRankTwo(['A',2],shape=[4,1])
sage: C.cardinality()
24
sage: C.cartan_type()
['A', 2]
sage: TestSuite(C).run()
sage: C = crystals.FastRankTwo(['B',2],shape=[4,1])
sage: C.cardinality()
154
sage: TestSuite(C).run()
sage: C = crystals.FastRankTwo(['B',2],shape=[3/2,1/2])
sage: C.cardinality()
16
sage: TestSuite(C).run()
sage: C = crystals.FastRankTwo(['C',2],shape=[2,1])
sage: C.cardinality()
16
sage: C = crystals.FastRankTwo(['C',2],shape=[3,1])
sage: C.cardinality()
35
sage: TestSuite(C).run()
```

class **Element** (*parent*, *value*, *format*)

Bases: `sage.structure.element.Element`

EXAMPLES:

```
sage: C = crystals.FastRankTwo(['A',2],shape=[2,1])
sage: c = C(0); c
[0, 0, 0]
sage: C[0].parent()
The fast crystal for A2 with shape [2,1]
sage: TestSuite(c).run()
```

e (*i*)

Returns the action of e_i on self.

EXAMPLES:

```
sage: C = crystals.FastRankTwo(['A',2],shape=[2,1])
sage: C(1).e(1)
[0, 0, 0]
sage: C(0).e(1) is None
True
```

f(*i*)

Returns the action of f_i on self.

EXAMPLES:

```
sage: C = crystals.FastRankTwo(['A',2],shape=[2,1])
sage: C(6).f(1)
[1, 2, 1]
sage: C(7).f(1) is None
True
```

weight()

Returns the weight of self.

EXAMPLES:

```
sage: [v.weight() for v in crystals.FastRankTwo(['A',2], shape=[2,1])]
[(2, 1, 0), (1, 2, 0), (1, 1, 1), (1, 0, 2), (0, 1, 2), (2, 0, 1), (1, 1, 1), (0, 2, 1)]
sage: [v.weight() for v in crystals.FastRankTwo(['B',2], shape=[1,0])]
[(1, 0), (0, 1), (0, 0), (0, -1), (-1, 0)]
sage: [v.weight() for v in crystals.FastRankTwo(['B',2], shape=[1/2,1/2])]
[(1/2, 1/2), (1/2, -1/2), (-1/2, 1/2), (-1/2, -1/2)]
sage: [v.weight() for v in crystals.FastRankTwo(['C',2], shape=[1,0])]
[(1, 0), (0, 1), (0, -1), (-1, 0)]
sage: [v.weight() for v in crystals.FastRankTwo(['C',2], shape=[1,1])]
[(1, 1), (1, -1), (0, 0), (-1, 1), (-1, -1)]
```

FastCrystal.**cmp_elements**(*x*, *y*)

Returns True if and only if there is a path from x to y in the crystal graph.

Because the crystal graph is classical, it is a directed acyclic graph which can be interpreted as a poset. This function implements the comparison function of this poset.

EXAMPLES:

```
sage: C = crystals.FastRankTwo(['A',2],shape=[2,1])
sage: x = C(0)
sage: y = C(1)
sage: C.cmp_elements(x,y)
-1
sage: C.cmp_elements(y,x)
1
sage: C.cmp_elements(x,x)
0
```

FastCrystal.**digraph**()

Returns the digraph associated to self.

EXAMPLES:

```
sage: C = crystals.FastRankTwo(['A',2],shape=[2,1])
sage: C.digraph()
Digraph on 8 vertices
```

FastCrystal.**list**()

Returns a list of the elements of self.

EXAMPLES:

```
sage: C = crystals.FastRankTwo(['A',2],shape=[2,1])
sage: C.list()
[[0, 0, 0],
 [1, 0, 0],
 [0, 1, 1],
 [0, 2, 1],
 [1, 2, 1],
 [0, 1, 0],
 [1, 1, 0],
 [2, 1, 0]]
```

5.1.46 Crystals of Generalized Young Walls

AUTHORS:

- Lucas David-Roesler: Initial version
- Ben Salisbury: Initial version
- Travis Scrimshaw: Initial version

Generalized Young walls are certain generalizations of Young tableaux introduced in [KS10] and designed to be a realization of the crystals $\mathcal{B}(\infty)$ and $\mathcal{B}(\lambda)$ in type $A_n^{(1)}$.

REFERENCES:

class sage.combinat.crystals.generalized_young_walls.**CrystalOfGeneralizedYoungWalls**(*n*, *La*)

Bases: sage.combinat.crystals.generalized_young_walls.InfinityCrystalOfGeneralizedYoungWa

The crystal $\mathcal{Y}(\lambda)$ of generalized Young walls of the given type with highest weight λ.

These were characterized in Theorem 4.1 of [KS10]. See GeneralizedYoungWall.in_highest_weight_crystal().

INPUT:

- n – type $A_n^{(1)}$
- weight – dominant integral weight

EXAMPLES:

```
sage: La = RootSystem(['A',3,1]).weight_lattice(extended=True).fundamental_weights()[1]
sage: YLa = crystals.GeneralizedYoungWalls(3,La)
sage: y = YLa([[0],[1,0,3,2,1],[2,1,0],[3]])
sage: y.pp()
        3|
    0|1|2|
1|2|3|0|1|
        0|
sage: y.weight()
-Lambda[0] + Lambda[2] + Lambda[3] - 3*delta
sage: y.in_highest_weight_crystal(La)
True
sage: y.f(1)
[[0], [1, 0, 3, 2, 1], [2, 1, 0], [3], [], [1]]
sage: y.f(1).f(1)
sage: yy = crystals.infinity.GeneralizedYoungWalls(3)([[0], [1, 0, 3, 2, 1], [2, 1, 0], [3], [],
sage: yy.f(1)
[[0], [1, 0, 3, 2, 1], [2, 1, 0], [3], [], [1], [], [], [], [1]]
```

```
sage: yyy = yy.f(1)
sage: yyy.in_highest_weight_crystal(La)
False

sage: LS = crystals.LSPaths(['A',3,1],[1,0,0,0])
sage: C = LS.subcrystal(max_depth=4)
sage: G = LS.digraph(subset=C)
sage: P = RootSystem(['A',3,1]).weight_lattice(extended=True)
sage: La = P.fundamental_weights()
sage: YW = crystals.GeneralizedYoungWalls(3,La[0])
sage: CW = YW.subcrystal(max_depth=4)
sage: GW = YW.digraph(subset=CW)
sage: GW.is_isomorphic(G,edge_labels=True)
True
```

To display the crystal down to a specified depth:

```
sage: S = YLa.subcrystal(max_depth=4)
sage: G = YLa.digraph(subset=S)
sage: view(G, tightpage=True) # not tested
```

Element
alias of CrystalOfGeneralizedYoungWallsElement

class sage.combinat.crystals.generalized_young_walls.**CrystalOfGeneralizedYoungWallsElement**(*pa*
da

Bases: sage.combinat.crystals.generalized_young_walls.GeneralizedYoungWall

Element of the highest weight crystal of generalized Young walls.

e(*i*)
Compute the action of $\widetilde{e}_i$ restricted to the highest weight crystal.

EXAMPLES:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_weights()[1]
sage: hwy = crystals.GeneralizedYoungWalls(2,La)([[],[1,0],[2,1]])
sage: hwy.e(1)
[[], [1, 0], [2]]
sage: hwy.e(2)
sage: hwy.e(3)
```

f(*i*)
Compute the action of $\widetilde{f}_i$ restricted to the highest weight crystal.

EXAMPLES:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_weights()[1]
sage: GYW = crystals.infinity.GeneralizedYoungWalls(2)
sage: y = GYW([[],[1,0],[2,1]])
sage: y.f(1)
[[], [1, 0], [2, 1], [], [1]]
sage: hwy = crystals.GeneralizedYoungWalls(2,La)([[],[1,0],[2,1]])
sage: hwy.f(1)
```

phi(*i*)
Return the value $\varepsilon_i(Y) + \langle h_i, \mathrm{wt}(Y) \rangle$, where h_i is the i-th simple coroot and Y is `self`.

EXAMPLES:

```
sage: La = RootSystem(['A',3,1]).weight_lattice(extended=True).fundamental_weights()
sage: y = crystals.GeneralizedYoungWalls(3,La[0])([])
sage: y.phi(1)
0
sage: y.phi(2)
0
```

weight()

Return the weight of `self` in the highest weight crystal as an element of the weight lattice $\bigoplus_{i=0}^{n} \mathbf{Z}\Lambda_i$.

EXAMPLES:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_weights()[1]
sage: hwy = crystals.GeneralizedYoungWalls(2,La)([[],[1,0],[2,1]])
sage: hwy.weight()
Lambda[0] - Lambda[1] + Lambda[2] - delta
```

class sage.combinat.crystals.generalized_young_walls.**GeneralizedYoungWall**(*parent*, *data*)

Bases: sage.combinat.combinat.CombinatorialElement

A generalized Young wall.

For more information, see InfinityCrystalOfGeneralizedYoungWalls.

EXAMPLES:

```
sage: Y = crystals.infinity.GeneralizedYoungWalls(4)
sage: mg = Y.module_generators[0]; mg.pp()
0
sage: mg.f_string([1,2,0,1]).pp()
1|2|
0|1|
  |
```

Epsilon()

Return $\sum_{i=0}^{n} \varepsilon_i(Y)\Lambda_i$ where Y is `self`.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(3)([[0],[1,0,3,2],[2,1],[3,2,1,0,3,2],[0],
sage: y.Epsilon()
Lambda[0] + 3*Lambda[2]
```

Phi()

Return $\sum_{i=0}^{n} \varphi_i(Y)\Lambda_i$ where Y is `self`.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(3)([[0],[1,0,3,2],[2,1],[3,2,1,0,3,2],[0],
sage: y.Phi()
-Lambda[0] + 3*Lambda[1] - Lambda[2] + 3*Lambda[3]

sage: x = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,1,0,3,2],[],[]
sage: x.Phi()
2*Lambda[0] + Lambda[1] - Lambda[2] + Lambda[3]
```

a(*i*, *k*)

Return the number $a_i(k)$ of i-colored boxes in the k-th column of `self`.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(3)([[0],[1,0,3,2],[2,1],[3,2,1,0,3,2],[0],
sage: y.a(1,2)
1
sage: y.a(0,2)
1
sage: y.a(3,2)
0
```

column(k)

Return the list of boxes from the k-th column of self.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(3)([[0],[1,0,3,2],[2,1],[3,2,1,0,3,2],[0],
sage: y.column(2)
[None, 0, 1, 2, None, None, None]

sage: hw = crystals.infinity.GeneralizedYoungWalls(5)([])
sage: hw.column(1)
[]
```

content()

Return total number of blocks in self.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(2)([[0],[1,0],[2,1,0,2],[],[1]])
sage: y.content()
8

sage: x = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,1,0,3,2],[],[]
sage: x.content()
13
```

e(i)

Return the application of the Kashiwara raising operator $\widetilde{e}_i$ on self.

This will remove the i-colored box corresponding to the rightmost $+$ in self.signature(i).

EXAMPLES:

```
sage: x = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,1,0,3,2],[],[]
sage: x.e(2)
[[], [1, 0, 3, 2], [2, 1], [3, 2, 1, 0, 3, 2]]
sage: _.e(2)
[[], [1, 0, 3], [2, 1], [3, 2, 1, 0, 3, 2]]
sage: _.e(2)
[[], [1, 0, 3], [2, 1], [3, 2, 1, 0, 3]]
sage: _.e(2)
```

epsilon(i)

Return the number of i-colored arrows in the i-string above self in the crystal graph.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,1,0,3,2],[],[]
sage: y.epsilon(1)
0
sage: y.epsilon(2)
3
```

```
sage: y.epsilon(0)
0
```

f(*i*)

Return the application of the Kashiwara lowering operator $\widetilde{f}_i$ on `self`.

This will add an i-colored colored box to the site corresponding to the leftmost plus in `self.signature(i)`.

EXAMPLES:

```
sage: hw = crystals.infinity.GeneralizedYoungWalls(2)([])
sage: hw.f(1)
[[], [1]]
sage: _.f(2)
[[], [1], [2]]
sage: _.f(0)
[[], [1, 0], [2]]
sage: _.f(0)
[[0], [1, 0], [2]]
```

generate_signature(*i*)

The i-signature of `self` (with whitespace where cancellation occurs) together with the unreduced sequence from $\{+, -\}$. The result also records to the row and column position of the sign.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(2)([[0],[1,0],[2,1,0,2],[],[1]])
sage: y.generate_signature(1)
([('+', 2, 5], ['-', 4, 1]], '  ')
```

in_highest_weight_crystal(*La*)

Return a boolean indicating if the generalized Young wall element is in the highest weight crystal cut out by the given highest weight `La`.

By Theorem 4.1 of [KS10], a generalized Young wall Y represents a vertex in the highest weight crystal $Y(\lambda)$, with $\lambda = \Lambda_{i_1} + \Lambda_{i_2} + \cdots + \Lambda_{i_\ell}$ a dominant integral weight of level $\ell > 0$, if it satisfies the following condition. For each positive integer k, if there exists $j \in I$ such that $a_j(k) - a_{j-1}(k) > 0$, then for some $p = 1, \ldots, \ell$,

$$j + k \equiv i_p + 1 \bmod n + 1 \text{ and } a_j(k) - a_{j-1}(k) \leq \lambda(h_{i_p}),$$

where $\{h_0, h_1, \ldots, h_n\}$ is the set of simple coroots attached to $A_n^{(1)}$.

EXAMPLES:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_weights()[1]
sage: GYW = crystals.infinity.GeneralizedYoungWalls(2)
sage: y = GYW([[],[1,0],[2,1]])
sage: y.in_highest_weight_crystal(La)
True
sage: x = GYW([[],[1],[2],[],[],[2],[],[],[2]])
sage: x.in_highest_weight_crystal(La)
False
```

latex_large()

Generate LaTeX code for `self` but the output is larger. Requires TikZ.

EXAMPLES:

```
sage: x = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,1,0,3,2],[],[]
sage: x.latex_large()
'\\begin{tikzpicture}[baseline=5,scale=.45] \n \\foreach \\x [count=\\s from 0] in \n{{},{1,
```

number_of_parts()

Return the value of $\mathscr{N}$ on `self`.

In [KLRS], the statistic $\mathscr{N}$ was defined on elements in $\mathcal{Y}(\infty)$ which counts how many parts are in the corresponding Kostant partition. Specifically, the computation of $\mathscr{N}(Y)$ is done using the following algorithm:

- If Y has no rows whose right-most box is colored n and such that the length of this row is a multiple of $n+1$, then $\mathscr{N}(Y)$ is the total number of distinct rows in Y, not counting multiplicity.
- Otherwise, search Y for the longest row such that the right-most box is colored n and such that the total number of boxes in the row is $k(n+1)$ for some $k \geq 1$. Replace this row by $n+1$ distinct rows of length k, reordering all rows, if necessary, so that the result is a proper wall. (Note that the resulting wall may no longer be reduced.) Repeat the search and replace process for all other rows of the above form for each $k' < k$. Then $\mathscr{N}(Y)$ is the number of distinct rows, not counting multipicity, in the wall resulting from this process.

EXAMPLES:

```
sage: Y = crystals.infinity.GeneralizedYoungWalls(3)
sage: y = Y([[0],[],[],[],[0],[],[],[],[0]])
sage: y.number_of_parts()
1

sage: Y = crystals.infinity.GeneralizedYoungWalls(3)
sage: y = Y([[0,3,2],[1,0],[],[],[0,3],[1,0],[],[],[0]])
sage: y.number_of_parts()
4

sage: Y = crystals.infinity.GeneralizedYoungWalls(2)
sage: y = Y([[0,2,1],[1,0],[2,1,0,2,1,0,2,1,0],[],[2,1,0,2,1,0]])
sage: y.number_of_parts()
8
```

phi(i)

Return the value $\varepsilon_i(Y) + \langle h_i, \mathrm{wt}(Y) \rangle$, where h_i is the i-th simple coroot and Y is `self`.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(3)([[0],[1,0,3,2],[2,1],[3,2,1,0,3,2],[0],
sage: y.phi(1)
3
sage: y.phi(2)
-1
```

pp()

Return an ASCII drawing of `self`.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(2)([[0,2,1],[1,0,2,1,0],[],[0],[1,0,2],[],
sage: y.pp()
        1|
         |
         |
    2|0|1|
        0|
```

```
          |
0|1|2|0|1|
    1|2|0|
```

raw_signature(*i*)

Return the sequence from $\{+,-\}$ obtained from all i-admissible slots and removable i-boxes without canceling any $(+,-)$-pairs. The result also notes the row and column of the sign.

EXAMPLES:

```
sage: x = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,1,0,3,2],[],[]
sage: x.raw_signature(2)
[['-', 3, 6], ['-', 1, 4], ['-', 6, 1]]
```

signature(*i*)

Return the i-signature of `self`.

The signature is obtained by reading `self` in columns bottom to top starting from the left. Then add a $-$ at every i-box which may be removed from `self` and still obtain a legal generalized Young wall, and add a $+$ at each site for which an i-box may be added and still obtain a valid generalized Young wall. Then successively cancel any $(+,-)$-pair to obtain a sequence of the form $-\cdots-+\cdots+$. This resulting sequence is the output.

EXAMPLES:

```
sage: y = crystals.infinity.GeneralizedYoungWalls(2)([[0],[1,0],[2,1,0,2],[],[1]])
sage: y.signature(1)
''

sage: x = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,1,0,3,2],[],[]
sage: x.signature(2)
'---'
```

sum_of_weighted_row_lengths()

Return the value of $\mathscr{M}$ on `self`.

Let $\mathcal{Y}_0 \subset \mathcal{Y}(\infty)$ be the set of generalized Young walls which have no rows whose right-most box is colored n. For $Y \in \mathcal{Y}_0$,

$$\mathscr{M}(Y) = \sum_{i=1}^{n} (i+1) M_i(Y),$$

where $M_i(Y)$ is the number of nonempty rows in Y whose right-most box is colored $i-1$.

EXAMPLES:

```
sage: Y = crystals.infinity.GeneralizedYoungWalls(2)
sage: y = Y([[0,2,1,0,2],[1,0,2],[],[0,2],[1,0],[],[0],[1,0]])
sage: y.sum_of_weighted_row_lengths()
15
```

weight(*root_lattice=False*)

Returns the weight of `self`.

INPUT:

- *root_lattice* – boolean determining whether weight should appear in root lattice or not in extended affine weight lattice.

EXAMPLES:

```
sage: x = crystals.infinity.GeneralizedYoungWalls(3)([[],[1,0,3,2],[2,1],[3,2,1,0,3,2],[],[]
sage: x.weight()
2*Lambda[0] + Lambda[1] - 4*Lambda[2] + Lambda[3] - 2*delta
sage: x.weight(root_lattice=True)
-2*alpha[0] - 3*alpha[1] - 5*alpha[2] - 3*alpha[3]
```

class sage.combinat.crystals.generalized_young_walls.**InfinityCrystalOfGeneralizedYoungWalls**(*n*
c
e
g

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The crystal $\mathcal{Y}(\infty)$ of generalized Young walls of type $A_n^{(1)}$ as defined in [KS10].

A generalized Young wall is a collection of boxes stacked on a fixed board, such that color of the box at the site located in the j-th row from the bottom and the i-th column from the right is $j - 1 \mod n+1$. There are several growth conditions on elements in $Y \in \mathcal{Y}(\infty)$:

- Walls grow in rows from right to left. That is, for every box $y \in Y$ that is not in the rightmost column, there must be a box immediately to the right of y.
- For all $p > q$ such that $p - q \equiv 0 \mod n+1$, the p-th row has most as many boxes as the q-th row.
- There does not exist a column in the wall such that if one i-colored box, for every $i = 0, 1, \ldots, n$, is removed from that column, then the result satisfies the above conditions.

There is a crystal structure on $\mathcal{Y}(\infty)$ defined as follows. Define maps

$$\widetilde{e}_i,\ \widetilde{f}_i \colon \mathcal{Y}(\infty) \longrightarrow \mathcal{Y}(\infty) \sqcup \{0\}, \qquad \varepsilon_i,\ \varphi_i \colon \mathcal{Y}(\infty) \longrightarrow \mathbf{Z}, \qquad \mathrm{wt} \colon \mathcal{Y}(\infty) \longrightarrow \bigoplus_{i=0}^{n} \mathbf{Z}\Lambda_i \oplus \mathbf{Z}\delta,$$

by

$$\mathrm{wt}(Y) = -\sum_{i=0}^{n} m_i(Y)\alpha_i,$$

where $m_i(Y)$ is the number of i-boxes in Y, $\varepsilon_i(Y)$ is the number of $-$ in the i-signature of Y, and

$$\varphi_i(Y) = \varepsilon_i(Y) + \langle h_i, \mathrm{wt}(Y) \rangle.$$

See GeneralizedYoungWall.e(), GeneralizedYoungWall.f(), and GeneralizedYoungWall.signature() for more about $\widetilde{e}_i$, $\widetilde{f}_i$, and i-signatures.

INPUT:

- n – type $A_n^{(1)}$

EXAMPLES:

```
sage: Yinf = crystals.infinity.GeneralizedYoungWalls(3)
sage: y = Yinf([[0],[1,0,3,2],[],[3,2,1],[0],[1,0]])
sage: y.pp()
    0|1|
      0|
  1|2|3|
       |
2|3|0|1|
      0|
sage: y.weight(root_lattice=True)
```

```
-4*alpha[0] - 3*alpha[1] - 2*alpha[2] - 2*alpha[3]
sage: y.f(0)
[[0], [1, 0, 3, 2], [], [3, 2, 1], [0], [1, 0], [], [], [0]]
sage: y.e(0).pp()
    0|1|
      |
  1|2|3|
      |
2|3|0|1|
    0|
```

To display the crystal down to depth 3:

```
sage: S = Yinf.subcrystal(max_depth=3)
sage: G = Yinf.digraph(subset=S) # long time
sage: view(G, tightpage=True) # not tested
```

Element
alias of GeneralizedYoungWall

weight_lattice_realization()
Return the extended affine weight lattice of self.

EXAMPLES:

```
sage: Y = crystals.infinity.GeneralizedYoungWalls(3)
sage: Y.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 3, 1]
```

5.1.47 Highest weight crystals

class sage.combinat.crystals.highest_weight_crystals.**FiniteDimensionalHighestWeightCrystal_Ty**
Bases: sage.combinat.crystals.tensor_product.TensorProductOfCrystals

Commonalities for all finite dimensional type E highest weight crystals.

Subclasses should setup an attribute column_crystal in their __init__ method before calling the __init__ method of this class.

Element
alias of TensorProductOfRegularCrystalsElement

module_generator()
This yields the module generator (or highest weight element) of the classical crystal of given dominant weight in self.

EXAMPLES:

```
sage: C=CartanType(['E',6])
sage: La=C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[2])
sage: T.module_generator()
[[(2, -1), (1,)]]
sage: T = crystals.HighestWeight(0*La[2])
sage: T.module_generator()
[]

sage: C=CartanType(['E',7])
sage: La=C.root_system().weight_lattice().fundamental_weights()
```

```
sage: T = crystals.HighestWeight(La[1])
sage: T.module_generator()
[[(-7, 1), (7,)]]
```

class sage.combinat.crystals.highest_weight_crystals.**FiniteDimensionalHighestWeightCrystal_Ty**
Bases: sage.combinat.crystals.highest_weight_crystals.FiniteDimensionalHighestWeightCryst

Class of finite dimensional highest weight crystals of type E_6.

EXAMPLES:

```
sage: C=CartanType(['E',6])
sage: La=C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[2]); T
Finite dimensional highest weight crystal of type ['E', 6] and highest weight Lambda[2]
sage: B1 = T.column_crystal[1]; B1
The crystal of letters for type ['E', 6]
sage: B6 = T.column_crystal[6]; B6
The crystal of letters for type ['E', 6] (dual)
sage: t = T(B6([-1]),B1([-1,3])); t
[(-1,), (-1, 3)]
sage: [t.epsilon(i) for i in T.index_set()]
[2, 0, 0, 0, 0, 0]
sage: [t.phi(i) for i in T.index_set()]
[0, 0, 1, 0, 0, 0]
sage: TestSuite(t).run()
```

class sage.combinat.crystals.highest_weight_crystals.**FiniteDimensionalHighestWeightCrystal_Ty**
Bases: sage.combinat.crystals.highest_weight_crystals.FiniteDimensionalHighestWeightCryst

Class of finite dimensional highest weight crystals of type E_7.

EXAMPLES:

```
sage: C=CartanType(['E',7])
sage: La=C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[1])
sage: T.cardinality()
133
sage: B7 = T.column_crystal[7]; B7
The crystal of letters for type ['E', 7]
sage: t = T(B7([-5, 6]), B7([-2, 3])); t
[(-5, 6), (-2, 3)]
sage: [t.epsilon(i) for i in T.index_set()]
[0, 1, 0, 0, 1, 0, 0]
sage: [t.phi(i) for i in T.index_set()]
[0, 0, 1, 0, 0, 1, 0]
sage: TestSuite(t).run()
```

sage.combinat.crystals.highest_weight_crystals.**HighestWeightCrystal**(*dominant_weight*, *model=None*)

Return the highest weight crystal of highest weight `dominant_weight` of the given `model`.

INPUT:

- `dominant_weight` – a dominant weight
- `model` – (optional) if not specified, then we have the following default models:
  - types A_n, B_n, C_n, D_n, G_2 - `tableaux`
  - types $E_{6,7}$ - `type E finite dimensional crystal`

– all other types - LS paths

otherwise can be one of the following:

– 'Tableaux' - KN tableaux

– 'TypeE' - type E finite dimensional crystal

– 'NakajimaMonomials' - Nakajima monomials

– 'LSPaths' - LS paths

– 'AlcovePaths' - alcove paths

– 'GeneralizedYoungWalls' - generalized Young walls

– 'RiggedConfigurations' - rigged configurations

EXAMPLES:

```
sage: La = RootSystem(['A',2]).weight_lattice().fundamental_weights()
sage: wt = La[1] + La[2]
sage: crystals.HighestWeight(wt)
The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]]

sage: La = RootSystem(['C',2]).weight_lattice().fundamental_weights()
sage: wt = 5*La[1] + La[2]
sage: crystals.HighestWeight(wt)
The crystal of tableaux of type ['C', 2] and shape(s) [[6, 1]]
```

Some type E examples:

```
sage: C = CartanType(['E',6])
sage: La = C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[1])
sage: T.cardinality()
27
sage: T = crystals.HighestWeight(La[6])
sage: T.cardinality()
27
sage: T = crystals.HighestWeight(La[2])
sage: T.cardinality()
78
sage: T = crystals.HighestWeight(La[4])
sage: T.cardinality()
2925
sage: T = crystals.HighestWeight(La[3])
sage: T.cardinality()
351
sage: T = crystals.HighestWeight(La[5])
sage: T.cardinality()
351

sage: C = CartanType(['E',7])
sage: La = C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[1])
sage: T.cardinality()
133
sage: T = crystals.HighestWeight(La[2])
sage: T.cardinality()
912
sage: T = crystals.HighestWeight(La[3])
sage: T.cardinality()
```

```
8645
sage: T = crystals.HighestWeight(La[4])
sage: T.cardinality()
365750
sage: T = crystals.HighestWeight(La[5])
sage: T.cardinality()
27664
sage: T = crystals.HighestWeight(La[6])
sage: T.cardinality()
1539
sage: T = crystals.HighestWeight(La[7])
sage: T.cardinality()
56
```

An example with an affine type:

```
sage: C = CartanType(['C',2,1])
sage: La = C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[1])
sage: sorted(T.subcrystal(max_depth=3), key=str)
[(-Lambda[0] + 3*Lambda[1] - Lambda[2] - delta,),
 (-Lambda[0] + Lambda[1] + Lambda[2] - delta,),
 (-Lambda[1] + 2*Lambda[2] - delta,),
 (2*Lambda[0] - Lambda[1],),
 (Lambda[0] + Lambda[1] - Lambda[2],),
 (Lambda[0] - Lambda[1] + Lambda[2],),
 (Lambda[1],)]
```

Using the various models:

```
sage: La = RootSystem(['F',4]).weight_lattice().fundamental_weights()
sage: wt = La[1] + La[4]
sage: crystals.HighestWeight(wt)
The crystal of LS paths of type ['F', 4] and weight Lambda[1] + Lambda[4]
sage: crystals.HighestWeight(wt, model='NakajimaMonomials')
Highest weight crystal of modified Nakajima monomials of
 Cartan type ['F', 4] and highest weight Lambda[1] + Lambda[4]
sage: crystals.HighestWeight(wt, model='AlcovePaths')
Highest weight crystal of alcove paths of type ['F', 4] and weight Lambda[1] + Lambda[4]
sage: crystals.HighestWeight(wt, model='RiggedConfigurations')
Crystal of rigged configurations of type ['F', 4] and weight Lambda[1] + Lambda[4]
```

5.1.48 Induced Crystals

We construct a crystal structure on a set induced by a bijection Φ.

AUTHORS:

- Travis Scrimshaw (2014-05-15): Initial implementation

class sage.combinat.crystals.induced_structure.**InducedCrystal**(*X*, *phi*, *inverse*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

A crystal induced from an injection.

Let X be a set and let C be crystal and consider any injection $\Phi : X \to C$. We induce a crystal structure on X by considering Φ to be a crystal morphism.

Alternatively we can induce a crystal structure on some (sub)set of X by considering an injection $\Phi : C \to X$ considered as a crystal morphism. This form is also useful when the set X is not explicitly known.

INPUT:

- `X` – the base set
- `phi` – the map Φ
- `inverse` – (optional) the inverse map Φ^{-1}
- `from_crystal` – (default: `False`) if the induced structure is of the second type $\Phi : C \to X$

EXAMPLES:

We construct a crystal structure of Gelfand-Tsetlin patterns by going through their bijection with semistandard tableaux:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,3))
sage: G = GelfandTsetlinPatterns(4, 3)
sage: phi = lambda x: D(x.to_tableau())
sage: phi_inv = lambda x: G(x.to_tableau())
sage: I = crystals.Induced(G, phi, phi_inv)
sage: I.digraph().is_isomorphic(D.digraph(), edge_labels=True)
True
```

Now we construct the above example but inducing the structure going the other way (from tableaux to Gelfand-Tsetlin patterns). This can also give us more information coming from the crystal.

```
sage: D2 = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,1))
sage: G2 = GelfandTsetlinPatterns(4, 1)
sage: phi2 = lambda x: D2(x.to_tableau())
sage: phi2_inv = lambda x: G2(x.to_tableau())
sage: I2 = crystals.Induced(D2, phi2_inv, phi2, from_crystal=True)
sage: I2.module_generators
([[0, 0, 0, 0], [0, 0, 0], [0, 0], [0]],
 [[1, 0, 0, 0], [1, 0, 0], [1, 0], [1]],
 [[1, 1, 0, 0], [1, 1, 0], [1, 1], [1]],
 [[1, 1, 1, 0], [1, 1, 1], [1, 1], [1]],
 [[1, 1, 1, 1], [1, 1, 1], [1, 1], [1]])
```

We check an example when the codomain is larger than the domain (although here the crystal structure is trivial):

```
sage: P = Permutations(4)
sage: D = crystals.Tableaux(['A',3], shapes=Partitions(4))
sage: T = crystals.TensorProduct(D, D)
sage: phi = lambda p: T(D(RSK(p)[0]), D(RSK(p)[1]))
sage: phi_inv = lambda d: RSK_inverse(d[0].to_tableau(), d[1].to_tableau(), output='permutation'
sage: all(phi_inv(phi(p)) == p for p in P) # Check it really is the inverse
True
sage: I = crystals.Induced(P, phi, phi_inv)
sage: I.digraph()
Multi-digraph on 24 vertices
```

We construct an example without a specified inverse map:

```
sage: X = Words(2,4)
sage: L = crystals.Letters(['A',1])
sage: T = crystals.TensorProduct(*[L]*4)
sage: Phi = lambda x : T(*[L(i) for i in x])
sage: I = crystals.Induced(X, Phi)
sage: I.digraph()
Digraph on 16 vertices
```

class **Element**

Bases: sage.structure.element_wrapper.ElementWrapper

An element of an induced crystal.

e(i)

Return e_i of self.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,3))
sage: G = GelfandTsetlinPatterns(4, 3)
sage: phi = lambda x: D(x.to_tableau())
sage: phi_inv = lambda x: G(x.to_tableau())
sage: I = crystals.Induced(G, phi, phi_inv)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: elt.e(1)
sage: elt.e(2)
[[1, 1, 0, 0], [1, 1, 0], [1, 1], [1]]
sage: elt.e(3)
```

epsilon(i)

Return ε_i of self.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,3))
sage: G = GelfandTsetlinPatterns(4, 3)
sage: phi = lambda x: D(x.to_tableau())
sage: phi_inv = lambda x: G(x.to_tableau())
sage: I = crystals.Induced(G, phi, phi_inv)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: [elt.epsilon(i) for i in I.index_set()]
[0, 1, 0]
```

f(i)

Return f_i of self.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,3))
sage: G = GelfandTsetlinPatterns(4, 3)
sage: phi = lambda x: D(x.to_tableau())
sage: phi_inv = lambda x: G(x.to_tableau())
sage: I = crystals.Induced(G, phi, phi_inv)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: elt.f(1)
[[1, 1, 0, 0], [1, 1, 0], [1, 0], [0]]
sage: elt.f(2)
sage: elt.f(3)
[[1, 1, 0, 0], [1, 0, 0], [1, 0], [1]]
```

phi(i)

Return φ_i of self.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,3))
sage: G = GelfandTsetlinPatterns(4, 3)
sage: phi = lambda x: D(x.to_tableau())
sage: phi_inv = lambda x: G(x.to_tableau())
sage: I = crystals.Induced(G, phi, phi_inv)
```

```
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: [elt.phi(i) for i in I.index_set()]
[1, 0, 1]
```

weight()
Return the weight of `self`.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,3))
sage: G = GelfandTsetlinPatterns(4, 3)
sage: phi = lambda x: D(x.to_tableau())
sage: phi_inv = lambda x: G(x.to_tableau())
sage: I = crystals.Induced(G, phi, phi_inv)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: elt.weight()
(1, 0, 1, 0)
```

InducedCrystal.**cardinality**()
Return the cardinality of `self`.

EXAMPLES:

```
sage: P = Permutations(4)
sage: D = crystals.Tableaux(['A',3], shapes=Partitions(4))
sage: T = crystals.TensorProduct(D, D)
sage: phi = lambda p: T(D(RSK(p)[0]), D(RSK(p)[1]))
sage: phi_inv = lambda d: RSK_inverse(d[0].to_tableau(), d[1].to_tableau(), output='permutat
sage: I = crystals.Induced(P, phi, phi_inv)
sage: I.cardinality() == factorial(4)
True
```

class sage.combinat.crystals.induced_structure.**InducedFromCrystal**(*X*, *phi*, *inverse*)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

A crystal induced from an injection.

Alternatively we can induce a crystal structure on some (sub)set of X by considering an injection $\Phi : C \to X$ considered as a crystal morphism.

See also:

InducedCrystal

INPUT:

- `X` – the base set
- `phi` – the map Φ
- `inverse` – (optional) the inverse map Φ^{-1}

EXAMPLES:

We construct a crystal structure on generalized permutations with a fixed first row by using RSK:

```
sage: C = crystals.Tableaux(['A',3], shape=[2,1])
sage: def psi(x):
....:     ret = RSK_inverse(x.to_tableau(), Tableau([[1,1],[2]]))
....:     return (tuple(ret[0]), tuple(ret[1]))
sage: psi_inv = lambda x: C(RSK(*x)[0])
sage: I = crystals.Induced(C, psi, psi_inv, from_crystal=True)
```

class **Element**

Bases: sage.structure.element_wrapper.ElementWrapper

An element of an induced crystal.

e(i)

Return e_i of self.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,1))
sage: G = GelfandTsetlinPatterns(4, 1)
sage: def phi(x): return G(x.to_tableau())
sage: def phi_inv(x): return D(G(x).to_tableau())
sage: I = crystals.Induced(D, phi, phi_inv, from_crystal=True)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: elt.e(1)
sage: elt.e(2)
[[1, 1, 0, 0], [1, 1, 0], [1, 1], [1]]
sage: elt.e(3)
```

epsilon(i)

Return ε_i of self.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,1))
sage: G = GelfandTsetlinPatterns(4, 1)
sage: def phi(x): return G(x.to_tableau())
sage: def phi_inv(x): return D(G(x).to_tableau())
sage: I = crystals.Induced(D, phi, phi_inv, from_crystal=True)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: [elt.epsilon(i) for i in I.index_set()]
[0, 1, 0]
```

f(i)

Return f_i of self.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,1))
sage: G = GelfandTsetlinPatterns(4, 1)
sage: def phi(x): return G(x.to_tableau())
sage: def phi_inv(x): return D(G(x).to_tableau())
sage: I = crystals.Induced(D, phi, phi_inv, from_crystal=True)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: elt.f(1)
[[1, 1, 0, 0], [1, 1, 0], [1, 0], [0]]
sage: elt.f(2)
sage: elt.f(3)
[[1, 1, 0, 0], [1, 0, 0], [1, 0], [1]]
```

phi(i)

Return φ_i of self.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,1))
sage: G = GelfandTsetlinPatterns(4, 1)
sage: def phi(x): return G(x.to_tableau())
sage: def phi_inv(x): return D(G(x).to_tableau())
sage: I = crystals.Induced(D, phi, phi_inv, from_crystal=True)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
```

```
sage: [elt.epsilon(i) for i in I.index_set()]
[0, 1, 0]
```

weight()
Return the weight of `self`.

EXAMPLES:

```
sage: D = crystals.Tableaux(['A',3], shapes=PartitionsInBox(4,1))
sage: G = GelfandTsetlinPatterns(4, 1)
sage: def phi(x): return G(x.to_tableau())
sage: def phi_inv(x): return D(G(x).to_tableau())
sage: I = crystals.Induced(D, phi, phi_inv, from_crystal=True)
sage: elt = I([[1, 1, 0, 0], [1, 1, 0], [1, 0], [1]])
sage: elt.weight()
(1, 0, 1, 0)
```

InducedFromCrystal.**cardinality**()
Return the cardinality of `self`.

EXAMPLES:

```
sage: C = crystals.Tableaux(['A',3], shape=[2,1])
sage: def psi(x):
....:     ret = RSK_inverse(x.to_tableau(), Tableau([[1,1],[2]]))
....:     return (tuple(ret[0]), tuple(ret[1]))
sage: psi_inv = lambda x: C(RSK(*x)[0])
sage: I = crystals.Induced(C, psi, psi_inv, from_crystal=True)
sage: I.cardinality() == C.cardinality()
True
```

5.1.49 $\mathcal{B}(\infty)$ Crystals of Tableaux in Nonexceptional Types and G_2

A tableau model for $\mathcal{B}(\infty)$. For more information, see `InfinityCrystalOfTableaux`.

AUTHORS:

- Ben Salisbury: Initial version
- Travis Scrimshaw: Initial version

class sage.combinat.crystals.infinity_crystals.**InfinityCrystalOfTableaux**(*cartan_type*)
Bases: sage.combinat.crystals.tensor_product.CrystalOfWords

$\mathcal{B}(\infty)$ crystal of tableaux.

A tableaux model $\mathcal{T}(\infty)$ for the crystal $\mathcal{B}(\infty)$ is introduced by Hong and Lee in [HL08]. This model is currently valid for types A_n, B_n, C_n, D_n, and G_2, and builds on the tableaux model given by Kashiwara and Nakashima [KN94] in types A_n, B_n, C_n, and D_n, and by Kang and Misra [KM94] in type G_2.

Note: We are using the English convention for our tableaux.

We say a tableau T is *marginally large* if:

- for each $1 \leq i \leq n$, the leftmost box in the i-th row from the top in T is an i-box,
- for each $1 \leq i \leq n$, the number of i-boxes in the i-th row from the top in T is greater than the total number of boxes in the $(i+1)$-th row by exactly one.

We now will describe this tableaux model type-by-type.

Type A_n

$\mathcal{T}(\infty)$ is the set of marginally large semistandard tableaux with exactly n rows over the alphabet $\{1 \prec 2 \prec \cdots \prec n+1\}$.

Type B_n

$\mathcal{T}(\infty)$ is the set of marginally large semistandard tableaux with exactly n rows over the alphabet $\{1 \prec \cdots \prec n \prec 0 \prec \overline{n} \prec \cdots \prec \overline{1}\}$ and subject to the following constraints:

- for each $1 \leq i \leq n$, the contents of the boxes in the i-th row are $\preceq \overline{i}$,
- the entry 0 can appear at most once in a single row.

Type C_n

$\mathcal{T}(\infty)$ is the set of marginally large semistandard tableaux with exactly n rows over the alphabet $\{1 \prec \cdots \prec n \prec \overline{n} \prec \cdots \prec \overline{1}\}$ and for each $1 \leq i \leq n$, the contents of the boxes in the i-th row are $\preceq \overline{i}$.

Type D_n

$\mathcal{T}(\infty)$ is the set of marginally large semistandard tableaux with exactly $n-1$ rows over the alphabet $\{1 \prec \cdots \prec n, \overline{n} \prec \cdots \prec \overline{1}\}$ and subject to the following constraints:

- for each $1 \leq i \leq n$, the contents of the boxes in the i-th row are $\preceq \overline{i}$,
- the entries n and $\overline{n}$ may not appear simultaneously in a single row.

Type G_2

$\mathcal{T}(\infty)$ is the set of marginally large semistandard tableaux with exactly 2 rows over the ordered alphabet $\{1 \prec 2 \prec 3 \prec 0 \prec \overline{3} \prec \overline{2} \prec \overline{1}\}$ and subject to the following constraints:

- the contents of the boxes in the first row are $\preceq \overline{i}$,
- the contents of the boxes in the second row are $\preceq 3$,
- the entry 0 can appear at most once in the first row and not at all in the second row.

In particular, the shape of the tableaux is not fixed in any instance of $\mathcal{T}(\infty)$; the row lengths of a tableau can be arbitrarily long.

REFERENCES:

INPUT:

- `cartan_type` – One of `['A',n]`, `['B',n]`, `['C',n]`, `['D',n]`, or `['G',2]`, where `n` is a positive integer

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['A',2])
sage: b = B.highest_weight_vector(); b.pp()
1  1
2
sage: b.f_string([2,1,1,2,2,2]).pp()
1  1  1  1  1  2  3
```

```
2  3  3  3

sage: B = crystals.infinity.Tableaux(['G',2])
sage: b = B(rows=[[1,1,1,1,1,2,3,3,0,-3,-1,-1,-1],[2,3,3,3]])
sage: b.e_string([2,1,1,1,1,1,1]).pp()
1  1  1  1  2  3  3  3  3 -2 -2 -2
2  3  3
sage: b.e_string([2,1,1,1,1,1,1,1])
```

We check that a few classical crystals embed into $\mathcal{T}(\infty)$:

```
sage: def crystal_test(B, C):
....:     T = crystals.elementary.T(C.cartan_type(), C.module_generators[0].weight())
....:     TP = crystals.TensorProduct(T, B)
....:     mg = TP(T[0], B.module_generators[0])
....:     g = {C.module_generators[0]: mg}
....:     f = C.crystal_morphism(g, category=HighestWeightCrystals())
....:     G = B.digraph(subset=[f(x) for x in C])
....:     return G.is_isomorphic(C.digraph(), edge_labels=True)
sage: B = crystals.infinity.Tableaux(['A',2])
sage: C = crystals.Tableaux(['A',2], shape=[2,1])
sage: crystal_test(B, C)
True
sage: C = crystals.Tableaux(['A',2], shape=[6,2])
sage: crystal_test(B, C)
True
sage: B = crystals.infinity.Tableaux(['B',2])
sage: C = crystals.Tableaux(['B',2], shape=[3])
sage: crystal_test(B, C)
True
sage: C = crystals.Tableaux(['B',2], shape=[2,1])
sage: crystal_test(B, C)
True
sage: B = crystals.infinity.Tableaux(['C',3])
sage: C = crystals.Tableaux(['C',3], shape=[2,1])
sage: crystal_test(B, C)
True
sage: B = crystals.infinity.Tableaux(['D',4])
sage: C = crystals.Tableaux(['D',4], shape=[2])
sage: crystal_test(B, C)
True
sage: C = crystals.Tableaux(['D',4], shape=[1,1,1,1])
sage: crystal_test(B, C)
True
sage: B = crystals.infinity.Tableaux(['G',2])
sage: C = crystals.Tableaux(['G',2], shape=[3])
sage: crystal_test(B, C)
True
```

class **Element** (*parent*, **args*, ***options*)

Bases: sage.combinat.crystals.tensor_product.CrystalOfTableauxElement

Elements in $\mathcal{B}(\infty)$ crystal of tableaux.

content ()

Return the content of `self`.

The content $|T|$ of $T \in \mathcal{B}(\infty)$ is the number of blocks added to the highest weight to obtain T with any $\bar{\imath}$-boxes in the i-th row counted with multiplicity 2 provided the underlying Cartan type is of type

B, D, or G.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("D5")
sage: b = B.highest_weight_vector().f_string([5,4,3,1,1,3,4,5,3,4,5,1,4,5,2,3,5,3,2,4])
sage: b.content()
13

sage: B = crystals.infinity.Tableaux("B2")
sage: b = B(rows=[[1,1,1,1,1,1,2,2,2,-2,-2],[2,0,-2,-2,-2]])
sage: b.content()
12

sage: B = crystals.infinity.Tableaux("C2")
sage: b = B(rows=[[1,1,1,1,1,1,2,2,2,-2,-2],[2,-2,-2,-2]])
sage: b.content()
8
```

e(i)

Return the action of $\widetilde{e}_i$ on `self`.

INPUT:

- `i` – An element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['B',3])
sage: b = B(rows=[[1,1,1,1,1,1,1,2,0,-3,-1,-1,-1,-1],[2,2,2,2,-2,-2],[3,-3,-3]])
sage: b.e(3).pp()
1  1  1  1  1  1  1  2  0 -3 -1 -1 -1 -1
2  2  2  2 -2 -2
3  0 -3
sage: b.e(1).pp()
1  1  1  1  1  1  1  0 -3 -1 -1 -1 -1
2  2  2  2 -2 -2
3 -3 -3
```

f(i)

Return the action of $\widetilde{f}_i$ on `self`.

INPUT:

- `i` – An element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['C',4])
sage: b = B.highest_weight_vector()
sage: b.f(1).pp()
1  1  1  1  2
2  2  2
3  3
4
sage: b.f(3).pp()
1  1  1  1  1
2  2  2  2
3  3  4
4
sage: b.f(3).f(4).pp()
1  1  1  1  1
2  2  2  2
3  3 -4
4
```

phi(*i*)

Return φ_i of `self`.

Let $T \in \mathcal{B}(\infty)$ Define $\varphi_i(T) := \varepsilon_i(T) + \langle h_i, \mathrm{wt}(T) \rangle$, where h_i is the i-th simple coroot and $\mathrm{wt}(T)$ is the `weight()` of T.

INPUT:

- `i` – An element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("A3")
sage: [B.highest_weight_vector().f_string([1,3,2,3,1,3,2,1]).phi(i) for i in B.index_set
[-3, 4, -3]

sage: B = crystals.infinity.Tableaux("G2")
sage: [B.highest_weight_vector().f_string([2,2,1,2,1,1,1,2]).phi(i) for i in B.index_set
[5, -3]
```

reduced_form()

Return the reduced form of `self`.

The reduced form of a tableaux $T \in \mathcal{T}(\infty)$ is the (not necessarily semistandard) tableaux obtained from T by removing all i-boxes in the i-th row, subject to the condition that if the row is empty, a $*$ is put as a placeholder. This is described in [BN10] and [LS12].

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['A',3])
sage: b = B.highest_weight_vector().f_string([2,2,2,3,3,3,3,3])
sage: b.pp()
1  1  1  1  1  1  1  1
2  2  2  2  4  4  4
3  4  4
sage: b.reduced_form()
[['*'], [4, 4, 4], [4, 4]]
```

seg()

Returns the statistic seg of `self`.

More precisely, following [LS12], define a k-segment of a tableau T in $\mathcal{B}(\infty)$ to be a maximal string of k-boxes in a single row of T. Set $\mathrm{seg}'(T)$ to be the number of k-segments in T, as k varies over all possible values. Then $\mathrm{seg}(T)$ is determined type-by-type.

- In types A_n and C_n, define $\mathrm{seg}(T) := \mathrm{seg}'(T)$.
- In types B_n and G_2, set $e(T)$ to be the number of rows in T which contain both a 0-box and an $\bar{\imath}$-box. Define $\mathrm{seg}(T) := \mathrm{seg}'(T) - e(T)$.
- In type D_n, set $d(T)$ to be the number of rows in T which contain an $\bar{\imath}$-box, but no n-box nor $\overline{n}$-box. Define $\mathrm{seg}(T) := \mathrm{seg}'(T) + d(T)$.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['A',3])
sage: b = B.highest_weight_vector().f_string([1,3,2,2,3,1,1,3])
sage: b.pp()
1  1  1  1  1  1  2  2  4
2  2  2  2  3
3  4  4
sage: b.seg()
4

sage: B = crystals.infinity.Tableaux(['D',4])
sage: b = B(rows=[[1,1,1,1,1,1,3,-2,-1],[2,2,2,4,-2],[3,3],[4]])
sage: b.pp()
1  1  1  1  1  1  3 -2 -1
```

```
2  2  2  4 -2
3  3
4
sage: b.seg()
6

sage: B = crystals.infinity.Tableaux(['G',2])
sage: b = B.highest_weight_vector().f_string([2,1,1,1,2,1,2,2,1,2,2,2,1,2,2,1])
sage: b.pp()
1  1  1  1  1  1  1  1  2  3  0 -3
2  3  3  3  3  3  3
sage: b.seg()
5
```

string_parameters(*word*)

Return the string parametrization of `self` with respect to `word`.

For w in the Weyl group, let $R(w)$ denote the set of reduced expressions for w; that is, if $w = s_{i_1} s_{i_2} \cdots s_{i_\ell}$ is a reduced expression, then $(i_1, i_2, \ldots, i_\ell) \in R(w)$. For brevity, such words are denoted by $\mathbf{i}$.

For $T \in \mathcal{T}(\infty)$ and $\mathbf{i} = (i_1, \ldots, i_\ell) \in R(w)$, the string parametrization $(a_1, \ldots, a_\ell)$ of T in the direction $\mathbf{i}$ is defined recursively by $a_1 = \varepsilon_{i_1}(T)$ and $a_j = \varepsilon_{i_j}(\widetilde{e}_{i_{j-1}}^{a_{j-1}} \cdots \widetilde{e}_{i_1}^{a_1} T)$ for $j = 2, \ldots, \ell$. If $w = w_0$ is the longest element of the Weyl group, then the path determined by the string parametrization terminates at the highest weight vector.

INPUT:

- `word` – A word in the alphabet of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("A5")
sage: b = B(rows=[[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,3,6,6,6,6,6,6],[2,2,2,2,2,2,2,2,2,4,5,5
sage: b.string_parameters([1,2,1,3,2,1,4,3,2,1,5,4,3,2,1])
[0, 1, 1, 1, 1, 0, 4, 4, 3, 0, 11, 10, 7, 7, 6]

sage: B = crystals.infinity.Tableaux("G2")
sage: b = B(rows=[[1,1,1,1,1,3,3,0,-3,-3,-2,-2,-1,-1,-1,-1],[2,3,3,3]])
sage: b.string_parameters([2,1,2,1,2,1])
[5, 13, 11, 15, 4, 4]
sage: b.string_parameters([1,2,1,2,1,2])
[7, 12, 15, 8, 10, 0]
```

weight()

Return the weight of `self`.

From the definition of a crystal and that the highest weight element b_∞ of $\mathcal{B}(\infty)$ is 0, the weight of $T \in \mathcal{B}(\infty)$ can be defined as $\mathrm{wt}(T) := -\sum_j \alpha_{i_j}$ where $\widetilde{e}_{i_1} \cdots \widetilde{e}_{i_\ell} T = b_\infty$ and $\{\alpha_i\}$ is the set of simple roots. (Note that the weight is independent of the path chosen to get to the highest weight.)

However we can also take advantage of the fact that $\rho\colon R_\lambda \otimes \mathcal{B}(\infty) \longrightarrow B(\lambda)$, where λ is the shape of T, preserves the tableau representation of T. Therefore

$$\mathrm{wt}(T) = \mathrm{wt}\big(\rho(T)\big) - \lambda$$

where $\mathrm{wt}\big(\rho(T)\big)$ is just the usual weight of the tableau T.

Let Λ_i be the i-th fundamental weight. In type D, the height $n-1$ columns corresponds to $\Lambda_{n-1} + \Lambda_n$ and the in type B, the height n columns corresponds to $2\Lambda_n$.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("C7")
sage: b = B.highest_weight_vector().f_string([1,6,4,7,4,2,4,6,2,4,6,7,1,2,4,7])
sage: b.weight()
(-2, -1, 3, -5, 5, -3, -3)
```

Check that the definitions agree:

```
sage: P = B.weight_lattice_realization()
sage: alpha = P.simple_roots()
sage: b.weight() == -2*alpha[1] - 3*alpha[2] - 5*alpha[4] - 3*alpha[6] - 3*alpha[7]
True
```

Check that it works for type B:

```
sage: B = crystals.infinity.Tableaux("B2")
sage: B.highest_weight_vector().weight()
(0, 0)
sage: b = B.highest_weight_vector().f_string([1,2,2,2,1,2])
sage: P = B.weight_lattice_realization()
sage: alpha = P.simple_roots()
sage: b.weight() == -2*alpha[1] - 4*alpha[2]
True
```

Check that it works for type D:

```
sage: B = crystals.infinity.Tableaux("D4")
sage: B.highest_weight_vector().weight()
(0, 0, 0, 0)
sage: b = B.highest_weight_vector().f_string([1,4,4,2,4,3,2,4,1,3,2,4])
sage: P = B.weight_lattice_realization()
sage: alpha = P.simple_roots()
sage: b.weight() == -2*alpha[1] - 3*alpha[2] - 2*alpha[3] - 5*alpha[4]
True
```

InfinityCrystalOfTableaux.**module_generator**()

Return the module generator (or highest weight element) of `self`.

The module generator is the unique tableau of shape $(n, n-1, \ldots, 2, 1)$ with weight 0.

EXAMPLES:

```
sage: T = crystals.infinity.Tableaux(['A',3])
sage: T.module_generator()
[[1, 1, 1], [2, 2], [3]]
```

class sage.combinat.crystals.infinity_crystals.**InfinityCrystalOfTableauxTypeD**(*cartan_type*)

Bases: sage.combinat.crystals.infinity_crystals.InfinityCrystalOfTableaux

$\mathcal{B}(\infty)$ crystal of tableaux for type D_n.

This is the set $\mathcal{T}(\infty)$ of marginally large semistandard tableaux with exactly $n-1$ rows over the alphabet $\{1 \prec \cdots \prec n, \overline{n} \prec \cdots \prec \overline{1}\}$ and subject to the following constraints:

- for each $1 \leq i \leq n$, the contents of the boxes in the i-th row are $\preceq \overline{i}$,
- the entries n and $\overline{n}$ may not appear simultaneously in a single row.

For more information, see InfinityCrystalOfTableaux.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("D4")
sage: b = B.highest_weight_vector().f_string([4,3,2,1,4])
sage: b.pp()
1  1  1  1  1  1  2
```

```
2  2  2  2  3
3 -4 -3
sage: b.weight()
(-1, 0, -2, -1)
```

class **Element**(*parent*, **args*, ***options*)

Bases: sage.combinat.crystals.infinity_crystals.InfinityCrystalOfTableaux.Element

Elements in $\mathcal{B}(\infty)$ crystal of tableaux for type D_n.

e(*i*)

Return the action of $\widetilde{e}_i$ on `self`.

INPUT:

- `i` – An element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['D',4])
sage: b = B.highest_weight_vector().f_string([1,4,3,1,2]); b.pp()
1  1  1  1  2  3
2  2  2
3 -3
sage: b.e(2).pp()
1  1  1  1  2  2
2  2  2
3 -3
```

f(*i*)

Return the action of $\widetilde{f}_i$ on `self`.

INPUT:

- `i` – An element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['D',5])
sage: b = B.highest_weight_vector().f_string([1,4,3,1,5]); b.pp()
1  1  1  1  1  1  2  2
2  2  2  2  2
3  3  3 -5
4  5
sage: b.f(1).pp()
1  1  1  1  1  1  2  2  2
2  2  2  2  2
3  3  3 -5
4  5
sage: b.f(5).pp()
1  1  1  1  1  1  2  2
2  2  2  2  2
3  3  3 -5
4 -4
```

InfinityCrystalOfTableauxTypeD.**module_generator**()

Return the module generator (or highest weight element) of `self`.

The module generator is the unique tableau of shape $(n-1,\ldots,2,1)$ with weight 0.

EXAMPLES:

```
sage: T = crystals.infinity.Tableaux(['D',4])
sage: T.module_generator()
[[1, 1, 1], [2, 2], [3]]
```

5.1.50 Polyhedral Realization of $B(\infty)$

class sage.combinat.crystals.polyhedral_realization.**InfinityCrystalAsPolyhedralRealization**(*ca*
se

Bases: sage.combinat.crystals.tensor_product.TensorProductOfCrystals

The polyhedral realization of $B(\infty)$.

Note: Here we are using anti-Kashiwara notation and might differ from some of the literature.

Consider a Kac-Moody algebra $\mathfrak{g}$ of Cartan type X with index set I, and consider a finite sequence $J = (j_1, j_2, \ldots, j_m)$ whose support equals I. We extend this to an infinite sequence by taking $\bar{J} = J \cdot J \cdot J \cdots$, where $\cdot$ denotes concatenation of sequences. Let

$$B_J = B_{j_m} \otimes \cdots \otimes B_{j_2} \otimes B_{j_1},$$

where B_i is an ElementaryCrystal.

As given in Theorem 2.1.1 of [K93], there exists a strict crystal embedding $\Psi_i \colon B(\infty) \to B_i \otimes B(\infty)$ defined by $u_\infty \mapsto b_i(0) \otimes u_\infty$, where $b_i(0) \in B_i$ and u_∞ is the (unique) highest weight element in $B(\infty)$. This is sometimes known as the *Kashiwara embedding* [NZ97] (though, in [NZ97], the target of this map is denoted by $\mathbf{Z}_J^\infty$). By iterating this embedding by taking $\Psi_J = \Psi_{j_n} \circ \Psi_{j_{n-1}} \circ \cdots \circ \Psi_{j_1}$, we obtain the following strict crystal embedding:

$$\Psi_J^n \colon B(\infty) \to B_J^{\otimes n} \otimes B(\infty).$$

We note there is a natural analog of Lemma 10.6.2 in [HK02] that for any $b \in B(\infty)$, there exists a positive integer N such that

$$\Psi_J^N(b) = \left(\bigotimes_{k=1}^{N} b^{(k)} \right) \otimes u_\infty.$$

Therefore we can model elements $b \in B(\infty)$ by considering an infinite list of elements $b^{(k)} \in B_J$ and defining the crystal structure by:

$$\begin{aligned}
\mathrm{wt}(b) &= \sum_{k=1}^{N} \mathrm{wt}(b^{(k)}) \\
e_i(b) &= e_i\left(\left(\bigotimes_{k=1}^{N} b^{(k)} \right) \right) \otimes u_\infty, \\
f_i(b) &= f_i\left(\left(\bigotimes_{k=1}^{N} b^{(k)} \right) \right) \otimes u_\infty, \\
\varepsilon_i(b) &= \max_{e_i^k(b) \neq 0} k, \\
\varphi_i(b) &= \varepsilon_i(b) - \langle \mathrm{wt}(b), h_i^\vee \rangle.
\end{aligned}$$

To translate this into a finite list, we consider a finite sequence $b_1 \otimes \cdots \otimes b_N$ and if

$$f_i\left(b^{(1)} \otimes \cdots b^{(N-1)} \otimes b^{(N)} \right) = b^{(1)} \otimes \cdots \otimes b^{(N-1)} \otimes f_i\left(b^{(N)} \right),$$

then we take the image as $b^{(1)} \otimes \cdots \otimes f_i\left(b^{(N)}\right) \otimes b^{(N+1)}$. Similarly we remove $b^{(N)}$ if we have $b^{(N)} = \bigotimes_{k=1}^{m} b_{j_k}(0)$. Additionally if

$$e_i\left(b^{(1)} \otimes \cdots \otimes b^{(N-1)} \otimes b^{(N)} \right) = b^{(1)} \otimes \cdots \otimes b^{(N-1)} \otimes e_i\left(b^{(N)} \right),$$

then we consider this to be 0.

REFERENCES:

INPUT:

- `cartan_type` – a Cartan type
- `seq` – (default: `None`) a finite sequence whose support equals the index set of the Cartan type; if `None`, then this is the index set

EXAMPLES:

```
sage: B = crystals.infinity.PolyhedralRealization(['A',2])
sage: mg = B.module_generators[0]; mg
[0, 0]
sage: mg.f_string([2,1,2,2])
[0, -3, -1, 0, 0, 0]
```

An example of type B_2:

```
sage: B = crystals.infinity.PolyhedralRealization(['B',2])
sage: mg = B.module_generators[0]; mg
[0, 0]
sage: mg.f_string([2,1,2,2])
[0, -2, -1, -1, 0, 0]
```

An example of type G_2:

```
sage: B = crystals.infinity.PolyhedralRealization(['G',2])
sage: mg = B.module_generators[0]; mg
[0, 0]
sage: mg.f_string([2,1,2,2])
[0, -3, -1, 0, 0, 0]
```

class **Element** (*parent*, **args*, ***kwds*)

Bases: `sage.combinat.crystals.tensor_product.TensorProductOfCrystalsElement`

An element in the polyhedral realization of $B(\infty)$.

e (i)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PolyhedralRealization(['A',2])
sage: mg = B.module_generators[0]
sage: all(mg.e(i) is None for i in B.index_set())
True
sage: mg.f(1).e(1) == mg
True
```

epsilon (i)

Return ε_i of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PolyhedralRealization(['A',2,1])
sage: mg = B.module_generators[0]
sage: [mg.epsilon(i) for i in B.index_set()]
[0, 0, 0]
sage: elt = mg.f(0)
sage: [elt.epsilon(i) for i in B.index_set()]
[1, 0, 0]
```

```
sage: elt = mg.f_string([0,1,2])
sage: [elt.epsilon(i) for i in B.index_set()]
[0, 0, 1]
sage: elt = mg.f_string([0,1,2,2])
sage: [elt.epsilon(i) for i in B.index_set()]
[0, 0, 2]
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PolyhedralRealization(['A',2])
sage: mg = B.module_generators[0]
sage: mg.f(1)
[-1, 0, 0, 0]
sage: mg.f_string([1,2,2,1])
[-1, -2, -1, 0, 0, 0]
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PolyhedralRealization(['A',2,1])
sage: mg = B.module_generators[0]
sage: [mg.phi(i) for i in B.index_set()]
[0, 0, 0]
sage: elt = mg.f(0)
sage: [elt.phi(i) for i in B.index_set()]
[-1, 1, 1]
sage: elt = mg.f_string([0,1])
sage: [elt.phi(i) for i in B.index_set()]
[-1, 0, 2]
sage: elt = mg.f_string([0,1,2,2])
sage: [elt.phi(i) for i in B.index_set()]
[1, 1, 0]
```

truncate(*k=None*)

Truncate `self` to have length `k` and return as an element in a (finite) tensor product of crystals.

INPUT:

- `k` – (optional) the length of the truncation; if not specified, then returns one more than the current non-ground-state elements (i.e. the current list in `self`)

EXAMPLES:

```
sage: B = crystals.infinity.PolyhedralRealization(['A',2])
sage: mg = B.module_generators[0]
sage: elt = mg.f_string([1,2,2,1]); elt
[-1, -2, -1, 0, 0, 0]
sage: t = elt.truncate(); t
[-1, -2, -1, 0, 0, 0]
sage: t.parent() is B.finite_tensor_product(6)
True
sage: elt.truncate(2)
[-1, -2]
sage: elt.truncate(10)
[-1, -2, -1, 0, 0, 0, 0, 0, 0, 0]
```

InfinityCrystalAsPolyhedralRealization.**finite_tensor_product**(*k*)

Return the finite tensor product of crystals of length `k` by truncating `self`.

EXAMPLES:

```
sage: B = crystals.infinity.PolyhedralRealization(['A',2])
sage: B.finite_tensor_product(5)
Full tensor product of the crystals
 [The 1-elementary crystal of type ['A', 2],
  The 2-elementary crystal of type ['A', 2],
  The 1-elementary crystal of type ['A', 2],
  The 2-elementary crystal of type ['A', 2],
  The 1-elementary crystal of type ['A', 2]]
```

5.1.51 Kirillov-Reshetikhin Crystals

class sage.combinat.crystals.kirillov_reshetikhin.**AmbientRetractMap**(*base*, *ambient*, *pdict_inv*, *index_set*, *similarity_factor_domain=None*, *automorphism=None*)

Bases: sage.categories.map.Map

The retraction map from the ambient crystal.

Consider a crystal embedding $\phi : X \to Y$, then the elements X can be considered as a subcrystal of the ambient crystal Y. The ambient retract is the partial map $\tilde{\phi} : Y \to X$ such that $\tilde{\phi} \circ \phi$ is the identity on X.

class sage.combinat.crystals.kirillov_reshetikhin.**CrystalDiagramAutomorphism**(*C*, *on_hw*, *index_set=None*, *automorphism=None*, *cache=True*)

Bases: sage.categories.crystals.CrystalMorphism

The crystal automorphism induced from the diagram automorphism.

For example, in type $A_n^{(1)}$ this is the promotion operator and in type $D_n^{(1)}$, this corresponds to the automorphism induced from interchanging the 0 and 1 nodes in the Dynkin diagram.

INPUT:

- C – a crystal
- on_hw – a function for the images of the index_set-highest weight elements
- index_set – (default: the empty set) the index set
- automorphism – (default: the identity) the twisting automorphism
- cache – (default: True) cache the result

is_embedding()

Return True as self is a crystal isomorphism.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.promotion().is_isomorphism()
True
```

is_isomorphism()
Return `True` as `self` is a crystal isomorphism.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.promotion().is_isomorphism()
True
```

is_strict()
Return `True` as `self` is a crystal isomorphism.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.promotion().is_isomorphism()
True
```

is_surjective()
Return `True` as `self` is a crystal isomorphism.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.promotion().is_isomorphism()
True
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_A**(*cartan_type*, *r*, *s*)
Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinCrystalFromPromoti

Class of Kirillov-Reshetikhin crystals of type $A_n^{(1)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: b = K(rows=[[1,2],[2,4]])
sage: b.f(0)
[[1, 1], [2, 2]]
```

classical_decomposition()
Specifies the classical crystal underlying the KR crystal of type A.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['A', 3] and shape(s) [[2, 2]]
```

dynkin_diagram_automorphism(*i*)
Specifies the Dynkin diagram automorphism underlying the promotion action on the crystal elements. The automorphism needs to map node 0 to some other Dynkin node.

For type A we use the Dynkin diagram automorphism which $i \mapsto i+1 \mod n+1$, where n is the rank.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.dynkin_diagram_automorphism(0)
1
```

```
sage: K.dynkin_diagram_automorphism(3)
0
```

promotion()

Specifies the promotion operator used to construct the affine type A crystal.

For type A this corresponds to the Dynkin diagram automorphism which $i \mapsto i+1 \mod n+1$, where n is the rank.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: b = K.classical_decomposition()(rows=[[1,2],[3,4]])
sage: K.promotion()(b)
[[1, 3], [2, 4]]
```

promotion_inverse()

Specifies the inverse promotion operator used to construct the affine type A crystal.

For type A this corresponds to the Dynkin diagram automorphism which $i \mapsto i-1 \mod n+1$, where n is the rank.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: b = K.classical_decomposition()(rows=[[1,3],[2,4]])
sage: K.promotion_inverse()(b)
[[1, 2], [3, 4]]
sage: b = K.classical_decomposition()(rows=[[1,2],[3,3]])
sage: K.promotion_inverse()(K.promotion()(b))
[[1, 2], [3, 3]]
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_A2**(*cartan_type*, *r*, *s*, *dual=None*)

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystal

Class of Kirillov-Reshetikhin crystals $B^{r,s}$ of type $A_{2n}^{(2)}$ for $1 \leq r \leq n$ in the realization with classical subalgebra B_n. The Cartan type in this case is inputted as the dual of $A_{2n}^{(2)}$.

This is an alternative implementation to KR_type_box which uses the classical decomposition into type C_n crystals.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 1)
sage: K
Kirillov-Reshetikhin crystal of type ['BC', 2, 2]^* with (r,s)=(1,1)
sage: b = K(rows=[[-1]])
sage: b.f(0)
[[1]]
sage: b.e(0)
```

We can now check whether the two KR crystals of type $A_4^{(2)}$ (namely the KR crystal and its dual construction) are isomorphic up to relabelling of the edges:

```
sage: C = CartanType(['A',4,2])
sage: K = crystals.KirillovReshetikhin(C,1,1)
sage: Kdual = crystals.KirillovReshetikhin(C.dual(),1,1)
sage: G = K.digraph()
sage: Gdual = Kdual.digraph()
```

```
sage: f = {0:2, 1:1, 2:0}
sage: Gnew = DiGraph(); Gnew.add_vertices(Gdual.vertices()); Gnew.add_edges([(u,v,f[i]) for (u,v
sage: G.is_isomorphic(Gnew, edge_labels = True)
True
```

Element
alias of KR_type_A2Element

ambient_crystal()
Returns the ambient crystal $B^{r,s}$ of type $B^{(1)}_{n+1}$ associated to the Kirillov-Reshetikhin crystal of type $A^{(2)}_{2n}$ dual. This ambient crystal is used to construct the zero arrows.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 2, 3)
sage: K.ambient_crystal()
Kirillov-Reshetikhin crystal of type ['B', 3, 1] with (r,s)=(2,3)
```

ambient_dict_pm_diagrams()
Gives a dictionary of all self-dual $\pm$ diagrams for the ambient crystal. Their key is their inner shape.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 1)
sage: K.ambient_dict_pm_diagrams()
{[1]: [[0, 0], [1]]}
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 2)
sage: K.ambient_dict_pm_diagrams()
{[]: [[1, 1], [0]], [2]: [[0, 0], [2]]}
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 2, 2)
sage: K.ambient_dict_pm_diagrams()
{[]: [[1, 1], [0, 0], [0]],
 [2]: [[0, 0], [1, 1], [0]],
 [2, 2]: [[0, 0], [0, 0], [2]]}
```

ambient_highest_weight_dict()
Gives a dictionary of all $2, ..., n+1$-highest weight vectors in the ambient crystal. Their key is the inner shape of their corresponding $\pm$ diagram, or equivalently, their $2, ..., n+1$ weight.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 2)
sage: K.ambient_highest_weight_dict()
{[]: [[1, -1]], [2]: [[2, 2]]}
```

classical_decomposition()
Specifies the classical crystal underlying the Kirillov-Reshetikhin crystal of type $A^{(2)}_{2n}$ with B_n as classical subdiagram.

It is given by $B^{r,s} \cong \bigoplus_\Lambda B(\Lambda)$ where $B(\Lambda)$ is a highest weight crystal of type B_n of highest weight Λ. The sum is over all weights Λ obtained from a rectangle of width s and height r by removing horizontal dominoes. Here we identify the fundamental weight Λ_i with a column of height i.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 2, 2)
```

```
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 2] and shape(s) [[], [2], [2, 2]]
```

from_ambient_crystal()

Provides a map from the ambient crystal of type $B^{(1)}_{n+1}$ to the Kirillov-Reshetikhin crystal of type $A^{(2)}_{2n}$.

Note that this map is only well-defined on type $A^{(2)}_{2n}$ elements that are in the image under to_ambient_crystal().

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 2)
sage: b = K.ambient_crystal()(rows=[[2,2]])
sage: K.from_ambient_crystal()(b)
[[1, 1]]
```

highest_weight_dict()

Gives a dictionary of the classical highest weight vectors of self. Their key is their shape.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 2)
sage: K.highest_weight_dict()
{[]: [], [2]: [[1, 1]]}
```

to_ambient_crystal()

Provides a map from the Kirillov-Reshetikhin crystal of type $A^{(2)}_{2n}$ to the ambient crystal of type $B^{(1)}_{n+1}$.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 2)
sage: b=K(rows=[[1,1]])
sage: K.to_ambient_crystal()(b)
[[2, 2]]
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 2, 2)
sage: b=K(rows=[[1,1]])
sage: K.to_ambient_crystal()(b)
[[1, 2], [2, -1]]
sage: K.to_ambient_crystal()(b).parent()
Kirillov-Reshetikhin crystal of type ['B', 3, 1] with (r,s)=(2,2)
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_A2Element**

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystalEleme

Class for the elements in the Kirillov-Reshetikhin crystals $B^{r,s}$ of type $A^{(2)}_{2n}$ for $r < n$ with underlying classcial algebra B_n.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 2)
sage: type(K.module_generators[0])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2_with_category.element_class'>
```

e0()

Gives e_0 on self by mapping self to the ambient crystal, calculating $e_1 e_0$ there and pulling the element back.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 1)
sage: b = K(rows=[[1]])
sage: b.e(0) # indirect doctest
[[-1]]
```

epsilon0()

Calculate ε_0 of `self` by mapping the element to the ambient crystal and calculating `\varepsilon_1` there.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 1)
sage: b=K(rows=[[1]])
sage: b.epsilon(0) # indirect doctest
1
```

f0()

Gives f_0 on self by mapping self to the ambient crystal, calculating $f_1 f_0$ there and pulling the element back.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 1)
sage: b = K(rows=[[-1]])
sage: b.f(0) # indirect doctest
[[1]]
```

phi0()

Calculate φ_0 of `self` by mapping the element to the ambient crystal and calculating φ_1 there.

EXAMPLES:

```
sage: C = CartanType(['A',4,2]).dual()
sage: K = sage.combinat.crystals.kirillov_reshetikhin.KR_type_A2(C, 1, 1)
sage: b=K(rows=[[-1]])
sage: b.phi(0) # indirect doctest
1
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_Bn**(*cartan_type*, *r*, *s*, *dual=None*)

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystal

Class of Kirillov-Reshetikhin crystals $B^{n,s}$ of type $B_n^{(1)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,2)
sage: K
Kirillov-Reshetikhin crystal of type ['B', 3, 1] with (r,s)=(3,2)
sage: b = K(rows=[[1],[2],[3]])
sage: b.f(0)
sage: b.e(0)
[[3]]

sage: K = crystals.KirillovReshetikhin(['B',3,1],3,2)
sage: [b.weight() for b in K if b.is_highest_weight([1,2,3])]
[-Lambda[0] + Lambda[1], -2*Lambda[0] + 2*Lambda[3]]
```

```
sage: [b.weight() for b in K if b.is_highest_weight([0,2,3])]
[Lambda[0] - Lambda[1], -2*Lambda[1] + 2*Lambda[3]]
```

Element
alias of KR_type_BnElement

ambient_crystal()
Returns the ambient crystal $B^{n,s}$ of type $A^{(2)}_{2n-1}$ associated to the Kirillov-Reshetikhin crystal; see Lemma 4.2 of reference [4]. This ambient crystal is used to construct the zero arrows.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,2)
sage: K.ambient_crystal()
Kirillov-Reshetikhin crystal of type ['B', 3, 1]^* with (r,s)=(3,2)
```

ambient_highest_weight_dict()
Gives a dictionary of the classical highest weight vectors of the ambient crystal of self. Their key is their shape.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,2)
sage: K.ambient_highest_weight_dict()
{(2,): [[1, 1]], (2, 1, 1): [[1, 1], [2], [3]], (2, 2, 2): [[1, 1], [2, 2], [3, 3]]}

sage: K = crystals.KirillovReshetikhin(['B',3,1],3,3)
sage: K.ambient_highest_weight_dict()
{(3,): [[1, 1, 1]],
 (3, 1, 1): [[1, 1, 1], [2], [3]],
 (3, 2, 2): [[1, 1, 1], [2, 2], [3, 3]],
 (3, 3, 3): [[1, 1, 1], [2, 2, 2], [3, 3, 3]]}
```

classical_decomposition()
Specifies the classical crystal underlying the Kirillov-Reshetikhin crystal $B^{n,s}$ of type $B^{(1)}_n$.

It is the same as for $r < n$, given by $B^{n,s} \cong \bigoplus_\Lambda B(\Lambda)$ where Λ are weights obtained from a rectangle of width $s/2$ and height n by removing horizontal dominoes. Here we identify the fundamental weight Λ_i with a column of height i for $i < n$ and a column of width $1/2$ for $i = n$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1], 3, 2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[1], [1, 1, 1]]
sage: K = crystals.KirillovReshetikhin(['B',3,1], 3, 3)
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[3/2, 1/2, 1/2], [3/2, 3/2, 3/2]]
```

from_ambient_crystal()
Provides a map from the ambient crystal of type $A^{(2)}_{2n-1}$ to the Kirillov-Reshetikhin crystal `self`.

Note that this map is only well-defined on elements that are in the image under to_ambient_crystal().

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,1)
sage: [b == K.from_ambient_crystal()(K.to_ambient_crystal()(b)) for b in K]
[True, True, True, True, True, True, True, True]
sage: b = K.ambient_crystal()(rows=[[1],[2],[-3]])
```

```
sage: K.from_ambient_crystal()(b)
[++-, []]
```

highest_weight_dict()

Gives a dictionary of the classical highest weight vectors of self. Their key is 2 times their shape.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,2)
sage: K.highest_weight_dict()
{(2,): [[1]], (2, 2, 2): [[1], [2], [3]]}
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,3)
sage: K.highest_weight_dict()
{(3, 1, 1): [+++, [[1]]], (3, 3, 3): [+++, [[1], [2], [3]]]}
```

similarity_factor()

Sets the similarity factor used to map to the ambient crystal.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,2)
sage: K.similarity_factor()
{1: 2, 2: 2, 3: 1}
```

to_ambient_crystal()

Provides a map from self to the ambient crystal of type $A_{2n-1}^{(2)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,1)
sage: [K.to_ambient_crystal()(b) for b in K]
[[[1], [2], [3]], [[1], [2], [-3]], [[1], [3], [-2]], [[2], [3], [-1]], [[1], [-3], [-2]],
[[2], [-3], [-1]], [[3], [-2], [-1]], [[-3], [-2], [-1]]]
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_BnElement**

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystalEleme

Class for the elements in the Kirillov-Reshetikhin crystals $B^{n,s}$ of type $B_n^{(1)}$.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['B',3,1],3,2)
sage: type(K.module_generators[0])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_Bn_with_category.element_class'>
```

e0()

Gives e_0 on self by mapping self to the ambient crystal, calculating e_0 there and pulling the element back.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['B',3,1],3,1)
sage: b = K.module_generators[0]
sage: b.e(0) # indirect doctest
[--+, []]
```

epsilon0()

Calculate ε_0 of `self` by mapping the element to the ambient crystal and calculating ε_0 there.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['B',3,1],3,1)
sage: b = K.module_generators[0]
```

```
sage: b.epsilon(0) # indirect doctest
1
```

f0()

Gives f_0 on self by mapping self to the ambient crystal, calculating f_0 there and pulling the element back.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['B',3,1],3,1)
sage: b = K.module_generators[0]
sage: b.f(0) # indirect doctest
```

phi0()

Calculate φ_0 of `self` by mapping the element to the ambient crystal and calculating φ_0 there.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['B',3,1],3,1)
sage: b = K.module_generators[0]
sage: b.phi(0) # indirect doctest
0
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_C**(*cartan_type*, *r*, *s*, *dual=None*)

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystal

Class of Kirillov-Reshetikhin crystals $B^{r,s}$ of type $C_n^{(1)}$ for $r < n$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1,2)
sage: K
Kirillov-Reshetikhin crystal of type ['C', 2, 1] with (r,s)=(1,2)
sage: b = K(rows=[])
sage: b.f(0)
[[1, 1]]
sage: b.e(0)
[[-1, -1]]
```

Element

alias of KR_type_CElement

ambient_crystal()

Returns the ambient crystal $B^{r,s}$ of type $A_{2n+1}^{(2)}$ associated to the Kirillov-Reshetikhin crystal of type $C_n^{(1)}$. This ambient crystal is used to construct the zero arrows.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,3)
sage: K.ambient_crystal()
Kirillov-Reshetikhin crystal of type ['B', 4, 1]^* with (r,s)=(2,3)
```

ambient_dict_pm_diagrams()

Gives a dictionary of all self-dual $\pm$ diagrams for the ambient crystal. Their key is their inner shape.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1,2)
sage: K.ambient_dict_pm_diagrams()
{[]: [[1, 1], [0]], [2]: [[0, 0], [2]]}
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,2)
sage: K.ambient_dict_pm_diagrams()
```

```
{[]: [[1, 1], [0, 0], [0]],
 [2]: [[0, 0], [1, 1], [0]],
 [2, 2]: [[0, 0], [0, 0], [2]]}
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,3)
sage: K.ambient_dict_pm_diagrams()
{[1, 1]: [[1, 1], [0, 0], [1]],
 [3, 1]: [[0, 0], [1, 1], [1]],
 [3, 3]: [[0, 0], [0, 0], [3]]}
```

ambient_highest_weight_dict()

Gives a dictionary of all $2, ..., n+1$-highest weight vectors in the ambient crystal. Their key is the inner shape of their corresponding $\pm$ diagram, or equivalently, their $2, ..., n+1$ weight.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,2)
sage: K.ambient_highest_weight_dict()
{[]: [[2], [-2]], [2]: [[1, 2], [2, -1]], [2, 2]: [[2, 2], [3, 3]]}
```

classical_decomposition()

Specifies the classical crystal underlying the Kirillov-Reshetikhin crystal of type $C_n^{(1)}$.

It is given by $B^{r,s} \cong \bigoplus_\Lambda B(\Lambda)$ where Λ are weights obtained from a rectangle of width s and height r by removing horizontal dominoes. Here we identify the fundamental weight Λ_i with a column of height i.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['C', 3] and shape(s) [[], [2], [2, 2]]
```

from_ambient_crystal()

Provides a map from the ambient crystal of type $A_{2n+1}^{(2)}$ to the Kirillov-Reshetikhin crystal of type $C_n^{(1)}$.

Note that this map is only well-defined on type $C_n^{(1)}$ elements that are in the image under to_ambient_crystal().

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,2)
sage: b = K.ambient_crystal()(rows=[[2,2],[3,3]])
sage: K.from_ambient_crystal()(b)
[[1, 1], [2, 2]]
```

highest_weight_dict()

Gives a dictionary of the classical highest weight vectors of self. Their key is their shape.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,2)
sage: K.highest_weight_dict()
{[]: [], [2]: [[1, 1]], [2, 2]: [[1, 1], [2, 2]]}
```

to_ambient_crystal()

Provides a map from the Kirillov-Reshetikhin crystal of type $C_n^{(1)}$ to the ambient crystal of type $A_{2n+1}^{(2)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1], 2,2)
sage: b=K(rows=[[1,1]])
sage: K.to_ambient_crystal()(b)
```

```
[[1, 2], [2, -1]]
sage: b=K(rows=[])
sage: K.to_ambient_crystal()(b)
[[2], [-2]]
sage: K.to_ambient_crystal()(b).parent()
Kirillov-Reshetikhin crystal of type ['B', 4, 1]^* with (r,s)=(2,2)
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_CElement**

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystalEleme

Class for the elements in the Kirillov-Reshetikhin crystals $B^{r,s}$ of type $C_n^{(1)}$ for $r < n$.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['C',3,1],1,2)
sage: type(K.module_generators[0])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_C_with_category.element_class'>
```

e0()

Gives e_0 on self by mapping self to the ambient crystal, calculating $e_1 e_0$ there and pulling the element back.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['C',3,1],1,2)
sage: b = K(rows=[])
sage: b.e(0) # indirect doctest
[[-1, -1]]
```

epsilon0()

Calculate ε_0 of `self` by mapping the element to the ambient crystal and calculating ε_1 there.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1,2)
sage: b=K(rows=[[1,1]])
sage: b.epsilon(0) # indirect doctest
2
```

f0()

Gives f_0 on self by mapping self to the ambient crystal, calculating $f_1 f_0$ there and pulling the element back.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['C',3,1],1,2)
sage: b = K(rows=[])
sage: b.f(0) # indirect doctest
[[1, 1]]
```

phi0()

Calculate φ_0 of `self` by mapping the element to the ambient crystal and calculating φ_1 there.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1,2)
sage: b=K(rows=[[-1,-1]])
sage: b.phi(0) # indirect doctest
2
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_Cn**(*cartan_type*, *r*, *s*, *dual=None*)

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystal

Class of Kirillov-Reshetikhin crystals $B^{n,s}$ of type $C_n^{(1)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1],3,1)
sage: [[b,b.f(0)] for b in K]
[[[[1], [2], [3]], None], [[[1], [2], [-3]], None], [[[1], [3], [-3]], None],
[[[2], [3], [-3]], None], [[[1], [3], [-2]], None], [[[2], [3], [-2]], None],
[[[2], [3], [-1]], [[1], [2], [3]]], [[[1], [-3], [-2]], None], [[[2], [-3], [-2]], None],
[[[2], [-3], [-1]], [[1], [2], [-3]]], [[[3], [-3], [-2]], None], [[[3], [-3], [-1]],
[[1], [3], [-3]]], [[[3], [-2], [-1]], [[1], [3], [-2]]], [[[-3], [-2], [-1]], [[1], [-3], [-2]]
```

Element

alias of KR_type_CnElement

classical_decomposition()

Specifies the classical crystal underlying the Kirillov-Reshetikhin crystal $B^{n,s}$ of type $C_n^{(1)}$. It is given by $B^{n,s} \cong B(s\Lambda_n)$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1],3,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['C', 3] and shape(s) [[2, 2, 2]]
```

from_highest_weight_vector_to_pm_diagram(*b*)

This gives the bijection between an element b in the classical decomposition of the KR crystal that is $2, 3, .., n$-highest weight and $\pm$ diagrams.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1],3,2)
sage: T = K.classical_decomposition()
sage: b = T(rows=[[2, 2], [3, 3], [-3, -1]])
sage: pm = K.from_highest_weight_vector_to_pm_diagram(b); pm
[[0, 0], [1, 0], [0, 1], [0]]
sage: pm.pp()
.  .
.  +
-  -

sage: hw = [ b for b in T if all(b.epsilon(i)==0 for i in [2,3]) ]
sage: all(K.from_pm_diagram_to_highest_weight_vector(K.from_highest_weight_vector_to_pm_diag
True
```

from_pm_diagram_to_highest_weight_vector(*pm*)

This gives the bijection between a $\pm$ diagram and an element b in the classical decomposition of the KR crystal that is {2,3,..,n}-highest weight.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1],3,2)
sage: pm = sage.combinat.crystals.kirillov_reshetikhin.PMDiagram([[0, 0], [1, 0], [0, 1], [0
sage: K.from_pm_diagram_to_highest_weight_vector(pm)
[[2, 2], [3, 3], [-3, -1]]
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_CnElement**

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystalEleme

Class for the elements in the Kirillov-Reshetikhin crystals $B^{n,s}$ of type $C_n^{(1)}$.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['C',3,1],3,2)
sage: type(K.module_generators[0])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_Cn_with_category.element_class'>
```

e0()

Gives e_0 on self by going to the $\pm$-diagram corresponding to the $2, ..., n$-highest weight vector in the component of $self$, then applying [Definition 6.1, 4], and pulling back from $\pm$-diagrams.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['C',3,1],3,2)
sage: b = K.module_generators[0]
sage: b.e(0) # indirect doctest
[[1, 2], [2, 3], [3, -1]]
sage: b = K(rows=[[1,2],[2,3],[3,-1]])
sage: b.e(0)
[[2, 2], [3, 3], [-1, -1]]
sage: b=K(rows=[[1, -3], [3, -2], [-3, -1]])
sage: b.e(0)
[[3, -3], [-3, -2], [-1, -1]]
```

epsilon0()

Calculate ε_0 of `self` using Lemma 6.1 of [4].

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['C',3,1],3,1)
sage: b = K.module_generators[0]
sage: b.epsilon(0) # indirect doctest
1
```

f0()

Gives e_0 on self by going to the $\pm$-diagram corresponding to the $2, ..., n$-highest weight vector in the component of $self$, then applying [Definition 6.1, 4], and pulling back from $\pm$-diagrams.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['C',3,1],3,1)
sage: b = K.module_generators[0]
sage: b.f(0) # indirect doctest
```

phi0()

Calculate φ_0 of `self`.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['C',3,1],3,1)
sage: b = K.module_generators[0]
sage: b.phi(0) # indirect doctest
0
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_D_tri1**(*ct*, *s*)

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystal

Class of Kirillov-Reshetikhin crystals $B^{1,s}$ of type $D_4^{(3)}$.

The crystal structure was defined in Section 4 of [KMOY07] using the coordinate representation.

REFERENCES:

class **Element**

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystalE

coordinates()

Return `self` as coordinates.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1, 3)
sage: all(K.from_coordinates(x.coordinates()) == x for x in K)
True
```

e0()

Return the action of e_0 on `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1,1)
sage: [x.e0() for x in K]
[[[-1]], [], [[-3]], [[-2]], None, None, None, None]
```

epsilon0()

Return ε_0 of `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1, 5)
sage: [mg.epsilon0() for mg in K.module_generators]
[5, 6, 7, 8, 9, 10]
```

f0()

Return the action of f_0 on `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1,1)
sage: [x.f0() for x in K]
[[[1]], None, None, None, None, [[2]], [[3]], []]
```

phi0()

Return φ_0 of `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1, 5)
sage: [mg.phi0() for mg in K.module_generators]
[5, 4, 3, 2, 1, 0]
```

KR_type_D_tri1.**classical_decomposition**()

Return the classical decomposition of `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1, 5)
sage: K.classical_decomposition()
The crystal of tableaux of type ['G', 2]
 and shape(s) [[], [1], [2], [3], [4], [5]]
```

KR_type_D_tri1.**from_coordinates**(*coords*)

Return an element of `self` from the coordinates `coords`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1, 5)
sage: K.from_coordinates((0, 2, 3, 1, 0, 1))
[[2, 2, 3, 0, -1]]
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_Dn_twisted**(*cartan_type*, *r*, *s*, *dual=None*)

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystal

Class of Kirillov-Reshetikhin crystals $B^{n,s}$ of type $D^{(2)}_{n+1}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,2],3,1)
sage: [[b,b.f(0)] for b in K]
[[[+++, []], None], [[++-, []], None], [[+-+, []], None], [[-++, []],
[+++, []]], [[+--, []], None], [[-+-, []], [++-, []]], [[--+, []], [+-+, []]],
[[---, []], [+--, []]]]
```

Element

alias of KR_type_Dn_twistedElement

classical_decomposition()

Specifies the classical crystal underlying the Kirillov-Reshetikhin crystal $B^{n,s}$ of type $D^{(2)}_{n+1}$. It is given by $B^{n,s} \cong B(s\Lambda_n)$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,2],3,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[1/2, 1/2, 1/2]]
sage: K = crystals.KirillovReshetikhin(['D',4,2],3,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[1, 1, 1]]
```

from_highest_weight_vector_to_pm_diagram(*b*)

This gives the bijection between an element b in the classical decomposition of the KR crystal that is $2, 3, .., n$-highest weight and $\pm$ diagrams.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,2],3,1)
sage: T = K.classical_decomposition()
sage: hw = [ b for b in T if all(b.epsilon(i)==0 for i in [2,3]) ]
sage: [K.from_highest_weight_vector_to_pm_diagram(b) for b in hw]
[[[0, 0], [0, 0], [1, 0], [0]], [[0, 0], [0, 0], [0, 1], [0]]]

sage: K = crystals.KirillovReshetikhin(['D',4,2],3,2)
sage: T = K.classical_decomposition()
sage: hw = [ b for b in T if all(b.epsilon(i)==0 for i in [2,3]) ]
sage: [K.from_highest_weight_vector_to_pm_diagram(b) for b in hw]
[[[0, 0], [0, 0], [2, 0], [0]], [[0, 0], [0, 0], [0, 0], [2]], [[0, 0], [2, 0], [0, 0], [0]]
[[0, 0], [0, 0], [0, 2], [0]]]
```

Note that, since the classical decomposition of this crystal is of type B_n, there can be (at most one) entry 0 in the $2, 3, ..., n$-highest weight elements at height n. In the following implementation this is realized as an empty column of height n since this uniquely specifies the existence of the 0:

EXAMPLES:

```
sage: b = hw[1]
sage: pm = K.from_highest_weight_vector_to_pm_diagram(b)
sage: pm.pp()
.  .
```

```
. .
. .
```

TESTS:

```
sage: all(K.from_pm_diagram_to_highest_weight_vector(K.from_highest_weight_vector_to_pm_diag
True
sage: K = crystals.KirillovReshetikhin(['D',4,2],3,2)
sage: T = K.classical_decomposition()
sage: hw = [ b for b in T if all(b.epsilon(i)==0 for i in [2,3]) ]
sage: all(K.from_pm_diagram_to_highest_weight_vector(K.from_highest_weight_vector_to_pm_diag
True
sage: K = crystals.KirillovReshetikhin(['D',4,2],3,3)
sage: T = K.classical_decomposition()
sage: hw = [ b for b in T if all(b.epsilon(i)==0 for i in [2,3]) ]
sage: all(K.from_pm_diagram_to_highest_weight_vector(K.from_highest_weight_vector_to_pm_diag
True
```

from_pm_diagram_to_highest_weight_vector(*pm*)

This gives the bijection between a $\pm$ diagram and an element b in the classical decomposition of the KR crystal that is {2,3,..,n}-highest weight.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,2],3,2)
sage: pm = sage.combinat.crystals.kirillov_reshetikhin.PMDiagram([[0, 0], [0, 0], [0, 0], [2
sage: K.from_pm_diagram_to_highest_weight_vector(pm)
[[2], [3], [0]]
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_Dn_twistedElement**

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystalEleme

Class for the elements in the Kirillov-Reshetikhin crystals $B^{n,s}$ of type $D_{n+1}^{(2)}$.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['D',4,2],3,2)
sage: type(K.module_generators[0])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_Dn_twisted_with_category.element_cla
```

e0()

Gives e_0 on self by going to the $\pm$-diagram corresponding to the $2, ..., n$-highest weight vector in the component of $self$, then applying [Definition 6.2, 4], and pulling back from $\pm$-diagrams.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['D',4,2],3,3)
sage: b = K.module_generators[0]
sage: b.e(0) # indirect doctest
[+++, [[2], [3], [0]]]
```

epsilon0()

Calculate ε_0 of `self` using Lemma 6.2 of [4].

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['D',4,2],3,1)
sage: b = K.module_generators[0]
sage: b.epsilon(0) # indirect doctest
1
```

TESTS:

Check that trac ticket #19982 is fixed:

```
sage: K = crystals.KirillovReshetikhin(['D',3,2], 2,3)
sage: def eps0_defn(elt):
....:     x = elt.e(0)
....:     eps = 0
....:     while x is not None:
....:         x = x.e(0)
....:         eps = eps + 1
....:     return eps
sage: all(eps0_defn(x) == x.epsilon0() for x in K)
True
```

f0()

Gives e_0 on self by going to the $\pm$-diagram corresponding to the $2, ..., n$-highest weight vector in the component of $self$, then applying [Definition 6.2, 4], and pulling back from $\pm$-diagrams.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['D',4,2],3,2)
sage: b = K.module_generators[0]
sage: b.f(0) # indirect doctest
```

phi0()

Calculate φ_0 of `self`.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['D',4,2],3,1)
sage: b = K.module_generators[0]
sage: b.phi(0) # indirect doctest
0
```

TESTS:

Check that trac ticket #19982 is fixed:

```
sage: K = crystals.KirillovReshetikhin(['D',3,2], 2,3)
sage: def phi0_defn(elt):
....:     x = elt.f(0)
....:     phi = 0
....:     while x is not None:
....:         x = x.f(0)
....:         phi = phi + 1
....:     return phi
sage: all(phi0_defn(x) == x.phi0() for x in K)
True
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_E6**(*cartan_type*, *r*, *s*)

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinCrystalFromPromoti

Class of Kirillov-Reshetikhin crystals of type $E_6^{(1)}$ for $r = 1, 2, 6$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: K.module_generator().e(0)
[]
sage: K.module_generator().e(0).f(0)
[[(2, -1), (1,)]]
sage: K = crystals.KirillovReshetikhin(['E',6,1], 1,1)
sage: b = K.module_generator()
```

```
sage: b
[(1,)]
sage: b.e(0)
[(-2, 1)]
sage: b = [t for t in K if t.epsilon(1) == 1 and t.phi(3) == 1 and t.phi(2) == 0 and t.epsilon(2
sage: b
[(-1, 3)]
sage: b.e(0)
[(-1, -2, 3)]
```

The elements of the Kirillov-Reshetikhin crystals can be constructed from a classical crystal element using `retract()`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: La = K.cartan_type().classical().root_system().weight_lattice().fundamental_weights()
sage: H = crystals.HighestWeight(La[2])
sage: t = H.module_generator()
sage: t
[[(2, -1), (1,)]]
sage: type(K.retract(t))
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_E6_with_category.element_class'>
sage: K.retract(t).e(0)
[]
```

TESTS:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1], 2,1)
sage: La = K.weight_lattice_realization().fundamental_weights()
sage: all(b.weight() == sum( (K.affine_weight(b.lift())[i] * La[i] for i in K.index_set()), 0*La
True
```

affine_weight(*b*)

Returns the affine level zero weight corresponding to the element b of the classical crystal underlying self. For the coefficients to calculate the level, see Kac pg. 48.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: [K.affine_weight(x.lift()) for x in K if all(x.epsilon(i) == 0 for i in [2,3,4,5])]
[(0, 0, 0, 0, 0, 0, 0),
(-2, 0, 1, 0, 0, 0, 0),
(-1, -1, 0, 0, 0, 1, 0),
(0, 0, 0, 0, 0, 0, 0),
(0, 0, 0, 0, 0, 1, -2),
(0, -1, 1, 0, 0, 0, -1),
(-1, 0, 0, 1, 0, 0, -1),
(-1, -1, 0, 0, 1, 0, -1),
(0, 0, 0, 0, 0, 0, 0),
(0, -2, 0, 1, 0, 0, 0)]
```

automorphism_on_affine_weight(*weight*)

Acts with the Dynkin diagram automorphism on affine weights as outputted by the `affine_weight` method.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: [[x[0], K.automorphism_on_affine_weight(x[0])] for x in K.highest_weight_dict().values
```

```
[[(0, 0, 0, 0, 0, 1, -2), (-2, 0, 1, 0, 0, 0, 0)],
 [(-1, 0, 0, 1, 0, 0, -1), (-1, -1, 0, 0, 0, 1, 0)],
 [(0, 0, 0, 0, 0, 0, 0), (0, 0, 0, 0, 0, 0, 0)],
 [(-2, 0, 1, 0, 0, 0, 0), (0, -2, 0, 1, 0, 0, 0)],
 [(0, 0, 0, 0, 0, 0, 0), (0, 0, 0, 0, 0, 0, 0)]]
```

classical_decomposition()

Specifies the classical crystal underlying the KR crystal of type $E_6^{(1)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1], 2,2)
sage: K.classical_decomposition()
Direct sum of the crystals Family
 (Finite dimensional highest weight crystal of type ['E', 6] and highest weight 0,
  Finite dimensional highest weight crystal of type ['E', 6] and highest weight Lambda[2],
  Finite dimensional highest weight crystal of type ['E', 6] and highest weight 2*Lambda[2])
sage: K = crystals.KirillovReshetikhin(['E',6,1], 1,2)
sage: K.classical_decomposition()
Direct sum of the crystals Family
 (Finite dimensional highest weight crystal of type ['E', 6] and highest weight 2*Lambda[1],
```

dynkin_diagram_automorphism(i)

Specifies the Dynkin diagram automorphism underlying the promotion action on the crystal elements. The automorphism needs to map node 0 to some other Dynkin node.

Here we use the Dynkin diagram automorphism of order 3 which maps node 0 to node 1.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: [K.dynkin_diagram_automorphism(i) for i in K.index_set()]
[1, 6, 3, 5, 4, 2, 0]
```

highest_weight_dict()

Returns a dictionary between $1, 2, 3, 4, 5$ highest weight elements, and a tuple of affine weights and its classical component.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: K.highest_weight_dict()
{[[(2, -1), (1,)]]: ((-2, 0, 1, 0, 0, 0, 0), 1),
 [[(3, -1, -6), (1,)]]: ((-1, 0, 0, 1, 0, 0, -1), 1),
 [[(6, -2), (-6, 2)]]: ((0, 0, 0, 0, 0, 0, 0), 1),
 [[(5, -2, -6), (-6, 2)]]: ((0, 0, 0, 0, 0, 1, -2), 1),
 []: ((0, 0, 0, 0, 0, 0, 0), 0)}
```

highest_weight_dict_inv()

Return a dictionary between a tuple of affine weights and a classical component, and $2, 3, 4, 5, 6$ highest weight elements.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: K.highest_weight_dict_inv()
{((-2, 0, 1, 0, 0, 0, 0), 1): [[(2, -1), (1,)]],
 ((-1, -1, 0, 0, 0, 1, 0), 1): [[(5, -3), (-1, 3)]],
 ((0, -2, 0, 1, 0, 0, 0), 1): [[(-1,), (-1, 3)]],
 ((0, 0, 0, 0, 0, 0, 0), 0): [],
 ((0, 0, 0, 0, 0, 0, 0), 1): [[(1, -3), (-1, 3)]]}
```

hw_auxiliary()

Returns the $2, 3, 4, 5$ highest weight elements of self.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: K.hw_auxiliary()
[[], [[(2, -1), (1,)]],
 [[(5, -3), (-1, 3)]],
 [[(6, -2), (-6, 2)]],
 [[(5, -2, -6), (-6, 2)]],
 [[(-1,), (-6, 2)]],
 [[(3, -1, -6), (1,)]],
 [[(4, -3, -6), (-1, 3)]],
 [[(1, -3), (-1, 3)]],
 [[(-1,), (-1, 3)]]]
```

promotion()

Specifies the promotion operator used to construct the affine type $E_6^{(1)}$ crystal.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1], 2,1)
sage: promotion = K.promotion()
sage: all(promotion(promotion(promotion(b))) == b for b in K.classical_decomposition())
True
sage: K = crystals.KirillovReshetikhin(['E',6,1],1,1)
sage: promotion = K.promotion()
sage: all(promotion(promotion(promotion(b))) == b for b in K.classical_decomposition())
True
```

promotion_inverse()

Return the inverse promotion. Since promotion is of order 3, the inverse promotion is the same as promotion applied twice.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1], 2,1)
sage: p = K.promotion()
sage: p_inv = K.promotion_inverse()
sage: all(p_inv(p(b)) == b for b in K.classical_decomposition())
True
```

promotion_on_highest_weight_vectors()

Gives a dictionary of the promotion map on $1, 2, 3, 4, 5$ highest weight elements to $2, 3, 4, 5, 6$ elements in `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: dic = K.promotion_on_highest_weight_vectors()
sage: dic
{[[(2, -1), (1,)]]: [[(-1,), (-1, 3)]],
 [[(3, -1, -6), (1,)]]: [[(5, -3), (-1, 3)]],
 [[(6, -2), (-6, 2)]]: [],
 [[(5, -2, -6), (-6, 2)]]: [[(2, -1), (1,)]],
 []: [[(1, -3), (-1, 3)]]}
```

promotion_on_highest_weight_vectors_function()

Return a lambda function on x defined by `self.promotion_on_highest_weight_vectors()[x]`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1], 2,1)
sage: f = K.promotion_on_highest_weight_vectors_function()
sage: f(K.module_generator().lift())
[[(-1,), (-1, 3)]]
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_box**(*cartan_type*, *r*, *s*)

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystal, sage.combinat.crystals.affine.AffineCrystalFromClassical

Class of Kirillov-Reshetikhin crystals $B^{r,s}$ of type $A_{2n}^{(2)}$ for $r \leq n$ and type $D_{n+1}^{(2)}$ for $r < n$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',4,2], 1,1)
sage: K
Kirillov-Reshetikhin crystal of type ['BC', 2, 2] with (r,s)=(1,1)
sage: b = K(rows=[])
sage: b.f(0)
[[1]]
sage: b.e(0)
[[-1]]
```

Element

alias of KR_type_boxElement

ambient_crystal()

Returns the ambient crystal $B^{r,2s}$ of type $C_n^{(1)}$ associated to the Kirillov-Reshetikhin crystal. This ambient crystal is used to construct the zero arrows.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',4,2], 2,2)
sage: K.ambient_crystal()
Kirillov-Reshetikhin crystal of type ['C', 2, 1] with (r,s)=(2,4)
```

ambient_highest_weight_dict()

Gives a dictionary of the classical highest weight vectors of the ambient crystal of self. Their key is their shape.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',6,2], 2,2)
sage: K.ambient_highest_weight_dict()
{[]: [],
 [2]: [[1, 1]],
 [2, 2]: [[1, 1], [2, 2]],
 [4]: [[1, 1, 1, 1]],
 [4, 2]: [[1, 1, 1, 1], [2, 2]],
 [4, 4]: [[1, 1, 1, 1], [2, 2, 2, 2]]}
```

classical_decomposition()

Specifies the classical crystal underlying the Kirillov-Reshetikhin crystal of type $A_{2n}^{(2)}$ and $D_{n+1}^{(2)}$.

It is given by $B^{r,s} \cong \bigoplus_{\Lambda} B(\Lambda)$ where Λ are weights obtained from a rectangle of width s and height r by removing boxes. Here we identify the fundamental weight Λ_i with a column of height i.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',4,2], 2,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['C', 2] and shape(s) [[], [1], [2], [1, 1], [2, 1], [2, 2]]
sage: K = crystals.KirillovReshetikhin(['D',4,2], 2,3)
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[], [1], [2], [1, 1], [3], [2, 1], [3
```

from_ambient_crystal()

Provides a map from the ambient crystal of type $C_n^{(1)}$ to the Kirillov-Reshetikhin crystal `self`.

Note that this map is only well-defined on elements that are in the image under `to_ambient_crystal()`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,2], 1,1)
sage: b = K.ambient_crystal()(rows=[[3,-3]])
sage: K.from_ambient_crystal()(b)
[[0]]
sage: K = crystals.KirillovReshetikhin(['A',4,2], 1,1)
sage: b = K.ambient_crystal()(rows=[])
sage: K.from_ambient_crystal()(b)
[]
```

highest_weight_dict()

Gives a dictionary of the classical highest weight vectors of self. Their key is 2 times their shape.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',6,2], 2,2)
sage: K.highest_weight_dict()
{[]: [],
 [2]: [[1]],
 [2, 2]: [[1], [2]],
 [4]: [[1, 1]],
 [4, 2]: [[1, 1], [2]],
 [4, 4]: [[1, 1], [2, 2]]}
```

similarity_factor()

Sets the similarity factor used to map to the ambient crystal.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',6,2], 2,2)
sage: K.similarity_factor()
{1: 2, 2: 2, 3: 2}
sage: K = crystals.KirillovReshetikhin(['D',5,2], 1,1)
sage: K.similarity_factor()
{1: 2, 2: 2, 3: 2, 4: 1}
```

to_ambient_crystal()

Provides a map from self to the ambient crystal of type $C_n^{(1)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,2], 1,1)
sage: [K.to_ambient_crystal()(b) for b in K]
[[], [[1, 1]], [[2, 2]], [[3, 3]], [[3, -3]], [[-3, -3]], [[-2, -2]], [[-1, -1]]]
sage: K = crystals.KirillovReshetikhin(['A',4,2], 1,1)
sage: [K.to_ambient_crystal()(b) for b in K]
[[], [[1, 1]], [[2, 2]], [[-2, -2]], [[-1, -1]]]
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_boxElement**

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystalEleme

Class for the elements in the Kirillov-Reshetikhin crystals $B^{r,s}$ of type $A_{2n}^{(2)}$ for $r \leq n$ and type $D_{n+1}^{(2)}$ for $r < n$.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['A',4,2],1,2)
sage: type(K.module_generators[0])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_box_with_category.element_class'>
```

e0()

Gives e_0 on self by mapping self to the ambient crystal, calculating e_0 there and pulling the element back.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['A',4,2],1,1)
sage: b = K(rows=[])
sage: b.e(0) # indirect doctest
[[-1]]
```

epsilon0()

Calculate ε_0 of `self` by mapping the element to the ambient crystal and calculating ε_0 there.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',4,2], 1,1)
sage: b=K(rows=[[1]])
sage: b.epsilon(0) # indirect doctest
2
```

f0()

Gives f_0 on self by mapping self to the ambient crystal, calculating f_0 there and pulling the element back.

EXAMPLES:

```
sage: K=crystals.KirillovReshetikhin(['A',4,2],1,1)
sage: b = K(rows=[])
sage: b.f(0) # indirect doctest
[[1]]
```

phi0()

Calculate φ_0 of `self` by mapping the element to the ambient crystal and calculating φ_0 there.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',3,2], 1,1)
sage: b=K(rows=[[-1]])
sage: b.phi(0) # indirect doctest
2
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_spin**(*cartan_type*, *r*, *s*)

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinCrystalFromPromotic

Class of Kirillov-Reshetikhin crystals $B^{n,s}$ of type $D_n^{(1)}$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1],4,1); K
Kirillov-Reshetikhin crystal of type ['D', 4, 1] with (r,s)=(4,1)
sage: [[b,b.f(0)] for b in K]
[[[++++, []], None], [[++--, []], None], [[+-+-, []], None], [[-++-, []], None],
```

```
[[+--+, []], None], [[-+-+, []], None], [[--++, []], [++++, []]], [[----, []], [++--, []]]]

sage: K = crystals.KirillovReshetikhin(['D',4,1],4,2); K
Kirillov-Reshetikhin crystal of type ['D', 4, 1] with (r,s)=(4,2)
sage: [[b,b.f(0)] for b in K]
[[[[1], [2], [3], [4]], None], [[[1], [2], [-4], [4]], None], [[[1], [3], [-4], [4]], None],
[[[2], [3], [-4], [4]], None], [[[1], [4], [-4], [4]], None], [[[2], [4], [-4], [4]], None],
[[[3], [4], [-4], [4]], [[1], [2], [3], [4]]], [[[-4], [4], [-4], [4]], [[1], [2], [-4], [4]]],
[[[-4], [4], [-4], [-3]], [[1], [2], [-4], [-3]]], [[[-4], [4], [-4], [-2]], [[1], [3], [-4], [-
[[[-4], [4], [-4], [-1]], [[2], [3], [-4], [-3]]], [[[-4], [4], [-3], [-2]], [[1], [4], [-4], [-
[[[-4], [4], [-3], [-1]], [[2], [4], [-4], [-3]]], [[[-4], [4], [-2], [-1]], [[-4], [4], [-4], [
[[[-4], [-3], [-2], [-1]], [[-4], [4], [-4], [-3]]], [[[1], [2], [-4], [-3]], None], [[[1], [3],
[[[2], [3], [-4], [-3]], None], [[[1], [3], [-4], [-2]], None], [[[2], [3], [-4], [-2]], None],
[[[2], [3], [-4], [-1]], None], [[[1], [4], [-4], [-3]], None], [[[2], [4], [-4], [-3]], None],
[[[3], [4], [-4], [-3]], None], [[[3], [4], [-4], [-2]], [[1], [3], [-4], [4]]],
[[[3], [4], [-4], [-1]], [[2], [3], [-4], [4]]], [[[1], [4], [-4], [-2]], None], [[[2], [4], [-4
[[[2], [4], [-4], [-1]], None], [[[1], [4], [-3], [-2]], None], [[[2], [4], [-3], [-2]], None],
[[[2], [4], [-3], [-1]], None], [[[3], [4], [-3], [-2]], [[1], [4], [-4], [4]]],
[[[3], [4], [-3], [-1]], [[2], [4], [-4], [4]]], [[[3], [4], [-2], [-1]], [[3], [4], [-4], [4]]]
```

TESTS:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1],3,1)
sage: all(b.e(0).f(0) == b for b in K if b.epsilon(0)>0)
True

sage: K = crystals.KirillovReshetikhin(['D',5,1],5,2)
sage: all(b.f(0).e(0) == b for b in K if b.phi(0)>0)
True
```

classical_decomposition()

Returns the classical crystal underlying the Kirillov-Reshetikhin crystal $B^{r,s}$ of type $D_n^{(1)}$ for $r = n-1, n$. It is given by $B^{n,s} \cong B(s\Lambda_r)$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1],4,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[1/2, 1/2, 1/2, 1/2]]
sage: K = crystals.KirillovReshetikhin(['D',4,1],3,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[1/2, 1/2, 1/2, -1/2]]
sage: K = crystals.KirillovReshetikhin(['D',4,1],3,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[1, 1, 1, -1]]
```

TESTS:

Check that this is robust against python ints:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 4, int(1))
sage: K.classical_crystal
The crystal of tableaux of type ['D', 4] and shape(s) [[1/2, 1/2, 1/2, 1/2]]
```

dynkin_diagram_automorphism(i)

Specifies the Dynkin diagram automorphism underlying the promotion action on the crystal elements. The automorphism needs to map node 0 to some other Dynkin node.

Here we use the Dynkin diagram automorphism which interchanges nodes 0 and 1 and leaves all other nodes unchanged.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1],4,1)
sage: K.dynkin_diagram_automorphism(0)
1
sage: K.dynkin_diagram_automorphism(1)
0
sage: K.dynkin_diagram_automorphism(4)
4
```

promotion()

Return the promotion operator on $B^{r,s}$ of type $D_n^{(1)}$ for $r = n-1, n$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1],3,1)
sage: T = K.classical_decomposition()
sage: promotion = K.promotion()
sage: for t in T:
....:     print t, promotion(t)
[+++-, []] [-++-, []]
[++-+, []] [-+-+, []]
[+-++, []] [--++, []]
[-+++, []] [++++, []]
[+---, []] [----, []]
[-+--, []] [++--, []]
[--+-, []] [+-+-, []]
[---+, []] [+--+, []]
```

promotion_inverse()

Returns the inverse promotion operator on $B^{r,s}$ of type $D_n^{(1)}$ for $r = n-1, n$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1],3,1)
sage: T = K.classical_decomposition()
sage: promotion = K.promotion()
sage: promotion_inverse = K.promotion_inverse()
sage: all(promotion_inverse(promotion(t)) == t for t in T)
True
```

promotion_on_highest_weight_vectors()

Returns the promotion operator on $\{2, 3, \ldots, n\}$-highest weight vectors.

A $\{2, 3, \ldots, n\}$-highest weight vector in $B(s\Lambda_n)$ of weight $w = (w_1, \ldots, w_n)$ is mapped to a $\{2, 3, \ldots, n\}$-highest weight vector in $B(s\Lambda_{n-1})$ of weight $(-w_1, w_2, \ldots, w_n)$ and vice versa.

See also promotion_on_highest_weight_vectors_inverse() and promotion().

EXAMPLES:

```
sage: KR = crystals.KirillovReshetikhin(['D',4,1],4,2)
sage: prom = KR.promotion_on_highest_weight_vectors()
sage: T = KR.classical_decomposition()
sage: HW = [t for t in T if t.is_highest_weight([2,3,4])]
sage: for t in HW:
....:     print t, prom[t]
[4, 3, 2, 1] [-1, 4, 3, 2]
[4, -4, 3, 2] [-4, 4, 3, 2]
[-1, -4, 3, 2] [-4, 3, 2, 1]

sage: KR = crystals.KirillovReshetikhin(['D',4,1],4,1)
```

```
sage: prom = KR.promotion_on_highest_weight_vectors()
sage: T = KR.classical_decomposition()
sage: HW = [t for t in T if t.is_highest_weight([2,3,4])]
sage: for t in HW:
....:     print t, prom[t]
[++++, []] [-+++, []]
[-++-, []] [+++-, []]
```

promotion_on_highest_weight_vectors_inverse()

Returns the inverse promotion operator on $\{2, 3, \ldots, n\}$-highest weight vectors.

See also promotion_on_highest_weight_vectors() and promotion_inverse().

EXAMPLES:

```
sage: KR = crystals.KirillovReshetikhin(['D',4,1],3,2)
sage: prom = KR.promotion_on_highest_weight_vectors()
sage: prom_inv = KR.promotion_on_highest_weight_vectors_inverse()
sage: T = KR.classical_decomposition()
sage: HW = [t for t in T if t.is_highest_weight([2,3,4])]
sage: all(prom_inv[prom[t]] == t for t in HW)
True
```

class sage.combinat.crystals.kirillov_reshetikhin.**KR_type_vertical**(*cartan_type*, *r*, *s*)

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinCrystalFromPromoti

Class of Kirillov-Reshetikhin crystals $B^{r,s}$ of type $D_n^{(1)}$ for $r \leq n-2$, $B_n^{(1)}$ for $r < n$, and $A_{2n-1}^{(2)}$ for $r \leq n$.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: b = K(rows=[])
sage: b.f(0)
[[1], [2]]
sage: b.f(0).f(0)
[[1, 1], [2, 2]]
sage: b.e(0)
[[-2], [-1]]
sage: b.e(0).e(0)
[[-2, -2], [-1, -1]]

sage: K = crystals.KirillovReshetikhin(['D',5,1], 3,1)
sage: b = K(rows=[[1]])
sage: b.e(0)
[[3], [-3], [-2]]

sage: K = crystals.KirillovReshetikhin(['B',3,1], 1,1)
sage: [[b,b.f(0)] for b in K]
[[[[1]], None], [[[2]], None], [[[3]], None], [[[0]], None], [[[-3]], None], [[[-2]], [[1]]], [[
sage: K = crystals.KirillovReshetikhin(['A',5,2], 1,1)
sage: [[b,b.f(0)] for b in K]
[[[[1]], None], [[[2]], None], [[[3]], None], [[[-3]], None], [[[-2]], [[1]]], [[[-1]], [[2]]]]
```

classical_decomposition()

Specifies the classical crystal underlying the Kirillov-Reshetikhin crystal of type $D_n^{(1)}$, $B_n^{(1)}$, and $A_{2n-1}^{(2)}$.

It is given by $B^{r,s} \cong \bigoplus_{\Lambda} B(\Lambda)$ where Λ are weights obtained from a rectangle of width s and height r by removing verticle dominoes. Here we identify the fundamental weight Λ_i with a column of height i.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[], [1, 1], [2, 2]]
```

dynkin_diagram_automorphism(*i*)

Specifies the Dynkin diagram automorphism underlying the promotion action on the crystal elements. The automorphism needs to map node 0 to some other Dynkin node.

Here we use the Dynkin diagram automorphism which interchanges nodes 0 and 1 and leaves all other nodes unchanged.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1],1,1)
sage: K.dynkin_diagram_automorphism(0)
1
sage: K.dynkin_diagram_automorphism(1)
0
sage: K.dynkin_diagram_automorphism(4)
4
```

from_highest_weight_vector_to_pm_diagram(*b*)

This gives the bijection between an element `b` in the classical decomposition of the KR crystal that is $2, 3, \ldots, n$-highest weight and $\pm$ diagrams.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: T = K.classical_decomposition()
sage: b = T(rows=[[2],[-2]])
sage: pm = K.from_highest_weight_vector_to_pm_diagram(b); pm
[[1, 1], [0, 0], [0]]
sage: pm.pp()
+
-
sage: b = T(rows=[])
sage: pm=K.from_highest_weight_vector_to_pm_diagram(b); pm
[[0, 2], [0, 0], [0]]
sage: pm.pp()

sage: hw = [ b for b in T if all(b.epsilon(i)==0 for i in [2,3,4]) ]
sage: all(K.from_pm_diagram_to_highest_weight_vector(K.from_highest_weight_vector_to_pm_diag
True
```

from_pm_diagram_to_highest_weight_vector(*pm*)

This gives the bijection between a $\pm$ diagram and an element `b` in the classical decomposition of the KR crystal that is $2, 3, \ldots, n$-highest weight.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: pm = sage.combinat.crystals.kirillov_reshetikhin.PMDiagram([[1, 1], [0, 0], [0]])
sage: K.from_pm_diagram_to_highest_weight_vector(pm)
[[2], [-2]]
```

promotion()

Specifies the promotion operator used to construct the affine type $D_n^{(1)}$ etc. crystal.

This corresponds to the Dynkin diagram automorphism which interchanges nodes 0 and 1, and leaves all

other nodes unchanged. On the level of crystals it is constructed using $\pm$ diagrams.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: promotion = K.promotion()
sage: b = K.classical_decomposition()(rows=[])
sage: promotion(b)
[[1, 2], [-2, -1]]
sage: b = K.classical_decomposition()(rows=[[1,3],[2,-1]])
sage: promotion(b)
[[1, 3], [2, -1]]
sage: b = K.classical_decomposition()(rows=[[1],[-3]])
sage: promotion(b)
[[2, -3], [-2, -1]]
```

promotion_inverse()

Return inverse of promotion.

In this case promotion is an involution, so promotion inverse equals promotion.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: promotion = K.promotion()
sage: promotion_inverse = K.promotion_inverse()
sage: all( promotion_inverse(promotion(b.lift())) == b.lift() for b in K )
True
```

promotion_on_highest_weight_vectors()

Calculates promotion on $2, 3, ..., n$ highest weight vectors.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: T = K.classical_decomposition()
sage: hw = [ b for b in T if all(b.epsilon(i)==0 for i in [2,3,4]) ]
sage: [K.promotion_on_highest_weight_vectors()(b) for b in hw]
[[[1, 2], [-2, -1]], [[2, 2], [-2, -1]], [[1, 2], [3, -1]], [[2], [-2]],
[[1, 2], [2, -2]], [[2, 2], [-1, -1]], [[2, 2], [3, -1]], [[2, 2], [3, 3]],
[], [[1], [2]], [[1, 1], [2, 2]], [[2], [-1]], [[1, 2], [2, -1]], [[2], [3]],
[[1, 2], [2, 3]]]
```

sage.combinat.crystals.kirillov_reshetikhin.**KashiwaraNakashimaTableaux**(*cartan_type*, *r*, *s*)

Return the Kashiwara-Nakashima model for the Kirillov-Reshetikhin crystal $B^{r,s}$ in the given type.

The Kashiwara-Nakashima (KN) model constructs the KR crystal from the KN tableaux model for the corresponding classical crystals. This model is named for the underlying KN tableaux.

Many Kirillov-Reshetikhin crystals are constructed from a classical crystal together with an automorphism p on the level of crystals which corresponds to a Dynkin diagram automorphism mapping node 0 to some other node i. The action of f_0 and e_0 is then constructed using $f_0 = p^{-1} \circ f_i \circ p$.

For example, for type $A_n^{(1)}$ the Kirillov-Reshetikhin crystal $B^{r,s}$ is obtained from the classical crystal $B(s\omega_r)$ using the promotion operator. For other types, see [Shimozono02], [Schilling08], and [JS2010].

Other Kirillov-Reshetikhin crystals are constructed using similarity methods. See Section 4 of [FOS09].

For more information on Kirillov-Reshetikhin crystals, see `KirillovReshetikhinCrystal()`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2, 1)
sage: K2 = crystals.kirillov_reshetikhin.KashiwaraNakashimaTableaux(['A',3,1], 2, 1)
sage: K is K2
True
```

sage.combinat.crystals.kirillov_reshetikhin.**KirillovReshetikhinCrystal**(*cartan_type*, *r*, *s*, *model='KN'*)

Return the Kirillov-Reshetikhin crystal $B^{r,s}$ of the given type in the given model.

For more information about general crystals see `sage.combinat.crystals.crystals`.

There are a variety of models for Kirillov-Reshetikhin crystals. There is one using the classical crystal with `Kashiwara-Nakashima tableaux`. There is one using `rigged configurations`. Another tableaux model comes from the bijection between rigged configurations and tensor products of tableaux called `Kirillov-Reshetikhin tableaux` Lastly there is a model of Kirillov-Reshetikhin crystals for $s = 1$ from crystals of `LS paths`.

INPUT:

- `cartan_type` – an affine Cartan type
- `r` – a label of finite Dynkin diagram
- `s` – a positive integer
- `model` – (default: `'KN'`) can be one of the following:
  - `'KN'` or `'KashiwaraNakashimaTableaux'` - use the Kashiwara-Nakashima tableaux model
  - `'KR'` or `'KirillovReshetkihinTableaux'` - use the Kirillov-Reshetkihin tableaux model
  - `'RC'` or `'RiggedConfiguration'` - use the rigged configuration model
  - `'LSPaths'` - use the LS path model

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2, 1)
sage: K.index_set()
(0, 1, 2, 3)
sage: K.list()
[[[1], [2]], [[1], [3]], [[2], [3]], [[1], [4]], [[2], [4]], [[3], [4]]]
sage: b=K(rows=[[1],[2]])
sage: b.weight()
-Lambda[0] + Lambda[2]

sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.automorphism(K.module_generators[0])
[[2, 2], [3, 3]]
sage: K.module_generators[0].e(0)
[[1, 2], [2, 4]]
sage: K.module_generators[0].f(2)
[[1, 1], [2, 3]]
sage: K.module_generators[0].f(1)
sage: K.module_generators[0].phi(0)
0
sage: K.module_generators[0].phi(1)
0
sage: K.module_generators[0].phi(2)
2
sage: K.module_generators[0].epsilon(0)
2
```

```
sage: K.module_generators[0].epsilon(1)
0
sage: K.module_generators[0].epsilon(2)
0
sage: b = K(rows=[[1,2],[2,3]])
sage: b
[[1, 2], [2, 3]]
sage: b.f(2)
[[1, 2], [3, 3]]

sage: K = crystals.KirillovReshetikhin(['D',4,1], 2, 1)
sage: K.cartan_type()
['D', 4, 1]
sage: type(K.module_generators[0])
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_vertical_with_category.element_class
```

The following gives some tests with regards to Lemma 3.11 in [LOS12].

TESTS:

```
sage: K = crystals.KirillovReshetikhin(['A',4,2],2,1)
sage: Lambda = K.weight_lattice_realization().fundamental_weights()
sage: [b for b in K if b.Epsilon() == Lambda[0]]
[[]]

sage: K = crystals.KirillovReshetikhin(['D',4,2],1,2)
sage: Lambda = K.weight_lattice_realization().fundamental_weights()
sage: [b for b in K if b.Epsilon() == 2*Lambda[0]]
[[]]
sage: [b for b in K if b.Epsilon() == 2*Lambda[3]]
[[[3, -3]]]
sage: K = crystals.KirillovReshetikhin(['D',4,2],1,1)
sage: [b for b in K if b.Epsilon() == Lambda[3]]
[[[0]]]

sage: K = crystals.KirillovReshetikhin(['B',3,1],2,1)
sage: Lambda = K.weight_lattice_realization().fundamental_weights()
sage: [b for b in K if b.Epsilon() == Lambda[0]]
[[]]
sage: [b for b in K if b.Epsilon() == Lambda[1]]
[[[2], [-2]]]
sage: K = crystals.KirillovReshetikhin(['B',3,1],2,2)
sage: [b for b in K if b.Epsilon() == 2*Lambda[0]]
[[]]
sage: [b for b in K if b.Epsilon() == 2*Lambda[1]]
[[[1, 2], [-2, -1]]]
sage: K = crystals.KirillovReshetikhin(['B',3,1],2,3)
sage: [b for b in K if b.Epsilon() == 3*Lambda[1]] # long time
[[[1, 2, 2], [-2, -2, -1]]]

sage: K = crystals.KirillovReshetikhin(['D',4,1],2,2)
sage: Lambda = K.weight_lattice_realization().fundamental_weights()
sage: [b for b in K if b.Epsilon() == 2*Lambda[0]] # long time
[[]]
sage: [b for b in K if b.Epsilon() == 2*Lambda[4]] # long time
[[[3, -4], [4, -3]]]

sage: K = crystals.KirillovReshetikhin(['B',3,1],3,1)
sage: Lambda = K.weight_lattice_realization().fundamental_weights()
```

```
sage: [b for b in K if b.Epsilon() == Lambda[0]]
[[+++, []]]
sage: [b for b in K if b.Epsilon() == Lambda[1]]
[[-++, []]]
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,3)
sage: [b for b in K if b.Epsilon() == 2*Lambda[0]] # long time
[[+++, [[1]]]]
sage: [b for b in K if b.Epsilon() == 2*Lambda[1]] # long time
[[-++, [[-1]]]]

sage: K = crystals.KirillovReshetikhin(['B',4,1],4,1)
sage: Lambda = K.weight_lattice_realization().fundamental_weights()
sage: [b for b in K if b.Epsilon() == Lambda[0]]
[[++++, []]]
sage: [b for b in K if b.Epsilon() == Lambda[1]]
[[-+++, []]]

sage: K = crystals.KirillovReshetikhin(['C',3,1],1,1)
sage: Lambda = K.weight_lattice_realization().fundamental_weights()
sage: [b for b in K if b.Epsilon() == Lambda[0]]
[[[1]]]
sage: [b for b in K if b.Epsilon() == Lambda[3]]
[[[-3]]]
sage: K = crystals.KirillovReshetikhin(['C',3,1],1,3)
sage: [b for b in K if b.Epsilon() == 2*Lambda[3]] # long time
[[[3, -3, -3]]]
sage: [b for b in K if b.Epsilon() == 2*Lambda[0]] # long time
[[[1]]]
```

We check the various models agree:

```
sage: KN = crystals.KirillovReshetikhin(['D',4,1], 2, 1)
sage: KR = crystals.KirillovReshetikhin(['D',4,1], 2, 1, model='KR')
sage: RC = crystals.KirillovReshetikhin(['D',4,1], 2, 1, model='RC')
sage: LS = crystals.KirillovReshetikhin(['D',4,1], 2, 1, model='LSPaths')
sage: G = KN.digraph()
sage: G.is_isomorphic(KR.digraph(), edge_labels=True)
True
sage: G.is_isomorphic(RC.digraph(), edge_labels=True)
True
sage: G.is_isomorphic(LS.digraph(), edge_labels=True)
True

sage: KN = crystals.KirillovReshetikhin(['D',4,1], 2, 1)
sage: KN2 = crystals.KirillovReshetikhin(['D',4,1], 2, 1, model='KN')
sage: KN3 = crystals.KirillovReshetikhin(['D',4,1], 2, 1, model='KashiwaraNakashimaTableaux')
sage: KN is KN2 and KN is KN3
True
```

REFERENCES:

sage.combinat.crystals.kirillov_reshetikhin.**KirillovReshetikhinCrystalFromLSPaths**(*cartan_type*, *r*, *s=1*)

Single column Kirillov-Reshetikhin crystals.

This yields the single column Kirillov-Reshetikhin crystals from the projected level zero LS paths, see `CrystalOfLSPaths`. This works for all types (even exceptional types). The weight of the canonical element in this crystal is Λ_r. For other implementation see `KirillovReshetikhinCrystal()`.

EXAMPLES:

```
sage: K = crystals.kirillov_reshetikhin.LSPaths(['A',2,1],2) # indirect doctest
sage: KR = crystals.KirillovReshetikhin(['A',2,1],2,1)
sage: G = K.digraph()
sage: GR = KR.digraph()
sage: G.is_isomorphic(GR, edge_labels = True)
True

sage: K = crystals.kirillov_reshetikhin.LSPaths(['C',3,1],2)
sage: KR = crystals.KirillovReshetikhin(['C',3,1],2,1)
sage: G = K.digraph()
sage: GR = KR.digraph()
sage: G.is_isomorphic(GR, edge_labels = True)
True

sage: K = crystals.kirillov_reshetikhin.LSPaths(['E',6,1],1)
sage: KR = crystals.KirillovReshetikhin(['E',6,1],1,1)
sage: G = K.digraph()
sage: GR = KR.digraph()
sage: G.is_isomorphic(GR, edge_labels = True)
True
sage: K.cardinality()
27

sage: K = crystals.kirillov_reshetikhin.LSPaths(['G',2,1],1)
sage: K.cardinality()
7

sage: K = crystals.kirillov_reshetikhin.LSPaths(['B',3,1],2)
sage: KR = crystals.KirillovReshetikhin(['B',3,1],2,1)
sage: KR.cardinality()
22
sage: K.cardinality()
22
sage: G = K.digraph()
sage: GR = KR.digraph()
sage: G.is_isomorphic(GR, edge_labels = True)
True
```

TESTS:

```
sage: K = crystals.kirillov_reshetikhin.LSPaths(['G',2,1],2)
sage: K.cardinality()
15
```

For $s > 1$ these crystals yield s-fold tensor products of Kirillov-Reshetikhin crystals:

```
sage: K = crystals.kirillov_reshetikhin.LSPaths(['A',1,1],1,3)
sage: B = crystals.KirillovReshetikhin(['A',1,1],1,1)
sage: T = crystals.TensorProduct(B,B,B)
sage: G = K.digraph()
sage: GT = T.digraph()
sage: G.is_isomorphic(GT, edge_labels = True)
True

sage: K = crystals.kirillov_reshetikhin.LSPaths(['B',2,1],1,2)
sage: B = crystals.KirillovReshetikhin(['B',2,1],1,1)
sage: T = crystals.TensorProduct(B,B)
sage: G = K.digraph()
```

```
sage: GT = T.digraph()
sage: G.is_isomorphic(GT, edge_labels = True)
True

sage: K = crystals.kirillov_reshetikhin.LSPaths(['B',2,1],2,3)
sage: B = crystals.KirillovReshetikhin(['B',2,1],2,1)
sage: T = crystals.TensorProduct(B,B,B)
sage: GT = T.digraph()
sage: G = K.digraph()
sage: G.is_isomorphic(GT, edge_labels = True)
True
```

class sage.combinat.crystals.kirillov_reshetikhin.**KirillovReshetikhinCrystalFromPromotion**(*cartan_type*, *r*, *s*)

Bases: sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystal, sage.combinat.crystals.affine.AffineCrystalFromClassicalAndPromotion

This generic class assumes that the Kirillov-Reshetikhin crystal is constructed from a classical crystal 'classical_decomposition' and an automorphism 'promotion' and its inverse which corresponds to a Dynkin diagram automorphism 'dynkin_diagram_automorphism'.

Each instance using this class needs to implement the methods:

- classical_decomposition
- promotion
- promotion_inverse
- dynkin_diagram_automorphism

Element
alias of KirillovReshetikhinCrystalFromPromotionElement

class sage.combinat.crystals.kirillov_reshetikhin.**KirillovReshetikhinCrystalFromPromotionElement**
Bases: sage.combinat.crystals.affine.AffineCrystalFromClassicalAndPromotionElement, sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinGenericCrystalElement

Element for a Kirillov-Reshetikhin crystal from promotion.

class sage.combinat.crystals.kirillov_reshetikhin.**KirillovReshetikhinGenericCrystal**(*cartan_type*, *r*, *s*, *dual=None*)

Bases: sage.combinat.crystals.affine.AffineCrystalFromClassical

Generic class for Kirillov-Reshetikhin crystal $B^{r,s}$ of the given type.

Input is a Dynkin node `r`, a positive integer `s`, and a Cartan type `cartan_type`.

Element
alias of KirillovReshetikhinGenericCrystalElement

R_matrix(*K*)
INPUT:

- `self` – a crystal L
- `K` – a Kirillov-Reshetikhin crystal of the same type as L

Returns the *combinatorial 'R'-matrix* from $L \otimes K \to K \otimes L$, where the combinatorial R-matrix is the affine crystal isomorphism which maps $u_L \otimes u_K$ to $u_K \otimes u_L$, where u_K is the unique element in $K = B^{r,s}$ of weight $s\Lambda_r - sc\Lambda_0$ (see module_generator).

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: L = crystals.KirillovReshetikhin(['A',2,1],1,2)
sage: f = K.R_matrix(L)
sage: [[b,f(b)] for b in crystals.TensorProduct(K,L)]
[[[[1]], [[1, 1]]], [[[1, 1]], [[1]]]],
[[[1]], [[1, 2]]], [[[1, 1]], [[2]]]],
[[[1]], [[2, 2]]], [[[1, 2]], [[2]]]],
[[[1]], [[1, 3]]], [[[1, 1]], [[3]]]],
[[[1]], [[2, 3]]], [[[1, 2]], [[3]]]],
[[[1]], [[3, 3]]], [[[1, 3]], [[3]]]],
[[[2]], [[1, 1]]], [[[1, 2]], [[1]]]],
[[[2]], [[1, 2]]], [[[2, 2]], [[1]]]],
[[[2]], [[2, 2]]], [[[2, 2]], [[2]]]],
[[[2]], [[1, 3]]], [[[2, 3]], [[1]]]],
[[[2]], [[2, 3]]], [[[2, 2]], [[3]]]],
[[[2]], [[3, 3]]], [[[2, 3]], [[3]]]],
[[[3]], [[1, 1]]], [[[1, 3]], [[1]]]],
[[[3]], [[1, 2]]], [[[1, 3]], [[2]]]],
[[[3]], [[2, 2]]], [[[2, 3]], [[2]]]],
[[[3]], [[1, 3]]], [[[3, 3]], [[1]]]],
[[[3]], [[2, 3]]], [[[3, 3]], [[2]]]],
[[[3]], [[3, 3]]], [[[3, 3]], [[3]]]]

sage: K = crystals.KirillovReshetikhin(['D',4,1],1,1)
sage: L = crystals.KirillovReshetikhin(['D',4,1],2,1)
sage: f = K.R_matrix(L)
sage: T = crystals.TensorProduct(K,L)
sage: b = T( K(rows=[[1]]), L(rows=[]) )
sage: f(b)
[[[2], [-2]], [[1]]]
```

Alternatively, one can compute the combinatorial R-matrix using the isomorphism method of digraphs:

```
sage: K1 = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: K2 = crystals.KirillovReshetikhin(['A',2,1],2,1)
sage: T1 = crystals.TensorProduct(K1,K2)
sage: T2 = crystals.TensorProduct(K2,K1)
sage: T1.digraph().is_isomorphic(T2.digraph(), edge_labels = True, certify = True) #todo: no
(True, {[[[1]], [[2], [3]]]: [[[1], [3]], [[2]]], [[[3]], [[2], [3]]]: [[[2], [3]], [[3]]],
[[[3]], [[1], [3]]]: [[[1], [3]], [[3]]], [[[1]], [[1], [3]]]: [[[1], [3]], [[1]]], [[[1]],
[[1], [2]]]: [[[1], [2]], [[1]]], [[[2]], [[1], [2]]]: [[[1], [2]], [[2]]], [[[3]],
[[1], [2]]]: [[[2], [3]], [[1]]], [[[2]], [[1], [3]]]: [[[1], [2]], [[3]]], [[[2]], [[2], [3
```

affinization()

Return the corresponding affinization crystal of `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
sage: K.affinization()
Affinization of Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(1,1)
```

classical_decomposition()

Return the classical decomposition of `self`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['A', 3] and shape(s) [[2, 2]]
```

is_perfect()

Returns True or False depending on whether `self` is a perfect crystal or not, respectively.

If `self` is the Kirillov-Reshetikhin crystal $B^{r,s}$, then it was proven in [FOS2010] that it is perfect if and only if s/c_r is an integer (where c_r is a constant related to the type of the crystal).

REFERENCES:

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
sage: K.is_perfect()
True

sage: K = crystals.KirillovReshetikhin(['C',2,1], 1, 1)
sage: K.is_perfect()
False

sage: K = crystals.KirillovReshetikhin(['C',2,1], 1, 2)
sage: K.is_perfect()
True
```

kirillov_reshetikhin_tableaux()

Return the corresponding set of KirillovReshetikhinTableaux.

EXAMPLES:

```
sage: KRC = crystals.KirillovReshetikhin(['D', 4, 1], 2, 2)
sage: KRC.kirillov_reshetikhin_tableaux()
Kirillov-Reshetikhin tableaux of type ['D', 4, 1] and shape (2, 2)
```

level()

Returns the level of `self` assuming that it is a perfect crystal.

If `self` is the Kirillov-Reshetikhin crystal $B^{r,s}$, then it was proven in [FOS2010] that its level is s/c_r which is an integer if `self` is perfect (here c_r is a constant related to the type of the crystal).

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
sage: K.level()
1
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1, 2)
sage: K.level()
1
sage: K = crystals.KirillovReshetikhin(['D',4,1], 1, 3)
sage: K.level()
3

sage: K = crystals.KirillovReshetikhin(['C',2,1], 1, 1)
sage: K.level()
Traceback (most recent call last):
...
ValueError: this crystal is not perfect
```

module_generator()
Returns the unique module generator of classical weight $s\Lambda_r$ of a Kirillov-Reshetikhin crystal $B^{r,s}$

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['C',2,1],1,2)
sage: K.module_generator()
[[1, 1]]
sage: K = crystals.KirillovReshetikhin(['E',6,1],1,1)
sage: K.module_generator()
[(1,)]

sage: K = crystals.KirillovReshetikhin(['D',4,1],2,1)
sage: K.module_generator()
[[1], [2]]
```

q_dimension(*q=None*, *prec=None*, *use_product=False*)
Return the q-dimension of `self`.

The q-dimension of a KR crystal is defined as the q-dimension of the underlying classical crystal.

EXAMPLES:

```
sage: KRC = crystals.KirillovReshetikhin(['A',2,1], 2,2)
sage: KRC.q_dimension()
q^4 + q^3 + 2*q^2 + q + 1
sage: KRC = crystals.KirillovReshetikhin(['D',4,1], 2,1)
sage: KRC.q_dimension()
q^10 + q^9 + 3*q^8 + 3*q^7 + 4*q^6 + 4*q^5 + 4*q^4 + 3*q^3 + 3*q^2 + q + 2
```

r()
Returns r of the underlying Kirillov-Reshetikhin crystal $B^{r,s}$

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2, 1)
sage: K.r()
2
```

s()
Returns s of the underlying Kirillov-Reshetikhin crystal $B^{r,s}$

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2, 1)
sage: K.s()
1
```

class sage.combinat.crystals.kirillov_reshetikhin.**KirillovReshetikhinGenericCrystalElement**
Bases: sage.combinat.crystals.affine.AffineCrystalFromClassicalElement

Abstract class for all Kirillov-Reshetikhin crystal elements.

lusztig_involution()
Return the classical Lusztig involution on `self`.

EXAMPLES:

```
sage: KRC = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: elt = KRC(-1,2); elt
[[2], [-1]]
sage: elt.lusztig_involution()
[[1], [-2]]
```

pp()

Pretty print `self`.

EXAMPLES:

```
sage: C = crystals.KirillovReshetikhin(['D',4,1], 2,1)
sage: C(2,1).pp()
  1
  2
sage: C = crystals.KirillovReshetikhin(['B',3,1], 3,3)
sage: C.module_generators[0].pp()
+ (X)   1
+
+
```

to_kirillov_reshetikhin_tableau()

Construct the corresponding `KirillovReshetikhinTableauxElement` from `self`.

We construct the Kirillov-Reshetikhin tableau element as follows:

1. Let λ be the shape of `self`.
2. Determine a path $e_{i_1} e_{i_2} \cdots e_{i_k}$ to the highest weight.
3. Apply $f_{i_k} \cdots f_{i_2} f_{i_1}$ to a highest weight KR tableau from filling the shape λ.

EXAMPLES:

```
sage: KRC = crystals.KirillovReshetikhin(['A', 4, 1], 2, 1)
sage: KRC(columns=[[2,1]]).to_kirillov_reshetikhin_tableau()
[[1], [2]]
sage: KRC = crystals.KirillovReshetikhin(['D', 4, 1], 2, 1)
sage: KRC(rows=[]).to_kirillov_reshetikhin_tableau()
[[1], [-1]]
```

to_tableau()

Return the `Tableau` corresponding to `self`.

EXAMPLES:

```
sage: C = crystals.KirillovReshetikhin(['D',4,1], 2,1)
sage: t = C(2,1).to_tableau(); t
[[1], [2]]
sage: type(t)
<class 'sage.combinat.tableau.Tableaux_all_with_category.element_class'>
```

class sage.combinat.crystals.kirillov_reshetikhin.**PMDiagram**(*pm_diagram*, *from_shapes=None*)

Bases: `sage.combinat.combinat.CombinatorialObject`

Class of $\pm$ diagrams. These diagrams are in one-to-one bijection with X_{n-1} highest weight vectors in an X_n highest weight crystal $X = B, C, D$. See Section 4.1 of [Schilling08].

The input is a list $pm = [[a_0, b_0], [a_1, b_1], ..., [a_{n-1}, b_{n-1}], [b_n]]$ of pairs and a last 1-tuple (or list of length 1). The pair $[a_i, b_i]$ specifies the number of a_i $+$ and b_i $-$ in the i-th row of the $\pm$ diagram if $n - i$ is odd and the number of a_i $\pm$ pairs above row i and b_i columns of height i not containing any $+$ or $-$ if $n - i$ is even.

Setting the option `from_shapes = True` one can also input a $\pm$ diagram in terms of its outer, intermediate, and inner shape by specifying a list `[n, s, outer, intermediate, inner]` where `s` is the width of the $\pm$ diagram, and `outer`, `intermediate`, and `inner` are the outer, intermediate, and inner shapes, respectively.

EXAMPLES:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
sage: pm = PMDiagram([[0,1],[1,2],[1]])
sage: pm.pm_diagram
[[0, 1], [1, 2], [1]]
sage: pm._list
[1, 1, 2, 0, 1]
sage: pm.n
2
sage: pm.width
5
sage: pm.pp()
.  .  .  .
.  +  -  -
sage: PMDiagram([2,5,[4,4],[4,2],[4,1]], from_shapes=True)
[[0, 1], [1, 2], [1]]
```

TESTS:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
sage: pm = PMDiagram([[1,2],[1,1],[1,1],[1,1],[1]])
sage: PMDiagram([pm.n, pm.width, pm.outer_shape(), pm.intermediate_shape(), pm.inner_shape()], f
True
sage: pm = PMDiagram([[1,2],[1,2],[1,1],[1,1],[1,1],[1]])
sage: PMDiagram([pm.n, pm.width, pm.outer_shape(), pm.intermediate_shape(), pm.inner_shape()], f
True
```

heights_of_addable_plus()

Return a list with the heights of all addable plus in the $\pm$ diagram.

EXAMPLES:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
sage: pm = PMDiagram([[1,2],[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.heights_of_addable_plus()
[1, 1, 2, 3, 4, 5]
sage: pm = PMDiagram([[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.heights_of_addable_plus()
[1, 2, 3, 4]
```

heights_of_minus()

Return a list with the heights of all minus in the $\pm$ diagram.

EXAMPLES:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
sage: pm = PMDiagram([[1,2],[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.heights_of_minus()
[5, 5, 3, 3, 1, 1]
sage: pm = PMDiagram([[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.heights_of_minus()
[4, 4, 2, 2]
```

inner_shape()

Return the inner shape of the pm diagram

EXAMPLES:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
sage: pm = PMDiagram([[0,1],[1,2],[1]])
sage: pm.inner_shape()
```

```
[4, 1]
sage: pm = PMDiagram([[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.inner_shape()
[7, 5, 3, 1]
sage: pm = PMDiagram([[1,2],[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.inner_shape()
[10, 7, 5, 3, 1]
```

intermediate_shape()

Return the intermediate shape of the pm diagram (inner shape plus positions of plusses)

EXAMPLES:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
sage: pm = PMDiagram([[0,1],[1,2],[1]])
sage: pm.intermediate_shape()
[4, 2]
sage: pm = PMDiagram([[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.intermediate_shape()
[8, 6, 4, 2]
sage: pm = PMDiagram([[1,2],[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.intermediate_shape()
[11, 8, 6, 4, 2]
sage: pm = PMDiagram([[1,0],[0,1],[2,0],[0,0],[0]])
sage: pm.intermediate_shape()
[4, 2, 2]
sage: pm = PMDiagram([[1, 0], [0, 0], [0, 0], [0, 0], [0]])
sage: pm.intermediate_shape()
[1]
```

outer_shape()

Return the outer shape of the $\pm$ diagram

EXAMPLES:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
sage: pm = PMDiagram([[0,1],[1,2],[1]])
sage: pm.outer_shape()
[4, 4]
sage: pm = PMDiagram([[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.outer_shape()
[8, 8, 4, 4]
sage: pm = PMDiagram([[1,2],[1,2],[1,1],[1,1],[1,1],[1]])
sage: pm.outer_shape()
[13, 8, 8, 4, 4]
```

pp()

Pretty print `self`.

EXAMPLES:

```
sage: from sage.combinat.crystals.kirillov_reshetikhin import PMDiagram
sage: pm = PMDiagram([[1,0],[0,1],[2,0],[0,0],[0]])
sage: pm.pp()
.  .  .  +
.  .  -  -
+  +
-  -
sage: pm = PMDiagram([[0,2], [0,0], [0]])
sage: pm.pp()
```

sigma()
Return sigma on pm diagrams as needed for the analogue of the Dynkin diagram automorphism that interchanges nodes 0 and 1 for type $D_n(1)$, $B_n(1)$, $A_{2n-1}(2)$ for Kirillov-Reshetikhin crystals.

EXAMPLES:
```
sage: pm = sage.combinat.crystals.kirillov_reshetikhin.PMDiagram([[0,1],[1,2],[1]])
sage: pm.sigma()
[[1, 0], [2, 1], [1]]
```

sage.combinat.crystals.kirillov_reshetikhin.**horizontal_dominoes_removed**(*r*, *s*)
Returns all partitions obtained from a rectangle of width s and height r by removing horizontal dominoes.

EXAMPLES:
```
sage: sage.combinat.crystals.kirillov_reshetikhin.horizontal_dominoes_removed(2,2)
[[], [2], [2, 2]]
sage: sage.combinat.crystals.kirillov_reshetikhin.horizontal_dominoes_removed(3,2)
[[], [2], [2, 2], [2, 2, 2]]
```

sage.combinat.crystals.kirillov_reshetikhin.**partitions_in_box**(*r*, *s*)
Returns all partitions in a box of width s and height r.

EXAMPLES:
```
sage: sage.combinat.crystals.kirillov_reshetikhin.partitions_in_box(3,2)
[[], [1], [2], [1, 1], [2, 1], [1, 1, 1], [2, 2], [2, 1, 1],
[2, 2, 1], [2, 2, 2]]
```

sage.combinat.crystals.kirillov_reshetikhin.**vertical_dominoes_removed**(*r*, *s*)
Returns all partitions obtained from a rectangle of width s and height r by removing vertical dominoes.

EXAMPLES:
```
sage: sage.combinat.crystals.kirillov_reshetikhin.vertical_dominoes_removed(2,2)
[[], [1, 1], [2, 2]]
sage: sage.combinat.crystals.kirillov_reshetikhin.vertical_dominoes_removed(3,2)
[[2], [2, 1, 1], [2, 2, 2]]
sage: sage.combinat.crystals.kirillov_reshetikhin.vertical_dominoes_removed(4,2)
[[], [1, 1], [1, 1, 1, 1], [2, 2], [2, 2, 1, 1], [2, 2, 2, 2]]
```

5.1.52 Kyoto Path Model for Affine Highest Weight Crystals

class sage.combinat.crystals.kyoto_path_model.**KyotoPathModel**(*crystals*, *weight*, *P*)
Bases: sage.combinat.crystals.tensor_product.TensorProductOfCrystals

The Kyoto path model for an affine highest weight crystal.

Note: Here we are using anti-Kashiwara notation and might differ from some of the literature.

Consider a Kac–Moody algebra $\mathfrak{g}$ of affine Cartan type X, and we want to model the $U'_q(\mathfrak{g})$-crystal $B(\lambda)$. First we consider the set of fundamental weights $\{\Lambda_i\}_{i\in I}$ of $\mathfrak{g}$ and let $\{\overline{\Lambda}_i\}_{i\in I_0}$ be the corresponding fundamental weights of the corresponding classical Lie algebra $\mathfrak{g}_0$. To model $B(\lambda)$, we start with a sequence of perfect $U'_q(\mathfrak{g})$-crystals $(B^{(i)})_i$ of level l such that

$$\lambda \in \overline{P}_l^+ = \left\{ \mu \in \overline{P}^+ \mid \langle c, \mu \rangle = l \right\}$$

where c is the canonical central element of $U_q'(\mathfrak{g})$ and $\overline{P}^+$ is the nonnegative weight lattice spanned by $\{\overline{\Lambda}_i \mid i \in I\}$.

Next we consider the crystal isomorphism $\Phi_0 : B(\lambda_0) \to B^{(0)} \otimes B(\lambda_1)$ defined by $u_{\lambda_0} \mapsto b^{(0)}_{\lambda_0} \otimes u_{\lambda_1}$ where $b^{(0)}_{\lambda_0}$ is the unique element in $B^{(0)}$ such that $\varphi\left(b^{(0)}_{\lambda_0}\right) = \lambda_0$ and $\lambda_1 = \varepsilon\left(b^{(0)}_{\lambda_0}\right)$ and u_μ is the highest weight element in $B(\mu)$. Iterating this, we obtain the following isomorphism:

$$\Phi_n : B(\lambda) \to B^{(0)} \otimes B^{(1)} \otimes \cdots \otimes B^{(N)} \otimes B(\lambda_{N+1}).$$

We note by Lemma 10.6.2 in [HK02] that for any $b \in B(\lambda)$ there exists a finite N such that

$$\Phi_N(b) = \left(\bigotimes_{k=0}^{N-1} b^{(k)}\right) \otimes u_{\lambda_N}.$$

Therefore we can model elements $b \in B(\lambda)$ as a $U_q'(\mathfrak{g})$-crystal by considering an infinite list of elements $b^{(k)} \in B^{(k)}$ and defining the crystal structure by:

$$\begin{aligned}
\overline{\mathrm{wt}}(b) &= \lambda_N + \sum_{k=0}^{N-1} \overline{\mathrm{wt}}\left(b^{(k)}\right) \\
e_i(b) &= e_i\left(b' \otimes b^{(N)}\right) \otimes u_{\lambda_N}, \\
f_i(b) &= f_i\left(b' \otimes b^{(N)}\right) \otimes u_{\lambda_N}, \\
\varepsilon_i(b) &= \max\left(\varepsilon_i(b') - \varphi_i\left(b^{(N)}\right), 0\right), \\
\varphi_i(b) &= \varphi_i(b') + \max\left(\varphi_i\left(b^{(N)}\right) - \varepsilon_i(b'), 0\right),
\end{aligned}$$

where $b' = b^{(0)} \otimes \cdots \otimes b^{(N-1)}$. To translate this into a finite list, we consider a finite sequence $b^{(0)} \otimes \cdots \otimes b^{(N-1)} \otimes b^{(N)}_{\lambda_N}$ and if

$$f_i\left(b^{(0)} \otimes \cdots b^{(N-1)} \otimes b^{(N)}_{\lambda_N}\right) = b_0 \otimes \cdots \otimes b^{(N-1)} \otimes f_i\left(b^{(N)}_{\lambda_N}\right),$$

then we take the image as $b^{(0)} \otimes \cdots \otimes f_i\left(b^{(N)}_{\lambda_N}\right) \otimes b^{(N+1)}_{\lambda_{N+1}}$. Similarly we remove $b^{(N)}_{\lambda_N}$ if we have $b_0 \otimes \cdots \otimes b^{(N-1)} \otimes b^{(N-1)}_{\lambda_{N-1}} \otimes b^{(N)}_{\lambda_N}$. Additionally if

$$e_i\left(b^{(0)} \otimes \cdots \otimes b^{(N-1)} \otimes b^{(N)}_{\lambda_N}\right) = b^{(0)} \otimes \cdots \otimes b^{(N-1)} \otimes e_i\left(b^{(N)}_{\lambda_N}\right),$$

then we consider this to be 0.

We can then lift the $U_q'(\mathfrak{g})$-crystal structure to a $U_q(\mathfrak{g})$-crystal structure by using a tensor product of the `affinization` of the of crystals $B^{(i)}$ for all i.

REFERENCES:

INPUT:

- `B` – a single or list of U_q' perfect crystal(s) of level l
- `weight` – a weight in $\overline{P}_l^+$

EXAMPLES:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
```

```
sage: mg = C.module_generators[0]; mg
[[[3]]]
sage: mg.f_string([0,1,2,2])
[[[3]], [[3]], [[1]]]
sage: x = mg.f_string([0,1,2]); x
[[[2]], [[3]], [[1]]]
sage: x.weight()
Lambda[0]
```

An example of type $A_5^{(2)}$:

```
sage: B = crystals.KirillovReshetikhin(['A',5,2], 1,1)
sage: La = RootSystem(['A',5,2]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]; mg
[[[-1]]]
sage: mg.f_string([0,2,1,3])
[[[-3]], [[2]], [[-1]]]
sage: mg.f_string([0,2,3,1])
[[[-3]], [[2]], [[-1]]]
```

An example of type $D_3^{(2)}$:

```
sage: B = crystals.KirillovReshetikhin(['D',3,2], 1,1)
sage: La = RootSystem(['D',3,2]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]; mg
[[]]
sage: mg.f_string([0,1,2,0])
[[[0]], [[1]], []]
```

An example using multiple crystals of the same level:

```
sage: B1 = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: B2 = crystals.KirillovReshetikhin(['A',2,1], 2,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel([B1, B2, B1], La[0])
sage: mg = C.module_generators[0]; mg
[[[3]]]
sage: mg.f_string([0,1,2,2])
[[[3]], [[1], [3]], [[3]]]
sage: mg.f_string([0,1,2,2,2])
sage: mg.f_string([0,1,2,2,1,0])
[[[3]], [[2], [3]], [[1]], [[2]]]
sage: mg.f_string([0,1,2,2,1,0,0,2])
[[[3]], [[1], [2]], [[1]], [[3]], [[1], [3]]]
```

By using the extended weight lattice, the Kyoto path model lifts the perfect crystals to their affinizations:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: P = RootSystem(['A',2,1]).weight_lattice(extended=True)
sage: La = P.fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]; mg
[[[3]](0)]
sage: x = mg.f_string([0,1,2]); x
[[[2]](-1), [[3]](0), [[1]](0)]
sage: x.weight()
Lambda[0] - delta
```

class **Element**(*parent*, **args*, ***kwds*)

Bases: sage.combinat.crystals.tensor_product.TensorProductOfRegularCrystalsElement

An element in the Kyoto path model.

e(*i*)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]
sage: all(mg.e(i) is None for i in C.index_set())
True
sage: mg.f(0).e(0) == mg
True
```

epsilon(*i*)

Return ε_i of `self`.

EXAMPLES:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]
sage: [mg.epsilon(i) for i in C.index_set()]
[0, 0, 0]
sage: elt = mg.f(0)
sage: [elt.epsilon(i) for i in C.index_set()]
[1, 0, 0]
sage: elt = mg.f_string([0,1,2])
sage: [elt.epsilon(i) for i in C.index_set()]
[0, 0, 1]
sage: elt = mg.f_string([0,1,2,2])
sage: [elt.epsilon(i) for i in C.index_set()]
[0, 0, 2]
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]
sage: mg.f(2)
sage: mg.f(0)
[[[1]], [[2]]]
sage: mg.f_string([0,1,2])
[[[2]], [[3]], [[1]]]
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]
```

```
sage: [mg.phi(i) for i in C.index_set()]
[1, 0, 0]
sage: elt = mg.f(0)
sage: [elt.phi(i) for i in C.index_set()]
[0, 1, 1]
sage: elt = mg.f_string([0,1])
sage: [elt.phi(i) for i in C.index_set()]
[0, 0, 2]
```

truncate(*k=None*)

Truncate `self` to have length `k` and return as an element in a (finite) tensor product of crystals.

INPUT:

- `k` – (optional) the length to truncate to; if not specified, then returns one more than the current non-ground-state elements (i.e. the current list in `self`)

EXAMPLES:

```
sage: B1 = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: B2 = crystals.KirillovReshetikhin(['A',2,1], 2,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel([B1,B2,B1], La[0])
sage: mg = C.highest_weight_vector()
sage: elt = mg.f_string([0,1,2,2,1,0]); elt
[[[3]], [[2], [3]], [[1]], [[2]]]
sage: t = elt.truncate(); t
[[[3]], [[2], [3]], [[1]], [[2]]]
sage: t.parent() is C.finite_tensor_product(4)
True
sage: elt.truncate(2)
[[[3]], [[2], [3]]]
sage: elt.truncate(10)
[[[3]], [[2], [3]], [[1]], [[2]], [[1], [3]],
 [[2]], [[1]], [[2], [3]], [[1]], [[3]]]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: P = RootSystem(['A',2,1]).weight_lattice(extended=True)
sage: La = P.fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: mg = C.module_generators[0]
sage: mg.weight()
Lambda[0]
sage: mg.f_string([0,1,2]).weight()
Lambda[0] - delta
```

KyotoPathModel.**finite_tensor_product**(*k*)

Return the finite tensor product of crystals of length `k` from truncating `self`.

EXAMPLES:

```
sage: B1 = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: B2 = crystals.KirillovReshetikhin(['A',2,1], 2,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel([B1,B2,B1], La[0])
sage: C.finite_tensor_product(5)
Full tensor product of the crystals
 [Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(1,1),
```

```
Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(2,1),
Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(1,1),
Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(1,1),
Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(2,1)]
```

KyotoPathModel.**weight_lattice_realization**()

Return the weight lattice realization used to express weights.

EXAMPLES:

```
sage: B = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: C = crystals.KyotoPathModel(B, La[0])
sage: C.weight_lattice_realization()
Weight lattice of the Root system of type ['A', 2, 1]

sage: P = RootSystem(['A',2,1]).weight_lattice(extended=True)
sage: C = crystals.KyotoPathModel(B, P.fundamental_weight(0))
sage: C.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]
```

5.1.53 Crystals of letters

class sage.combinat.crystals.letters.**ClassicalCrystalOfLetters**(*cartan_type*, *element_class*, *element_print_style=None*, *dual=None*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

A generic class for classical crystals of letters.

All classical crystals of letters should be instances of this class or of subclasses. To define a new crystal of letters, one only needs to implement a class for the elements (which subclasses Letter), with appropriate e_i and f_i operations. If the module generator is not 1, one also needs to define the subclass ClassicalCrystalOfLetters for the crystal itself.

The basic assumption is that crystals of letters are small, but used intensively as building blocks. Therefore, we explicitly build in memory the list of all elements, the crystal graph and its transitive closure, so as to make the following operations constant time: list, cmp, (todo: phi, epsilon, e, and f with caching)

list()

Return a list of the elements of self.

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: C.list()
[1, 2, 3, 4, 5, 6]
```

lt_elements(*x*, *y*)

Return True if and only if there is a path from x to y in the crystal graph, when x is not equal to y.

Because the crystal graph is classical, it is a directed acyclic graph which can be interpreted as a poset. This function implements the comparison function of this poset.

EXAMPLES:

```
sage: C = crystals.Letters(['A', 5])
sage: x = C(1)
sage: y = C(2)
sage: C.lt_elements(x,y)
True
sage: C.lt_elements(y,x)
False
sage: C.lt_elements(x,x)
False
sage: C = crystals.Letters(['D', 4])
sage: C.lt_elements(C(4),C(-4))
False
sage: C.lt_elements(C(-4),C(4))
False
```

sage.combinat.crystals.letters.**CrystalOfLetters**(*cartan_type*, *element_print_style=None*, *dual=None*)

Return the crystal of letters of the given type.

For classical types, this is a combinatorial model for the crystal with highest weight Λ_1 (the first fundamental weight).

Any irreducible classical crystal appears as the irreducible component of the tensor product of several copies of this crystal (plus possibly one copy of the spin crystal, see `CrystalOfSpins`). See [KN94]. Elements of this irreducible component have a fixed shape, and can be fit inside a tableau shape. Otherwise said, any irreducible classical crystal is isomorphic to a crystal of tableaux with cells filled by elements of the crystal of letters (possibly tensored with the crystal of spins).

INPUT:

- `T` – A Cartan type

REFERENCES:

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: C.list()
[1, 2, 3, 4, 5, 6]
sage: C.cartan_type()
['A', 5]
```

For type E_6, one can also specify how elements are printed. This option is usually set to None and the default representation is used. If one chooses the option 'compact', the elements are printed in the more compact convention with 27 letters `+abcdefghijklmnopqrstuvwxyz` and the 27 letters `-ABCDEFGHIJKLMNOPQRSTUVWXYZ` for the dual crystal.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6], element_print_style = 'compact')
sage: C
The crystal of letters for type ['E', 6]
sage: C.list()
[+, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z]
sage: C = crystals.Letters(['E',6], element_print_style = 'compact', dual = True)
sage: C
The crystal of letters for type ['E', 6] (dual)
sage: C.list()
[-, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z]
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_A_element**

Bases: sage.combinat.crystals.letters.Letter

Type A crystal of letters elements.

TESTS:

```
sage: C = crystals.Letters(['A',3])
sage: C.list()
[1, 2, 3, 4]
sage: [ [x < y for y in C] for x in C ]
[[False, True, True, True],
 [False, False, True, True],
 [False, False, False, True],
 [False, False, False, False]]

sage: C = crystals.Letters(['A',5])
sage: C(1) < C(1), C(1) < C(2), C(1) < C(3), C(2) < C(1)
(False, True, True, False)

sage: TestSuite(C).run()
```

e(i)

Return the action of e_i on self.

EXAMPLES:

```
sage: C = crystals.Letters(['A',4])
sage: [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[(2, 1, 1), (3, 2, 2), (4, 3, 3), (5, 4, 4)]
```

epsilon(i)

Return ε_i of self.

EXAMPLES:

```
sage: C = crystals.Letters(['A',4])
sage: [(c,i) for i in C.index_set() for c in C if c.epsilon(i) != 0]
[(2, 1), (3, 2), (4, 3), (5, 4)]
```

f(i)

Return the action of f_i on self.

EXAMPLES:

```
sage: C = crystals.Letters(['A',4])
sage: [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[(1, 1, 2), (2, 2, 3), (3, 3, 4), (4, 4, 5)]
```

phi(i)

Return φ_i of self.

EXAMPLES:

```
sage: C = crystals.Letters(['A',4])
sage: [(c,i) for i in C.index_set() for c in C if c.phi(i) != 0]
[(1, 1), (2, 2), (3, 3), (4, 4)]
```

weight()

Return the weight of self.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Letters(['A',3])]
[(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1)]
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_B_element**

Bases: sage.combinat.crystals.letters.Letter

Type B crystal of letters elements.

TESTS:

```
sage: C = crystals.Letters(['B',3])
sage: TestSuite(C).run()
```

e(i)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['B',4])
sage: [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[(2, 1, 1),
 (-1, 1, -2),
 (3, 2, 2),
 (-2, 2, -3),
 (4, 3, 3),
 (-3, 3, -4),
 (0, 4, 4),
 (-4, 4, 0)]
```

epsilon(i)

Return ε_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['B',3])
sage: [(c,i) for i in C.index_set() for c in C if c.epsilon(i) != 0]
[(2, 1), (-1, 1), (3, 2), (-2, 2), (0, 3), (-3, 3)]
```

f(i)

Return the actions of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['B',4])
sage: [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[(1, 1, 2),
 (-2, 1, -1),
 (2, 2, 3),
 (-3, 2, -2),
 (3, 3, 4),
 (-4, 3, -3),
 (4, 4, 0),
 (0, 4, -4)]
```

phi(i)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['B',3])
sage: [(c,i) for i in C.index_set() for c in C if c.phi(i) != 0]
[(1, 1), (-2, 1), (2, 2), (-3, 2), (3, 3), (0, 3)]
```

weight()
Return the weight of self.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Letters(['B',3])]
[(1, 0, 0),
 (0, 1, 0),
 (0, 0, 1),
 (0, 0, 0),
 (0, 0, -1),
 (0, -1, 0),
 (-1, 0, 0)]
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_C_element**
Bases: sage.combinat.crystals.letters.Letter

Type C crystal of letters elements.

TESTS:

```
sage: C = crystals.Letters (['C',3])
sage: C.list()
[1, 2, 3, -3, -2, -1]
sage: [ [x < y for y in C] for x in C ]
[[False, True, True, True, True, True],
 [False, False, True, True, True, True],
 [False, False, False, True, True, True],
 [False, False, False, False, True, True],
 [False, False, False, False, False, True],
 [False, False, False, False, False, False]]
sage: TestSuite(C).run()
```

e(i)
Return the action of e_i on self.

EXAMPLES:

```
sage: C = crystals.Letters(['C',4])
sage: [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[(2, 1, 1),
 (-1, 1, -2),
 (3, 2, 2),
 (-2, 2, -3),
 (4, 3, 3),
 (-3, 3, -4),
 (-4, 4, 4)]
```

epsilon(i)
Return ε_i of self.

EXAMPLES:

```
sage: C = crystals.Letters(['C',3])
sage: [(c,i) for i in C.index_set() for c in C if c.epsilon(i) != 0]
[(2, 1), (-1, 1), (3, 2), (-2, 2), (-3, 3)]
```

f(i)
Return the action of f_i on self.

EXAMPLES:

```
sage: C = crystals.Letters(['C',4])
sage: [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[(1, 1, 2), (-2, 1, -1), (2, 2, 3),
 (-3, 2, -2), (3, 3, 4), (-4, 3, -3), (4, 4, -4)]
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['C',3])
sage: [(c,i) for i in C.index_set() for c in C if c.phi(i) != 0]
[(1, 1), (-2, 1), (2, 2), (-3, 2), (3, 3)]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Letters(['C',3])]
[(1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 0, -1), (0, -1, 0), (-1, 0, 0)]
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_D_element**

Bases: sage.combinat.crystals.letters.Letter

Type D crystal of letters elements.

TESTS:

```
sage: C = crystals.Letters(['D',4])
sage: C.list()
[1, 2, 3, 4, -4, -3, -2, -1]
sage: TestSuite(C).run()
```

e(*i*)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['D',5])
sage: [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[(2, 1, 1),
 (-1, 1, -2),
 (3, 2, 2),
 (-2, 2, -3),
 (4, 3, 3),
 (-3, 3, -4),
 (5, 4, 4),
 (-4, 4, -5),
 (-5, 5, 4),
 (-4, 5, 5)]
```

epsilon(*i*)

Return ε_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['D',4])
sage: [(c,i) for i in C.index_set() for c in C if c.epsilon(i) != 0]
[(2, 1), (-1, 1), (3, 2), (-2, 2), (4, 3), (-3, 3), (-4, 4), (-3, 4)]
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['D',5])
sage: [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[(1, 1, 2),
 (-2, 1, -1),
 (2, 2, 3),
 (-3, 2, -2),
 (3, 3, 4),
 (-4, 3, -3),
 (4, 4, 5),
 (-5, 4, -4),
 (4, 5, -5),
 (5, 5, -4)]
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['D',4])
sage: [(c,i) for i in C.index_set() for c in C if c.phi(i) != 0]
[(1, 1), (-2, 1), (2, 2), (-3, 2), (3, 3), (-4, 3), (3, 4), (4, 4)]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Letters(['D',4])]
[(1, 0, 0, 0),
 (0, 1, 0, 0),
 (0, 0, 1, 0),
 (0, 0, 0, 1),
 (0, 0, 0, -1),
 (0, 0, -1, 0),
 (0, -1, 0, 0),
 (-1, 0, 0, 0)]
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_E6_element**

Bases: sage.combinat.crystals.letters.LetterTuple

Type E_6 crystal of letters elements. This crystal corresponds to the highest weight crystal $B(\Lambda_1)$.

TESTS:

```
sage: C = crystals.Letters(['E',6])
sage: C.module_generators
((1,),)
sage: C.list()
[(1,), (-1, 3), (-3, 4), (-4, 2, 5), (-2, 5), (-5, 2, 6), (-2, -5, 4, 6),
(-4, 3, 6), (-3, 1, 6), (-1, 6), (-6, 2), (-2, -6, 4), (-4, -6, 3, 5),
(-3, -6, 1, 5), (-1, -6, 5), (-5, 3), (-3, -5, 1, 4), (-1, -5, 4), (-4, 1, 2),
(-1, -4, 2, 3), (-3, 2), (-2, -3, 4), (-4, 5), (-5, 6), (-6,), (-2, 1), (-1, -2, 3)]
sage: TestSuite(C).run()
sage: all(b.f(i).e(i) == b for i in C.index_set() for b in C if b.f(i) is not None)
True
sage: all(b.e(i).f(i) == b for i in C.index_set() for b in C if b.e(i) is not None)
True
```

```
sage: G = C.digraph()
sage: G.show(edge_labels=true, figsize=12, vertex_size=1)
```

e(*i*)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6])
sage: C((-1,3)).e(1)
(1,)
sage: C((-2,-3,4)).e(2)
(-3, 2)
sage: C((1,)).e(1)
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6])
sage: C((1,)).f(1)
(-1, 3)
sage: C((-6,)).f(1)
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Letters(['E',6])]
[(0, 0, 0, 0, 0, -2/3, -2/3, 2/3),
 (-1/2, 1/2, 1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
 (1/2, -1/2, 1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
 (1/2, 1/2, -1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
 (-1/2, -1/2, -1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
 (1/2, 1/2, 1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
 (-1/2, -1/2, 1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
 (-1/2, 1/2, -1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
 (1/2, -1/2, -1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
 (0, 0, 0, 0, 1, 1/3, 1/3, -1/3),
 (1/2, 1/2, 1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
 (-1/2, -1/2, 1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
 (-1/2, 1/2, -1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
 (1/2, -1/2, -1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
 (0, 0, 0, 1, 0, 1/3, 1/3, -1/3),
 (-1/2, 1/2, 1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
 (1/2, -1/2, 1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
 (0, 0, 1, 0, 0, 1/3, 1/3, -1/3),
 (1/2, 1/2, -1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
 (0, 1, 0, 0, 0, 1/3, 1/3, -1/3),
 (1, 0, 0, 0, 0, 1/3, 1/3, -1/3),
 (0, -1, 0, 0, 0, 1/3, 1/3, -1/3),
 (0, 0, -1, 0, 0, 1/3, 1/3, -1/3),
 (0, 0, 0, -1, 0, 1/3, 1/3, -1/3),
 (0, 0, 0, 0, -1, 1/3, 1/3, -1/3),
 (-1/2, -1/2, -1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
 (-1, 0, 0, 0, 0, 1/3, 1/3, -1/3)]
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_E6_element_dual**

Bases: sage.combinat.crystals.letters.LetterTuple

Type E_6 crystal of letters elements. This crystal corresponds to the highest weight crystal $B(\Lambda_6)$. This crystal is dual to $B(\Lambda_1)$ of type E_6.

TESTS:

```
sage: C = crystals.Letters(['E',6], dual = True)
sage: C.module_generators
((6,),)
sage: all(b==b.retract(b.lift()) for b in C)
True
sage: C.list()
[(6,), (5, -6), (4, -5), (2, 3, -4), (3, -2), (1, 2, -3), (2, -1), (1, 4, -2, -3),
 (4, -1, -2), (1, 5, -4), (3, 5, -1, -4), (5, -3), (1, 6, -5), (3, 6, -1, -5), (4, 6, -3, -5),
 (2, 6, -4), (6, -2), (1, -6), (3, -1, -6), (4, -3, -6), (2, 5, -4, -6), (5, -2, -6), (2, -5),
 (4, -2, -5), (3, -4), (1, -3), (-1,)]
sage: TestSuite(C).run()
sage: all(b.f(i).e(i) == b for i in C.index_set() for b in C if b.f(i) is not None)
True
sage: all(b.e(i).f(i) == b for i in C.index_set() for b in C if b.e(i) is not None)
True
sage: G = C.digraph()
sage: G.show(edge_labels=true, figsize=12, vertex_size=1)
```

e(*i*)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6], dual = True)
sage: C((-1,)).e(1)
(1, -3)
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6], dual = True)
sage: C((6,)).f(6)
(5, -6)
sage: C((6,)).f(1)
```

lift()

Lift an element of `self` to the crystal of letters `crystals.Letters(['E',6])` by taking its inverse weight.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6], dual = True)
sage: b = C.module_generators[0]
sage: b.lift()
(-6,)
```

retract(*p*)

Retract element `p`, which is an element in `crystals.Letters(['E',6])` to an element in `crystals.Letters(['E',6], dual=True)` by taking its inverse weight.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6])
sage: Cd = crystals.Letters(['E',6], dual = True)
sage: b = Cd.module_generators[0]
sage: p = C((-1,3))
sage: b.retract(p)
(1, -3)
sage: b.retract(None)
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6], dual = True)
sage: b=C.module_generators[0]
sage: b.weight()
(0, 0, 0, 0, 1, -1/3, -1/3, 1/3)
sage: [v.weight() for v in C]
[(0, 0, 0, 0, 1, -1/3, -1/3, 1/3),
(0, 0, 0, 1, 0, -1/3, -1/3, 1/3),
(0, 0, 1, 0, 0, -1/3, -1/3, 1/3),
(0, 1, 0, 0, 0, -1/3, -1/3, 1/3),
(-1, 0, 0, 0, 0, -1/3, -1/3, 1/3),
(1, 0, 0, 0, 0, -1/3, -1/3, 1/3),
(1/2, 1/2, 1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(0, -1, 0, 0, 0, -1/3, -1/3, 1/3),
(-1/2, -1/2, 1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(0, 0, -1, 0, 0, -1/3, -1/3, 1/3),
(-1/2, 1/2, -1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, -1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(0, 0, 0, -1, 0, -1/3, -1/3, 1/3),
(-1/2, 1/2, 1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, 1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(1/2, 1/2, -1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(-1/2, -1/2, -1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(0, 0, 0, 0, -1, -1/3, -1/3, 1/3),
(-1/2, 1/2, 1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, 1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, 1/2, -1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(-1/2, -1/2, -1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, 1/2, 1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(-1/2, -1/2, 1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(-1/2, 1/2, -1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, -1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(0, 0, 0, 0, 0, 2/3, 2/3, -2/3)]
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_E7_element**

Bases: sage.combinat.crystals.letters.LetterTuple

Type E_7 crystal of letters elements. This crystal corresponds to the highest weight crystal $B(\Lambda_7)$.

TESTS:

```
sage: C = crystals.Letters(['E',7])
sage: C.module_generators
((7,),)
sage: C.list()
[(7,), (-7, 6), (-6, 5), (-5, 4), (-4, 2, 3), (-2, 3), (-3, 1, 2), (-1,
2), (-3, -2, 1, 4), (-1, -2, 4), (-4, 1, 5), (-4, -1, 3, 5), (-3, 5),
(-5, 6, 1), (-5, -1, 3, 6), (-5, -3, 4, 6), (-4, 2, 6), (-2, 6), (-6, 7,
```

```
1), (-1, -6, 3, 7), (-6, -3, 7, 4), (-6, -4, 2, 7, 5), (-6, -2, 7, 5),
(-5, 7, 2), (-5, -2, 4, 7), (-4, 7, 3), (-3, 1, 7), (-1, 7), (-7, 1),
(-1, -7, 3), (-7, -3, 4), (-4, -7, 2, 5), (-7, -2, 5), (-5, -7, 6, 2),
(-5, -2, -7, 4, 6), (-7, -4, 6, 3), (-3, -7, 1, 6), (-7, -1, 6), (-6,
2), (-2, -6, 4), (-6, -4, 5, 3), (-3, -6, 1, 5), (-6, -1, 5), (-5, 3),
(-3, -5, 4, 1), (-5, -1, 4), (-4, 1, 2), (-1, -4, 3, 2), (-3, 2), (-2,
-3, 4), (-4, 5), (-5, 6), (-6, 7), (-7,), (-2, 1), (-2, -1, 3)]
sage: TestSuite(C).run()
sage: all(b.f(i).e(i) == b for i in C.index_set() for b in C if b.f(i) is not None)
True
sage: all(b.e(i).f(i) == b for i in C.index_set() for b in C if b.e(i) is not None)
True
sage: G = C.digraph()
sage: G.show(edge_labels=true, figsize=12, vertex_size=1)
```

e(*i*)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E',7])
sage: C((7,)).e(7)
sage: C((-7,6)).e(7)
(7,)
```

f(*i*)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E',7])
sage: C((-7,)).f(7)
sage: C((7,)).f(7)
(-7, 6)
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Letters(['E',7])]
[(0, 0, 0, 0, 0, 1, -1/2, 1/2), (0, 0, 0, 0, 1, 0, -1/2, 1/2), (0, 0, 0,
1, 0, 0, -1/2, 1/2), (0, 0, 1, 0, 0, 0, -1/2, 1/2), (0, 1, 0, 0, 0, 0,
-1/2, 1/2), (-1, 0, 0, 0, 0, 0, -1/2, 1/2), (1, 0, 0, 0, 0, 0, -1/2,
1/2), (1/2, 1/2, 1/2, 1/2, 1/2, 1/2, 0, 0), (0, -1, 0, 0, 0, 0, -1/2,
1/2), (-1/2, -1/2, 1/2, 1/2, 1/2, 1/2, 0, 0), (0, 0, -1, 0, 0, 0, -1/2,
1/2), (-1/2, 1/2, -1/2, 1/2, 1/2, 1/2, 0, 0), (1/2, -1/2, -1/2, 1/2,
1/2, 1/2, 0, 0), (0, 0, 0, -1, 0, 0, -1/2, 1/2), (-1/2, 1/2, 1/2, -1/2,
1/2, 1/2, 0, 0), (1/2, -1/2, 1/2, -1/2, 1/2, 1/2, 0, 0), (1/2, 1/2,
-1/2, -1/2, 1/2, 1/2, 0, 0), (-1/2, -1/2, -1/2, -1/2, 1/2, 1/2, 0, 0),
(0, 0, 0, 0, -1, 0, -1/2, 1/2), (-1/2, 1/2, 1/2, 1/2, -1/2, 1/2, 0, 0),
(1/2, -1/2, 1/2, 1/2, -1/2, 1/2, 0, 0), (1/2, 1/2, -1/2, 1/2, -1/2, 1/2,
0, 0), (-1/2, -1/2, -1/2, 1/2, -1/2, 1/2, 0, 0), (1/2, 1/2, 1/2, -1/2,
-1/2, 1/2, 0, 0), (-1/2, -1/2, 1/2, -1/2, -1/2, 1/2, 0, 0), (-1/2, 1/2,
-1/2, -1/2, -1/2, 1/2, 0, 0), (1/2, -1/2, -1/2, -1/2, -1/2, 1/2, 0, 0),
(0, 0, 0, 0, 0, 1, 1/2, -1/2), (0, 0, 0, 0, 0, -1, -1/2, 1/2), (-1/2,
1/2, 1/2, 1/2, 1/2, -1/2, 0, 0), (1/2, -1/2, 1/2, 1/2, 1/2, -1/2, 0, 0),
(1/2, 1/2, -1/2, 1/2, 1/2, -1/2, 0, 0), (-1/2, -1/2, -1/2, 1/2, 1/2,
-1/2, 0, 0), (1/2, 1/2, 1/2, -1/2, 1/2, -1/2, 0, 0), (-1/2, -1/2, 1/2,
-1/2, 1/2, -1/2, 0, 0), (-1/2, 1/2, -1/2, -1/2, 1/2, -1/2, 0, 0), (1/2,
```

```
-1/2, -1/2, -1/2, 1/2, -1/2, 0, 0), (0, 0, 0, 0, 1, 0, 1/2, -1/2), (1/2,
1/2, 1/2, 1/2, -1/2, -1/2, 0, 0), (-1/2, -1/2, 1/2, 1/2, -1/2, -1/2, 0,
0), (-1/2, 1/2, -1/2, 1/2, -1/2, -1/2, 0, 0), (1/2, -1/2, -1/2, 1/2,
-1/2, -1/2, 0, 0), (0, 0, 0, 1, 0, 0, 1/2, -1/2), (-1/2, 1/2, 1/2, -1/2,
-1/2, -1/2, 0, 0), (1/2, -1/2, 1/2, -1/2, -1/2, -1/2, 0, 0), (0, 0, 1,
0, 0, 0, 1/2, -1/2), (1/2, 1/2, -1/2, -1/2, -1/2, -1/2, 0, 0), (0, 1, 0,
0, 0, 0, 1/2, -1/2), (1, 0, 0, 0, 0, 0, 1/2, -1/2), (0, -1, 0, 0, 0, 0,
1/2, -1/2), (0, 0, -1, 0, 0, 0, 1/2, -1/2), (0, 0, 0, -1, 0, 0, 1/2,
-1/2), (0, 0, 0, 0, -1, 0, 1/2, -1/2), (0, 0, 0, 0, 0, -1, 1/2, -1/2),
(-1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 0, 0), (-1, 0, 0, 0, 0, 0, 1/2,
-1/2)]
```

class sage.combinat.crystals.letters.**Crystal_of_letters_type_G_element**

Bases: sage.combinat.crystals.letters.Letter

Type G_2 crystal of letters elements.

TESTS:

```
sage: C = crystals.Letters(['G',2])
sage: C.list()
[1, 2, 3, 0, -3, -2, -1]
sage: TestSuite(C).run()
```

e (i)

Return the action of e_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['G',2])
sage: [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[(2, 1, 1),
 (0, 1, 3),
 (-3, 1, 0),
 (-1, 1, -2),
 (3, 2, 2),
 (-2, 2, -3)]
```

epsilon (i)

Return ε_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['G',2])
sage: [(c,i,c.epsilon(i)) for i in C.index_set() for c in C if c.epsilon(i) != 0]
[(2, 1, 1), (0, 1, 1), (-3, 1, 2), (-1, 1, 1), (3, 2, 1), (-2, 2, 1)]
```

f (i)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['G',2])
sage: [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[(1, 1, 2),
 (3, 1, 0),
 (0, 1, -3),
 (-2, 1, -1),
 (2, 2, 3),
 (-3, 2, -2)]
```

phi(*i*)
Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['G',2])
sage: [(c,i,c.phi(i)) for i in C.index_set() for c in C if c.phi(i) != 0]
[(1, 1, 1), (3, 1, 2), (0, 1, 1), (-2, 1, 1), (2, 2, 1), (-3, 2, 1)]
```

weight()
Return the weight of `self`.

EXAMPLES:

```
sage: [v.weight() for v in crystals.Letters(['G',2])]
[(1, 0, -1), (1, -1, 0), (0, 1, -1), (0, 0, 0), (0, -1, 1), (-1, 1, 0), (-1, 0, 1)]
```

class sage.combinat.crystals.letters.**EmptyLetter**
Bases: sage.structure.element.Element

The affine letter $\emptyset$ thought of as a classical crystal letter in classical type B_n and C_n.

Warning: This is not a classical letter.

Used in the rigged configuration bijections.

e(*i*)
Return e_i of `self` which is `None`.

EXAMPLES:

```
sage: C = crystals.Letters(['C', 3])
sage: C('E').e(1)
```

epsilon(*i*)
Return ε_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['C', 3])
sage: C('E').epsilon(1)
0
```

f(*i*)
Return f_i of `self` which is `None`.

EXAMPLES:

```
sage: C = crystals.Letters(['C', 3])
sage: C('E').f(1)
```

phi(*i*)
Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['C', 3])
sage: C('E').phi(1)
0
```

value

weight()
Return the weight of self.

EXAMPLES:

```
sage: C = crystals.Letters(['C', 3])
sage: C('E').weight()
(0, 0, 0)
```

class sage.combinat.crystals.letters.**Letter**
Bases: sage.structure.element.Element

A class for letters.

Like ElementWrapper, plus delegates __lt__ (comparison) to the parent.

EXAMPLES:

```
sage: from sage.combinat.crystals.letters import Letter
sage: a = Letter(ZZ, 1)
sage: Letter(ZZ, 1).parent()
Integer Ring

sage: Letter(ZZ, 1)._repr_()
'1'

sage: parent1 = ZZ  # Any fake value ...
sage: parent2 = QQ  # Any fake value ...
sage: l11 = Letter(parent1, 1)
sage: l12 = Letter(parent1, 2)
sage: l21 = Letter(parent2, 1)
sage: l22 = Letter(parent2, 2)
sage: l11 == l11
True
sage: l11 == l12
False
sage: l11 == l21 # not tested
False

sage: C = crystals.Letters(['B', 3])
sage: C(0) != C(0)
False
sage: C(1) != C(-1)
True
```

value

class sage.combinat.crystals.letters.**LetterTuple**
Bases: sage.structure.element.Element

Abstract class for type E letters.

epsilon(i)
Return ε_i of self.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6])
sage: C((-6,)).epsilon(1)
0
sage: C((-6,)).epsilon(6)
1
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['E',6])
sage: C((1,)).phi(1)
1
sage: C((1,)).phi(6)
0
```

value

5.1.54 Littelmann paths

AUTHORS:

- Mark Shimozono, Anne Schilling (2012): Initial version
- Anne Schilling (2013): Implemented `CrystalOfProjectedLevelZeroLSPaths`
- Travis Scrimshaw (2016): Implemented `InfinityCrystalOfLSPaths`

class sage.combinat.crystals.littelmann_path.**CrystalOfLSPaths**(*starting_weight*, *starting_weight_parent*)

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.Parent`

Crystal graph of LS paths generated from the straight-line path to a given weight.

INPUT:

- `cartan_type` – (optional) the Cartan type of a finite or affine root system
- `starting_weight` – a weight; if `cartan_type` is given, then the weight should be given as a list of coefficients of the fundamental weights, otherwise it should be given in the `weight_space` basis; for affine highest weight crystals, one needs to use the extended weight space.

The crystal class of piecewise linear paths in the weight space, generated from a straight-line path from the origin to a given element of the weight lattice.

OUTPUT:

- a tuple of weights defining the directions of the piecewise linear segments

EXAMPLES:

```
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space(extended = True).basis()
sage: B = crystals.LSPaths(La[2]-La[0]); B
The crystal of LS paths of type ['A', 2, 1] and weight -Lambda[0] + Lambda[2]

sage: C = crystals.LSPaths(['A',2,1],[-1,0,1]); C
The crystal of LS paths of type ['A', 2, 1] and weight -Lambda[0] + Lambda[2]
sage: B == C
True
sage: c = C.module_generators[0]; c
(-Lambda[0] + Lambda[2],)
sage: [c.f(i) for i in C.index_set()]
[None, None, (Lambda[1] - Lambda[2],)]

sage: R = C.R; R
```

```
Root system of type ['A', 2, 1]
sage: Lambda = R.weight_space().basis(); Lambda
Finite family {0: Lambda[0], 1: Lambda[1], 2: Lambda[2]}
sage: b=C(tuple([-Lambda[0]+Lambda[2]]))
sage: b==c
True
sage: b.f(2)
(Lambda[1] - Lambda[2],)
```

For classical highest weight crystals we can also compare the results with the tableaux implementation:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: sorted(C, key=str)
[(-2*Lambda[1] + Lambda[2],), (-Lambda[1] + 1/2*Lambda[2], Lambda[1] - 1/2*Lambda[2]),
 (-Lambda[1] + 2*Lambda[2],), (-Lambda[1] - Lambda[2],),
 (1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2]), (2*Lambda[1] - Lambda[2],),
 (Lambda[1] + Lambda[2],), (Lambda[1] - 2*Lambda[2],)]
sage: C.cardinality()
8
sage: B = crystals.Tableaux(['A',2],shape=[2,1])
sage: B.cardinality()
8
sage: B.digraph().is_isomorphic(C.digraph())
True
```

Make sure you use the weight space and not the weight lattice for your weights:

```
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_lattice(extended = True).basis()
sage: B = crystals.LSPaths(La[2]); B
Traceback (most recent call last):
...
ValueError: Please use the weight space, rather than weight lattice for your weights
```

REFERENCES:

class Element

Bases: sage.structure.element_wrapper.ElementWrapper

TESTS:

```
sage: C = crystals.LSPaths(['E',6],[1,0,0,0,0,0])
sage: c = C.an_element()
sage: TestSuite(c).run()
```

compress()

Merges consecutive positively parallel steps present in the path.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: Lambda = C.R.weight_space().fundamental_weights(); Lambda
Finite family {1: Lambda[1], 2: Lambda[2]}
sage: c = C(tuple([1/2*Lambda[1]+1/2*Lambda[2], 1/2*Lambda[1]+1/2*Lambda[2]]))
sage: c.compress()
(Lambda[1] + Lambda[2],)
```

dualize()

Returns dualized path.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: for c in C:
...       print c, c.dualize()
...
(Lambda[1] + Lambda[2],) (-Lambda[1] - Lambda[2],)
(-Lambda[1] + 2*Lambda[2],) (Lambda[1] - 2*Lambda[2],)
(1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2]) (1/2*Lambda[1] - Lambda[2], -1/2
(Lambda[1] - 2*Lambda[2],) (-Lambda[1] + 2*Lambda[2],)
(-Lambda[1] - Lambda[2],) (Lambda[1] + Lambda[2],)
(2*Lambda[1] - Lambda[2],) (-2*Lambda[1] + Lambda[2],)
(-Lambda[1] + 1/2*Lambda[2], Lambda[1] - 1/2*Lambda[2]) (-Lambda[1] + 1/2*Lambda[2], Lam
(-2*Lambda[1] + Lambda[2],) (2*Lambda[1] - Lambda[2],)
```

e(*i*, *power=1*, *to_string_end=False*, *length_only=False*)

Returns the i-th crystal raising operator on `self`.

INPUT:

- `i` – element of the index set of the underlying root system
- `power` – positive integer; specifies the power of the raising operator to be applied (default: 1)
- `to_string_end` – boolean; if set to True, returns the dominant end of the i-string of `self`. (default: False)
- `length_only` – boolean; if set to True, returns the distance to the dominant end of the i-string of `self`.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: c = C[2]; c
(1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2])
sage: c.e(1)
sage: c.e(2)
(-Lambda[1] + 2*Lambda[2],)
sage: c.e(2,to_string_end=True)
(-Lambda[1] + 2*Lambda[2],)
sage: c.e(1,to_string_end=True)
(1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2])
sage: c.e(1,length_only=True)
0
```

endpoint()

Computes the endpoint of the path.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: b = C.module_generators[0]
sage: b.endpoint()
Lambda[1] + Lambda[2]
sage: b.f_string([1,2,2,1])
(-Lambda[1] - Lambda[2],)
sage: b.f_string([1,2,2,1]).endpoint()
-Lambda[1] - Lambda[2]
sage: b.f_string([1,2])
(1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2])
sage: b.f_string([1,2]).endpoint()
0
sage: b = C([])
sage: b.endpoint()
0
```

epsilon(*i*)

Returns the distance to the beginning of the i-string.

This method overrides the generic implementation in the category of crystals since this computation is more efficient.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: [c.epsilon(1) for c in C]
[0, 1, 0, 0, 1, 0, 1, 2]
sage: [c.epsilon(2) for c in C]
[0, 0, 1, 2, 1, 1, 0, 0]
```

f(*i*, *power=1*, *to_string_end=False*, *length_only=False*)

Returns the i-th crystal lowering operator on `self`.

INPUT:

- `i` – element of the index set of the underlying root system
- `power` – positive integer; specifies the power of the lowering operator to be applied (default: 1)
- `to_string_end` – boolean; if set to True, returns the anti-dominant end of the i-string of `self`. (default: False)
- `length_only` – boolean; if set to True, returns the distance to the anti-dominant end of the i-string of `self`.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: c = C.module_generators[0]
sage: c.f(1)
(-Lambda[1] + 2*Lambda[2],)
sage: c.f(1,power=2)
sage: c.f(2)
(2*Lambda[1] - Lambda[2],)
sage: c.f(2,to_string_end=True)
(2*Lambda[1] - Lambda[2],)
sage: c.f(2,length_only=True)
1

sage: C = crystals.LSPaths(['A',2,1],[-1,-1,2])
sage: c = C.module_generators[0]
sage: c.f(2,power=2)
(Lambda[0] + Lambda[1] - 2*Lambda[2],)
```

phi(*i*)

Returns the distance to the end of the i-string.

This method overrides the generic implementation in the category of crystals since this computation is more efficient.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: [c.phi(1) for c in C]
[1, 0, 0, 1, 0, 2, 1, 0]
sage: [c.phi(2) for c in C]
[1, 2, 1, 0, 0, 0, 0, 1]
```

reflect_step(*which_step*, *i*)

Apply the i-th simple reflection to the indicated step in `self`.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: b = C.module_generators[0]
```

```
sage: b.reflect_step(0,1)
(-Lambda[1] + 2*Lambda[2],)
sage: b.reflect_step(0,2)
(2*Lambda[1] - Lambda[2],)
```

s(*i*)

Computes the reflection of `self` along the i-string.

This method is more efficient than the generic implementation since it uses powers of e and f in the Littelmann model directly.

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: c = C.module_generators[0]
sage: c.s(1)
(-Lambda[1] + 2*Lambda[2],)
sage: c.s(2)
(2*Lambda[1] - Lambda[2],)

sage: C = crystals.LSPaths(['A',2,1],[-1,0,1])
sage: c = C.module_generators[0]; c
(-Lambda[0] + Lambda[2],)
sage: c.s(2)
(Lambda[1] - Lambda[2],)
sage: c.s(1)
(-Lambda[0] + Lambda[2],)
sage: c.f(2).s(1)
(Lambda[0] - Lambda[1],)
```

split_step(*which_step*, *r*)

Splits indicated step into two parallel steps of relative lengths r and $1-r$.

INPUT:

- `which_step` – a position in the tuple `self`
- `r` – a rational number between 0 and 1

EXAMPLES:

```
sage: C = crystals.LSPaths(['A',2],[1,1])
sage: b = C.module_generators[0]
sage: b.split_step(0,1/3)
(1/3*Lambda[1] + 1/3*Lambda[2], 2/3*Lambda[1] + 2/3*Lambda[2])
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: B = crystals.LSPaths(['A',1,1],[1,0])
sage: b = B.highest_weight_vector()
sage: b.f(0).weight()
-Lambda[0] + 2*Lambda[1] - delta
```

CrystalOfLSPaths.**weight_lattice_realization**()

Return weight lattice realization of `self`.

EXAMPLES:

```
sage: B = crystals.LSPaths(['B',3],[1,1,0])
sage: B.weight_lattice_realization()
Weight space over the Rational Field of the Root system of type ['B', 3]
sage: B = crystals.LSPaths(['B',3,1],[1,1,1,0])
```

```
sage: B.weight_lattice_realization()
Extended weight space over the Rational Field of the Root system of type ['B', 3, 1]
```

class sage.combinat.crystals.littelmann_path.**CrystalOfProjectedLevelZeroLSPaths**(*starting_weight*, *starting_weight_parent*

Bases: sage.combinat.crystals.littelmann_path.CrystalOfLSPaths

Crystal of projected level zero LS paths.

INPUT:

- `weight` – a dominant weight of the weight space of an affine Kac-Moody root system

When `weight` is just a single fundamental weight Λ_r, this crystal is isomorphic to a Kirillov-Reshetikhin (KR) crystal, see also `sage.combinat.crystals.kirillov_reshetikhin.KirillovReshetikhinFromLSPaths()`. For general weights, it is isomorphic to a tensor product of single-column KR crystals.

EXAMPLES:

```
sage: R = RootSystem(['C',3,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[3])
sage: LS.cardinality()
84
sage: GLS = LS.digraph()

sage: K1 = crystals.KirillovReshetikhin(['C',3,1],1,1)
sage: K3 = crystals.KirillovReshetikhin(['C',3,1],3,1)
sage: T = crystals.TensorProduct(K3,K1)
sage: T.cardinality()
84
sage: GT = T.digraph() # long time
sage: GLS.is_isomorphic(GT, edge_labels = True) # long time
True
```

TESTS:

```
sage: ct = CartanType(['A',4,2]).dual()
sage: P = RootSystem(ct).weight_space()
sage: La = P.fundamental_weights()
sage: C = crystals.ProjectedLevelZeroLSPaths(La[1])
sage: sorted(C, key=str)
[(-Lambda[0] + Lambda[1],),
 (-Lambda[1] + 2*Lambda[2],),
 (1/2*Lambda[1] - Lambda[2], -1/2*Lambda[1] + Lambda[2]),
 (Lambda[0] - Lambda[1],),
 (Lambda[1] - 2*Lambda[2],)]
```

class Element

Bases: sage.combinat.crystals.littelmann_path.CrystalOfLSPaths.Element

Element of a crystal of projected level zero LS paths.

energy_function()

Return the energy function of `self`.

The energy function $D(\pi)$ of the level zero LS path $\pi \in \mathbb{B}_{\mathrm{cl}}(\lambda)$ requires a series of definitions; for simplicity the root system is assumed to be untwisted affine.

The LS path π is a piecewise linear map from the unit interval $[0,1]$ to the weight lattice. It is specified by "times" $0 = \sigma_0 < \sigma_1 < \cdots < \sigma_s = 1$ and "direction vectors" $x_u\lambda$ where $x_u \in W/W_J$ for $1 \leq u \leq s$, and W_J is the stabilizer of λ in the finite Weyl group W. Precisely,

$$\pi(t) = \sum_{u'=1}^{u-1} (\sigma_{u'} - \sigma_{u'-1}) x_{u'}\lambda + (t - \sigma_{u-1}) x_u \lambda$$

for $1 \leq u \leq s$ and $\sigma_{u-1} \leq t \leq \sigma_u$.

For any $x, y \in W/W_J$ let

$$d : x = w_0 \overset{\beta_1}{\leftarrow} w_1 \overset{\beta_2}{\leftarrow} \cdots \overset{\beta_n}{\leftarrow} w_n = y$$

be a shortest directed path in the parabolic quantum Bruhat graph. Define

$$\mathrm{wt}(d) := \sum_{\substack{1 \leq k \leq n \\ \ell(w_{k-1}) < \ell(w_k)}} \beta_k^\vee$$

It can be shown that $\mathrm{wt}(d)$ depends only on x, y; call its value $\mathrm{wt}(x, y)$. The energy function $D(\pi)$ is defined by

$$D(\pi) = -\sum_{u=1}^{s-1} (1 - \sigma_u) \langle \lambda, \mathrm{wt}(x_u, x_{u+1}) \rangle$$

For more information, see [LNSSS2013].

REFERENCES:

Note: In the dual-of-untwisted case the parabolic quantum Bruhat graph that is used is obtained by exchanging the roles of roots and coroots. Moreover, in the computation of the pairing the short roots must be doubled (or tripled for type G). This factor is determined by the translation factor of the corresponding root. Type BC is viewed as untwisted type, whereas the dual of BC is viewed as twisted. Except for the untwisted cases, these formulas are currently still conjectural.

EXAMPLES:

```
sage: R = RootSystem(['C',3,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[3])
sage: b = LS.module_generators[0]
sage: c = b.f(1).f(3).f(2)
sage: c.energy_function()
0
sage: c=b.e(0)
sage: c.energy_function()
1

sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: b = LS.module_generators[0]
sage: c = b.e(0)
sage: c.energy_function()
1
sage: for c in sorted(LS, key=str): print c,c.energy_function()
(-2*Lambda[0] + 2*Lambda[1],)                    0
(-2*Lambda[1] + 2*Lambda[2],)                    0
```

```
(-Lambda[0] + Lambda[1], -Lambda[1] + Lambda[2]) 1
(-Lambda[0] + Lambda[1], Lambda[0] - Lambda[2])  1
(-Lambda[1] + Lambda[2], -Lambda[0] + Lambda[1]) 0
(-Lambda[1] + Lambda[2], Lambda[0] - Lambda[2])  1
(2*Lambda[0] - 2*Lambda[2],)                     0
(Lambda[0] - Lambda[2], -Lambda[0] + Lambda[1])  0
(Lambda[0] - Lambda[2], -Lambda[1] + Lambda[2])  0
```

The next test checks that the energy function is constant on classically connected components:

```
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1]+La[2])
sage: G = LS.digraph(index_set=[1,2])
sage: C = G.connected_components()
sage: [all(c[0].energy_function()==a.energy_function() for a in c) for c in C]
[True, True, True, True]

sage: R = RootSystem(['D',4,2])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[2])
sage: J = R.cartan_type().classical().index_set()
sage: hw = [x for x in LS if x.is_highest_weight(J)]
sage: [(x.weight(), x.energy_function()) for x in hw]
[(-2*Lambda[0] + Lambda[2], 0), (-2*Lambda[0] + Lambda[1], 1), (0, 2)]
sage: G = LS.digraph(index_set=J)
sage: C = G.connected_components()
sage: [all(c[0].energy_function()==a.energy_function() for a in c) for c in C]
[True, True, True]

sage: R = RootSystem(CartanType(['G',2,1]).dual())
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[2])
sage: G = LS.digraph(index_set=[1,2])
sage: C = G.connected_components()
sage: [all(c[0].energy_function()==a.energy_function() for a in c) for c in C] # long ti
[True, True, True, True, True, True, True, True, True, True, True, True, True, Tru

sage: ct = CartanType(['BC',2,2]).dual()
sage: R = RootSystem(ct)
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1]+La[2])
sage: G = LS.digraph(index_set=R.cartan_type().classical().index_set())
sage: C = G.connected_components()
sage: [all(c[0].energy_function()==a.energy_function() for a in c) for c in C] # long ti
[True, True, True, True, True, True, True, True, True, True, True]

sage: R = RootSystem(['BC',2,2])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1]+La[2])
sage: G = LS.digraph(index_set=R.cartan_type().classical().index_set())
sage: C = G.connected_components()
sage: [all(c[0].energy_function()==a.energy_function() for a in c) for c in C] # long ti
[True, True, True, True, True, True, True, True, True, True, True, True, True, Tru
True, True, True, True, True, True, True, True, True, True, True, True, True, True
```

scalar_factors()

Obtain the scalar factors for `self`.

Each LS path (or `self`) can be written as a piecewise linear map

$$\pi(t) = \sum_{u'=1}^{u-1} (\sigma_{u'} - \sigma_{u'-1})\nu_{u'} + (t - \sigma_{u-1})\nu_u$$

for $0 < \sigma_1 < \sigma_2 < \cdots < \sigma_s = 1$ and $\sigma_{u-1} \leq t \leq \sigma_u$ and $1 \leq u \leq s$. This method returns the tuple of $(\sigma_1, \ldots, \sigma_s)$.

EXAMPLES:

```
sage: R = RootSystem(['C',3,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[3])
sage: b = LS.module_generators[0]
sage: b.scalar_factors()
[1]
sage: c = b.f(1).f(3).f(2)
sage: c.scalar_factors()
[1/3, 1]
```

weyl_group_representation()

Transforms the weights in the LS path `self` to elements in the Weyl group.

Each LS path can be written as the piecewise linear map:

$$\pi(t) = \sum_{u'=1}^{u-1} (\sigma_{u'} - \sigma_{u'-1})\nu_{u'} + (t - \sigma_{u-1})\nu_u$$

for $0 < \sigma_1 < \sigma_2 < \cdots < \sigma_s = 1$ and $\sigma_{u-1} \leq t \leq \sigma_u$ and $1 \leq u \leq s$. Each weight ν_u is also associated to a Weyl group element. This method returns the list of Weyl group elements associated to the ν_u for $1 \leq u \leq s$.

EXAMPLES:

```
sage: R = RootSystem(['C',3,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[3])
sage: b = LS.module_generators[0]
sage: c = b.f(1).f(3).f(2)
sage: c.weyl_group_representation()
[s2*s3*s1, s3*s1]
```

CrystalOfProjectedLevelZeroLSPaths.**is_perfect**(*level=1*)

Checks whether the crystal `self` is perfect (of level `level`).

INPUT:

- `level` – (default: 1) positive integer

A crystal $\mathcal{B}$ is perfect of level ℓ if:

1. $\mathcal{B}$ is isomorphic to the crystal graph of a finite-dimensional $U_q'(\mathfrak{g})$-module.
2. $\mathcal{B} \otimes \mathcal{B}$ is connected.
3. There exists a $\lambda \in X$, such that $\mathrm{wt}(\mathcal{B}) \subset \lambda + \sum_{i \in I} \mathbb{Z}_{\leq 0}\alpha_i$ and there is a unique element in $\mathcal{B}$ of classical weight λ.
4. $\forall b \in \mathcal{B}, \mathrm{level}(\varepsilon(b)) \geq \ell$.
5. $\forall \Lambda$ dominant weights of level ℓ, there exist unique elements $b_\Lambda, b^\Lambda \in \mathcal{B}$, such that $\varepsilon(b_\Lambda) = \Lambda = \varphi(b^\Lambda)$.

Points (1)-(3) are known to hold. This method checks points (4) and (5).

EXAMPLES:

```
sage: C = CartanType(['C',2,1])
sage: R = RootSystem(C)
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1])
sage: LS.is_perfect()
False
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[2])
sage: LS.is_perfect()
True

sage: C = CartanType(['E',6,1])
sage: R = RootSystem(C)
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1])
sage: LS.is_perfect()
True
sage: LS.is_perfect(2)
False

sage: C = CartanType(['D',4,1])
sage: R = RootSystem(C)
sage: La = R.weight_space().basis()
sage: all(crystals.ProjectedLevelZeroLSPaths(La[i]).is_perfect() for i in [1,2,3,4])
True

sage: C = CartanType(['A',6,2])
sage: R = RootSystem(C)
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[2])
sage: LS.is_perfect()
True
sage: LS.is_perfect(2)
False
```

CrystalOfProjectedLevelZeroLSPaths.**one_dimensional_configuration_sum**(*q=None*, *group_components=True*)

Compute the one-dimensional configuration sum.

INPUT:

- `q` – (default: `None`) a variable or `None`; if `None`, a variable `q` is set in the code
- `group_components` – (default: `True`) boolean; if `True`, then the terms are grouped by classical component

The one-dimensional configuration sum is the sum of the weights of all elements in the crystal weighted by the energy function. For untwisted types it uses the parabolic quantum Bruhat graph, see [LNSSS2013]. In the dual-of-untwisted case, the parabolic quantum Bruhat graph is defined by exchanging the roles of roots and coroots (which is still conjectural at this point).

EXAMPLES:

```
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: LS.one_dimensional_configuration_sum() # long time
B[-2*Lambda[1] + 2*Lambda[2]] + (q+1)*B[-Lambda[1]]
+ (q+1)*B[Lambda[1] - Lambda[2]] + B[2*Lambda[1]] + B[-2*Lambda[2]] + (q+1)*B[Lambda[2]]
```

```
sage: R.<t> = ZZ[]
sage: LS.one_dimensional_configuration_sum(t, False) # long time
B[-2*Lambda[1] + 2*Lambda[2]] + (t+1)*B[-Lambda[1]] + (t+1)*B[Lambda[1] - Lambda[2]]
+ B[2*Lambda[1]] + B[-2*Lambda[2]] + (t+1)*B[Lambda[2]]
```

TESTS:

```
sage: R = RootSystem(['B',3,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[2])
sage: LS.one_dimensional_configuration_sum() == LS.one_dimensional_configuration_sum(group_c
True
sage: K1 = crystals.KirillovReshetikhin(['B',3,1],1,1)
sage: K2 = crystals.KirillovReshetikhin(['B',3,1],2,1)
sage: T = crystals.TensorProduct(K2,K1)
sage: T.one_dimensional_configuration_sum() == LS.one_dimensional_configuration_sum() # long
True

sage: R = RootSystem(['D',4,2])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[2])
sage: K1 = crystals.KirillovReshetikhin(['D',4,2],1,1)
sage: K2 = crystals.KirillovReshetikhin(['D',4,2],2,1)
sage: T = crystals.TensorProduct(K2,K1)
sage: T.one_dimensional_configuration_sum() == LS.one_dimensional_configuration_sum() # long
True

sage: R = RootSystem(['A',5,2])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(3*La[1])
sage: K1 = crystals.KirillovReshetikhin(['A',5,2],1,1)
sage: T = crystals.TensorProduct(K1,K1,K1)
sage: T.one_dimensional_configuration_sum() == LS.one_dimensional_configuration_sum() # long
True
```

class sage.combinat.crystals.littelmann_path.**InfinityCrystalOfLSPaths**(*cartan_type*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

LS path model for $\mathcal{B}(\infty)$.

Elements of $\mathcal{B}(\infty)$ are equivalence classes of paths $[\pi]$ in $\mathcal{B}(k\rho)$ for $k \gg 0$, where ρ is the Weyl vector. A canonical representative for an element of $\mathcal{B}(\infty)$ is chosen by taking k to be minimal such that the endpoint of π is strictly dominant but its representative in $\mathcal{B}((k-1)\rho)$ is on the wall of the dominant chamber.

REFERENCES:

class Element

Bases: sage.combinat.crystals.littelmann_path.CrystalOfLSPaths.Element

e(*i*, *power=1*, *length_only=False*)

Return the i-th crystal raising operator on `self`.

INPUT:

- `i` – element of the index set
- `power` – (default: 1) positive integer; specifies the power of the lowering operator to be applied
- `length_only` – (default: `False`) boolean; if `True`, then return the distance to the anti-dominant end of the i-string of `self`

EXAMPLES:

```
sage: B = crystals.infinity.LSPaths(['B',3,1])
sage: mg = B.module_generator()
sage: mg.e(0)
sage: mg.e(1)
sage: mg.e(2)
sage: x = mg.f_string([1,0,2,1,0,2,1,1,0])
sage: all(x.f(i).e(i) == x for i in B.index_set())
True
sage: all(x.e(i).f(i) == x for i in B.index_set() if x.epsilon(i) > 0)
True
```

TESTS:

Check that this works in affine types:

```
sage: B = crystals.infinity.LSPaths(['A',3,1])
sage: mg = B.highest_weight_vector()
sage: x = mg.f_string([0,1,2,3])
sage: x.e_string([3,2,1,0]) == mg
True
```

We check that `epsilon()` works:

```
sage: B = crystals.infinity.LSPaths(['D',4])
sage: mg = B.highest_weight_vector()
sage: x = mg.f_string([1,3,4,2,4,3,2,1,4])
sage: [x.epsilon(i) for i in B.index_set()]
[1, 1, 0, 1]
```

f (*i*, *power=1*, *length_only=False*)

Return the i-th crystal lowering operator on `self`.

INPUT:

- `i` – element of the index set
- `power` – (default: 1) positive integer; specifies the power of the lowering operator to be applied
- `length_only` – (default: `False`) boolean; if `True`, then return the distance to the anti-dominant end of the i-string of `self`

EXAMPLES:

```
sage: B = crystals.infinity.LSPaths(['D',3,2])
sage: mg = B.highest_weight_vector()
sage: mg.f(1)
(3*Lambda[0] - Lambda[1] + 3*Lambda[2],
 2*Lambda[0] + 2*Lambda[1] + 2*Lambda[2])
sage: mg.f(2)
(Lambda[0] + 2*Lambda[1] - Lambda[2],
 2*Lambda[0] + 2*Lambda[1] + 2*Lambda[2])
sage: mg.f(0)
(-Lambda[0] + 2*Lambda[1] + Lambda[2] - delta,
 2*Lambda[0] + 2*Lambda[1] + 2*Lambda[2])
```

phi (*i*)

Return φ_i of `self`.

Let $\pi \in \mathcal{B}(\infty)$. Define

$$\varphi_i(\pi) := \varepsilon_i(\pi) + \langle h_i, \mathrm{wt}(\pi) \rangle,$$

where h_i is the i-th simple coroot and $\mathrm{wt}(\pi)$ is the `weight()` of π.

INPUT:

- `i` – element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.LSPaths(['D',4])
sage: mg = B.highest_weight_vector()
sage: x = mg.f_string([1,3,4,2,4,3,2,1,4])
sage: [x.phi(i) for i in B.index_set()]
[-1, 4, -2, -3]
```

weight()

Return the weight of `self`.

Todo

This is a generic algorithm. We should find a better description and implement it.

EXAMPLES:

```
sage: B = crystals.infinity.LSPaths(['E',6])
sage: mg = B.highest_weight_vector()
sage: f_seq = [1,4,2,6,4,2,3,1,5,5]
sage: x = mg.f_string(f_seq)
sage: x.weight()
-3*Lambda[1] - 2*Lambda[2] + 2*Lambda[3] + Lambda[4] - Lambda[5]

sage: al = B.cartan_type().root_system().weight_space().simple_roots()
sage: x.weight() == -sum(al[i] for i in f_seq)
True
```

InfinityCrystalOfLSPaths.**module_generator**()

Return the module generator (or highest weight element) of `self`.

The module generator is the unique path $\pi_\infty : t \mapsto t\rho$, for $t \in [0, \infty)$.

EXAMPLES:

```
sage: B = crystals.infinity.LSPaths(['A',6,2])
sage: mg = B.module_generator(); mg
(Lambda[0] + Lambda[1] + Lambda[2] + Lambda[3],)
sage: mg.weight()
0
```

InfinityCrystalOfLSPaths.**weight_lattice_realization**()

Return the weight lattice realization of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.LSPaths(['C',4])
sage: B.weight_lattice_realization()
Weight space over the Rational Field of the Root system of type ['C', 4]
```

sage.combinat.crystals.littelmann_path.**positively_parallel_weights**(v, w)

Check whether the vectors `v` and `w` are positive scalar multiples of each other.

EXAMPLES:

```
sage: from sage.combinat.crystals.littelmann_path import positively_parallel_weights
sage: La = RootSystem(['A',5,2]).weight_space(extended=True).fundamental_weights()
sage: rho = sum(La)
sage: positively_parallel_weights(rho, 4*rho)
True
sage: positively_parallel_weights(4*rho, rho)
True
sage: positively_parallel_weights(rho, -rho)
```

```
False
sage: positively_parallel_weights(rho, La[1] + La[2])
False
```

5.1.55 Crystals of Modified Nakajima Monomials

AUTHORS:

- Arthur Lubovsky: Initial version
- Ben Salisbury: Initial version

Let $Y_{i,k}$, for $i \in I$ and $k \in \mathbf{Z}$, be a commuting set of variables, and let $\mathbf{1}$ be a new variable which commutes with each $Y_{i,k}$. (Here, I represents the index set of a Cartan datum.) One may endow the structure of a crystal on the set $\widehat{\mathcal{M}}$ of monomials of the form

$$M = \prod_{(i,k) \in I \times \mathbf{Z}_{\geq 0}} Y_{i,k}^{y_i(k)} \mathbf{1}.$$

Elements of $\widehat{\mathcal{M}}$ are called *modified Nakajima monomials*. We will omit the $\mathbf{1}$ from the end of a monomial if there exists at least one $y_i(k) \neq 0$. The crystal structure on this set is defined by

$$\begin{aligned}
\mathrm{wt}(M) &= \sum_{i \in I} \Big(\sum_{k \geq 0} y_i(k) \Big) \Lambda_i, \\
\varphi_i(M) &= \max\Big\{ \sum_{0 \leq j \leq k} y_i(j) : k \geq 0 \Big\}, \\
\varepsilon_i(M) &= \varphi_i(M) - \langle h_i, \mathrm{wt}(M) \rangle, \\
k_f = k_f(M) &= \min\Big\{ k \geq 0 : \varphi_i(M) = \sum_{0 \leq j \leq k} y_i(j) \Big\}, \\
k_e = k_e(M) &= \max\Big\{ k \geq 0 : \varphi_i(M) = \sum_{0 \leq j \leq k} y_i(j) \Big\},
\end{aligned}$$

where $\{h_i : i \in I\}$ and $\{\Lambda_i : i \in I\}$ are the simple coroots and fundamental weights, respectively. With a chosen set of integers $C = (c_{ij})_{i \neq j}$ such that $c_{ij} + cji = 1$, one defines

$$A_{i,k} = Y_{i,k} Y_{i,k+1} \prod_{j \neq i} Y_{j,k+c_{ji}}^{a_{ji}},$$

where (a_{ij}) is a Cartan matrix. Then

$$\begin{aligned}
e_i M &= \begin{cases} 0 & \text{if } \varepsilon_i(M) = 0, \\ A_{i,k_e} M & \text{if } \varepsilon_i(M) > 0, \end{cases} \\
f_i M &= A_{i,k_f}^{-1} M.
\end{aligned}$$

It is shown in [KKS07] that the connected component of $\widehat{\mathcal{M}}$ containing the element $\mathbf{1}$, which we denote by $\mathcal{M}(\infty)$, is crystal isomorphic to the crystal $B(\infty)$.

Let $\widetilde{\mathcal{M}}$ be $\widehat{\mathcal{M}}$ as a set, and with crystal structure defined as on $\widehat{\mathcal{M}}$ with the exception that

$$f_i M = \begin{cases} 0 & \text{if } \varphi_i(M) = 0, \\ A_{i,k_f}^{-1} M & \text{if } \varphi_i(M) > 0. \end{cases}$$

Then Kashiwara [Kash03] showed that the connected component in $\widetilde{\mathcal{M}}$ containing a monomial M such that $e_i M = 0$, for all $i \in I$, is crystal isomorphic to the irreducible highest weight crystal $B(\mathrm{wt}(M))$.

WARNING:

Monomial crystals depend on the choice of positive integers $C = (c_{ij})_{i \neq j}$ satisfying the condition $c_{ij} + c_{ji} = 1$. We have chosen such integers uniformly such that $c_{ij} = 1$ if $i < j$ and $c_{ij} = 0$ if $i > j$.

REFERENCES:

class sage.combinat.crystals.monomial_crystals.**CrystalOfNakajimaMonomials**(*ct*, *La*)

Bases: sage.combinat.crystals.monomial_crystals.InfinityCrystalOfNakajimaMonomials

Let $\widetilde{\mathcal{M}}$ be $\widehat{\mathcal{M}}$ as a set, and with crystal structure defined as on $\widehat{\mathcal{M}}$ with the exception that

$$f_i M = \begin{cases} 0 & \text{if } \varphi_i(M) = 0, \\ A_{i,k_f}^{-1} M & \text{if } \varphi_i(M) > 0. \end{cases}$$

Then Kashiwara [Kash03] showed that the connected component in $\widetilde{\mathcal{M}}$ containing a monomial M such that $e_i M = 0$, for all $i \in I$, is crystal isomorphic to the irreducible highest weight crystal $B(\mathrm{wt}(M))$.

INPUT:

- `ct` – a Cartan type
- `La` – an element of the weight lattice

EXAMPLES:

```
sage: La = RootSystem("A2").weight_lattice().fundamental_weights()
sage: M = crystals.NakajimaMonomials("A2",La[1]+La[2])
sage: B = crystals.Tableaux("A2",shape=[2,1])
sage: GM = M.digraph()
sage: GB = B.digraph()
sage: GM.is_isomorphic(GB,edge_labels=True)
True

sage: La = RootSystem("G2").weight_lattice().fundamental_weights()
sage: M = crystals.NakajimaMonomials("G2",La[1]+La[2])
sage: B = crystals.Tableaux("G2",shape=[2,1])
sage: GM = M.digraph()
sage: GB = B.digraph()
sage: GM.is_isomorphic(GB,edge_labels=True)
True

sage: La = RootSystem("B2").weight_lattice().fundamental_weights()
sage: M = crystals.NakajimaMonomials(['B',2],La[1]+La[2])
sage: B = crystals.Tableaux("B2",shape=[3/2,1/2])
sage: GM = M.digraph()
sage: GB = B.digraph()
sage: GM.is_isomorphic(GB,edge_labels=True)
True

sage: La = RootSystem(['A',3,1]).weight_lattice(extended=True).fundamental_weights()
sage: M = crystals.NakajimaMonomials(['A',3,1],La[0]+La[2])
sage: B = crystals.GeneralizedYoungWalls(3,La[0]+La[2])
sage: SM = M.subcrystal(max_depth=4)
sage: SB = B.subcrystal(max_depth=4)
sage: GM = M.digraph(subset=SM) # long time
sage: GB = B.digraph(subset=SB) # long time
sage: GM.is_isomorphic(GB,edge_labels=True) # long time
```

```
True

sage: La = RootSystem(['A',5,2]).weight_lattice(extended=True).fundamental_weights()
sage: LA = RootSystem(['A',5,2]).weight_space().fundamental_weights()
sage: M = crystals.NakajimaMonomials(['A',5,2],3*La[0])
sage: B = crystals.LSPaths(3*LA[0])
sage: SM = M.subcrystal(max_depth=4)
sage: SB = B.subcrystal(max_depth=4)
sage: GM = M.digraph(subset=SM)
sage: GB = B.digraph(subset=SB)
sage: GM.is_isomorphic(GB,edge_labels=True)
True
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: La = RootSystem(['A',2]).weight_lattice().fundamental_weights()
sage: M = crystals.NakajimaMonomials(['A',2], La[1])
sage: M.cardinality()
3

sage: La = RootSystem(['D',4,2]).weight_lattice(extended=True).fundamental_weights()
sage: M = crystals.NakajimaMonomials(['D',4,2], La[1])
sage: M.cardinality()
+Infinity
```

class sage.combinat.crystals.monomial_crystals.**CrystalOfNakajimaMonomialsElement**(*parent*, *dict*)

Bases: sage.combinat.crystals.monomial_crystals.NakajimaYMonomial

Element class for CrystalOfNakajimaMonomials.

The f_i operators need to be modified from the version in monomial_crystalsNakajimaYMonomial in order to create irreducible highest weight realizations. This modified f_i is defined as

$$f_i M = \begin{cases} 0 & \text{if } \varphi_i(M) = 0, \\ A_{i,k_f}^{-1} M & \text{if } \varphi_i(M) > 0. \end{cases}$$

EXAMPLES:

```
sage: La = RootSystem(['A',5,2]).weight_lattice(extended=True).fundamental_weights()
sage: M = crystals.NakajimaMonomials(['A',5,2],3*La[0])
sage: m = M.module_generators[0].f(0); m
Y(0,0)^2 Y(0,1)^-1 Y(2,0)
sage: TestSuite(m).run()
```

f(*i*)

Return the action of f_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: La = RootSystem(['A',5,2]).weight_lattice(extended=True).fundamental_weights()
sage: M = crystals.NakajimaMonomials(['A',5,2],3*La[0])
sage: m = M.module_generators[0]
sage: [m.f(i) for i in M.index_set()]
[Y(0,0)^2 Y(0,1)^-1 Y(2,0) , None, None, None]
```

weight()

Return the weight of `self` as an element of the weight lattice.

EXAMPLES:

```
sage: La = RootSystem("A2").weight_lattice().fundamental_weights()
sage: M = crystals.NakajimaMonomials("A2",La[1]+La[2])
sage: M.module_generators[0].weight()
(2, 1, 0)
```

class sage.combinat.crystals.monomial_crystals.**InfinityCrystalOfNakajimaMonomials**(*ct*, *category*, *elt_class*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Let $Y_{i,k}$, for $i \in I$ and $k \in \mathbf{Z}$, be a commuting set of variables, and let $\mathbf{1}$ be a new variable which commutes with each $Y_{i,k}$. (Here, I represents the index set of a Cartan datum.) One may endow the structure of a crystal on the set $\widehat{\mathcal{M}}$ of monomials of the form

$$M = \prod_{(i,k) \in I \times \mathbf{Z}_{\geq 0}} Y_{i,k}^{y_i(k)} \mathbf{1}.$$

Elements of $\widehat{\mathcal{M}}$ are called *modified Nakajima monomials*. We will omit the $\mathbf{1}$ from the end of a monomial if there exists at least one $y_i(k) \neq 0$. The crystal structure on this set is defined by

$$\begin{aligned}
\mathrm{wt}(M) &= \sum_{i \in I} \Big(\sum_{k \geq 0} y_i(k) \Big) \Lambda_i, \\
\varphi_i(M) &= \max\Big\{ \sum_{0 \leq j \leq k} y_i(j) : k \geq 0 \Big\}, \\
\varepsilon_i(M) &= \varphi_i(M) - \langle h_i, \mathrm{wt}(M) \rangle, \\
k_f = k_f(M) &= \min\Big\{ k \geq 0 : \varphi_i(M) = \sum_{0 \leq j \leq k} y_i(j) \Big\}, \\
k_e = k_e(M) &= \max\Big\{ k \geq 0 : \varphi_i(M) = \sum_{0 \leq j \leq k} y_i(j) \Big\},
\end{aligned}$$

where $\{h_i : i \in I\}$ and $\{\Lambda_i : i \in I\}$ are the simple coroots and fundamental weights, respectively. With a chosen set of integers $C = (c_{ij})_{i \neq j}$ such that $c_{ij} + cji = 1$, one defines

$$A_{i,k} = Y_{i,k} Y_{i,k+1} \prod_{j \neq i} Y_{j,k+c_{ji}}^{a_{ji}},$$

where (a_{ij}) is a Cartan matrix. Then

$$\begin{aligned}
e_i M &= \begin{cases} 0 & \text{if } \varepsilon_i(M) = 0, \\ A_{i,k_e} M & \text{if } \varepsilon_i(M) > 0, \end{cases} \\
f_i M &= A_{i,k_f}^{-1} M.
\end{aligned}$$

It is shown in [KKS07] that the connected component of $\widehat{\mathcal{M}}$ containing the element $\mathbf{1}$, which we denote by $\mathcal{M}(\infty)$, is crystal isomorphic to the crystal $B(\infty)$.

INPUT:

- `cartan_type` – a Cartan type
- `use_Y` – choice of monomials in terms of A or Y

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("C3")
sage: S = B.subcrystal(max_depth=4)
sage: G = B.digraph(subset=S) # long time
sage: M = crystals.infinity.NakajimaMonomials("C3") # long time
sage: T = M.subcrystal(max_depth=4) # long time
sage: H = M.digraph(subset=T) # long time
sage: G.is_isomorphic(H,edge_labels=True) # long time
True

sage: M = crystals.infinity.NakajimaMonomials(['A',2,1])
sage: T = M.subcrystal(max_depth=3)
sage: H = M.digraph(subset=T) # long time
sage: Y = crystals.infinity.GeneralizedYoungWalls(2)
sage: YS = Y.subcrystal(max_depth=3)
sage: YG = Y.digraph(subset=YS) # long time
sage: YG.is_isomorphic(H,edge_labels=True) # long time
True

sage: M = crystals.infinity.NakajimaMonomials("D4")
sage: B = crystals.infinity.Tableaux("D4")
sage: MS = M.subcrystal(max_depth=3)
sage: BS = B.subcrystal(max_depth=3)
sage: MG = M.digraph(subset=MS) # long time
sage: BG = B.digraph(subset=BS) # long time
sage: BG.is_isomorphic(MG,edge_labels=True) # long time
True
```

cardinality()

Return the cardinality of `self`, which is always ∞.

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['A',5,2])
sage: M.cardinality()
+Infinity
```

weight_lattice_realization()

Return the weight lattice realization of `self`.

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['A',3,2])
sage: M.weight_lattice_realization()
Extended weight lattice of the Root system of type ['B', 2, 1]^*
sage: M = crystals.infinity.NakajimaMonomials(['A',2])
sage: M.weight_lattice_realization()
Ambient space of the Root system of type ['A', 2]
sage: A = CartanMatrix([[2,-3],[-3,2]])
sage: M = crystals.infinity.NakajimaMonomials(A)
sage: M.weight_lattice_realization()
Weight lattice of the Root system of type Dynkin diagram of rank 2
```

class sage.combinat.crystals.monomial_crystals.**NakajimaAMonomial**(*parent*, *dict*)

Bases: sage.combinat.crystals.monomial_crystals.NakajimaYMonomial

Monomials of the form $A_{i_1,k_1}^{a_1} \cdots A_{i_t,k_t}^{a_t}$, where $i_1, \ldots, i_t$ are elements of the index set, $k_1, \ldots, k_t$ are nonnegative integers, and $a_1, \ldots, a_t$ are integers.

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials("A3",use_Y=False)
sage: mg = M.module_generators[0]
sage: mg.f_string([1,2,3,2,1])
A(1,0)^-1 A(1,1)^-1 A(2,0)^-2 A(3,0)^-1
sage: mg.f_string([3,2,1])
A(1,2)^-1 A(2,1)^-1 A(3,0)^-1
```

e (i)

Return the action of e_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['D',4,1],use_Y=False)
sage: m = M.module_generators[0].f_string([4,2,3,0])
sage: [m.e(i) for i in M.index_set()]
[A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 ,
 None,
 None,
 A(0,2)^-1 A(2,1)^-1 A(4,0)^-1 ,
 None]
```

epsilon (i)

Return the action of ε_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['C',4,1],use_Y=False)
sage: m = M.module_generators[0].f_string([4,2,3])
sage: [m.epsilon(i) for i in M.index_set()]
[0, 0, 0, 1, 0]
```

f (i)

Return the action of f_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials("E8",use_Y=False)
sage: m = M.module_generators[0].f_string([4,2,3,8])
sage: m
A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(8,0)^-1
sage: [m.f(i) for i in M.index_set()]
[A(1,2)^-1 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(8,0)^-1 ,
 A(2,0)^-1 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(8,0)^-1 ,
 A(2,1)^-1 A(3,0)^-1 A(3,1)^-1 A(4,0)^-1 A(8,0)^-1 ,
 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(4,1)^-1 A(8,0)^-1 ,
 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(5,0)^-1 A(8,0)^-1 ,
 A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(6,0)^-1 A(8,0)^-1 ,
```

```
A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(7,1)^-1 A(8,0)^-1 ,
A(2,1)^-1 A(3,1)^-1 A(4,0)^-1 A(8,0)^-2 ]
```

phi(*i*)

Return the action of φ_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['C',4,1],use_Y=False)
sage: m = M.module_generators[0].f_string([4,2,3])
sage: [m.phi(i) for i in M.index_set()]
[0, 1, -1, 2, -1]
```

to_Y_monomial()

Represent $\prod_{(i,k)} A_{i,k}^{a_i(k)}$ in the form $\prod_{(i,k)} Y_{i,k}^{y_i(k)}$ using the formula

$$A_{i,k} = Y_{i,k} Y_{i,k+1} \prod_{\substack{j \in I \\ j \neq i}} Y_{i,k+c_{ji}}^{a_{ji}}.$$

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['A',2,1],use_Y=False)
sage: m = M.module_generators[0].f_string([2,0,1,2,1])
sage: m
A(0,0)^-1 A(1,0)^-1 A(1,1)^-1 A(2,0)^-1 A(2,1)^-1
sage: m.to_Y_monomial()
Y(0,1) Y(0,2) Y(1,1)^-1 Y(2,2)^-1
```

weight()

Return the weight of `self` as an element of `self.parent().weight_lattice_realization()`.

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['A',4,2],use_Y=False)
sage: m = M.module_generators[0].f_string([1,2,0,1])
sage: m.weight()
2*Lambda[0] - Lambda[1] - delta
```

weight_in_root_lattice()

Return the weight of `self` as an element of the root lattice.

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['C',3,1],use_Y=False)
sage: m = M.module_generators[0].f_string([3,0,1,2,0])
sage: m.weight_in_root_lattice()
-2*alpha[0] - alpha[1] - alpha[2] - alpha[3]
```

class `sage.combinat.crystals.monomial_crystals.`**NakajimaYMonomial**(*parent*, *dict*)

Bases: `sage.structure.element.Element`

Monomials of the form $Y_{i_1,k_1}^{a_1} \cdots Y_{i_t,k_t}^{y_t}$, where $i_1, \ldots, i_t$ are elements of the index set, $k_1, \ldots, k_t$ are nonnegative integers, and $y_1, \ldots, y_t$ are integers.

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['B',3,1])
sage: mg = M.module_generators[0]
sage: mg
1
sage: mg.f_string([1,3,2,0,1,2,3,0,0,1])
Y(0,0)^-1 Y(0,1)^-1 Y(0,2)^-1 Y(0,3)^-1 Y(1,0)^-3 Y(1,1)^-2 Y(1,2) Y(2,0)^3 Y(2,2) Y(3,0) Y(3,2)
```

e(i)

Return the action of e_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['E',7,1])
sage: m = M.module_generators[0].f_string([0,1,4,3])
sage: [m.e(i) for i in M.index_set()]
[None,
 None,
 None,
 Y(0,0)^-1 Y(1,1)^-1 Y(2,1) Y(3,0) Y(3,1) Y(4,0)^-1 Y(4,1)^-1 Y(5,0) ,
 None,
 None,
 None,
 None]

sage: M = crystals.infinity.NakajimaMonomials("C5")
sage: m = M.module_generators[0].f_string([1,3])
sage: [m.e(i) for i in M.index_set()]
[Y(2,1) Y(3,0)^-1 Y(3,1)^-1 Y(4,0) ,
 None,
 Y(1,0)^-1 Y(1,1)^-1 Y(2,0) ,
 None,
 None]
```

epsilon(i)

Return the value of ε_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['G',2,1])
sage: m = M.module_generators[0].f(2)
sage: [m.epsilon(i) for i in M.index_set()]
[0, 0, 1]
```

f(i)

Return the action of f_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials("B4")
sage: m = M.module_generators[0].f_string([1,3,4])
```

```
sage: [m.f(i) for i in M.index_set()]
[Y(1,0)^-2 Y(1,1)^-2 Y(2,0)^2 Y(2,1) Y(3,0)^-1 Y(4,0) Y(4,1)^-1 ,
 Y(1,0)^-1 Y(1,1)^-1 Y(1,2) Y(2,0) Y(2,2)^-1 Y(3,0)^-1 Y(3,1) Y(4,0) Y(4,1)^-1 ,
 Y(1,0)^-1 Y(1,1)^-1 Y(2,0) Y(2,1)^2 Y(3,0)^-2 Y(3,1)^-1 Y(4,0)^3 Y(4,1)^-1 ,
 Y(1,0)^-1 Y(1,1)^-1 Y(2,0) Y(2,1) Y(3,0)^-1 Y(3,1) Y(4,1)^-2 ]
```

phi(i)

Return the value of φ_i on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['D',4,3])
sage: m = M.module_generators[0].f(1)
sage: [m.phi(i) for i in M.index_set()]
[1, -1, 1]
```

weight()

Return the weight of `self` as an element of the weight lattice.

EXAMPLES:

```
sage: C = crystals.infinity.NakajimaMonomials(['A',1,1])
sage: v=C.highest_weight_vector()
sage: v.f(1).weight()+v.f(0).weight()
-delta
```

weight_in_root_lattice()

Return the weight of `self` as an element of the root lattice.

EXAMPLES:

```
sage: M = crystals.infinity.NakajimaMonomials(['F',4])
sage: m = M.module_generators[0].f_string([3,3,1,2,4])
sage: m.weight_in_root_lattice()
-alpha[1] - alpha[2] - 2*alpha[3] - alpha[4]

sage: M = crystals.infinity.NakajimaMonomials(['B',3,1])
sage: mg = M.module_generators[0]
sage: m = mg.f_string([1,3,2,0,1,2,3,0,0,1])
sage: m.weight_in_root_lattice()
-3*alpha[0] - 3*alpha[1] - 2*alpha[2] - 2*alpha[3]
```

5.1.56 Spin Crystals

These are the crystals associated with the three spin representations: the spin representations of odd orthogonal groups (or rather their double covers); and the $+$ and $-$ spin representations of the even orthogonal groups.

We follow Kashiwara and Nakashima (Journal of Algebra 165, 1994) in representing the elements of the spin crystal by sequences of signs $\pm$.

sage.combinat.crystals.spins.**CrystalOfSpins**(ct)

Return the spin crystal of the given type B.

This is a combinatorial model for the crystal with highest weight $Lambda_n$ (the n-th fundamental weight). It has 2^n elements, here called Spins. See also `CrystalOfLetters()`, `CrystalOfSpinsPlus()`, and `CrystalOfSpinsMinus()`.

INPUT:

- ['B', n] - A Cartan type B_n.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: C.list()
[+++, ++-, +-+, -++, +--, -+-, --+, ---]
sage: C.cartan_type()
['B', 3]

sage: [x.signature() for x in C]
['+++', '++-', '+-+', '-++', '+--', '-+-', '--+', '---']
```

TESTS:

```
sage: crystals.TensorProduct(C,C,generators=[[C.list()[0],C.list()[0]]]).cardinality()
35
```

sage.combinat.crystals.spins.**CrystalOfSpinsMinus**(*ct*)

Return the minus spin crystal of the given type D.

This is the crystal with highest weight $Lambda_{n-1}$ (the $(n-1)$-st fundamental weight).

INPUT:

- ['D', n] - A Cartan type D_n.

EXAMPLES:

```
sage: E = crystals.SpinsMinus(['D',4])
sage: E.list()
[+++-, ++-+, +-++, -+++, +---, -+--, --+-, ---+]
sage: [x.signature() for x in E]
['+++-', '++-+', '+-++', '-+++', '+---', '-+--', '--+-', '---+']
```

TESTS:

```
sage: len(crystals.TensorProduct(E,E,generators=[[E[0],E[0]]]).list())
35
sage: D = crystals.SpinsPlus(['D',4])
sage: len(crystals.TensorProduct(D,E,generators=[[D.list()[0],E.list()[0]]]).list())
56
```

sage.combinat.crystals.spins.**CrystalOfSpinsPlus**(*ct*)

Return the plus spin crystal of the given type D.

This is the crystal with highest weight $Lambda_n$ (the n-th fundamental weight).

INPUT:

- ['D', n] - A Cartan type D_n.

EXAMPLES:

```
sage: D = crystals.SpinsPlus(['D',4])
sage: D.list()
[++++, ++--, +-+-, -++-, +--+, -+-+, --++, ----]

sage: [x.signature() for x in D]
['++++', '++--', '+-+-', '-++-', '+--+', '-+-+', '--++', '----']
```

TESTS:

```
sage: TestSuite(D).run()
```

class sage.combinat.crystals.spins.**GenericCrystalOfSpins**(*ct*, *element_class*, *case*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

A generic crystal of spins.

digraph()

Return the directed graph associated to self.

EXAMPLES:

```
sage: crystals.Spins(['B',3]).digraph()
Digraph on 8 vertices
```

lt_elements(*x*, *y*)

Return True if and only if there is a path from x to y in the crystal graph.

Because the crystal graph is classical, it is a directed acyclic graph which can be interpreted as a poset. This function implements the comparison function of this poset.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: x = C([1,1,1])
sage: y = C([-1,-1,-1])
sage: C.lt_elements(x,y)
True
sage: C.lt_elements(y,x)
False
sage: C.lt_elements(x,x)
False
```

class sage.combinat.crystals.spins.**Spin**

Bases: sage.combinat.crystals.letters.LetterTuple

A spin letter in the crystal of spins.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: c = C([1,1,1])
sage: TestSuite(c).run()

sage: C([1,1,1]).parent()
The crystal of spins for type ['B', 3]

sage: c = C([1,1,1])
sage: c._repr_()
'+++'

sage: D = crystals.Spins(['B',4])
sage: a = C([1,1,1])
sage: b = C([-1,-1,-1])
sage: c = D([1,1,1,1])
sage: a == a
True
sage: a == b
False
```

```
sage: b == c
False
```

epsilon(*i*)

Return ε_i of `self`.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: [[C[m].epsilon(i) for i in range(1,4)] for m in range(8)]
[[0, 0, 0], [0, 0, 1], [0, 1, 0], [1, 0, 0],
 [0, 0, 1], [1, 0, 1], [0, 1, 0], [0, 0, 1]]
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: [[C[m].phi(i) for i in range(1,4)] for m in range(8)]
[[0, 0, 1], [0, 1, 0], [1, 0, 1], [0, 0, 1],
 [1, 0, 0], [0, 1, 0], [0, 0, 1], [0, 0, 0]]
```

pp()

Pretty print `self` as a column.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: b = C([1,1,-1])
sage: b.pp()
+
+
-
```

signature()

Return the signature of `self`.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: C([1,1,1]).signature()
'+++'
sage: C([1,1,-1]).signature()
'++-'
```

class sage.combinat.crystals.spins.**Spin_crystal_type_B_element**

Bases: sage.combinat.crystals.spins.Spin

Type B spin representation crystal element

e(*i*)

Returns the action of e_i on self.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: [[C[m].e(i) for i in range(1,4)] for m in range(8)]
[[None, None, None], [None, None, +++], [None, ++-, None], [+-+, None, None],
[None, None, +-+], [+--, None, -++], [None, -+-, None], [None, None, --+]]
```

f (*i*)
Returns the action of f_i on self.

EXAMPLES:

```
sage: C = crystals.Spins(['B',3])
sage: [[C[m].f(i) for i in range(1,4)] for m in range(8)]
[[None, None, ++-], [None, +-+, None], [-++, None, +--], [None, None, -+-],
[-+-, None, None], [None, --+, None], [None, None, ---], [None, None, None]]
```

class sage.combinat.crystals.spins.**Spin_crystal_type_D_element**
Bases: sage.combinat.crystals.spins.Spin

Type D spin representation crystal element

e (*i*)
Returns the action of e_i on self.

EXAMPLES:

```
sage: D = crystals.SpinsPlus(['D',4])
sage: [[D.list()[m].e(i) for i in range(1,4)] for m in range(8)]
[[None, None, None], [None, None, None], [None, ++--, None], [+-+-, None, None],
[None, None, +-+-], [+--+, None, -++-], [None, -+-+, None], [None, None, None]]

sage: E = crystals.SpinsMinus(['D',4])
sage: [[E[m].e(i) for i in range(1,4)] for m in range(8)]
[[None, None, None], [None, None, +++-], [None, ++-+, None], [+-++, None, None],
[None, None, None], [+---, None, None], [None, -+--, None], [None, None, --+-]]
```

f (*i*)
Returns the action of f_i on self.

EXAMPLES:

```
sage: D = crystals.SpinsPlus(['D',4])
sage: [[D.list()[m].f(i) for i in range(1,4)] for m in range(8)]
[[None, None, None], [None, +-+-, None], [-++-, None, +--+], [None, None, -+-+],
[-+-+, None, None], [None, --++, None], [None, None, None], [None, None, None]]

sage: E = crystals.SpinsMinus(['D',4])
sage: [[E[m].f(i) for i in range(1,4)] for m in range(8)]
[[None, None, ++-+], [None, +-++, None], [-+++, None, None], [None, None, None],
[-+--, None, None], [None, --+-, None], [None, None, ---+], [None, None, None]]
```

5.1.57 Star-Crystal Structure On $B(\infty)$

AUTHORS:

- Ben Salisbury: Initial version
- Travis Scrimshaw: Initial version

class sage.combinat.crystals.star_crystal.**StarCrystal**(*Binf*)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The star-crystal or $*$-crystal version of a highest weight crystal.

The $*$-crystal structure on $B(\infty)$ is the structure induced by the algebra antiautomorphism $*\colon U_q(\mathfrak{g}) \longrightarrow U_q(\mathfrak{g})$

that stabilizes the negative half $U_q^-(\mathfrak{g})$. It is defined by

$$E_i^* = E_i, \quad F_i^* = F_i, \quad q^* = q, \quad (q^h)^* = q^{-h},$$

where E_i and F_i are the Chevalley generators of $U_q(\mathfrak{g})$ and h is an element of the Cartan subalgebra.

The induced operation on the crystal $B(\infty)$ is called the *Kashiwara involution*. Its implementation here is based on the recursive algorithm from Theorem 2.2.1 of [Kash95], which states that for any $i \in I$ there is a unique strict crystal embedding

$$\Psi_i \colon B(\infty) \longrightarrow B_i \otimes B(\infty)$$

such that

- $u_\infty \mapsto b_i(0) \otimes u_\infty$, where u_∞ is the highest weight vector in $B(\infty)$;
- if $\Psi_i(b) = f_i^m b_i(0) \otimes b_0$, then $\Psi_i(f_i^* b) = f_i^{m+1} b_i(0) \otimes b_0$ and $\varepsilon_i(b^*) = m$;
- the image of Ψ_i is $\{f_i^m b_i(0) \otimes b : \varepsilon_i(b^*) = 0,\ m \geq 0\}$.

Here, B_i is the i-th elementary crystal. See `ElementaryCrystal` for more information.

INPUT:

- `Binf` – a crystal from `catalog_infinity_crystals`

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux(['A',2])
sage: Bstar = crystals.infinity.Star(B)
sage: mg = Bstar.highest_weight_vector()
sage: mg
[[1, 1], [2]]
sage: mg.f_string([1,2,1,2,2])
[[1, 1, 1, 1, 2, 2], [2, 3, 3, 3]]
```

REFERENCES:

class `Element`

Bases: `sage.structure.element_wrapper.ElementWrapper`

`e`(*i*)

Return the action of e_i^* on `self`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['E',6,1])
sage: RCstar = crystals.infinity.Star(RC)
sage: nuJ = RCstar.module_generators[0].f_string([0,4,6,1,2])
sage: ascii_art(nuJ.e(1))
-1[ ]-1   (/)   0[ ]1   (/)   -1[ ]-1   (/)   -2[ ]-1

sage: M = crystals.infinity.NakajimaMonomials(['B',2,1])
sage: Mstar = crystals.infinity.Star(M)
sage: m = Mstar.module_generators[0].f_string([0,1,2,2,1,0])
sage: m.e(1)
Y(0,0)^-1 Y(0,2)^-1 Y(1,1) Y(1,2)^-1 Y(2,1)^2
```

`epsilon`(*i*)

Return ε_i^* of `self`.

INPUT:

- i – an element of the index set

EXAMPLES:

```
sage: Y = crystals.infinity.GeneralizedYoungWalls(3)
sage: Ystar = crystals.infinity.Star(Y)
sage: y = Ystar.module_generators[0].f_string([0,1,3,2,1,0])
sage: [y.epsilon(i) for i in y.index_set()]
[1, 0, 1, 0]

sage: RC = crystals.infinity.RiggedConfigurations(['E',6,1])
sage: RCstar = crystals.infinity.Star(RC)
sage: nuJ = RCstar.module_generators[0].f_string([0,4,6,1,2])
sage: [nuJ.epsilon(i) for i in nuJ.index_set()]
[0, 1, 1, 0, 0, 0, 1]
```

f (i)

Return the action of f_i^* on `self`.

INPUT:

- i – an element of the index set

EXAMPLES:

```
sage: T = crystals.infinity.Tableaux("G2")
sage: Tstar = crystals.infinity.Star(T)
sage: t = Tstar.module_generators[0].f_string([1,2,1,1,2])
sage: t
[[1, 1, 1, 2, 0], [2, 3]]

sage: M = crystals.infinity.NakajimaMonomials(['B',2,1])
sage: Mstar = crystals.infinity.Star(M)
sage: m = Mstar.module_generators[0].f_string([0,1,2,2,1,0])
sage: m
Y(0,0)^-1 Y(0,2)^-1 Y(1,0)^-1 Y(1,2)^-1 Y(2,0)^2 Y(2,1)^2
```

jump (i)

Return the i-jump of `self`.

For $b \in B(\infty)$,

$$\mathrm{jump}_i(b) = \varepsilon_i(b) + \varepsilon_i^*(b) + \langle h_i, \mathrm{wt}(b) \rangle,$$

where h_i is a simple coroot.

INPUT:

- i – an element of the index set

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations("D4")
sage: RCstar = crystals.infinity.Star(RC)
sage: nu0star = RCstar.module_generators[0]
sage: nustar = nu0star.f_string([2,1,3,4,2])
sage: [nustar.jump(i) for i in RC.index_set()]
[0, 1, 0, 0]
sage: nustar = nu0star.f_string([2,1,3,4,2,2,1,3,2]) # long time
sage: [nustar.jump(i) for i in RC.index_set()] # long time
[1, 0, 1, 2]
```

phi (i)

Return φ_i^* of `self`.

For $b \in B(\infty)$,

$$\varphi_i^*(b) = \varepsilon_i^*(b) + \langle h_i, \mathrm{wt}(b) \rangle,$$

where h_i is a simple coroot.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: T = crystals.infinity.Tableaux("A2")
sage: Tstar = crystals.infinity.Star(T)
sage: t = Tstar.module_generators[0].f_string([1,2,1,1,2])
sage: [t.phi(i) for i in t.index_set()]
[-3, 1]

sage: M = crystals.infinity.NakajimaMonomials(['B',2,1])
sage: Mstar = crystals.infinity.Star(M)
sage: m = Mstar.module_generators[0].f_string([0,1,2,2,1,0])
sage: [m.phi(i) for i in m.index_set()]
[-1, -1, 4]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['E',6,1])
sage: RCstar = crystals.infinity.Star(RC)
sage: nuJ = RCstar.module_generators[0].f_string([0,4,6,1,2])
sage: nuJ.weight()
-Lambda[0] - 2*Lambda[1] + 2*Lambda[3] - Lambda[4]
 + 2*Lambda[5] - 2*Lambda[6] - delta
```

5.1.58 Tensor Products of Crystals

Main entry points:

- `TensorProductOfCrystals`
- `CrystalOfTableaux`

AUTHORS:

- Anne Schilling, Nicolas Thiery (2007): Initial version
- Ben Salisbury, Travis Scrimshaw (2013): Refactored tensor products to handle non-regular crystals and created new subclass to take advantage of the regularity

class `sage.combinat.crystals.tensor_product.`**CrystalOfTableaux**(*cartan_type*, *shapes*)

Bases: `sage.combinat.crystals.tensor_product.CrystalOfWords`

A class for crystals of tableaux with integer valued shapes

INPUT:

- `cartan_type` – a Cartan type
- `shape` – a partition of length at most `cartan_type.rank()`
- `shapes` – a list of such partitions

This constructs a classical crystal with the given Cartan type and highest weight(s) corresponding to the given shape(s).

If the type is D_r, the shape is permitted to have a negative value in the r-th position. Thus if the shape equals $[s_1, \ldots, s_r]$, then s_r may be negative but in any case $s_1 \geq \cdots \geq s_{r-1} \geq |s_r|$. This crystal is related to that of shape $[s_1, \ldots, |s_r|]$ by the outer automorphism of $SO(2r)$.

If the type is D_r or B_r, the shape is permitted to be of length r with all parts of half integer value. This corresponds to having one spin column at the beginning of the tableau. If several shapes are provided, they currently should all or none have this property.

Crystals of tableaux are constructed using an embedding into tensor products following Kashiwara and Nakashima [KN94]. Sage's tensor product rule for crystals differs from that of Kashiwara and Nakashima by reversing the order of the tensor factors. Sage produces the same crystals of tableaux as Kashiwara and Nakashima. With Sage's convention, the tensor product of crystals is the same as the monoid operation on tableaux and hence the plactic monoid.

See also:

`sage.combinat.crystals.crystals` for general help on crystals, and in particular plotting and LaTeX output.

EXAMPLES:

We create the crystal of tableaux for type A_2, with highest weight given by the partition $[2,1,1]$:

```
sage: T = crystals.Tableaux(['A',3], shape = [2,1,1])
```

Here is the list of its elements:

```
sage: T.list()
[[[1, 1], [2], [3]], [[1, 2], [2], [3]], [[1, 3], [2], [3]],
 [[1, 4], [2], [3]], [[1, 4], [2], [4]], [[1, 4], [3], [4]],
 [[2, 4], [3], [4]], [[1, 1], [2], [4]], [[1, 2], [2], [4]],
 [[1, 3], [2], [4]], [[1, 3], [3], [4]], [[2, 3], [3], [4]],
 [[1, 1], [3], [4]], [[1, 2], [3], [4]], [[2, 2], [3], [4]]]
```

Internally, a tableau of a given Cartan type is represented as a tensor product of letters of the same type. The order in which the tensor factors appear is by reading the columns of the tableaux left to right, top to bottom (in French notation). As an example:

```
sage: T = crystals.Tableaux(['A',2], shape = [3,2])
sage: T.module_generators[0]
[[1, 1, 1], [2, 2]]
sage: T.module_generators[0]._list
[2, 1, 2, 1, 1]
```

To create a tableau, one can use:

```
sage: Tab = crystals.Tableaux(['A',3], shape = [2,2])
sage: Tab(rows=[[1,2],[3,4]])
[[1, 2], [3, 4]]
sage: Tab(columns=[[3,1],[4,2]])
[[1, 2], [3, 4]]
```

Todo

FIXME:

- Do we want to specify the columns increasingly or decreasingly? That is, should this be `Tab(columns = [[1,3],[2,4]])`?
- Make this fully consistent with `Tableau()`!

We illustrate the use of a shape with a negative last entry in type D:

```
sage: T = crystals.Tableaux(['D',4],shape=[1,1,1,-1])
sage: T.cardinality()
```

```
35
sage: TestSuite(T).run()
```

We illustrate the construction of crystals of spin tableaux when the partitions have half integer values in type B and D:

```
sage: T = crystals.Tableaux(['B',3],shape=[3/2,1/2,1/2]); T
The crystal of tableaux of type ['B', 3] and shape(s) [[3/2, 1/2, 1/2]]
sage: T.cardinality()
48
sage: T.module_generators
[[+++, [[1]]]]
sage: TestSuite(T).run()

sage: T = crystals.Tableaux(['D',3],shape=[3/2,1/2,-1/2]); T
The crystal of tableaux of type ['D', 3] and shape(s) [[3/2, 1/2, -1/2]]
sage: T.cardinality()
20
sage: T.module_generators
[[++-, [[1]]]]
sage: TestSuite(T).run()
```

TESTS:

Base cases:

```
sage: T = crystals.Tableaux(['A',2], shape = [])
sage: T.list()
[[]]
sage: TestSuite(T).run()

sage: T = crystals.Tableaux(['C',2], shape = [1])
sage: T.list()
[[[1]], [[2]], [[-2]], [[-1]]]
sage: TestSuite(T).run()

sage: T = crystals.Tableaux(['A',2], shapes = [[],[1],[2]])
sage: T.list()
[[], [[1]], [[2]], [[3]], [[1, 1]], [[1, 2]], [[2, 2]], [[1, 3]], [[2, 3]], [[3, 3]]]
sage: T.module_generators
([], [[1]], [[1, 1]])

sage: T = crystals.Tableaux(['B',2], shape=[3])
sage: T(rows=[[1,1,0]])
[[1, 1, 0]]
```

Input tests:

```
sage: T = crystals.Tableaux(['A',3], shape = [2,2])
sage: C = T.letters
sage: Tab(rows    = [[1,2],[3,4]])._list == [C(3),C(1),C(4),C(2)]
True
sage: Tab(columns = [[3,1],[4,2]])._list == [C(3),C(1),C(4),C(2)]
True
```

For compatibility with `TensorProductOfCrystals()` we need to accept as input the internal list or sequence of elements:

```
sage: Tab(list    = [3,1,4,2])._list     == [C(3),C(1),C(4),C(2)]
True
```

```
sage: Tab(3,1,4,2)._list                  == [C(3),C(1),C(4),C(2)]
True
```

The next example checks whether a given tableau is in fact a valid type C tableau or not:

```
sage: T = crystals.Tableaux(['C',3], shape = [2,2,2])
sage: Tab = T(rows=[[1,3],[2,-3],[3,-1]])
sage: Tab in T.list()
True
sage: Tab = T(rows=[[2,3],[3,-3],[-3,-2]])
sage: Tab in T.list()
False
```

Element
alias of CrystalOfTableauxElement

cartan_type()
Returns the Cartan type of the associated crystal

EXAMPLES:

```
sage: T = crystals.Tableaux(['A',3], shape = [2,2])
sage: T.cartan_type()
['A', 3]
```

module_generator(*shape*)
This yields the module generator (or highest weight element) of a classical crystal of given shape. The module generator is the unique tableau with equal shape and content.

EXAMPLE:

```
sage: T = crystals.Tableaux(['D',3], shape = [1,1])
sage: T.module_generator([1,1])
[[1], [2]]

sage: T = crystals.Tableaux(['D',4],shape=[2,2,2,-2])
sage: T.module_generator(tuple([2,2,2,-2]))
[[1, 1], [2, 2], [3, 3], [-4, -4]]
sage: T.cardinality()
294
sage: T = crystals.Tableaux(['D',4],shape=[2,2,2,2])
sage: T.module_generator(tuple([2,2,2,2]))
[[1, 1], [2, 2], [3, 3], [4, 4]]
sage: T.cardinality()
294
```

class sage.combinat.crystals.tensor_product.**CrystalOfTableauxElement**(*parent*, **args*, ***options*)
Bases: sage.combinat.crystals.tensor_product.TensorProductOfRegularCrystalsElement

Element in a crystal of tableaux.

pp()
EXAMPLES:

```
sage: T = crystals.Tableaux(['A',3], shape = [2,2])
sage: t = T(rows=[[1,2],[3,4]])
sage: t.pp()
```

```
1  2
3  4
```

promotion()

Promotion for type A crystals of tableaux of rectangular shape

Returns the result of applying promotion on this tableau.

This method only makes sense in type A with rectangular shapes.

EXAMPLES:

```
sage: C = crystals.Tableaux(["A",3], shape = [3,3,3])
sage: t = C(Tableau([[1,1,1],[2,2,3],[3,4,4]]))
sage: t
[[1, 1, 1], [2, 2, 3], [3, 4, 4]]
sage: t.promotion()
[[1, 1, 2], [2, 2, 3], [3, 4, 4]]
sage: t.promotion().parent()
The crystal of tableaux of type ['A', 3] and shape(s) [[3, 3, 3]]
```

promotion_inverse()

Inverse promotion for type A crystals of tableaux of rectangular shape

Returns the result of applying inverse promotion on this tableau.

This method only makes sense in type A with rectangular shapes.

EXAMPLES:

```
sage: C = crystals.Tableaux(["A",3], shape = [3,3,3])
sage: t = C(Tableau([[1,1,1],[2,2,3],[3,4,4]]))
sage: t
[[1, 1, 1], [2, 2, 3], [3, 4, 4]]
sage: t.promotion_inverse()
[[1, 1, 2], [2, 3, 3], [4, 4, 4]]
sage: t.promotion_inverse().parent()
The crystal of tableaux of type ['A', 3] and shape(s) [[3, 3, 3]]
```

to_tableau()

Returns the Tableau object corresponding to self.

EXAMPLES:

```
sage: T = crystals.Tableaux(['A',3], shape = [2,2])
sage: t = T(rows=[[1,2],[3,4]]).to_tableau(); t
[[1, 2], [3, 4]]
sage: type(t)
<class 'sage.combinat.tableau.Tableaux_all_with_category.element_class'>
sage: type(t[0][0])
<type 'int'>
sage: T = crystals.Tableaux(['D',3], shape = [1,1])
sage: t=T(rows=[[-3],[3]]).to_tableau(); t
[[-3], [3]]
sage: t=T(rows=[[3],[-3]]).to_tableau(); t
[[3], [-3]]
sage: T = crystals.Tableaux(['B',2], shape = [1,1])
sage: t = T(rows=[[0],[0]]).to_tableau(); t
[[0], [0]]
```

class sage.combinat.crystals.tensor_product.**CrystalOfWords**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Auxiliary class to provide a call method to create tensor product elements. This class is shared with several tensor product classes and is also used in CrystalOfTableaux to allow tableaux of different tensor product structures in column-reading (and hence different shapes) to be considered elements in the same crystal.

Element
alias of TensorProductOfCrystalsElement

one_dimensional_configuration_sum(*q=None*, *group_components=True*)
Computes the one-dimensional configuration sum.

INPUT:

- q – (default: None) a variable or None; if None, a variable q is set in the code
- group_components – (default: True) boolean; if True, then the terms are grouped by classical component

The one-dimensional configuration sum is the sum of the weights of all elements in the crystal weighted by the energy function.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: T.one_dimensional_configuration_sum()
B[-2*Lambda[1] + 2*Lambda[2]] + (q+1)*B[-Lambda[1]] + (q+1)*B[Lambda[1] - Lambda[2]]
+ B[2*Lambda[1]] + B[-2*Lambda[2]] + (q+1)*B[Lambda[2]]
sage: R.<t> = ZZ[]
sage: T.one_dimensional_configuration_sum(t, False)
B[-2*Lambda[1] + 2*Lambda[2]] + (t+1)*B[-Lambda[1]] + (t+1)*B[Lambda[1] - Lambda[2]]
+ B[2*Lambda[1]] + B[-2*Lambda[2]] + (t+1)*B[Lambda[2]]

sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: LS.one_dimensional_configuration_sum() == T.one_dimensional_configuration_sum() # long
True
```

TESTS:

```
sage: K1 = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: K2 = crystals.KirillovReshetikhin(['A',2,1],2,1)
sage: T = crystals.TensorProduct(K1,K2)
sage: T.one_dimensional_configuration_sum() == T.one_dimensional_configuration_sum(group_com
True
```

class sage.combinat.crystals.tensor_product.**FullTensorProductOfCrystals**(*crystals*, ***options*)
Bases: sage.combinat.crystals.tensor_product.TensorProductOfCrystals

Full tensor product of crystals.

Todo

Merge this into TensorProductOfCrystals.

---

cardinality()
Return the cardinality of self.

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C,C)
sage: T.cardinality()
9
```

weight_lattice_realization()

Return the weight lattice realization used to express weights.

The weight lattice realization is the common parent which all weight lattice realizations of the crystals of `self` coerce into.

EXAMPLES:

```
sage: B = crystals.elementary.B(['A',4], 2)
sage: B.weight_lattice_realization()
Root lattice of the Root system of type ['A', 4]
sage: T = crystals.infinity.Tableaux(['A',4])
sage: T.weight_lattice_realization()
Ambient space of the Root system of type ['A', 4]
sage: TP = crystals.TensorProduct(B, T)
sage: TP.weight_lattice_realization()
Ambient space of the Root system of type ['A', 4]
```

class sage.combinat.crystals.tensor_product.**FullTensorProductOfRegularCrystals**(*crystals*, ***options*)

Bases: sage.combinat.crystals.tensor_product.FullTensorProductOfCrystals

Full tensor product of regular crystals.

Element

alias of TensorProductOfRegularCrystalsElement

class sage.combinat.crystals.tensor_product.**ImmutableListWithParent**(*parent*, **args*, ***kwds*)

Bases: sage.combinat.combinat.CombinatorialElement

A class for lists having a parent

Specification: any subclass `C` should implement `__init__` which accepts the following form `C(parent, list = list)`

EXAMPLES:

We create an immutable list whose parent is the class list:

```
sage: from sage.combinat.crystals.tensor_product import ImmutableListWithParent, TestParent
sage: l = ImmutableListWithParent(TestParent(), [1,2,3])
sage: l._list
[1, 2, 3]
sage: l.parent()
A parent for tests
sage: l.sibling([2,1]) == ImmutableListWithParent(TestParent(), [2,1])
True
sage: l.reversed()
[3, 2, 1]
sage: l.set_index(1,4)
[1, 4, 3]
```

TESTS:

```
sage: TestSuite(l).run(skip = "_test_category")
```

reversed()

Returns the sibling of `self` which is obtained by reversing the elements of‘‘ self‘‘.

EXAMPLES:

```
sage: from sage.combinat.crystals.tensor_product import ImmutableListWithParent, TestParent
sage: l = ImmutableListWithParent(TestParent(), [1,2,3])
sage: l.reversed()
[3, 2, 1]
```

set_index(*k*, *value*)

Returns the sibling of `self` obtained by setting the k^{th} entry of self to value.

EXAMPLES:

```
sage: from sage.combinat.crystals.tensor_product import ImmutableListWithParent, TestParent
sage: l = ImmutableListWithParent(TestParent(), [1,2,3])
sage: l.set_index(0,2)
[2, 2, 3]
sage: l.set_index(1,4)
[1, 4, 3]
sage: _.parent()
A parent for tests
```

sibling(*l*)

Returns an `ImmutableListWithParent` object whose list is `l` and whose parent is the same as the parent of `self`.

Note that the implementation of this function makes an assumption about the constructor for subclasses.

EXAMPLES:

```
sage: from sage.combinat.crystals.tensor_product import ImmutableListWithParent, TestParent
sage: l = ImmutableListWithParent(TestParent(), [1,2,3])
sage: m = l.sibling([2,3,4]); m
[2, 3, 4]
sage: m.parent()
A parent for tests
```

class sage.combinat.crystals.tensor_product.**TensorProductOfCrystals**

Bases: sage.combinat.crystals.tensor_product.CrystalOfWords

Tensor product of crystals.

Given two crystals B and B' of the same Cartan type, one can form the tensor product $B \otimes B'$. As a set $B \otimes B'$ is the Cartesian product $B \times B'$. The crystal operators f_i and e_i act on $b \otimes b' \in B \otimes B'$ as follows:

$$f_i(b \otimes b') = \begin{cases} f_i(b) \otimes b' & \text{if } \varepsilon_i(b) \geq \varphi_i(b') \\ b \otimes f_i(b') & \text{otherwise} \end{cases}$$

and

$$e_i(b \otimes b') = \begin{cases} e_i(b) \otimes b' & \text{if } \varepsilon_i(b) > \varphi_i(b') \\ b \otimes e_i(b') & \text{otherwise.} \end{cases}$$

We also define:

$$\begin{aligned} \varphi_i(b \otimes b') &= \max\left(\varphi_i(b), \varphi_i(b) + \varphi_i(b') - \varepsilon_i(b)\right) \\ \varepsilon_i(b \otimes b') &= \max\left(\varepsilon_i(b'), \varepsilon_i(b') + \varepsilon_i(b) - \varphi_i(b')\right). \end{aligned}$$

Note: This is the opposite of Kashiwara's convention for tensor products of crystals.

Since tensor products are associative $(\mathcal{B} \otimes \mathcal{C}) \otimes \mathcal{D} \cong \mathcal{B} \otimes (\mathcal{C} \otimes \mathcal{D})$ via the natural isomorphism $(b \otimes c) \otimes d \mapsto b \otimes (c \otimes d)$, we can generalizing this to arbitrary tensor products. Thus consider $B_N \otimes \cdots \otimes B_1$, where each B_k is an abstract crystal. The underlying set of the tensor product is $B_N \times \cdots \times B_1$, while the crystal structure is given as follows. Let I be the index set, and fix some $i \in I$ and $b_N \otimes \cdots \otimes b_1 \in B_N \otimes \cdots \otimes B_1$. Define

$$a_i(k) := \varepsilon_i(b_k) - \sum_{j=1}^{k-1} \langle \alpha_i^\vee, \mathrm{wt}(b_j) \rangle.$$

Then

$$\begin{aligned}
\mathrm{wt}(b_N \otimes \cdots \otimes b_1) &= \mathrm{wt}(b_N) + \cdots + \mathrm{wt}(b_1), \\
\varepsilon_i(b_N \otimes \cdots \otimes b_1) &= \max_{1 \le k \le n} \left(\sum_{j=1}^{k} \varepsilon_i(b_j) - \sum_{j=1}^{k-1} \varphi_i(b_j) \right) \\
&= \max_{1 \le k \le N} \big(a_i(k) \big), \\
\varphi_i(b_N \otimes \cdots \otimes b_1) &= \max_{1 \le k \le N} \left(\varphi_i(b_N) + \sum_{j=k}^{N-1} \big(\varphi_i(b_j) - \varepsilon_i(b_{j+1}) \big) \right) \\
&= \max_{1 \le k \le N} \big(\lambda_i + a_i(k) \big)
\end{aligned}$$

where $\lambda_i = \langle \alpha_i^\vee, \mathrm{wt}(b_N \otimes \cdots \otimes b_1) \rangle$. Then for $k = 1, \ldots, N$ the action of the Kashiwara operators is determined as follows.

•If $a_i(k) > a_i(j)$ for $1 \le j < k$ and $a_i(k) \ge a_i(j)$ for $k < j \le N$:

$$e_i(b_N \otimes \cdots \otimes b_1) = b_N \otimes \cdots \otimes e_i b_k \otimes \cdots \otimes b_1.$$

•If $a_i(k) \ge a_i(j)$ for $1 \le j < k$ and $a_i(k) > a_i(j)$ for $k < j \le N$:

$$f_i(b_N \otimes \cdots \otimes b_1) = b_N \otimes \cdots \otimes f_i b_k \otimes \cdots \otimes b_1.$$

Note that this is just recursively applying the definition of the tensor product on two crystals. Recall that $\langle \alpha_i^\vee, \mathrm{wt}(b_j) \rangle = \varphi_i(b_j) - \varepsilon_i(b_j)$ by the definition of a crystal.

Regular crystals

Now if all crystals B_k are regular crystals, all ε_i and φ_i are non-negative and we can define tensor product by the *signature rule*. We start by writing a word in $+$ and $-$ as follows:

$$\underbrace{- \cdots -}_{\varphi_i(b_N) \text{ times}} \quad \underbrace{+ \cdots +}_{\varepsilon_i(b_N) \text{ times}} \quad \cdots \quad \underbrace{- \cdots -}_{\varphi_i(b_1) \text{ times}} \quad \underbrace{+ \cdots +}_{\varepsilon_i(b_1) \text{ times}},$$

and then canceling ordered pairs of $+-$ until the word is in the reduced form:

$$\underbrace{- \cdots -}_{\varphi_i \text{ times}} \quad \underbrace{+ \cdots +}_{\varepsilon_i \text{ times}}.$$

Here e_i acts on the factor corresponding to the leftmost $+$ and f_i on the factor corresponding to the rightmost $-$. If there is no $+$ or $-$ respectively, then the result is 0 (`None`).

EXAMPLES:

We construct the type A_2-crystal generated by $2 \otimes 1 \otimes 1$:

```
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C,C,C,generators=[[C(2),C(1),C(1)]])
```

It has 8 elements:

```
sage: T.list()
[[2, 1, 1], [2, 1, 2], [2, 1, 3], [3, 1, 3], [3, 2, 3], [3, 1, 1], [3, 1, 2], [3, 2, 2]]
```

One can also check the Cartan type of the crystal:

```
sage: T.cartan_type()
['A', 2]
```

Other examples include crystals of tableaux (which internally are represented as tensor products obtained by reading the tableaux columnwise):

```
sage: C = crystals.Tableaux(['A',3], shape=[1,1,0])
sage: D = crystals.Tableaux(['A',3], shape=[1,0,0])
sage: T = crystals.TensorProduct(C,D, generators=[[C(rows=[[1], [2]]), D(rows=[[1]])], [C(rows=[
sage: T.cardinality()
24
sage: TestSuite(T).run()
sage: T.module_generators
[[[[1], [2]], [[1]]], [[[2], [3]], [[1]]]]
sage: [x.weight() for x in T.module_generators]
[(2, 1, 0, 0), (1, 1, 1, 0)]
```

If no module generators are specified, we obtain the full tensor product:

```
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C,C)
sage: T.list()
[[1, 1], [1, 2], [1, 3], [2, 1], [2, 2], [2, 3], [3, 1], [3, 2], [3, 3]]
sage: T.cardinality()
9
```

For a tensor product of crystals without module generators, the default implementation of `module_generators` contains all elements in the tensor product of the crystals. If there is a subset of elements in the tensor product that still generates the crystal, this needs to be implemented for the specific crystal separately:

```
sage: T.module_generators.list()
[[1, 1], [1, 2], [1, 3], [2, 1], [2, 2], [2, 3], [3, 1], [3, 2], [3, 3]]
```

For classical highest weight crystals, it is also possible to list all highest weight elements:

```
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C,C,C,generators=[[C(2),C(1),C(1)],[C(1),C(2),C(1)]])
sage: T.highest_weight_vectors()
([2, 1, 1], [1, 2, 1])
```

Examples with non-regular and infinite crystals (these did not work before trac ticket #14402):

```
sage: B = crystals.infinity.Tableaux(['D',10])
sage: T = crystals.TensorProduct(B,B)
sage: T
Full tensor product of the crystals
[The infinity crystal of tableaux of type ['D', 10],
```

```
 The infinity crystal of tableaux of type ['D', 10]]

sage: B = crystals.infinity.GeneralizedYoungWalls(15)
sage: T = crystals.TensorProduct(B,B,B)
sage: T
Full tensor product of the crystals
[Crystal of generalized Young walls of type ['A', 15, 1],
 Crystal of generalized Young walls of type ['A', 15, 1],
 Crystal of generalized Young walls of type ['A', 15, 1]]

sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_weights()
sage: B = crystals.GeneralizedYoungWalls(2,La[0]+La[1])
sage: C = crystals.GeneralizedYoungWalls(2,2*La[2])
sage: D = crystals.GeneralizedYoungWalls(2,3*La[0]+La[2])
sage: T = crystals.TensorProduct(B,C,D)
sage: T
Full tensor product of the crystals
[Highest weight crystal of generalized Young walls of Cartan type ['A', 2, 1] and highest weight
 Highest weight crystal of generalized Young walls of Cartan type ['A', 2, 1] and highest weight
 Highest weight crystal of generalized Young walls of Cartan type ['A', 2, 1] and highest weight
```

There is also a global option for setting the convention (by default Sage uses anti-Kashiwara):

```
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C,C)
sage: elt = T(C(1), C(2)); elt
[1, 2]
sage: crystals.TensorProduct.global_options['convention'] = "Kashiwara"
sage: elt
[2, 1]
sage: crystals.TensorProduct.global_options.reset()
```

global_options(**get_value*, ***set_value*)

Sets the global options for tensor products of crystals. The default is to use the anti-Kashiwara convention.

There are two conventions for how e_i and f_i act on tensor products, and the difference between the two is the order of the tensor factors are reversed. This affects both the input and output. See the example below.

OPTIONS:

- `convention` – (default: `antiKashiwara`) Sets the convention used for displaying/inputting tensor product of crystals
  - `Kashiwara` – use the Kashiwara convention
  - `anti` – alias for `antiKashiwara`
  - `antiKashiwara` – use the anti-Kashiwara convention
  - `opposite` – alias for `antiKashiwara`

Note: Changing the `convention` also changes how the input is handled.

Warning: Internally, the crystals are always stored using the anti-Kashiwara convention.

If no parameters are set, then the function returns a copy of the options dictionary.

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C,C)
sage: elt = T(C(1), C(2)); elt
[1, 2]
sage: crystals.TensorProduct.global_options['convention'] = "Kashiwara"
sage: elt
[2, 1]
sage: T(C(1), C(2)) == elt
False
sage: T(C(2), C(1)) == elt
True
sage: crystals.TensorProduct.global_options.reset()
```

See `GlobalOptions` for more features of these options.

class sage.combinat.crystals.tensor_product.**TensorProductOfCrystalsElement**(*parent*, **args*, ***kwds*)

Bases: sage.combinat.crystals.tensor_product.ImmutableListWithParent

A class for elements of tensor products of crystals.

e(*i*)

Return the action of e_i on `self`.

INPUT:

- `i` – An element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("D4")
sage: T = crystals.TensorProduct(B,B)
sage: b1 = B.highest_weight_vector().f_string([1,4,3])
sage: b2 = B.highest_weight_vector().f_string([2,2,3,1,4])
sage: t = T(b2, b1)
sage: t.e(1)
[[[1, 1, 1, 1, 1], [2, 2, 3, -3], [3]], [[1, 1, 1, 1, 2], [2, 2, 2], [3, -3]]]
sage: t.e(2)
sage: t.e(3)
[[[1, 1, 1, 1, 1, 2], [2, 2, 3, -4], [3]], [[1, 1, 1, 1, 2], [2, 2, 2], [3, -3]]]
sage: t.e(4)
[[[1, 1, 1, 1, 1, 2], [2, 2, 3, 4], [3]], [[1, 1, 1, 1, 2], [2, 2, 2], [3, -3]]]
```

epsilon(*i*)

Return ε_i of `self`.

INPUT:

- `i` – An element of the index set

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("G2")
sage: T = crystals.TensorProduct(B,B)
sage: b1 = B.highest_weight_vector().f(2)
sage: b2 = B.highest_weight_vector().f_string([2,2,1])
sage: t = T(b2, b1)
sage: [t.epsilon(i) for i in B.index_set()]
[0, 3]
```

f(*i*)

Return the action of f_i on `self`.

INPUT:

- `i` – An element of the index set

EXAMPLES:

```
sage: La = RootSystem(['A',3,1]).weight_lattice(extended=True).fundamental_weights()
sage: B = crystals.GeneralizedYoungWalls(3,La[0])
sage: T = crystals.TensorProduct(B,B,B)
sage: b1 = B.highest_weight_vector().f_string([0,3])
sage: b2 = B.highest_weight_vector().f_string([0])
sage: b3 = B.highest_weight_vector()
sage: t = T(b3, b2, b1)
sage: t.f(0)
[[[0]], [[0]], [[0, 3]]]
sage: t.f(1)
[[], [[0]], [[0, 3], [1]]]
sage: t.f(2)
[[], [[0]], [[0, 3, 2]]]
sage: t.f(3)
[[], [[0, 3]], [[0, 3]]]
```

phi(*i*)

Return φ_i of `self`.

INPUT:

- `i` – An element of the index set

EXAMPLES:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_weights()
sage: B = crystals.GeneralizedYoungWalls(2,La[0]+La[1])
sage: T = crystals.TensorProduct(B,B)
sage: b1 = B.highest_weight_vector().f_string([1,0])
sage: b2 = B.highest_weight_vector().f_string([0,1])
sage: t = T(b2, b1)
sage: [t.phi(i) for i in B.index_set()]
[1, 1, 4]
```

TESTS:

Check that trac ticket #15462 is fixed:

```
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: La = RootSystem(['A',2]).ambient_space().fundamental_weights()
sage: T = crystals.TensorProduct(crystals.elementary.T(['A',2], La[1]+La[2]), B)
sage: t = T.an_element()
sage: t.phi(1)
2
sage: t.phi(2)
2
```

pp()

Pretty print `self`.

EXAMPLES:

```
sage: C = crystals.Tableaux(['A',3], shape=[3,1])
sage: D = crystals.Tableaux(['A',3], shape=[1])
```

```
sage: E = crystals.Tableaux(['A',3], shape=[2,2,2])
sage: T = crystals.TensorProduct(C,D,E)
sage: T.module_generators[0].pp()
  1  1  1 (X)    1 (X)    1  1
  2                      2  2
                         3  3
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: B = crystals.infinity.Tableaux("A3")
sage: T = crystals.TensorProduct(B,B)
sage: b1 = B.highest_weight_vector().f_string([2,1,3])
sage: b2 = B.highest_weight_vector().f(1)
sage: t = T(b2, b1)
sage: t
[[[1, 1, 1, 2], [2, 2], [3]], [[1, 1, 1, 1, 2], [2, 2, 4], [3]]]
sage: t.weight()
(-2, 1, 0, 1)
```

class sage.combinat.crystals.tensor_product.**TensorProductOfCrystalsWithGenerators**(*crystals*, *generators*, *cartan_type*)

Bases: sage.combinat.crystals.tensor_product.TensorProductOfCrystals

Tensor product of crystals with a generating set.

Todo

Deprecate this class in favor of using `subcrystal()`.

---

class sage.combinat.crystals.tensor_product.**TensorProductOfRegularCrystalsElement**(*parent*, **args*, ***kwds*)

Bases: sage.combinat.crystals.tensor_product.TensorProductOfCrystalsElement

Element class for a tensor product of regular crystals.

TESTS:

```
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C, C)
sage: elt = T(C(1), C(2))
sage: from sage.combinat.crystals.tensor_product import TensorProductOfRegularCrystalsElement
sage: isinstance(elt, TensorProductOfRegularCrystalsElement)
True
```

affine_grading()

Returns the affine grading of $self$.

The affine grading is only defined when `self` is an element of a tensor product of affine Kirillov-Reshetikhin crystals. It is calculated by finding a path from `self` to a ground state path using the helper method `e_string_to_ground_state()` and counting the number of affine Kashiwara operators e_0 applied on the way.

INPUT:

- self – an element of a tensor product of Kirillov-Reshetikhin crystals

OUTPUT: an integer

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: t = T.module_generators[0]
sage: t.affine_grading()
1

sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K,K)
sage: hw = [b for b in T if all(b.epsilon(i)==0 for i in [1,2])]
sage: for b in hw:
....:     print b, b.affine_grading()
[[[1]], [[1]], [[1]]] 3
[[[1]], [[2]], [[1]]] 1
[[[2]], [[1]], [[1]]] 2
[[[3]], [[2]], [[1]]] 0

sage: K = crystals.KirillovReshetikhin(['C',2,1],1,1)
sage: T = crystals.TensorProduct(K,K,K)
sage: hw = [b for b in T if all(b.epsilon(i)==0 for i in [1,2])]
sage: for b in hw:
....:     print b, b.affine_grading()
[[[1]], [[1]], [[1]]] 2
[[[1]], [[2]], [[1]]] 1
[[[1]], [[-1]], [[1]]] 0
[[[2]], [[1]], [[1]]] 1
[[[-2]], [[2]], [[1]]] 0
[[[-1]], [[1]], [[1]]] 1
```

e(*i*)

Return the action of e_i on self.

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(1),C(2)).e(1) == T(C(1),C(1))
True
sage: T(C(2),C(1)).e(1) is None
True
sage: T(C(2),C(2)).e(1) == T(C(1),C(2))
True
```

e_string_to_ground_state()

Returns a string of integers in the index set $(i_1, \ldots, i_k)$ such that $e_{i_k} \cdots e_{i_1}$ of self is the ground state.

This method is only defined when self is an element of a tensor product of affine Kirillov-Reshetikhin crystals. It calculates a path from self to a ground state path using Demazure arrows as defined in Lemma 7.3 in [SchillingTingley2011].

INPUT:

- self – an element of a tensor product of Kirillov-Reshetikhin crystals

OUTPUT: a tuple of integers $(i_1, \ldots, i_k)$

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: t = T.module_generators[0]
sage: t.e_string_to_ground_state()
(0, 2)

sage: K = crystals.KirillovReshetikhin(['C',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: t = T.module_generators[0]; t
[[[1]], [[1]]]
sage: t.e_string_to_ground_state()
(0,)
sage: x=t.e(0)
sage: x.e_string_to_ground_state()
()
sage: y=t.f_string([1,2,1,1,0]); y
[[[2]], [[1]]]
sage: y.e_string_to_ground_state()
()
```

energy_function()

Return the energy function of `self`.

The energy is only defined when `self` is an element of a tensor product of affine Kirillov-Reshetikhin crystals. In this implementation, it is assumed that `self` is an element of a tensor product of perfect crystals of the same level, see Theorem 7.5 in [SchillingTingley2011].

INPUT:

- `self` – an element of a tensor product of perfect Kirillov-Reshetkhin crystals of the same level

OUTPUT: an integer

REFERENCES:

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K,K)
sage: hw = [b for b in T if all(b.epsilon(i)==0 for i in [1,2])]
sage: for b in hw:
....:     print b, b.energy_function()
[[[1]], [[1]], [[1]]] 0
[[[1]], [[2]], [[1]]] 2
[[[2]], [[1]], [[1]]] 1
[[[3]], [[2]], [[1]]] 3

sage: K = crystals.KirillovReshetikhin(['C',2,1],1,2)
sage: T = crystals.TensorProduct(K,K)
sage: hw = [b for b in T if all(b.epsilon(i)==0 for i in [1,2])]
sage: for b in hw:  # long time (5s on sage.math, 2011)
....:     print b, b.energy_function()
[[], []] 4
[[], [[1, 1]]] 1
[[[1, 1]], []] 3
[[[1, 1]], [[1, 1]]] 0
[[[1, 2]], [[1, 1]]] 1
[[[2, 2]], [[1, 1]]] 2
[[[-1, -1]], [[1, 1]]] 2
[[[1, -1]], [[1, 1]]] 2
```

```
[[[2, -1]], [[1, 1]]] 2

sage: K = crystals.KirillovReshetikhin(['C',2,1],1,1)
sage: T = crystals.TensorProduct(K)
sage: t = T.module_generators[0]
sage: t.energy_function()
Traceback (most recent call last):
...
ValueError: All crystals in the tensor product need to be perfect of the same level
```

epsilon(i)

Return ε_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(1),C(1)).epsilon(1)
0
sage: T(C(1),C(2)).epsilon(1)
1
sage: T(C(2),C(1)).epsilon(1)
0
```

f(i)

Return the action of f_i on `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(1),C(1)).f(1)
[1, 2]
sage: T(C(1),C(2)).f(1)
[2, 2]
sage: T(C(2),C(1)).f(1) is None
True
```

phi(i)

Return φ_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(1),C(1)).phi(1)
2
sage: T(C(1),C(2)).phi(1)
1
sage: T(C(2),C(1)).phi(1)
0
```

positions_of_unmatched_minus(i, *dual=False*, *reverse=False*)

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(2),C(1)).positions_of_unmatched_minus(1)
[]
sage: T(C(1),C(2)).positions_of_unmatched_minus(1)
[0]
```

positions_of_unmatched_plus(*i*)

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(2),C(1)).positions_of_unmatched_plus(1)
[]
sage: T(C(1),C(2)).positions_of_unmatched_plus(1)
[1]
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A',3])
sage: T = crystals.TensorProduct(C,C)
sage: T(C(1),C(2)).weight()
(1, 1, 0, 0)
sage: T = crystals.Tableaux(['D',4],shape=[])
sage: T.list()[0].weight()
(0, 0, 0, 0)
```

class sage.combinat.crystals.tensor_product.**TensorProductOfRegularCrystalsWithGenerators**(*crystal_generators*, *cartan_t*

Bases: sage.combinat.crystals.tensor_product.TensorProductOfCrystalsWithGenerators

Tensor product of regular crystals with a generating set.

Element

alias of TensorProductOfRegularCrystalsElement

class sage.combinat.crystals.tensor_product.**TestParent**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

A parent for tests.

sage.combinat.crystals.tensor_product.**trunc**(*i*)

Truncates to the integer closer to zero

EXAMPLES:

```
sage: from sage.combinat.crystals.tensor_product import trunc
sage: trunc(-3/2), trunc(-1), trunc(-1/2), trunc(0), trunc(1/2), trunc(1), trunc(3/2)
(-1, -1, 0, 0, 0, 1, 1)
sage: isinstance(trunc(3/2), Integer)
True
```

5.1.59 Cyclic sieving phenomenon

Implementation of the Cyclic Sieving Phenomenon as described in Reiner, Stanton, White - The cyclic sieving phenomenon, Journal of Combinatorial Theory A 108 (2004)

We define the CyclicSievingPolynomial of a finite set S together with cyclic action cyc_act (of order n) to be the unique polynomial P(q) of order < n such that the triple (S, cyc_act, P(q)) exhibits the cyclic sieving phenomenon.

AUTHORS:

- Christian Stump

sage.combinat.cyclic_sieving_phenomenon.**CyclicSievingCheck**(*L*, *cyc_act*, *f*, *order=None*)

Returns True if the triple (L, cyc_act, f) exhibits the cyclic sieving phenomenon. If cyc_act is None, L is expected to obtain the orbit lengths.

INPUT:

- L – if cyc_act is None: list of orbit sizes, otherwise list of objects
- cyc_act – (default:None) function taking an element of L and returning an element of L (must define a bijection on L)
- **order – (default:None) if set to an integer, this cyclic order of cyc_act is used (must be an integer multiple of the orde** otherwise, the order of cyc_action is used

EXAMPLES:

```
sage: from sage.combinat.cyclic_sieving_phenomenon import *
sage: from sage.combinat.q_analogues import q_binomial
sage: S42 = [ Set(S) for S in subsets([1,2,3,4]) if len(S) == 2 ]; S42
[{1, 2}, {1, 3}, {2, 3}, {1, 4}, {2, 4}, {3, 4}]
sage: cyc_act = lambda S: Set( i.mod(4)+1 for i in S)
sage: cyc_act([1,3])
{2, 4}
sage: cyc_act([1,4])
{1, 2}
sage: p = q_binomial(4,2); p
q^4 + q^3 + 2*q^2 + q + 1
sage: CyclicSievingPolynomial( S42, cyc_act )
q^3 + 2*q^2 + q + 2
sage: CyclicSievingCheck( S42, cyc_act, p )
True
```

sage.combinat.cyclic_sieving_phenomenon.**CyclicSievingPolynomial**(*L*, *cyc_act=None*, *order=None*, *get_order=False*)

Returns the unique polynomial p of degree smaller than order such that the triple (L, cyc_act, p) exhibits the CSP. If `cyc_act` is None, `L` is expected to contain the orbit lengths.

INPUT:

- L – if cyc_act is None: list of orbit sizes, otherwise list of objects
- cyc_act – (default:None) function taking an element of L and returning an element of L (must define a bijection on L)
- **order – (default:None) if set to an integer, this cyclic order of cyc_act is used (must be an integer multiple of the orde** otherwise, the order of cyc_action is used
- get_order – (default:False) if True, a tuple [p,n] is returned where p as above, and n is the order

EXAMPLES:

```
sage: from sage.combinat.cyclic_sieving_phenomenon import CyclicSievingPolynomial
sage: S42 = [ Set(S) for S in subsets([1,2,3,4]) if len(S) == 2 ]; S42
[{1, 2}, {1, 3}, {2, 3}, {1, 4}, {2, 4}, {3, 4}]
```

```
sage: cyc_act = lambda S: Set( i.mod(4)+1 for i in S)
sage: cyc_act([1,3])
{2, 4}
sage: cyc_act([1,4])
{1, 2}
sage: CyclicSievingPolynomial( S42, cyc_act )
q^3 + 2*q^2 + q + 2
sage: CyclicSievingPolynomial( S42, cyc_act, get_order=True )
[q^3 + 2*q^2 + q + 2, 4]
sage: CyclicSievingPolynomial( S42, cyc_act, order=8 )
q^6 + 2*q^4 + q^2 + 2
sage: CyclicSievingPolynomial([4,2])
q^3 + 2*q^2 + q + 2
```

TESTS:

We check that trac ticket #13997 is handled:

```
sage: CyclicSievingPolynomial( S42, cyc_act, order=8, get_order=True )
[q^6 + 2*q^4 + q^2 + 2, 8]
```

sage.combinat.cyclic_sieving_phenomenon.**orbit_decomposition**(*L*, *cyc_act*)

Returns the orbit decomposition of L by the action of cyc_act

INPUT:

- L – list
- cyc_act – function taking an element of L and returning an element of L (must define a bijection on L)

OUTPUT:

- a list of lists, the orbits under the cyc_act acting on L

EXAMPLES:

```
sage: from sage.combinat.cyclic_sieving_phenomenon import *
sage: S42 = [ Set(S) for S in subsets([1,2,3,4]) if len(S) == 2 ]; S42
[{1, 2}, {1, 3}, {2, 3}, {1, 4}, {2, 4}, {3, 4}]
sage: cyc_act = lambda S: Set( i.mod(4)+1 for i in S)
sage: cyc_act([1,3])
{2, 4}
sage: cyc_act([1,4])
{1, 2}
sage: orbit_decomposition( S42, cyc_act )
[[{2, 4}, {1, 3}], [{1, 2}, {2, 3}, {3, 4}, {1, 4}]]
```

5.1.60 De Bruijn sequences

A De Bruijn sequence is defined as the shortest cyclic sequence that incorporates all substrings of a certain length of an alphabet.

For instance, the $2^3 = 8$ binary strings of length 3 are all included in the following string:

```
sage: DeBruijnSequences(2,3).an_element()
[0, 0, 0, 1, 0, 1, 1, 1]
```

They can be obtained as a subsequence of the *cyclic* De Bruijn sequence of parameters $k = 2$ and $n = 3$:

```
sage: seq = DeBruijnSequences(2,3).an_element()
sage: print Word(seq).string_rep()
00010111
sage: shift = lambda i: [(i+j)%2**3 for j in range(3)]
sage: for i in range(2**3):
...       print (Word(map(lambda (j,b): b if j in shift(i) else '*',
...                                       enumerate(seq))).string_rep())
000*****
*001****
**010***
***101**
****011*
*****111
0*****11
00*****1
```

This sequence is of length k^n, which is best possible as it is the number of k-ary strings of length n. One can equivalently define a De Bruijn sequence of parameters k and n as a cyclic sequence of length k^n in which all substring of length n are different.

See also the Wikipedia article on De Bruijn sequences.

TESTS:

Checking the sequences generated are indeed valid:

```
sage: for n in range(1, 7):
...       for k in range(1, 7):
...          D = DeBruijnSequences(k, n)
...          if not D.an_element() in D:
...              print "Something's dead wrong (n=%s, k=%s)!" %(n,k)
...              break
```

AUTHOR:

- Eviatar Bach (2011): initial version
- Nathann Cohen (2011): Some work on the documentation and defined the __contain__ method

class sage.combinat.debruijn_sequence.**DeBruijnSequences**(k, n)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Represents the De Bruijn sequences of given parameters k and n.

A De Bruijn sequence of parameters k and n is defined as the shortest cyclic sequence that incorporates all substrings of length n a k-ary alphabet.

This class can be used to generate the lexicographically smallest De Bruijn sequence, to count the number of existing De Bruijn sequences or to test whether a given sequence is De Bruijn.

INPUT:

- k – A natural number to define arity. The letters used are the integers $0..k-1$.
- n – A natural number that defines the length of the substring.

EXAMPLES:

Obtaining a De Bruijn sequence:

```
sage: seq = DeBruijnSequences(2, 3).an_element()
sage: print seq
[0, 0, 0, 1, 0, 1, 1, 1]
```

Testing whether it is indeed one:

```
sage: seq in DeBruijnSequences(2, 3)
True
```

The total number for these parameters:

```
sage: DeBruijnSequences(2, 3).cardinality()
2
```

Note: This function only generates one De Bruijn sequence (the smallest lexicographically). Support for generating all possible ones may be added in the future.

TESTS:

Setting `k` to 1 will return 0:

```
sage: DeBruijnSequences(1, 3).an_element()
[0]
```

Setting `n` to 1 will return the alphabet:

```
sage: DeBruijnSequences(3, 1).an_element()
[0, 1, 2]
```

The test suite:

```
sage: d=DeBruijnSequences(2, 3)
sage: TestSuite(d).run()
```

an_element()

Returns the lexicographically smallest De Bruijn sequence with the given parameters.

ALGORITHM:

The algorithm is described in the book "Combinatorial Generation" by Frank Ruskey. This program is based on a Ruby implementation by Jonas Elfström, which is based on the C program by Joe Sadawa.

EXAMPLE:

```
sage: DeBruijnSequences(2, 3).an_element()
[0, 0, 0, 1, 0, 1, 1, 1]
```

cardinality()

Returns the number of distinct De Bruijn sequences for the object's parameters.

EXAMPLE:

```
sage: DeBruijnSequences(2, 5).cardinality()
2048
```

ALGORITHM:

The formula for cardinality is $k!^{k^{n-1}}/k^n$ [1].

REFERENCES:

[1] Rosenfeld, Vladimir Raphael, 2002: Enumerating De Bruijn Sequences. *Communications in Math. and in Computer Chem.*

sage.combinat.debruijn_sequence.**debruijn_sequence**(*k*, *n*)

The generating function for De Bruijn sequences. This avoids the object creation, so is significantly faster than accessing from DeBruijnSequence. For more information, see the documentation there. The algorithm used is from Frank Ruskey's "Combinatorial Generation".

INPUT:

- k – Arity. Must be an integer.
- n – Substring length. Must be an integer.

EXAMPLES:

```
sage: from sage.combinat.debruijn_sequence import debruijn_sequence
sage: debruijn_sequence(3, 1)
[0, 1, 2]
```

sage.combinat.debruijn_sequence.**is_debruijn_sequence**(*seq*, *k*, *n*)

Given a sequence of integer elements in $0..k-1$, tests whether it corresponds to a De Bruijn sequence of parameters k and n.

INPUT:

- seq – Sequence of elements in $0..k-1$.
- n, k – Integers.

EXAMPLE:

```
sage: from sage.combinat.debruijn_sequence import is_debruijn_sequence
sage: s = DeBruijnSequences(2, 3).an_element()
sage: is_debruijn_sequence(s, 2, 3)
True
sage: is_debruijn_sequence(s + [0], 2, 3)
False
sage: is_debruijn_sequence([1] + s[1:], 2, 3)
False
```

5.1.61 Degree sequences

The present module implements the DegreeSequences class, whose instances represent the integer sequences of length n:

```
sage: DegreeSequences(6)
Degree sequences on 6 elements
```

With the object DegreeSequences(n), one can :

- Check whether a sequence is indeed a degree sequence

```
sage: DS = DegreeSequences(5)
sage: [4, 3, 3, 3, 3] in DS
True
sage: [4, 4, 0, 0, 0] in DS
False
```

- List all the possible degree sequences of length n:

```
sage: for seq in DegreeSequences(4):
...       print seq
[0, 0, 0, 0]
```

```
[1, 1, 0, 0]
[2, 1, 1, 0]
[3, 1, 1, 1]
[1, 1, 1, 1]
[2, 2, 1, 1]
[2, 2, 2, 0]
[3, 2, 2, 1]
[2, 2, 2, 2]
[3, 3, 2, 2]
[3, 3, 3, 3]
```

Note: Given a degree sequence, one can obtain a graph realizing it by using `sage.graphs.graph_generators.graphs.DegreeSequence()`. For instance

```
sage: ds = [3, 3, 2, 2, 2, 2, 2, 1, 1, 0]
sage: g = graphs.DegreeSequence(ds)
sage: g.degree_sequence()
[3, 3, 2, 2, 2, 2, 2, 1, 1, 0]
```

Definitions

A sequence of integers $d_1, ..., d_n$ is said to be a *degree sequence* (or *graphic* sequence) if there exists a graph in which vertex i is of degree d_i. It is often required to be *non-increasing*, i.e. that $d_1 \geq ... \geq d_n$. Finding a graph with given degree sequence is known as *graph realization problem*.

An integer sequence need not necessarily be a degree sequence. Indeed, in a degree sequence of length n no integer can be larger than $n-1$ – the degree of a vertex is at most $n-1$ – and the sum of them is at most $n(n-1)$.

Degree sequences are completely characterized by a result from Erdos and Gallai:

Erdos and Gallai: *The sequence of integers $d_1 \geq ... \geq d_n$ is a degree sequence if and only if* $\sum_i d_i$ is even and $\forall i$

$$\sum_{j \leq i} d_j \leq j(j-1) + \sum_{j>i} \min(d_j, i)$$

Alternatively, a degree sequence can be defined recursively :

Havel and Hakimi: *The sequence of integers $d_1 \geq ... \geq d_n$ is a degree sequence if and only if $d_2 - 1, ..., d_{d_1+1} - 1, d_{d_1+2}, ..., d_n$ is also a degree sequence.*

Or equivalently :

Havel and Hakimi (bis): *If there is a realization of an integer sequence as a graph (i.e. if the sequence is a degree sequence), then it can be realized in such a way that the vertex of maximum degree Δ is adjacent to the Δ vertices of highest degree (except itself, of course).*

Algorithms

Checking whether a given sequence is a degree sequence

This is tested using Erdos and Gallai's criterion. It is also checked that the given sequence is non-increasing and has length n.

Iterating through the sequences of length n

From Havel and Hakimi's recursive definition of a degree sequence, one can build an enumeration algorithm as done in [RCES]. It consists in trying to **extend** a current degree sequence on n elements into a degree sequence on $n+$

1 elements by adding a vertex of degree larger than those already present in the sequence. This can be seen as **reversing** the reduction operation described in Havel and Hakimi's characterization. This operation can appear in several different ways :

- Extensions of a degree sequence that do **not** change the value of the maximum element
  - If the maximum element of a given degree sequence is 0, then one can remove it to reduce the sequence, following Havel and Hakimi's rule. Conversely, if the maximum element of the (current) sequence is 0, then one can always extend it by adding a new element of degree 0 to the sequence.

$$0,0,0 \xrightarrow{Extension} \mathbf{0},0,0,0 \xrightarrow{Extension} \mathbf{0},0,0,...,0,0,0 \xrightarrow{Reduction} 0,0,0,0 \xrightarrow{Reduction} 0,0,0$$

  - If there are at least $\Delta+1$ elements of (maximum) degree Δ in a given degree sequence, then one can reduce it by removing a vertex of degree Δ and decreasing the values of Δ elements of value Δ to $\Delta-1$. Conversely, if the maximum element of the (current) sequence is $d>0$, then one can add a new element of degree d to the sequence if it can be linked to d elements of (current) degree $d-1$. Those d vertices of degree $d-1$ hence become vertices of degree d, and so d elements of degree $d-1$ are removed from the sequence while $d+1$ elements of degree d are added to it.

$$3,2,2,2,1 \xrightarrow{Extension} \mathbf{3},3,(2+1),(2+1),(2+1),1=\mathbf{3},3,3,3,3,1 \xrightarrow{Reduction} 3,2,2,2,1$$

- Extension of a degree sequence that changes the value of the maximum element :
  - In the general case, i.e. when the number of elements of value $\Delta,\Delta-1$ is small compared to Δ (i.e. the maximum element of a given degree sequence), reducing a sequence strictly decreases the value of the maximum element. According to Havel and Hakimi's characterization there is only **one** way to reduce a sequence, but reversing this operation is more complicated than in the previous cases. Indeed, the following extensions are perfectly valid according to the reduction rule.

$$2,1,1,0,0 \xrightarrow{Extension} \mathbf{3},(2+1),(1+1),(1+1),0,0=3,3,2,2,0,0 \xrightarrow{Reduction} 2,1,1,0,0$$
$$2,1,1,0,0 \xrightarrow{Extension} \mathbf{3},(2+1),(1+1),1,(0+1),0=3,3,2,1,1,0 \xrightarrow{Reduction} 2,1,1,0,0$$
$$2,1,1,0,0 \xrightarrow{Extension} \mathbf{3},(2+1),1,1,(0+1),(0+1)=3,3,1,1,1,1 \xrightarrow{Reduction} 2,1,1,0,0$$
$$...$$

    In order to extend a current degree sequence while strictly increasing its maximum degree, it is equivalent to pick a set I of elements of the degree sequence with $|I|>\Delta$ in such a way that the $(d_i+1)_{i\in I}$ are the $|I|$ maximum elements of the sequence $(d_i+{}^{1\text{ if }i\in I}_{0\text{ if }i\notin I})_{1\leq i\leq n}$, and to add to this new sequence an element of value $|I|$. The non-increasing sequence containing the elements $|I|$ and $(d_i+{}^{1\text{ if }i\in I}_{0\text{ if }i\notin I})_{1\leq i\leq n}$ can be reduced to $(d_i)_{1\leq i\leq n}$ by Havel and Hakimi's rule.

$$...1,1,2,\mathbf{2},\mathbf{2},2,2,3,3,\underline{3},\mathbf{3},\mathbf{3},\mathbf{4},\mathbf{6},... \xrightarrow{Extension} ...1,1,2,2,2,3,3,\underline{3},\mathbf{3},\mathbf{3},\mathbf{4},\mathbf{4},\mathbf{5},\mathbf{7},...$$

    The number of possible sets I having this property (i.e. the number of possible extensions of a sequence) is smaller than it seems. Indeed, by definition, if $j\notin I$ then for all $i\in I$ the inequality $d_j\leq d_i+1$ holds. Hence, each set I is entirely determined by the largest element d_k of the sequence that it does **not** contain (hence I contains $\{1,...,k-1\}$), and by the cardinalities of $\{i\in I:d_i=d_k\}$ and $\{i\in I:d_i=d_k-1\}$.

$$I=\{i\in I:d_i=d_k\}\cup\{i\in I:d_i=d_k-1\}\cup\{i:d_i>d_k\}$$

    The number of possible extensions is hence at most cubic, and is easily enumerated.

About the implementation

In the actual implementation of the enumeration algorithm, the degree sequence is stored differently for reasons of efficiency.

Indeed, when enumerating all the degree sequences of length n, Sage first allocates an array `seq` of $n+1$ integers where `seq[i]` is the number of elements of value `i` in the current sequence. Obviously, `seq[n]=0` holds in permanence : it is useful to allocate a larger array than necessary to simplify the code. The `seq` array is a global variable.

The recursive function `enum(depth, maximum)` is the one building the list of sequences. It builds the list of degree sequences of length n which *extend* the sequence currently stored in `seq[0]...seq[depth-1]`. When it is called, `maximum` must be set to the maximum value of an element in the partial sequence `seq[0]...seq[depth-1]`.

If during its run the function `enum` heavily works on the content of the `seq` array, the value of `seq` is the **same** before and after the run of `enum`.

Extending the current partial sequence

The two cases for which the maximum degree of the partial sequence does not change are easy to detect. It is (sligthly) harder to enumerate all the sets I corresponding to possible extensions of the partial sequence. As said previously, to each set I one can associate an integer `current_box` such that I contains all the i satisfying $d_i > current\_box$. The variable `taken` represents the number of all such elements i, so that when enumerating all possible sets I in the algorithm we have the equality

$$I = \text{taken } + \text{ number of elements of value } current\_box + \text{ number of elements of value } current\_box - 1$$

References

Author

Nathann Cohen

Tests

The sequences produced by random graphs *are* degree sequences:

```
sage: n = 30
sage: DS = DegreeSequences(30)
sage: for i in range(10):
...       g = graphs.RandomGNP(n,.2)
...       if not g.degree_sequence() in DS:
...           print "Something is very wrong !"
```

Checking that we indeed enumerate *all* the degree sequences for $n = 5$:

```
sage: ds1 = Set([tuple(g.degree_sequence()) for g in graphs(5)])
sage: ds2 = Set(map(tuple,list(DegreeSequences(5))))
sage: ds1 == ds2
True
```

Checking the consistency of enumeration and test:

```
sage: DS = DegreeSequences(6)
sage: all(seq in DS for seq in DS)
True
```

Warning: For the moment, iterating over all degree sequences involves building the list of them first, then iterate on this list. This is obviously bad, as it requires uselessly a **lot** of memory for large values of n.
This should be changed. Updating the code does not require more than a couple of minutes.

class sage.combinat.degree_sequences.**DegreeSequences**(n)
Degree Sequences

An instance of this class represents the degree sequences of graphs on a given number n of vertices. It can be used to list and count them, as well as to test whether a sequence is a degree sequence. For more information, please refer to the documentation of the DegreeSequence module.

EXAMPLE:

```
sage: DegreeSequences(8)
Degree sequences on 8 elements
sage: [3,3,2,2,2,2,2,2] in DegreeSequences(8)
True
```

5.1.62 Derangements

AUTHORS:

- Alasdair McAndrew (2010-05): Initial version
- Travis Scrimshaw (2013-03-30): Put derangements into category framework

class sage.combinat.derangements.**Derangement**(*parent*, **args*, ***kwds*)
Bases: sage.combinat.combinat.CombinatorialElement

A derangement.

A derangement on a set S is a permutation σ such that $\sigma(x) \neq x$ for all $x \in S$, i.e. σ is a permutation of S with no fixed points.

EXAMPLES:

```
sage: D = Derangements(4)
sage: elt = D([4,3,2,1])
sage: TestSuite(elt).run()
```

to_permutation()
Return the permutation corresponding to self.

EXAMPLES:

```
sage: D = Derangements(4)
sage: p = D([4,3,2,1]).to_permutation(); p
[4, 3, 2, 1]
sage: type(p)
<class 'sage.combinat.permutation.StandardPermutations_all_with_category.element_class'>
sage: D = Derangements([1, 3, 3, 4])
sage: D[0].to_permutation()
Traceback (most recent call last):
...
ValueError: Can only convert to a permutation for derangements of [1, 2, ..., n]
```

class sage.combinat.derangements.**Derangements**(x)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The class of all derangements of a set or multiset.

A derangement on a set S is a permutation σ such that $\sigma(x) \neq x$ for all $x \in S$, i.e. σ is a permutation of S with no fixed points.

For an integer, or a list or string with all elements distinct, the derangements are obtained by a standard result described in [DerUB]. For a list or string with repeated elements, the derangements are formed by computing all permutations of the input and discarding all non-derangements.

INPUT:

- x – Can be an integer which corresponds to derangements of $\{1, 2, 3, \ldots, x\}$, a list, or a string

REFERENCES:

- Wikipedia article Derangement

EXAMPLES:

```
sage: D1 = Derangements([2,3,4,5])
sage: D1.list()
[[3, 4, 5, 2],
 [5, 4, 2, 3],
 [3, 5, 2, 4],
 [4, 5, 3, 2],
 [4, 2, 5, 3],
 [5, 2, 3, 4],
 [5, 4, 3, 2],
 [4, 5, 2, 3],
 [3, 2, 5, 4]]
sage: D1.cardinality()
9
sage: D1.random_element() # random
[4, 2, 5, 3]
sage: D2 = Derangements([1,2,3,1,2,3])
sage: D2.cardinality()
10
sage: D2.list()
[[2, 1, 1, 3, 3, 2],
 [2, 1, 2, 3, 3, 1],
 [2, 3, 1, 2, 3, 1],
 [2, 3, 1, 3, 1, 2],
 [2, 3, 2, 3, 1, 1],
 [3, 1, 1, 2, 3, 2],
 [3, 1, 2, 2, 3, 1],
 [3, 1, 2, 3, 1, 2],
 [3, 3, 1, 2, 1, 2],
 [3, 3, 2, 2, 1, 1]]
sage: D2.random_element() # random
[2, 3, 1, 3, 1, 2]
```

Element
alias of Derangement

cardinality()
Counts the number of derangements of a positive integer, a list, or a string. The list or string may contain repeated elements. If an integer n is given, the the value returned is the number of derangements of $[1, 2, 3, \ldots, n]$.

For an integer, or a list or string with all elements distinct, the value is obtained by the standard result $D_2 = 1, D_3 = 2, D_n = (n-1)(D_{n-1} + D_{n-2})$.

For a list or string with repeated elements, the number of derangements is computed by Macmahon's theorem. If the numbers of repeated elements are $a_1, a_2, \ldots, a_k$ then the number of derangements is given by the coefficient of $x_1 x_2 \cdots x_k$ in the expansion of $\prod_{i=0}^{k}(S - s_i)^{a_i}$ where $S = x_1 + x_2 + \cdots + x_k$.

EXAMPLES:

```
sage: D = Derangements(5)
sage: D.cardinality()
44
sage: D = Derangements([1,44,918,67,254])
sage: D.cardinality()
44
sage: D = Derangements(['A','AT','CAT','CATS','CARTS'])
sage: D.cardinality()
44
sage: D = Derangements('UNCOPYRIGHTABLE')
sage: D.cardinality()
481066515734
sage: D = Derangements([1,1,2,2,3,3])
sage: D.cardinality()
10
sage: D = Derangements('SATTAS')
sage: D.cardinality()
10
sage: D = Derangements([1,1,2,2,2])
sage: D.cardinality()
0
```

random_element()

Produces all derangements of a positive integer, a list, or a string. The list or string may contain repeated elements. If an integer n is given, then a random derangements of $[1, 2, 3, \ldots, n]$ is returned

For an integer, or a list or string with all elements distinct, the value is obtained by an algorithm described in [Martinez08]. For a list or string with repeated elements the derangement is formed by choosing an element at random from the list of all possible derangements.

OUTPUT:

A single list or string containing a derangement, or an empty list if there are no derangements.

REFERENCES:

EXAMPLES:

```
sage: D = Derangements(4)
sage: D.random_element() # random
[2, 3, 4, 1]
sage: D = Derangements(['A','AT','CAT','CATS','CARTS','CARETS'])
sage: D.random_element() # random
['AT', 'CARTS', 'A', 'CAT', 'CARETS', 'CATS']
sage: D = Derangements('UNCOPYRIGHTABLE')
sage: D.random_element() # random
['C', 'U', 'I', 'H', 'O', 'G', 'N', 'B', 'E', 'L', 'A', 'R', 'P', 'Y', 'T']
sage: D = Derangements([1,1,1,1,2,2,2,2,3,3,3,3])
sage: D.random_element() # random
[3, 2, 2, 3, 1, 3, 1, 3, 2, 1, 1, 2]
sage: D = Derangements('ESSENCES')
sage: D.random_element() # random
['N', 'E', 'E', 'C', 'S', 'S', 'S', 'E']
sage: D = Derangements([1,1,2,2,2])
sage: D.random_element()
```

```
[]
```

5.1.63 Descent Algebras

AUTHORS:

- Travis Scrimshaw (2013-07-28): Initial version

class sage.combinat.descent_algebra.**DescentAlgebra**(*R*, *n*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Solomon's descent algebra.

The descent algebra Σ_n over a ring R is a subalgebra of the symmetric group algebra RS_n. (The product in the latter algebra is defined by $(pq)(i) = q(p(i))$ for any two permutations p and q in S_n and every $i \in \{1, 2, \ldots, n\}$. The algebra Σ_n inherits this product.)

There are three bases currently implemented for Σ_n:

- the standard basis D_S of (sums of) descent classes, indexed by subsets S of $\{1, 2, \ldots, n-1\}$,
- the subset basis B_p, indexed by compositions p of n,
- the idempotent basis I_p, indexed by compositions p of n, which is used to construct the mutually orthogonal idempotents of the symmetric group algebra.

The idempotent basis is only defined when R is a $\mathbf{Q}$-algebra.

We follow the notations and conventions in [GR1989], apart from the order of multiplication being different from the one used in that article. Schocker's exposition [Schocker2004], in turn, uses the same order of multiplication as we are, but has different notations for the bases.

INPUT:

- R – the base ring
- n – a nonnegative integer

REFERENCES:

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D(); D
Descent algebra of 4 over Rational Field in the standard basis
sage: B = DA.B(); B
Descent algebra of 4 over Rational Field in the subset basis
sage: I = DA.I(); I
Descent algebra of 4 over Rational Field in the idempotent basis
sage: basis_B = B.basis()
sage: elt = basis_B[Composition([1,2,1])] + 4*basis_B[Composition([1,3])]; elt
B[1, 2, 1] + 4*B[1, 3]
sage: D(elt)
5*D{} + 5*D{1} + D{1, 3} + D{3}
sage: I(elt)
7/6*I[1, 1, 1, 1] + 2*I[1, 1, 2] + 3*I[1, 2, 1] + 4*I[1, 3]
```

As syntactic sugar, one can use the notation D[i,...,l] to construct elements of the basis; note that for the empty set one must use D[[]] due to Python's syntax:

```
sage: D[[]] + D[2] + 2*D[1,2]
D{} + 2*D{1, 2} + D{2}
```

The same syntax works for the other bases:

```
sage: I[1,2,1] + 3*I[4] + 2*I[3,1]
I[1, 2, 1] + 2*I[3, 1] + 3*I[4]
```

TESTS:

We check that we can go back and forth between our bases:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D()
sage: B = DA.B()
sage: I = DA.I()
sage: all(D(B(b)) == b for b in D.basis())
True
sage: all(D(I(b)) == b for b in D.basis())
True
sage: all(B(D(b)) == b for b in B.basis())
True
sage: all(B(I(b)) == b for b in B.basis())
True
sage: all(I(D(b)) == b for b in I.basis())
True
sage: all(I(B(b)) == b for b in I.basis())
True
```

class B (*alg*, *prefix='B'*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The subset basis of a descent algebra (indexed by compositions).

The subset basis $(B_S)_{S\subseteq\{1,2,\ldots,n-1\}}$ of Σ_n is formed by

$$B_S = \sum_{T\subseteq S} D_T,$$

where $(D_S)_{S\subseteq\{1,2,\ldots,n-1\}}$ is the standard basis. However it is more natural to index the subset basis by compositions of n under the bijection $\{i_1, i_2, \ldots, i_k\} \mapsto (i_1, i_2 - i_1, i_3 - i_2, \ldots, i_k - i_{k-1}, n - i_k)$ (where $i_1 < i_2 < \cdots < i_k$), which is what Sage uses to index the basis.

The basis element B_p is denoted Ξ^p in [Schocker2004].

By using compositions of n, the product B_pB_q becomes a sum over the non-negative-integer matrices M with row sum p and column sum q. The summand corresponding to M is B_c, where c is the composition obtained by reading M row-by-row from left-to-right and top-to-bottom and removing all zeroes. This multiplication rule is commonly called "Solomon's Mackey formula".

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: list(B.basis())
[B[1, 1, 1, 1], B[1, 1, 2], B[1, 2, 1], B[1, 3],
 B[2, 1, 1], B[2, 2], B[3, 1], B[4]]
```

one_basis()

Return the identity element which is the composition $[n]$, as per `AlgebrasWithBasis.ParentMethods.one_basis`.

EXAMPLES:

```
sage: DescentAlgebra(QQ, 4).B().one_basis()
[4]
sage: DescentAlgebra(QQ, 0).B().one_basis()
[]

sage: all( U * DescentAlgebra(QQ, 3).B().one() == U
....:      for U in DescentAlgebra(QQ, 3).B().basis() )
True
```

product_on_basis(p, q)

Return $B_p B_q$, where p and q are compositions of n.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: p = Composition([1,2,1])
sage: q = Composition([3,1])
sage: B.product_on_basis(p, q)
B[1, 1, 1, 1] + 2*B[1, 2, 1]
```

to_D_basis(p)

Return B_p as a linear combination of D-basis elements.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: D = DA.D()
sage: map(D, B.basis()) # indirect doctest
[D{} + D{1} + D{1, 2} + D{1, 2, 3}
  + D{1, 3} + D{2} + D{2, 3} + D{3},
 D{} + D{1} + D{1, 2} + D{2},
 D{} + D{1} + D{1, 3} + D{3},
 D{} + D{1},
 D{} + D{2} + D{2, 3} + D{3},
 D{} + D{2},
 D{} + D{3},
 D{}]
```

TESTS:

Check to make sure the empty case is handled correctly:

```
sage: DA = DescentAlgebra(QQ, 0)
sage: B = DA.B()
sage: D = DA.D()
sage: map(D, B.basis())
[D{}]
```

to_I_basis(p)

Return B_p as a linear combination of I-basis elements.

This is done using the formula

$$B_p = \sum_{q \leq p} \frac{1}{\mathbf{k}!(q,p)} I_q,$$

where $\leq$ is the refinement order and $\mathbf{k}!(q, p)$ is defined as follows: When $q \leq p$, we can write q as a concatenation $q_{(1)}q_{(2)}\cdots q_{(k)}$ with each $q_{(i)}$ being a composition of the i-th entry of p, and then we set $\mathbf{k}!(q, p)$ to be $l(q_{(1)})!l(q_{(2)})!\cdots l(q_{(k)})!$, where $l(r)$ denotes the number of parts of any composition r.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: I = DA.I()
sage: map(I, B.basis()) # indirect doctest
[I[1, 1, 1, 1],
 1/2*I[1, 1, 1, 1] + I[1, 1, 2],
 1/2*I[1, 1, 1, 1] + I[1, 2, 1],
 1/6*I[1, 1, 1, 1] + 1/2*I[1, 1, 2] + 1/2*I[1, 2, 1] + I[1, 3],
 1/2*I[1, 1, 1, 1] + I[2, 1, 1],
 1/4*I[1, 1, 1, 1] + 1/2*I[1, 1, 2] + 1/2*I[2, 1, 1] + I[2, 2],
 1/6*I[1, 1, 1, 1] + 1/2*I[1, 2, 1] + 1/2*I[2, 1, 1] + I[3, 1],
 1/24*I[1, 1, 1, 1] + 1/6*I[1, 1, 2] + 1/6*I[1, 2, 1]
  + 1/2*I[1, 3] + 1/6*I[2, 1, 1] + 1/2*I[2, 2] + 1/2*I[3, 1] + I[4]]
```

to_nsym(*p*)

Return B_p as an element in $NSym$, the non-commutative symmetric functions.

This maps B_p to S_p where S denotes the Complete basis of $NSym$.

EXAMPLES:

```
sage: B = DescentAlgebra(QQ, 4).B()
sage: S = NonCommutativeSymmetricFunctions(QQ).Complete()
sage: map(S, B.basis()) # indirect doctest
[S[1, 1, 1, 1],
 S[1, 1, 2],
 S[1, 2, 1],
 S[1, 3],
 S[2, 1, 1],
 S[2, 2],
 S[3, 1],
 S[4]]
```

class DescentAlgebra.**D**(*alg*, *prefix='D'*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The standard basis of a descent algebra.

This basis is indexed by $S \subseteq \{1, 2, \ldots, n-1\}$, and the basis vector indexed by S is the sum of all permutations, taken in the symmetric group algebra RS_n, whose descent set is S. We denote this basis vector by D_S.

Occasionally this basis appears in literature but indexed by compositions of n rather than subsets of $\{1, 2, \ldots, n-1\}$. The equivalence between these two indexings is owed to the bijection from the power set of $\{1, 2, \ldots, n-1\}$ to the set of all compositions of n which sends every subset $\{i_1, i_2, \ldots, i_k\}$ of $\{1, 2, \ldots, n-1\}$ (with $i_1 < i_2 < \cdots < i_k$) to the composition $(i_1, i_2 - i_1, \ldots, i_k - i_{k-1}, n - i_k)$.

The basis element corresponding to a composition p (or to the subset of $\{1, 2, \ldots, n-1\}$) is denoted Δ^p in [Schocker2004].

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D()
sage: list(D.basis())
```

```
[D{}, D{1}, D{2}, D{3}, D{1, 2}, D{1, 3}, D{2, 3}, D{1, 2, 3}]

sage: DA = DescentAlgebra(QQ, 0)
sage: D = DA.D()
sage: list(D.basis())
[D{}]
```

one_basis()

Return the identity element, as per `AlgebrasWithBasis.ParentMethods.one_basis`.

EXAMPLES:

```
sage: DescentAlgebra(QQ, 4).D().one_basis()
()
sage: DescentAlgebra(QQ, 0).D().one_basis()
()

sage: all( U * DescentAlgebra(QQ, 3).D().one() == U
....:      for U in DescentAlgebra(QQ, 3).D().basis() )
True
```

product_on_basis(S, T)

Return $D_S D_T$, where S and T are subsets of $[n-1]$.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D()
sage: D.product_on_basis((1, 3), (2,))
D{} + D{1} + D{1, 2} + 2*D{1, 2, 3} + D{1, 3} + D{2} + D{2, 3} + D{3}
```

to_B_basis(S)

Return D_S as a linear combination of B_p-basis elements.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D()
sage: B = DA.B()
sage: map(B, D.basis()) # indirect doctest
[B[4],
 B[1, 3] - B[4],
 B[2, 2] - B[4],
 B[3, 1] - B[4],
 B[1, 1, 2] - B[1, 3] - B[2, 2] + B[4],
 B[1, 2, 1] - B[1, 3] - B[3, 1] + B[4],
 B[2, 1, 1] - B[2, 2] - B[3, 1] + B[4],
 B[1, 1, 1, 1] - B[1, 1, 2] - B[1, 2, 1] + B[1, 3]
  - B[2, 1, 1] + B[2, 2] + B[3, 1] - B[4]]
```

to_symmetric_group_algebra_on_basis(S)

Return D_S as a linear combination of basis elements in the symmetric group algebra.

EXAMPLES:

```
sage: D = DescentAlgebra(QQ, 4).D()
sage: [D.to_symmetric_group_algebra_on_basis(tuple(b))
....:  for b in Subsets(3)]
[[1, 2, 3, 4],
 [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3],
 [1, 3, 2, 4] + [1, 4, 2, 3] + [2, 3, 1, 4]
  + [2, 4, 1, 3] + [3, 4, 1, 2],
```

```
[1, 2, 4, 3] + [1, 3, 4, 2] + [2, 3, 4, 1],
[3, 2, 1, 4] + [4, 2, 1, 3] + [4, 3, 1, 2],
[2, 1, 4, 3] + [3, 1, 4, 2] + [3, 2, 4, 1]
 + [4, 1, 3, 2] + [4, 2, 3, 1],
[1, 4, 3, 2] + [2, 4, 3, 1] + [3, 4, 2, 1],
[4, 3, 2, 1]]
```

class DescentAlgebra.**I**(*alg*, *prefix='I'*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The idempotent basis of a descent algebra.

The idempotent basis $(I_p)_{p \models n}$ is a basis for Σ_n whenever the ground ring is a $\mathbf{Q}$-algebra. One way to compute it is using the formula (Theorem 3.3 in [GR1989])

$$I_p = \sum_{q \leq p} \frac{(-1)^{l(q)-l(p)}}{\mathbf{k}(q,p)} B_q,$$

where $\leq$ is the refinement order and $l(r)$ denotes the number of parts of any composition r, and where $\mathbf{k}(q,p)$ is defined as follows: When $q \leq p$, we can write q as a concatenation $q_{(1)}q_{(2)}\cdots q_{(k)}$ with each $q_{(i)}$ being a composition of the i-th entry of p, and then we set $\mathbf{k}(q,p)$ to be the product $l(q_{(1)})l(q_{(2)})\cdots l(q_{(k)})$.

Let $\lambda(p)$ denote the partition obtained from a composition p by sorting. This basis is called the idempotent basis since for any q such that $\lambda(p) = \lambda(q)$, we have:

$$I_p I_q = s(\lambda) I_p$$

where λ denotes $\lambda(p) = \lambda(q)$, and where $s(\lambda)$ is the stabilizer of λ in S_n. (This is part of Theorem 4.2 in [GR1989].)

It is also straightforward to compute the idempotents E_λ for the symmetric group algebra by the formula (Theorem 3.2 in [GR1989]):

$$E_\lambda = \frac{1}{k!} \sum_{\lambda(p)=\lambda} I_p.$$

Note: The basis elements are not orthogonal idempotents.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: I = DA.I()
sage: list(I.basis())
[I[1, 1, 1, 1], I[1, 1, 2], I[1, 2, 1], I[1, 3], I[2, 1, 1], I[2, 2], I[3, 1], I[4]]
```

idempotent(*la*)

Return the idemponent corresponding to the partition `la` of n.

EXAMPLES:

```
sage: I = DescentAlgebra(QQ, 4).I()
sage: E = I.idempotent([3,1]); E
1/2*I[1, 3] + 1/2*I[3, 1]
sage: E*E == E
True
sage: E2 = I.idempotent([2,1,1]); E2
1/6*I[1, 1, 2] + 1/6*I[1, 2, 1] + 1/6*I[2, 1, 1]
```

```
sage: E2*E2 == E2
True
sage: E*E2 == I.zero()
True
```

one()

Return the identity element, which is $B_{[n]}$, in the I basis.

EXAMPLES:

```
sage: DescentAlgebra(QQ, 4).I().one()
1/24*I[1, 1, 1, 1] + 1/6*I[1, 1, 2] + 1/6*I[1, 2, 1]
 + 1/2*I[1, 3] + 1/6*I[2, 1, 1] + 1/2*I[2, 2]
 + 1/2*I[3, 1] + I[4]
sage: DescentAlgebra(QQ, 0).I().one()
I[]
```

TESTS:

```
sage: all( U * DescentAlgebra(QQ, 3).I().one() == U
....:      for U in DescentAlgebra(QQ, 3).I().basis() )
True
```

one_basis()

The element 1 is not (generally) a basis vector in the I basis, thus this returns a `TypeError`.

EXAMPLES:

```
sage: DescentAlgebra(QQ, 4).I().one_basis()
Traceback (most recent call last):
...
TypeError: 1 is not a basis element in the I basis.
```

product_on_basis(p, q)

Return $I_p I_q$, where p and q are compositions of n.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: I = DA.I()
sage: p = Composition([1,2,1])
sage: q = Composition([3,1])
sage: I.product_on_basis(p, q)
0
sage: I.product_on_basis(p, p)
2*I[1, 2, 1]
```

to_B_basis(p)

Return I_p as a linear combination of B-basis elements.

This is computed using the formula (Theorem 3.3 in [GR1989])

$$I_p = \sum_{q \leq p} \frac{(-1)^{l(q)-l(p)}}{\mathbf{k}(q,p)} B_q,$$

where $\leq$ is the refinement order and $l(r)$ denotes the number of parts of any composition r, and where $\mathbf{k}(q,p)$ is defined as follows: When $q \leq p$, we can write q as a concatenation $q_{(1)}q_{(2)}\cdots q_{(k)}$ with each $q_{(i)}$ being a composition of the i-th entry of p, and then we set $\mathbf{k}(q,p)$ to be $l(q_{(1)})l(q_{(2)})\cdots l(q_{(k)})$.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: I = DA.I()
```

```
sage: map(B, I.basis()) # indirect doctest
[B[1, 1, 1, 1],
 -1/2*B[1, 1, 1, 1] + B[1, 1, 2],
 -1/2*B[1, 1, 1, 1] + B[1, 2, 1],
 1/3*B[1, 1, 1, 1] - 1/2*B[1, 1, 2] - 1/2*B[1, 2, 1] + B[1, 3],
 -1/2*B[1, 1, 1, 1] + B[2, 1, 1],
 1/4*B[1, 1, 1, 1] - 1/2*B[1, 1, 2] - 1/2*B[2, 1, 1] + B[2, 2],
 1/3*B[1, 1, 1, 1] - 1/2*B[1, 2, 1] - 1/2*B[2, 1, 1] + B[3, 1],
 -1/4*B[1, 1, 1, 1] + 1/3*B[1, 1, 2] + 1/3*B[1, 2, 1]
  - 1/2*B[1, 3] + 1/3*B[2, 1, 1] - 1/2*B[2, 2]
  - 1/2*B[3, 1] + B[4]]
```

DescentAlgebra.**a_realization**()

Return a particular realization of `self` (the B-basis).

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: DA.a_realization()
Descent algebra of 4 over Rational Field in the subset basis
```

class sage.combinat.descent_algebra.**DescentAlgebraBases**(*base*)

Bases: sage.categories.realizations.Category_realization_of_parent

The category of bases of a descent algebra.

class ElementMethods

to_symmetric_group_algebra()

Return `self` in the symmetric group algebra.

EXAMPLES:

```
sage: B = DescentAlgebra(QQ, 4).B()
sage: B[1,3].to_symmetric_group_algebra()
[1, 2, 3, 4] + [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3]
sage: I = DescentAlgebra(QQ, 4).I()
sage: elt = I(B[1,3])
sage: elt.to_symmetric_group_algebra()
[1, 2, 3, 4] + [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3]
```

class DescentAlgebraBases.**ParentMethods**

is_commutative()

Return whether this descent algebra is commutative.

EXAMPLES:

```
sage: B = DescentAlgebra(QQ, 4).B()
sage: B.is_commutative()
False
sage: B = DescentAlgebra(QQ, 1).B()
sage: B.is_commutative()
True
```

is_field(*proof=True*)

Return whether this descent algebra is a field.

EXAMPLES:

```
sage: B = DescentAlgebra(QQ, 4).B()
sage: B.is_field()
False
sage: B = DescentAlgebra(QQ, 1).B()
sage: B.is_field()
True
```

to_symmetric_group_algebra()

Morphism from `self` to the symmetric group algebra.

EXAMPLES:

```
sage: D = DescentAlgebra(QQ, 4).D()
sage: D.to_symmetric_group_algebra(D[1,3])
[2, 1, 4, 3] + [3, 1, 4, 2] + [3, 2, 4, 1] + [4, 1, 3, 2] + [4, 2, 3, 1]
sage: B = DescentAlgebra(QQ, 4).B()
sage: B.to_symmetric_group_algebra(B[1,2,1])
[1, 2, 3, 4] + [1, 2, 4, 3] + [1, 3, 4, 2] + [2, 1, 3, 4]
 + [2, 1, 4, 3] + [2, 3, 4, 1] + [3, 1, 2, 4] + [3, 1, 4, 2]
 + [3, 2, 4, 1] + [4, 1, 2, 3] + [4, 1, 3, 2] + [4, 2, 3, 1]
```

to_symmetric_group_algebra_on_basis(S)

Return the basis element index by `S` as a linear combination of basis elements in the symmetric group algebra.

EXAMPLES:

```
sage: B = DescentAlgebra(QQ, 3).B()
sage: [B.to_symmetric_group_algebra_on_basis(c)
....:  for c in Compositions(3)]
[[1, 2, 3] + [1, 3, 2] + [2, 1, 3]
  + [2, 3, 1] + [3, 1, 2] + [3, 2, 1],
 [1, 2, 3] + [2, 1, 3] + [3, 1, 2],
 [1, 2, 3] + [1, 3, 2] + [2, 3, 1],
 [1, 2, 3]]
sage: I = DescentAlgebra(QQ, 3).I()
sage: [I.to_symmetric_group_algebra_on_basis(c)
....:  for c in Compositions(3)]
[[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1]
  + [3, 1, 2] + [3, 2, 1],
 1/2*[1, 2, 3] - 1/2*[1, 3, 2] + 1/2*[2, 1, 3]
  - 1/2*[2, 3, 1] + 1/2*[3, 1, 2] - 1/2*[3, 2, 1],
 1/2*[1, 2, 3] + 1/2*[1, 3, 2] - 1/2*[2, 1, 3]
  + 1/2*[2, 3, 1] - 1/2*[3, 1, 2] - 1/2*[3, 2, 1],
 1/3*[1, 2, 3] - 1/6*[1, 3, 2] - 1/6*[2, 1, 3]
  - 1/6*[2, 3, 1] - 1/6*[3, 1, 2] + 1/3*[3, 2, 1]]
```

DescentAlgebraBases.**super_categories**()

The super categories of `self`.

EXAMPLES:

```
sage: from sage.combinat.descent_algebra import DescentAlgebraBases
sage: DA = DescentAlgebra(QQ, 4)
sage: bases = DescentAlgebraBases(DA)
sage: bases.super_categories()
[Category of finite dimensional algebras with basis over Rational Field,
 Category of realizations of Descent algebra of 4 over Rational Field]
```

5.1.64 Combinatorial Designs and Incidence Structures

All designs can be accessed by `designs.<tab>` and are listed in the design catalog:

- *Catalog of designs*

Design-related classes

- *Incidence structures (i.e. hypergraphs, i.e. set systems)*
- *Covering designs: coverings of -element subsets of a -set by -sets*

Constructions

- *Block designs*
- *Balanced Incomplete Block Designs (BIBD)*
- *Resolvable Balanced Incomplete Block Design (RBIBD)*
- *Group-Divisible Designs (GDD)*
- *Mutually Orthogonal Latin Squares (MOLS)*
- *Orthogonal arrays (OA)*
- *Orthogonal arrays (build recursive constructions)*
- *Orthogonal arrays (find recursive constructions)*
- *Difference families*
- *Difference Matrices*
- *Steiner Quadruple Systems*
- *Two-graphs*
- *Database of small combinatorial designs*

Technical things

- *External Representations of Block Designs*
- *Cython functions for combinatorial designs*
- *Hypergraph isomorphic copy search*
- *Evenly distributed sets in finite fields*

5.1.65 Combinatorial design features that are imported by default in the interpreter namespace

5.1.66 Balanced Incomplete Block Designs (BIBD)

This module gathers everything related to Balanced Incomplete Block Designs. One can build a BIBD (or check that it can be built) with `balanced_incomplete_block_design()`:

```
sage: BIBD = designs.balanced_incomplete_block_design(7,3)
```

In particular, Sage can build a $(v, k, 1)$-BIBD when one exists for all $k \leq 5$. The following functions are available:

balanced_incomplete_block_design()	Return a BIBD of parameters v, k.
BIBD_from_TD()	Return a BIBD through TD-based constructions.
BIBD_from_difference_family()	Return the BIBD associated to the difference family D on the group G.
BIBD_from_PBD()	Return a $(v, k, 1)$-BIBD from a (r, K)-PBD where $r = (v-1)/(k-1)$.
PBD_from_TD()	Return a $(kt, \{k, t\})$-PBD if $u = 0$ and a $(kt + u, \{k, k + 1, t, u\})$-PBD otherwise.
steiner_triple_system()	Return a Steiner Triple System.
v_5_1_BIBD()	Return a $(v, 5, 1)$-BIBD.
v_4_1_BIBD()	Return a $(v, 4, 1)$-BIBD.
PBD_4_5_8_9_12()	Return a $(v, \{4, 5, 8, 9, 12\})$-PBD on v elements.
BIBD_5q_5_for_q_prime_power()	Return a $(5q, 5, 1)$-BIBD with $q \equiv 1 \pmod 4$ a prime power.

Construction of BIBD when $k = 4$

Decompositions of K_v into K_4 (i.e. $(v, 4, 1)$-BIBD) are built following Douglas Stinson's construction as presented in [Stinson2004] page 167. It is based upon the construction of $(v\{4, 5, 8, 9, 12\})$-PBD (see the doc of PBD_4_5_8_9_12()), knowing that a $(v\{4, 5, 8, 9, 12\})$-PBD on v points can always be transformed into a $((k-1)v + 1, 4, 1)$-BIBD, which covers all possible cases of $(v, 4, 1)$-BIBD.

Construction of BIBD when $k = 5$

Decompositions of K_v into K_4 (i.e. $(v, 4, 1)$-BIBD) are built following Clayton Smith's construction [ClaytonSmith].

Functions

sage.combinat.designs.bibd.**BIBD_5q_5_for_q_prime_power**(q)

Return a $(5q, 5, 1)$-BIBD with $q \equiv 1 \pmod 4$ a prime power.

See Theorem 24 [ClaytonSmith].

INPUT:

- q (integer) – a prime power such that $q \equiv 1 \pmod 4$.

EXAMPLES:

```
sage: from sage.combinat.designs.bibd import BIBD_5q_5_for_q_prime_power
sage: for q in [25, 45, 65, 85, 125, 145, 185, 205, 305, 405, 605]: # long time
....:     _ = BIBD_5q_5_for_q_prime_power(q/5)                     # long time
```

sage.combinat.designs.bibd.**BIBD_from_PBD**(*PBD*, *v*, *k*, *check=True*, *base_cases={}*)

Return a $(v, k, 1)$-BIBD from a (r, K)-PBD where $r = (v-1)/(k-1)$.

This is Theorem 7.20 from [Stinson2004].

INPUT:

- v, k – integers.
- PBD – A PBD on $r = (v-1)/(k-1)$ points, such that for any block of PBD of size s there must exist a $((k-1)s + 1, k, 1)$-BIBD.
- check (boolean) – whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to True by default.
- base_cases – caching system, for internal use.

EXAMPLES:

```
sage: from sage.combinat.designs.bibd import PBD_4_5_8_9_12
sage: from sage.combinat.designs.bibd import BIBD_from_PBD
sage: from sage.combinat.designs.bibd import is_pairwise_balanced_design
sage: PBD = PBD_4_5_8_9_12(17)
sage: bibd = is_pairwise_balanced_design(BIBD_from_PBD(PBD,52,4),52,[4])
```

sage.combinat.designs.bibd.**BIBD_from_TD**(*v*, *k*, *existence=False*)

Return a BIBD through TD-based constructions.

INPUT:

- `v, k` (integers) – computes a $(v, k, 1)$-BIBD.
- `existence` (boolean) – instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`).
  - `False` – meaning that the design does not exist.

This method implements three constructions:

- If there exists a $TD(k, v)$ and a $(v, k, 1)$-BIBD then there exists a $(kv, k, 1)$-BIBD.

  The BIBD is obtained from all blocks of the TD, and from the blocks of the $(v, k, 1)$-BIBDs defined over the k groups of the TD.

- If there exists a $TD(k, v)$ and a $(v + 1, k, 1)$-BIBD then there exists a $(kv + 1, k, 1)$-BIBD.

  The BIBD is obtained from all blocks of the TD, and from the blocks of the $(v + 1, k, 1)$-BIBDs defined over the sets $V_1 \cup \infty, \ldots, V_k \cup \infty$ where the $V_1, \ldots, V_k$ are the groups of the TD.

- If there exists a $TD(k, v)$ and a $(v + k, k, 1)$-BIBD then there exists a $(kv + k, k, 1)$-BIBD.

  The BIBD is obtained from all blocks of the TD, and from the blocks of the $(v + k, k, 1)$-BIBDs defined over the sets $V_1 \cup \{\infty_1, \ldots, \infty_k\}, \ldots, V_k \cup \{\infty_1, \ldots, \infty_k\}$ where the $V_1, \ldots, V_k$ are the groups of the TD. By making sure that all copies of the $(v + k, k, 1)$-BIBD contain the block $\{\infty_1, \ldots, \infty_k\}$, the result is also a BIBD.

These constructions can be found in http://www.argilo.net/files/bibd.pdf.

EXAMPLES:

First construction:

```
sage: from sage.combinat.designs.bibd import BIBD_from_TD
sage: BIBD_from_TD(25,5,existence=True)
True
sage: _ = BlockDesign(25,BIBD_from_TD(25,5))
```

Second construction:

```
sage: from sage.combinat.designs.bibd import BIBD_from_TD
sage: BIBD_from_TD(21,5,existence=True)
True
sage: _ = BlockDesign(21,BIBD_from_TD(21,5))
```

Third construction:

```
sage: from sage.combinat.designs.bibd import BIBD_from_TD
sage: BIBD_from_TD(85,5,existence=True)
```

```
True
sage: _ = BlockDesign(85,BIBD_from_TD(85,5))
```

No idea:

```
sage: from sage.combinat.designs.bibd import BIBD_from_TD
sage: BIBD_from_TD(20,5,existence=True)
Unknown
sage: BIBD_from_TD(20,5)
Traceback (most recent call last):
...
NotImplementedError: I do not know how to build a (20,5,1)-BIBD!
```

sage.combinat.designs.bibd.**BIBD_from_arc_in_desarguesian_projective_plane**(*n*, *k*, *existence=False*)

Returns a $(n, k, 1)$-BIBD from a maximal arc in a projective plane.

This function implements a construction from Denniston [Denniston69], who describes a maximal `arc` in a `Desarguesian Projective Plane` of order 2^k. From two powers of two n, q with $n < q$, it produces a $((n-1)(q+1)+1, n, 1)$-BIBD.

INPUT:

- `n, k` (integers) – must be powers of two (among other restrictions).
- `existence` (boolean) – whether to return the BIBD obtained through this construction (default), or to merely indicate with a boolean return value whether this method *can* build the requested BIBD.

EXAMPLES:

A $(232, 8, 1)$-BIBD:

```
sage: from sage.combinat.designs.bibd import BIBD_from_arc_in_desarguesian_projective_plane
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: D = BIBD_from_arc_in_desarguesian_projective_plane(232,8)
sage: BalancedIncompleteBlockDesign(232,D)
(232,8,1)-Balanced Incomplete Block Design
```

A $(120, 8, 1)$-BIBD:

```
sage: D = BIBD_from_arc_in_desarguesian_projective_plane(120,8)
sage: BalancedIncompleteBlockDesign(120,D)
(120,8,1)-Balanced Incomplete Block Design
```

Other parameters:

```
sage: all(BIBD_from_arc_in_desarguesian_projective_plane(n,k,existence=True)
....:     for n,k in
....:       [(120, 8), (232, 8), (456, 8), (904, 8), (496, 16),
....:        (976, 16), (1936, 16), (2016, 32), (4000, 32), (8128, 64)])
True
```

Of course, not all can be built this way:

```
sage: BIBD_from_arc_in_desarguesian_projective_plane(7,3,existence=True)
False
sage: BIBD_from_arc_in_desarguesian_projective_plane(7,3)
Traceback (most recent call last):
```

```
...
ValueError: This function cannot produce a (7,3,1)-BIBD
```

REFERENCE:

sage.combinat.designs.bibd.**BIBD_from_difference_family**(*G*, *D*, *lambd=None*, *check=True*)

Return the BIBD associated to the difference family `D` on the group `G`.

Let G be a group. A (G, k, λ)-*difference family* is a family $B = \{B_1, B_2, \ldots, B_b\}$ of k-subsets of G such that for each element of $G \backslash \{0\}$ there exists exactly λ pairs of elements (x, y), x and y belonging to the same block, such that $x - y = g$ (or x y^{-1} = g‘ in multiplicative notation).

If $\{B_1, B_2, \ldots, B_b\}$ is a (G, k, λ)-difference family then its set of translates $\{B_i \cdot g; i \in \{1, \ldots, b\}, g \in G\}$ is a (v, k, λ)-BIBD where v is the cardinality of G.

INPUT:

- `G` - a finite additive Abelian group
- `D` - a difference family on `G` (short blocks are allowed).
- `lambd` - the λ parameter (optional, only used if `check` is `True`)
- `check` - whether or not we check the output (default: `True`)

EXAMPLES:

```
sage: G = Zmod(21)
sage: D = [[0,1,4,14,16]]
sage: print sorted(G(x-y) for x in D[0] for y in D[0] if x != y)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

sage: from sage.combinat.designs.bibd import BIBD_from_difference_family
sage: BIBD_from_difference_family(G, D)
[[0, 1, 4, 14, 16],
 [1, 2, 5, 15, 17],
 [2, 3, 6, 16, 18],
 [3, 4, 7, 17, 19],
 [4, 5, 8, 18, 20],
 [5, 6, 9, 19, 0],
 [6, 7, 10, 20, 1],
 [7, 8, 11, 0, 2],
 [8, 9, 12, 1, 3],
 [9, 10, 13, 2, 4],
 [10, 11, 14, 3, 5],
 [11, 12, 15, 4, 6],
 [12, 13, 16, 5, 7],
 [13, 14, 17, 6, 8],
 [14, 15, 18, 7, 9],
 [15, 16, 19, 8, 10],
 [16, 17, 20, 9, 11],
 [17, 18, 0, 10, 12],
 [18, 19, 1, 11, 13],
 [19, 20, 2, 12, 14],
 [20, 0, 3, 13, 15]]
```

class sage.combinat.designs.bibd.**BalancedIncompleteBlockDesign**(*points*, *blocks*, *k=None*, *lambd=1*, *check=True*, *copy=True*, ***kwds*)

Bases: sage.combinat.designs.bibd.PairwiseBalancedDesign

" Balanced Incomplete Block Design (BIBD)

INPUT:

- points – the underlying set. If points is an integer v, then the set is considered to be $\{0, ..., v-1\}$.
- blocks – collection of blocks
- k (integer) – size of the blocks. Set to None (automatic guess) by default.
- lambd (integer) – value of λ, set to 1 by default.
- check (boolean) – whether to check that the design is a PBD with the right parameters.
- copy – (use with caution) if set to False then blocks must be a list of lists of integers. The list will not be copied but will be modified in place (each block is sorted, and the whole list is sorted). Your blocks object will become the instance's internal data.

EXAMPLES:

```
sage: b=designs.balanced_incomplete_block_design(9,3); b
(9,3,1)-Balanced Incomplete Block Design
```

arc(*s=2*, *solver=None*, *verbose=0*)

Return the s-arc with maximum cardinality.

A s-arc is a subset of points in a BIBD that intersects each block on at most s points. It is one possible generalization of independent set for graphs.

A simple counting shows that the cardinality of a s-arc is at most $(s-1)*r+1$ where r is the number of blocks incident to any point. A s-arc in a BIBD with cardinality $(s-1)*r+1$ is called maximal and is characterized by the following property: it is not empty and each block either contains 0 or s points of this arc. Equivalently, the trace of the BIBD on these points is again a BIBD (with block size s).

For more informations, see Wikipedia article Arc_(projective_geometry).

INPUT:

- s - (default to 2) the maximum number of points from the arc in each block
- solver – (default: None) Specify a Linear Program (LP) solver to be used. If set to None, the default one is used. For more information on LP solvers and which default solver is used, see the method solve of the class MixedIntegerLinearProgram.
- verbose – integer (default: 0). Sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLES:

```
sage: B = designs.balanced_incomplete_block_design(21, 5)
sage: a2 = B.arc()
sage: a2 # random
[5, 9, 10, 12, 15, 20]
sage: len(a2)
6
sage: a4 = B.arc(4)
sage: a4 # random
[0, 1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 20]
sage: len(a4)
16
```

The 2-arc and 4-arc above are maximal. One can check that they intersect the blocks in either 0 or s points. Or equivalently that the traces are again BIBD:

```
sage: r = (21-1)/(5-1)
sage: 1 + r*1
6
sage: 1 + r*3
16

sage: B.trace(a2).is_t_design(2, return_parameters=True)
(True, (2, 6, 2, 1))
sage: B.trace(a4).is_t_design(2, return_parameters=True)
(True, (2, 16, 4, 1))
```

Some other examples which are not maximal:

```
sage: B = designs.balanced_incomplete_block_design(25, 4)
sage: a2 = B.arc(2)
sage: r = (25-1)/(4-1)
sage: print len(a2), 1 + r
8 9
sage: sa2 = set(a2)
sage: set(len(sa2.intersection(b)) for b in B.blocks())
{0, 1, 2}
sage: B.trace(a2).is_t_design(2)
False

sage: a3 = B.arc(3)
sage: print len(a3), 1 + 2*r
15 17
sage: sa3 = set(a3)
sage: set(len(sa3.intersection(b)) for b in B.blocks()) == set([0,3])
False
sage: B.trace(a3).is_t_design(3)
False
```

TESTS:

Test consistency with relabeling:

```
sage: b = designs.balanced_incomplete_block_design(7,3)
sage: b.relabel(list("abcdefg"))
sage: set(b.arc()).issubset(b.ground_set())
True
```

sage.combinat.designs.bibd.**PBD_4_5_8_9_12**(*v*, *check=True*)

Return a $(v, \{4, 5, 8, 9, 12\})$-PBD on v elements.

A $(v, \{4, 5, 8, 9, 12\})$-PBD exists if and only if $v \equiv 0, 1 \pmod 4$. The construction implemented here appears page 168 in [Stinson2004].

INPUT:

- v – an integer congruent to 0 or 1 modulo 4.
- check (boolean) – whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to True by default.

EXAMPLES:

```
sage: designs.balanced_incomplete_block_design(40,4).blocks() # indirect doctest
[[0, 1, 2, 12], [0, 3, 6, 9], [0, 4, 8, 10],
 [0, 5, 7, 11], [0, 13, 26, 39], [0, 14, 25, 28],
```

```
 [0, 15, 27, 38], [0, 16, 22, 32], [0, 17, 23, 34],
...
```

Check that trac ticket #16476 is fixed:

```
sage: from sage.combinat.designs.bibd import PBD_4_5_8_9_12
sage: for v in (0,1,4,5,8,9,12,13,16,17,20,21,24,25):
....:     _ = PBD_4_5_8_9_12(v)
```

sage.combinat.designs.bibd.**PBD_from_TD**(*k*, *t*, *u*)

Return a $(kt, \{k, t\})$-PBD if $u = 0$ and a $(kt + u, \{k, k + 1, t, u\})$-PBD otherwise.

This is theorem 23 from [ClaytonSmith]. The PBD is obtained from the blocks a truncated $TD(k + 1, t)$, to which are added the blocks corresponding to the groups of the TD. When $u = 0$, a $TD(k, t)$ is used instead.

INPUT:

- `k,t,u` – integers such that $0 \leq u \leq t$.

EXAMPLES:

```
sage: from sage.combinat.designs.bibd import PBD_from_TD
sage: from sage.combinat.designs.bibd import is_pairwise_balanced_design
sage: PBD = PBD_from_TD(2,2,1); PBD
[[0, 2, 4], [0, 3], [1, 2], [1, 3, 4], [0, 1], [2, 3]]
sage: is_pairwise_balanced_design(PBD,2*2+1,[2,3])
True
```

class sage.combinat.designs.bibd.**PairwiseBalancedDesign**(*points*, *blocks*, *K=None*, *lambd=1*, *check=True*, *copy=True*, ***kwds*)

Bases: sage.combinat.designs.group_divisible_designs.GroupDivisibleDesign

Pairwise Balanced Design (PBD)

A Pairwise Balanced Design, or (v, K, λ)-PBD, is a collection $\mathcal{B}$ of blocks defined on a set X of size v, such that any block pair of points $p_1, p_2 \in X$ occurs in exactly λ blocks of $\mathcal{B}$. Besides, for every block $B \in \mathcal{B}$ we must have $|B| \in K$.

INPUT:

- `points` – the underlying set. If `points` is an integer v, then the set is considered to be $\{0, ..., v - 1\}$.
- `blocks` – collection of blocks
- `K` – list of integers of which the sizes of the blocks must be elements. Set to `None` (automatic guess) by default.
- `lambd` (integer) – value of λ, set to 1 by default.
- `check` (boolean) – whether to check that the design is a PBD with the right parameters.
- `copy` – (use with caution) if set to `False` then `blocks` must be a list of lists of integers. The list will not be copied but will be modified in place (each block is sorted, and the whole list is sorted). Your `blocks` object will become the instance's internal data.

sage.combinat.designs.bibd.**balanced_incomplete_block_design**(*v*, *k*, *existence=False*, *use_LJCR=False*)

Return a BIBD of parameters v, k.

A Balanced Incomplete Block Design of parameters v, k is a collection $\mathcal{C}$ of k-subsets of $V = \{0, \ldots, v - 1\}$ such that for any two distinct elements $x, y \in V$ there is a unique element $S \in \mathcal{C}$ such that $x, y \in S$.

More general definitions sometimes involve a λ parameter, and we assume here that $\lambda = 1$.

For more information on BIBD, see the corresponding Wikipedia entry.

INPUT:

- v, k (integers)
- existence (boolean) – instead of building the design, return:
  - True – meaning that Sage knows how to build the design
  - Unknown – meaning that Sage does not know how to build the design, but that the design may exist (see sage.misc.unknown).
  - False – meaning that the design does not exist.
- use_LJCR (boolean) – whether to query the La Jolla Covering Repository for the design when Sage does not know how to build it (see best_known_covering_design_www()). This requires internet.

See also:

- steiner_triple_system()
- v_4_1_BIBD()
- v_5_1_BIBD()

TODO:

- Implement other constructions from the Handbook of Combinatorial Designs.

EXAMPLES:

```
sage: designs.balanced_incomplete_block_design(7, 3).blocks()
[[0, 1, 3], [0, 2, 4], [0, 5, 6], [1, 2, 6], [1, 4, 5], [2, 3, 5], [3, 4, 6]]
sage: B = designs.balanced_incomplete_block_design(66, 6, use_LJCR=True) # optional - internet
sage: B                                                                  # optional - internet
Incidence structure with 66 points and 143 blocks
sage: B.blocks()                                                         # optional - internet
[[0, 1, 2, 3, 4, 65], [0, 5, 24, 25, 39, 57], [0, 6, 27, 38, 44, 55], ...
sage: designs.balanced_incomplete_block_design(66, 6, use_LJCR=True)  # optional - internet
Incidence structure with 66 points and 143 blocks
sage: designs.balanced_incomplete_block_design(216, 6)
Traceback (most recent call last):
...
NotImplementedError: I don't know how to build a (216,6,1)-BIBD!
```

TESTS:

```
sage: designs.balanced_incomplete_block_design(85,5,existence=True)
True
sage: _ = designs.balanced_incomplete_block_design(85,5)
```

A BIBD from a Finite Projective Plane:

```
sage: _ = designs.balanced_incomplete_block_design(21,5)
```

Some trivial BIBD:

```
sage: designs.balanced_incomplete_block_design(10,10)
(10,10,1)-Balanced Incomplete Block Design
sage: designs.balanced_incomplete_block_design(1,10)
(1,0,1)-Balanced Incomplete Block Design
```

Existence of BIBD with $k = 3, 4, 5$:

```
sage: [v for v in xrange(50) if designs.balanced_incomplete_block_design(v,3,existence=True)]
[1, 3, 7, 9, 13, 15, 19, 21, 25, 27, 31, 33, 37, 39, 43, 45, 49]
sage: [v for v in xrange(100) if designs.balanced_incomplete_block_design(v,4,existence=True)]
[1, 4, 13, 16, 25, 28, 37, 40, 49, 52, 61, 64, 73, 76, 85, 88, 97]
sage: [v for v in xrange(150) if designs.balanced_incomplete_block_design(v,5,existence=True)]
[1, 5, 21, 25, 41, 45, 61, 65, 81, 85, 101, 105, 121, 125, 141, 145]
```

For $k > 5$ there are currently very few constructions:

```
sage: [v for v in xrange(300) if designs.balanced_incomplete_block_design(v,6,existence=True) is
[1, 6, 31, 66, 76, 91, 96, 106, 111, 121, 126, 136, 141, 151, 156, 171, 181, 186, 196, 201, 211,
sage: [v for v in xrange(300) if designs.balanced_incomplete_block_design(v,6,existence=True) is
[51, 61, 81, 166, 216, 226, 231, 246, 256, 261, 276, 286, 291]
```

Here are some constructions with $k \geq 7$ and v a prime power:

```
sage: designs.balanced_incomplete_block_design(169,7)
(169,7,1)-Balanced Incomplete Block Design
sage: designs.balanced_incomplete_block_design(617,8)
(617,8,1)-Balanced Incomplete Block Design
sage: designs.balanced_incomplete_block_design(433,9)
(433,9,1)-Balanced Incomplete Block Design
sage: designs.balanced_incomplete_block_design(1171,10)
(1171,10,1)-Balanced Incomplete Block Design
```

And we know some inexistence results:

```
sage: designs.balanced_incomplete_block_design(21,6,existence=True)
False
```

sage.combinat.designs.bibd.**steiner_triple_system**(*n*)

Return a Steiner Triple System

A Steiner Triple System (STS) of a set $\{0, ..., n-1\}$ is a family S of 3-sets such that for any $i \neq j$ there exists exactly one set of S in which they are both contained.

It can alternatively be thought of as a factorization of the complete graph K_n with triangles.

A Steiner Triple System of a n-set exists if and only if $n \equiv 1 \pmod 6$ or $n \equiv 3 \pmod 6$, in which case one can be found through Bose's and Skolem's constructions, respectively [AndHonk97].

INPUT:

- n return a Steiner Triple System of $\{0, ..., n-1\}$

EXAMPLE:

A Steiner Triple System on 9 elements

```
sage: sts = designs.steiner_triple_system(9)
sage: sts
(9,3,1)-Balanced Incomplete Block Design
sage: list(sts)
[[0, 1, 5], [0, 2, 4], [0, 3, 6], [0, 7, 8], [1, 2, 3],
 [1, 4, 7], [1, 6, 8], [2, 5, 8], [2, 6, 7], [3, 4, 8],
 [3, 5, 7], [4, 5, 6]]
```

As any pair of vertices is covered once, its parameters are

```
sage: sts.is_t_design(return_parameters=True)
(True, (2, 9, 3, 1))
```

An exception is raised for invalid values of n

```
sage: designs.steiner_triple_system(10)
Traceback (most recent call last):
...
EmptySetError: Steiner triple systems only exist for n = 1 mod 6 or n = 3 mod 6
```

REFERENCE:

sage.combinat.designs.bibd.**v_4_1_BIBD**(*v*, *check=True*)

Return a $(v, 4, 1)$-BIBD.

A $(v, 4, 1)$-BIBD is an edge-decomposition of the complete graph K_v into copies of K_4. For more information, see balanced_incomplete_block_design(). It exists if and only if $v \equiv 1, 4 \pmod{12}$.

See page 167 of [Stinson2004] for the construction details.

See also:

- balanced_incomplete_block_design()

INPUT:

- v (integer) – number of points.
- check (boolean) – whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to True by default.

EXAMPLES:

```
sage: from sage.combinat.designs.bibd import v_4_1_BIBD  # long time
sage: for n in range(13,100):                            # long time
....:     if n%12 in [1,4]:                              # long time
....:         _ = v_4_1_BIBD(n, check = True)            # long time
```

TESTS:

Check that the $(25, 4)$ and $(37, 4)$-difference family are available:

```
sage: assert designs.difference_family(25,4,existence=True)
sage: _ = designs.difference_family(25,4)
sage: assert designs.difference_family(37,4,existence=True)
sage: _ = designs.difference_family(37,4)
```

Check some larger $(v, 4, 1)$-BIBD (see trac ticket #17557):

```
sage: for v in range(400):                                    # long time
....:     if v%12 in [1,4]:                                   # long time
....:         _ = designs.balanced_incomplete_block_design(v,4) # long time
```

sage.combinat.designs.bibd.**v_5_1_BIBD**(*v*, *check=True*)

Return a $(v, 5, 1)$-BIBD.

This method follows the constuction from [ClaytonSmith].

INPUT:

- v (integer)

See also:

- balanced_incomplete_block_design()

EXAMPLES:

```
sage: from sage.combinat.designs.bibd import v_5_1_BIBD
sage: i = 0
sage: while i<200:
....:     i += 20
....:     _ = v_5_1_BIBD(i+1)
....:     _ = v_5_1_BIBD(i+5)
```

TESTS:

Check that the needed difference families are there:

```
sage: for v in [21,41,61,81,141,161,281]:
....:     assert designs.difference_family(v,5,existence=True)
....:     _ = designs.difference_family(v,5)
```

5.1.67 Resolvable Balanced Incomplete Block Design (RBIBD)

This module contains everything related to resolvable Balanced Incomplete Block Designs. The constructions implemented here can be obtained through the `designs.<tab>` object:

```
designs.resolvable_balanced_incomplete_block_design(15,3)
```

For Balanced Incomplete Block Design (BIBD) see the module `bibd`. A BIBD is said to be *resolvable* if its blocks can be partitionned into parallel classes, i.e. partitions of its ground set.

The main function of this module is `resolvable_balanced_incomplete_block_design()`, which calls all others.

`resolvable_balanced_incomplete_block_design()`	Return a resolvable BIBD of parameters v, k.
`kirkman_triple_system()`	Return a Kirkman Triple System on v points.
`v_4_1_rbibd()`	Return a $(v, 4, 1)$-RBIBD
`PBD_4_7()`	Return a $(v, \{4, 7\})$-PBD
`PBD_4_7_from_Y()`	Return a $(3v + 1, \{4, 7\})$-PBD from a $(v, \{4, 5, 7\}, \mathbf{N} - \{3, 6, 10\})$-GDD.

References:

Functions

`sage.combinat.designs.resolvable_bibd.`**`PBD_4_7`**(*v*, *check=True*, *existence=False*)

Return a $(v, \{4, 7\})$-PBD

For all v such that $n \equiv 1 \pmod 3$ and $n \neq 10, 19, 31$ there exists a $(v, \{4, 7\})$-PBD. This is proved in Proposition IX.4.5 from [BJL99], which this method implements.

This construction of PBD is used by the construction of Kirkman Triple Systems.

EXAMPLE:

```
sage: from sage.combinat.designs.resolvable_bibd import PBD_4_7
sage: PBD_4_7(22)
Pairwise Balanced Design on 22 points with sets of sizes in set([4, 7])
```

TESTS:

All values ≤ 300:

```
sage: for i in range(1,300,3):
....:     if i not in [10,19,31]:
....:         assert PBD_4_7(i,existence=True)
....:         _ = PBD_4_7(i,check=True)
```

sage.combinat.designs.resolvable_bibd.**PBD_4_7_from_Y**(*gdd*, *check=True*)

Return a $(3v+1, \{4,7\})$-PBD from a $(v, \{4,5,7\}, \mathbf{N} - \{3,6,10\})$-GDD.

This implements Lemma IX.3.11 from [BJL99] (p.625). All points of the GDD are tripled, and a $+\infty$ point is added to the design.

- A group of size $s \in Y = \mathbf{N} - \{3,6,10\}$ becomes a set of size $3s$. Adding ∞ to it gives it size $3s+1$, and this set is then replaced by a $(3s+1, \{4,7\})$-PBD.
- A block of size $s \in \{4,5,7\}$ becomes a $(3s, \{4,7\}, \{3\})$-GDD.

This lemma is part of the existence proof of $(v, \{4,7\})$-PBD as explained in IX.4.5 from [BJL99]).

INPUT:

- `gdd` – a $(v, \{4,5,7\}, Y)$-GDD where $Y = \mathbf{N} - \{3,6,10\}$.
- `check` – (boolean) Whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.

EXAMPLE:

```
sage: from sage.combinat.designs.resolvable_bibd import PBD_4_7_from_Y
sage: PBD_4_7_from_Y(designs.transversal_design(7,8))
Pairwise Balanced Design on 169 points with sets of sizes in set([4, 7])
```

TESTS:

```
sage: PBD_4_7_from_Y(designs.balanced_incomplete_block_design(10,10))
Traceback (most recent call last):
...
ValueError: The GDD should only contain blocks of size {4,5,7} but there are other: set([10])
sage: PBD_4_7_from_Y(designs.transversal_design(4,3))
Traceback (most recent call last):
...
RuntimeError: A group has size 3 but I do not know how to build a (10,[4,7])-PBD
```

sage.combinat.designs.resolvable_bibd.**kirkman_triple_system**(*v*, *existence=False*)

Return a Kirkman Triple System on v points.

A Kirkman Triple System $KTS(v)$ is a resolvable Steiner Triple System. It exists if and only if $v \equiv 3 \pmod 6$.

INPUT:

- n (integer)
- `existence` (boolean; `False` by default) – whether to build the $KTS(n)$ or only answer whether it exists.

See also:

`IncidenceStructure.is_resolvable()`

EXAMPLES:

A solution to Kirkmman's original problem:

```
sage: kts = designs.kirkman_triple_system(15)
sage: classes = kts.is_resolvable(1)[1]
sage: names = '0123456789abcde'
sage: to_name = lambda (r,s,t): ' '+names[r]+names[s]+names[t]+' '
sage: rows = ['   '.join(('Day {}'.format(i) for i in range(1,8)))]
sage: rows.extend('   '.join(map(to_name,row)) for row in zip(*classes))
sage: print '\n'.join(rows)
Day 1   Day 2   Day 3   Day 4   Day 5   Day 6   Day 7
 07e     18e     29e     3ae     4be     5ce     6de
 139     24a     35b     46c     05d     167     028
 26b     03c     14d     257     368     049     15a
 458     569     06a     01b     12c     23d     347
 acd     7bd     78c     89d     79a     8ab     9bc
```

TESTS:

```
sage: for i in range(3,300,6):
....:     _ = designs.kirkman_triple_system(i)
```

sage.combinat.designs.resolvable_bibd.**resolvable_balanced_incomplete_block_design**(*v*, *k*, *existence=False*

Return a resolvable BIBD of parameters v, k.

A BIBD is said to be *resolvable* if its blocks can be partitionned into parallel classes, i.e. partitions of the ground set.

INPUT:

- `v, k` (integers)
- `existence` (boolean) – instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`).
  - `False` – meaning that the design does not exist.

See also:

- `IncidenceStructure.is_resolvable()`
- `balanced_incomplete_block_design()`

EXAMPLE:

```
sage: KTS15 = designs.resolvable_balanced_incomplete_block_design(15,3); KTS15
(15,3,1)-Balanced Incomplete Block Design
sage: KTS15.is_resolvable()
True
```

TESTS:

```
sage: for v in range(40):
....:     for k in range(v):
....:         if designs.resolvable_balanced_incomplete_block_design(v,k,existence=True):
....:             _ = designs.resolvable_balanced_incomplete_block_design(v,k)
```

sage.combinat.designs.resolvable_bibd.**v_4_1_rbibd**(*v*, *existence=False*)

Return a $(v, 4, 1)$-RBIBD.

INPUT:

- n (integer)
- `existence` (boolean; `False` by default) – whether to build the design or only answer whether it exists.

See also:

- `IncidenceStructure.is_resolvable()`
- `resolvable_balanced_incomplete_block_design()`

Note: A resolvable $(v, 4, 1)$-BIBD exists whenever $1 \equiv 4 \pmod{()12}$. This function, however, only implements a construction of $(v, 4, 1)$-BIBD such that $v = 3q + 1 \equiv 1 \pmod 3$ where q is a prime power (see VII.7.5.a from [BJL99]).

EXAMPLE:

```
sage: rBIBD = designs.resolvable_balanced_incomplete_block_design(28,4)
sage: rBIBD.is_resolvable()
True
sage: rBIBD.is_t_design(return_parameters=True)
(True, (2, 28, 4, 1))
```

TESTS:

```
sage: for q in prime_powers(2,30):
....:     if (3*q+1)%12 == 4:
....:         _ = designs.resolvable_balanced_incomplete_block_design(3*q+1,4) # indirect doctes
```

5.1.68 Group-Divisible Designs (GDD)

This module gathers everything related to Group-Divisible Designs. The constructions defined here can be accessed through `designs.<tab>`:

```
sage: designs.group_divisible_design(14,{4},{2})
Group Divisible Design on 14 points of type 2^7
```

The main function implemented here is `group_divisible_design()` (which calls all others) and the main class is `GroupDivisibleDesign`. The following functions are available:

`group_divisible_design()`	Return a (v, K, G)-Group Divisible Design.
`GDD_4_2()`	Return a $(2q, \{4\}, \{2\})$-GDD for q a prime power with $q \equiv 1 \pmod 6$.

Functions

sage.combinat.designs.group_divisible_designs.**GDD_4_2**(*q*, *existence=False*, *check=True*)

Return a $(2q, \{4\}, \{2\})$-GDD for q a prime power with $q \equiv 1 \pmod 6$.

This method implements Lemma VII.5.17 from [BJL99] (p.495).

INPUT:

- `q` (integer)

- `existence` (boolean) – instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`).
  - `False` – meaning that the design does not exist.
- `check` – (boolean) Whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.

EXAMPLE:

```
sage: from sage.combinat.designs.group_divisible_designs import GDD_4_2
sage: GDD_4_2(7,existence=True)
True
sage: GDD_4_2(7)
Group Divisible Design on 14 points of type 2^7
sage: GDD_4_2(8,existence=True)
Unknown
sage: GDD_4_2(8)
Traceback (most recent call last):
...
NotImplementedError
```

class sage.combinat.designs.group_divisible_designs.**GroupDivisibleDesign**(*points*, *groups*, *blocks*, *G=None*, *K=None*, *lambd=1*, *check=True*, *copy=True*, ***kwds*)

Bases: `sage.combinat.designs.incidence_structures.IncidenceStructure`

Group Divisible Design (GDD)

Let K and G be sets of positive integers and let λ be a positive integer. A Group Divisible Design of index λ and order v is a triple $(V, \mathcal{G}, \mathcal{B})$ where:

- V is a set of cardinality v
- $\mathcal{G}$ is a partition of V into groups whose size belongs to G
- $\mathcal{B}$ is a family of subsets of V whose size belongs to K such that any two points $p_1, p_2 \in V$ from different groups appear simultaneously in exactly λ elements of $\mathcal{B}$. Besides, a group and a block intersect on at most one point.

If $K = \{k_1, ..., k_l\}$ and G has exactly m_i groups of cardinality k_i then G is said to have type $k_1^{m_1}...k_l^{m_l}$.

INPUT:

- `points` – the underlying set. If `points` is an integer v, then the set is considered to be $\{0, ..., v-1\}$.
- `groups` – the groups of the design. Set to `None` for an automatic guess (this triggers `check=True` and can thus cost some time).
- `blocks` – collection of blocks
- `G` – list of integers of which the sizes of the groups must be elements. Set to `None` (automatic guess) by default.

- K – list of integers of which the sizes of the blocks must be elements. Set to None (automatic guess) by default.
- lambd (integer) – value of λ, set to 1 by default.
- check (boolean) – whether to check that the design is indeed a GDD with the right parameters. Set to True by default.
- copy – (use with caution) if set to False then blocks must be a list of lists of integers. The list will not be copied but will be modified in place (each block is sorted, and the whole list is sorted). Your blocks object will become the instance's internal data.

EXAMPLE:

```
sage: from sage.combinat.designs.group_divisible_designs import GroupDivisibleDesign
sage: TD = designs.transversal_design(4,10)
sage: groups = [range(i*10,(i+1)*10) for i in range(4)]
sage: GDD = GroupDivisibleDesign(40,groups,TD); GDD
Group Divisible Design on 40 points of type 10^4
```

With unspecified groups:

```
sage: D = designs.transversal_design(4,3).relabel(list('abcdefghiklm'),inplace=False).blocks()
sage: GDD = GroupDivisibleDesign('abcdefghiklm',None,D)
sage: sorted(GDD.groups())
[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i'], ['k', 'l', 'm']]
```

groups()

Return the groups of the Group-Divisible Design.

EXAMPLE:

```
sage: from sage.combinat.designs.group_divisible_designs import GroupDivisibleDesign
sage: TD = designs.transversal_design(4,10)
sage: groups = [range(i*10,(i+1)*10) for i in range(4)]
sage: GDD = GroupDivisibleDesign(40,groups,TD); GDD
Group Divisible Design on 40 points of type 10^4
sage: GDD.groups()
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
 [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
 [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
 [30, 31, 32, 33, 34, 35, 36, 37, 38, 39]]
```

TESTS:

Non-integer ground set:

```
sage: TD=designs.transversal_design(5,5)
sage: TD.relabel({i:chr(97+i) for i in range(25)})
sage: TD.groups()
[['a', 'b', 'c', 'd', 'e'],
 ['f', 'g', 'h', 'i', 'j'],
 ['k', 'l', 'm', 'n', 'o'],
 ['p', 'q', 'r', 's', 't'],
 ['u', 'v', 'w', 'x', 'y']]
```

sage.combinat.designs.group_divisible_designs.**group_divisible_design**(*v*, *K*, *G*, *existence=False*, *check=False*)

Return a (v, K, G)-Group Divisible Design.

A (v, K, G)-GDD is a pair $(\mathcal{G}, \mathcal{B})$ where:

- $\mathcal{G}$ is a partition of $X = \bigcup \mathcal{G}$ where $|X| = v$
- $\forall S \in \mathcal{G}, |S| \in G$
- $\forall S \in \mathcal{B}, |S| \in K$
- $\mathcal{G} \cup \mathcal{B}$ is a $(v, K \cup G)$-PBD

For more information, see the documentation of `GroupDivisibleDesign` or `PairwiseBalancedDesign`.

INPUT:

- `v` (integer)
- `K, G` (sets of integers)
- `existence` (boolean) – instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`).
  - `False` – meaning that the design does not exist.
- `check` – (boolean) Whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.

Note: The GDD returned by this function are defined on `range(v)`, and its groups are sets of consecutive integers.

EXAMPLES:

```
sage: designs.group_divisible_design(14,{4},{2})
Group Divisible Design on 14 points of type 2^7
```

5.1.69 Block designs

A *block design* is a set together with a family of subsets (repeated subsets are allowed) whose members are chosen to satisfy some set of properties that are deemed useful for a particular application. See Wikipedia article Block_design.

REFERENCES:

AUTHORS:

- Quentin Honoré (2015): construction of Hughes plane trac ticket #18527
- Vincent Delecroix (2014): rewrite the part on projective planes trac ticket #16281
- Peter Dobcsanyi and David Joyner (2007-2008)

  This is a significantly modified form of the module block_design.py (version 0.6) written by Peter Dobcsanyi peter@designtheory.org. Thanks go to Robert Miller for lots of good design suggestions.

Todo

Implement more finite non-Desarguesian plane as in [We07] and Wikipedia article Non-Desarguesian_plane.

Functions and methods

sage.combinat.designs.block_design.**AffineGeometryDesign**(*n*, *d*, *F*)

Return an Affine Geometry Design.

INPUT:

- n (integer) – the Euclidean dimension. The number of points is $v = |F^n|$.
- d (integer) – the dimension of the (affine) subspaces of $P = GF(q)^n$ which make up the blocks.
- F – a Finite Field (i.e. `FiniteField(17)`), or a prime power (i.e. an integer)

$AG_{n,d}(F)$, as it is sometimes denoted, is a 2 - (v, k, λ) design of points and d- flats (cosets of dimension n) in the affine geometry $AG_n(F)$, where

$$v = q^n,\ k = q^d, \lambda = \frac{(q^{n-1} - 1) \cdots (q^{n+1-d} - 1)}{(q^{n-1} - 1) \cdots (q - 1)}.$$

Wraps some functions used in GAP Design's `PGPointFlatBlockDesign`. Does *not* require GAP's Design package.

EXAMPLES:

```
sage: BD = designs.AffineGeometryDesign(3, 1, GF(2))
sage: BD.is_t_design(return_parameters=True)
(True, (2, 8, 2, 1))
sage: BD = designs.AffineGeometryDesign(3, 2, GF(2))
sage: BD.is_t_design(return_parameters=True)
(True, (3, 8, 4, 1))
```

With an integer instead of a Finite Field:

```
sage: BD = designs.AffineGeometryDesign(3, 2, 4)
sage: BD.is_t_design(return_parameters=True)
(True, (2, 64, 16, 5))
```

sage.combinat.designs.block_design.**CremonaRichmondConfiguration**()

Return the Cremona-Richmond configuration

The Cremona-Richmond configuration is a set system whose incidence graph is equal to the `TutteCoxeterGraph()`. It is a generalized quadrangle of parameters $(2, 2)$.

For more information, see the Wikipedia article Cremona-Richmond_configuration.

EXAMPLE:

```
sage: H = designs.CremonaRichmondConfiguration(); H
Incidence structure with 15 points and 15 blocks
sage: g = graphs.TutteCoxeterGraph()
sage: H.incidence_graph().is_isomorphic(g)
True
```

sage.combinat.designs.block_design.**DesarguesianProjectivePlaneDesign**(*n*, *point_coordinates=True*, *check=True*)

Return the Desarguesian projective plane of order `n` as a 2-design.

The Desarguesian projective plane of order n can also be defined as the projective plane over a field of order n. For more information, have a look at Wikipedia article Projective_plane.

INPUT:

- `n` – an integer which must be a power of a prime number

- point_coordinates (boolean) – whether to label the points with their homogeneous coordinates (default) or with integers.
- check – (boolean) Whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to True by default.

See also:

ProjectiveGeometryDesign()

EXAMPLES:

```
sage: designs.DesarguesianProjectivePlaneDesign(2)
(7,3,1)-Balanced Incomplete Block Design
sage: designs.DesarguesianProjectivePlaneDesign(3)
(13,4,1)-Balanced Incomplete Block Design
sage: designs.DesarguesianProjectivePlaneDesign(4)
(21,5,1)-Balanced Incomplete Block Design
sage: designs.DesarguesianProjectivePlaneDesign(5)
(31,6,1)-Balanced Incomplete Block Design
sage: designs.DesarguesianProjectivePlaneDesign(6)
Traceback (most recent call last):
...
ValueError: the order of a finite field must be a prime power
```

sage.combinat.designs.block_design.**Hadamard3Design**(n)

Return the Hadamard 3-design with parameters $3-(n, \frac{n}{2}, \frac{n}{4}-1)$.

This is the unique extension of the Hadamard 2-design (see HadamardDesign()). We implement the description from pp. 12 in [CvL].

INPUT:

- n (integer) – a multiple of 4 such that $n > 4$.

EXAMPLES:

```
sage: designs.Hadamard3Design(12)
Incidence structure with 12 points and 22 blocks
```

We verify that any two blocks of the Hadamard 3-design $3-(8,4,1)$ design meet in 0 or 2 points. More generally, it is true that any two blocks of a Hadamard 3-design meet in 0 or $\frac{n}{4}$ points (for $n > 4$).

```
sage: D = designs.Hadamard3Design(8)
sage: N = D.incidence_matrix()
sage: N.transpose()*N
[4 2 2 2 2 2 2 2 2 2 2 2 2 0]
[2 4 2 2 2 2 2 2 2 2 2 2 0 2]
[2 2 4 2 2 2 2 2 2 2 2 0 2 2]
[2 2 2 4 2 2 2 2 2 2 0 2 2 2]
[2 2 2 2 4 2 2 2 2 0 2 2 2 2]
[2 2 2 2 2 4 2 2 0 2 2 2 2 2]
[2 2 2 2 2 2 4 0 2 2 2 2 2 2]
[2 2 2 2 2 2 0 4 2 2 2 2 2 2]
[2 2 2 2 2 0 2 2 4 2 2 2 2 2]
[2 2 2 2 0 2 2 2 2 4 2 2 2 2]
[2 2 2 0 2 2 2 2 2 2 4 2 2 2]
[2 2 0 2 2 2 2 2 2 2 2 4 2 2]
[2 0 2 2 2 2 2 2 2 2 2 2 4 2]
[0 2 2 2 2 2 2 2 2 2 2 2 2 4]
```

REFERENCES:

sage.combinat.designs.block_design.**HadamardDesign**(n)

As described in Section 1, p. 10, in [CvL]. The input n must have the property that there is a Hadamard matrix of order $n+1$ (and that a construction of that Hadamard matrix has been implemented...).

EXAMPLES:

```
sage: designs.HadamardDesign(7)
Incidence structure with 7 points and 7 blocks
sage: print designs.HadamardDesign(7)
Incidence structure with 7 points and 7 blocks
```

For example, the Hadamard 2-design with $n = 11$ is a design whose parameters are 2-(11, 5, 2). We verify that $NJ = 5J$ for this design.

```
sage: D = designs.HadamardDesign(11); N = D.incidence_matrix()
sage: J = matrix(ZZ, 11, 11, [1]*11*11); N*J
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
[5 5 5 5 5 5 5 5 5 5 5]
```

REFERENCES:

- [CvL] P. Cameron, J. H. van Lint, Designs, graphs, codes and their links, London Math. Soc., 1991.

sage.combinat.designs.block_design.**HughesPlane**(*q2*, *check=True*)

Return the Hughes projective plane of order `q2`.

Let q be an odd prime, the Hughes plane of order q^2 is a finite projective plane of order q^2 introduced by D. Hughes in [Hu57]. Its construction is as follows.

Let $K = GF(q^2)$ be a finite field with q^2 elements and $F = GF(q) \subset K$ be its unique subfield with q elements. We define a twisted multiplication on K as

$$x \circ y = \begin{cases} x\,y & \text{if y is a square in K} \\ x^q\,y & \text{otherwise} \end{cases}$$

The points of the Hughes plane are the triples (x, y, z) of points in $K^3 \backslash \{0, 0, 0\}$ up to the equivalence relation $(x, y, z) \sim (x \circ k, y \circ k, z \circ k)$ where $k \in K$.

For $a = 1$ or $a \in (K \backslash F)$ we define a block $L(a)$ as the set of triples (x, y, z) so that $x + a \circ y + z = 0$. The rest of the blocks are obtained by letting act the group $GL(3, F)$ by its standard action.

For more information, see Wikipedia article Hughes_plane and [We07].

See also:

`DesarguesianProjectivePlaneDesign()` to build the Desarguesian projective planes

INPUT:

- `q2` – an even power of an odd prime number

- `check` – (boolean) Whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.

EXAMPLES:

```
sage: H = designs.HughesPlane(9)
sage: H
(91,10,1)-Balanced Incomplete Block Design
```

We prove in the following computations that the Desarguesian plane `H` is not Desarguesian. Let us consider the two triangles $(0, 1, 10)$ and $(57, 70, 59)$. We show that the intersection points $D_{0,1} \cap D_{57,70}$, $D_{1,10} \cap D_{70,59}$ and $D_{10,0} \cap D_{59,57}$ are on the same line while $D_{0,70}$, $D_{1,59}$ and $D_{10,57}$ are not concurrent:

```
sage: blocks = H.blocks()
sage: line = lambda p,q: (b for b in blocks if p in b and q in b).next()

sage: b_0_1 = line(0, 1)
sage: b_1_10 = line(1, 10)
sage: b_10_0 = line(10, 0)
sage: b_57_70 = line(57, 70)
sage: b_70_59 = line(70, 59)
sage: b_59_57 = line(59, 57)

sage: set(b_0_1).intersection(b_57_70)
{2}
sage: set(b_1_10).intersection(b_70_59)
{73}
sage: set(b_10_0).intersection(b_59_57)
{60}

sage: line(2, 73) == line(73, 60)
True

sage: b_0_57 = line(0, 57)
sage: b_1_70 = line(1, 70)
sage: b_10_59 = line(10, 59)

sage: p = set(b_0_57).intersection(b_1_70)
sage: q = set(b_1_70).intersection(b_10_59)
sage: p == q
False
```

TESTS:

Some wrong input:

```
sage: designs.HughesPlane(5)
Traceback (most recent call last):
...
EmptySetError: No Hughes plane of non-square order exists.

sage: designs.HughesPlane(16)
Traceback (most recent call last):
...
EmptySetError: No Hughes plane of even order exists.
```

Check that it works for non-prime q:

```
sage: designs.HughesPlane(3**4)     # not tested - 10 secs
(6643,82,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.block_design.**ProjectiveGeometryDesign**(*n*, *d*, *F*, *algorithm=None*, *point_coordinates=True*, *check=True*)

Return a projective geometry design.

A projective geometry design of parameters n, d, F has for points the lines of F^{n+1}, and for blocks the $d+1$-dimensional subspaces of F^{n+1}, each of which contains $\frac{|F|^{d+1}-1}{|F|-1}$ lines.

INPUT:

- `n` is the projective dimension
- `d` is the dimension of the subspaces of $P = PPn(F)$ which make up the blocks.
- `F` is a finite field.
- `algorithm` – set to `None` by default, which results in using Sage's own implementation. In order to use GAP's implementation instead (i.e. its `PGPointFlatBlockDesign` function) set `algorithm="gap"`. Note that GAP's "design" package must be available in this case, and that it can be installed with the `gap_packages` spkg.
- `point_coordinates` – `True` by default. Ignored and assumed to be `False` if `algorithm="gap"`. If `True`, the ground set is indexed by coordinates in F^{n+1}. Otherwise the ground set is indexed by integers,

EXAMPLES:

The set of d-dimensional subspaces in a n-dimensional projective space forms 2-designs (or balanced incomplete block designs):

```
sage: PG = designs.ProjectiveGeometryDesign(4,2,GF(2))
sage: PG
Incidence structure with 31 points and 155 blocks
sage: PG.is_t_design(return_parameters=True)
(True, (2, 31, 7, 7))

sage: PG = designs.ProjectiveGeometryDesign(3,1,GF(4,'z'))
sage: PG.is_t_design(return_parameters=True)
(True, (2, 85, 5, 1))
```

Use coordinates:

```
sage: PG = designs.ProjectiveGeometryDesign(2,1,GF(3))
sage: PG.blocks()[0]
[(1, 0, 0), (1, 0, 1), (1, 0, 2), (0, 0, 1)]
```

Use indexing by integers:

```
sage: PG = designs.ProjectiveGeometryDesign(2,1,GF(3),point_coordinates=0)
sage: PG.blocks()[0]
[0, 1, 2, 12]
```

Check that the constructor using gap also works:

```
sage: BD = designs.ProjectiveGeometryDesign(2, 1, GF(2), algorithm="gap") # optional - gap_packa
sage: BD.is_t_design(return_parameters=True)                              # optional - gap_packa
(True, (2, 7, 3, 1))
```

sage.combinat.designs.block_design.**WittDesign**(*n*)

INPUT:

- n is in $9, 10, 11, 12, 21, 22, 23, 24$.

Wraps GAP Design's WittDesign. If `n=24` then this function returns the large Witt design W_{24}, the unique (up to isomorphism) $5-(24, 8, 1)$ design. If `n=12` then this function returns the small Witt design W_{12}, the unique (up to isomorphism) $5-(12, 6, 1)$ design. The other values of n return a block design derived from these.

EXAMPLES:

```
sage: BD = designs.WittDesign(9)             # optional - gap_packages (design package)
sage: BD.is_t_design(return_parameters=True) # optional - gap_packages (design package)
(True, (2, 9, 3, 1))
sage: BD                                     # optional - gap_packages (design package)
Incidence structure with 9 points and 12 blocks
sage: print BD                               # optional - gap_packages (design package)
Incidence structure with 9 points and 12 blocks
```

sage.combinat.designs.block_design.**are_hyperplanes_in_projective_geometry_parameters**(*v*, *k*, *lmbda*, *return_pa*

Return `True` if the parameters `(v,k,lmbda)` are the one of hyperplanes in a (finite Desarguesian) projective space.

In other words, test whether there exists a prime power `q` and an integer `d` greater than two such that:

- $v = (q^{d+1}-1)/(q-1) = q^d + q^{d-1} + ... + 1$
- $k = (q^{d}-1)/(q-1) = q^{d-1} + q^{d-2} + ... + 1$
- $lmbda = (q^{d-1}-1)/(q-1) = q^{d-2} + q^{d-3} + ... + 1$

If it exists, such a pair `(q,d)` is unique.

INPUT:

- `v,k,lmbda` (integers)

OUTPUT:

- a boolean or, if `return_parameters` is set to `True` a pair `(True, (q,d))` or `(False, (None,None))`.

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import are_hyperplanes_in_projective_geometry_para
sage: are_hyperplanes_in_projective_geometry_parameters(40,13,4)
True
sage: are_hyperplanes_in_projective_geometry_parameters(40,13,4,return_parameters=True)
(True, (3, 3))
sage: PG = designs.ProjectiveGeometryDesign(3,2,GF(3))
sage: PG.is_t_design(return_parameters=True)
(True, (2, 40, 13, 4))

sage: are_hyperplanes_in_projective_geometry_parameters(15,3,1)
False
sage: are_hyperplanes_in_projective_geometry_parameters(15,3,1,return_parameters=True)
(False, (None, None))
```

TESTS:

```
sage: sgp = lambda q,d: ((q**(d+1)-1)//(q-1), (q**d-1)//(q-1), (q**(d-1)-1)//(q-1))
sage: for q in [3,4,5,7,8,9,11]:
....:     for d in [2,3,4,5]:
```

```
....:         v,k,l = sgp(q,d)
....:         assert are_hyperplanes_in_projective_geometry_parameters(v,k,l,True) == (True, (q,
....:         assert are_hyperplanes_in_projective_geometry_parameters(v+1,k,l) is False
....:         assert are_hyperplanes_in_projective_geometry_parameters(v-1,k,l) is False
....:         assert are_hyperplanes_in_projective_geometry_parameters(v,k+1,l) is False
....:         assert are_hyperplanes_in_projective_geometry_parameters(v,k-1,l) is False
....:         assert are_hyperplanes_in_projective_geometry_parameters(v,k,l+1) is False
....:         assert are_hyperplanes_in_projective_geometry_parameters(v,k,l-1) is False
```

sage.combinat.designs.block_design.**normalize_hughes_plane_point**(*p*, *q*)

Return the normalized form of point p as a 3-tuple.

In the Hughes projective plane over the finite field K, all triples (xk, yk, zk) with $k \in K$ represent the same point (where the multiplication is over the nearfield built from K). This function chooses a canonical representative among them.

This function is used in HughesPlane().

INPUT:

- p - point with the coordinates (x,y,z) (a list, a vector, a tuple...)
- q - cardinality of the underlying finite field

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import normalize_hughes_plane_point
sage: K = FiniteField(9,'x')
sage: x = K.gen()
sage: normalize_hughes_plane_point((x, x+1, x), 9)
(1, x, 1)
sage: normalize_hughes_plane_point(vector((x,x,x)), 9)
(1, 1, 1)
sage: zero = K.zero()
sage: normalize_hughes_plane_point((2*x+2, zero, zero), 9)
(1, 0, 0)
sage: one = K.one()
sage: normalize_hughes_plane_point((2*x, one, zero), 9)
(2*x, 1, 0)
```

sage.combinat.designs.block_design.**projective_plane**(*n*, *check=True*, *existence=False*)

Return a projective plane of order n as a 2-design.

A finite projective plane is a 2-design with $n^2 + n + 1$ lines (or blocks) and $n^2 + n + 1$ points. For more information on finite projective planes, see the Wikipedia article Projective_plane#Finite_projective_planes.

If no construction is possible, then the function raises a EmptySetError whereas if no construction is available the function raises a NotImplementedError.

INPUT:

- n – the finite projective plane's order

EXAMPLES:

```
sage: designs.projective_plane(2)
(7,3,1)-Balanced Incomplete Block Design
sage: designs.projective_plane(3)
(13,4,1)-Balanced Incomplete Block Design
sage: designs.projective_plane(4)
(21,5,1)-Balanced Incomplete Block Design
sage: designs.projective_plane(5)
```

```
(31,6,1)-Balanced Incomplete Block Design
sage: designs.projective_plane(6)
Traceback (most recent call last):
...
EmptySetError: By the Bruck-Ryser theorem, no projective plane of order 6 exists.
sage: designs.projective_plane(10)
Traceback (most recent call last):
...
EmptySetError: No projective plane of order 10 exists by C. Lam, L. Thiel and S. Swiercz "The nc
sage: designs.projective_plane(12)
Traceback (most recent call last):
...
NotImplementedError: If such a projective plane exists, we do not know how to build it.
sage: designs.projective_plane(14)
Traceback (most recent call last):
...
EmptySetError: By the Bruck-Ryser theorem, no projective plane of order 14 exists.
```

TESTS:

```
sage: designs.projective_plane(2197, existence=True)
True
sage: designs.projective_plane(6, existence=True)
False
sage: designs.projective_plane(10, existence=True)
False
sage: designs.projective_plane(12, existence=True)
Unknown
```

sage.combinat.designs.block_design.**projective_plane_to_OA**(*pplane*, *pt=None*, *check=True*)

Return the orthogonal array built from the projective plane `pplane`.

The orthogonal array $OA(n+1, n, 2)$ is obtained from the projective plane `pplane` by removing the point `pt` and the $n+1$ lines that pass through it‘. These $n+1$ lines form the $n+1$ groups while the remaining n^2+n lines form the transversals.

INPUT:

- `pplane` - a projective plane as a 2-design
- `pt` - a point in the projective plane `pplane`. If it is not provided then it is set to n^2+n.
- `check` – (boolean) Whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import projective_plane_to_OA
sage: p2 = designs.DesarguesianProjectivePlaneDesign(2,point_coordinates=False)
sage: projective_plane_to_OA(p2)
[[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 0]]
sage: p3 = designs.DesarguesianProjectivePlaneDesign(3,point_coordinates=False)
sage: projective_plane_to_OA(p3)
[[0, 0, 0, 0],
 [0, 1, 2, 1],
 [0, 2, 1, 2],
 [1, 0, 2, 2],
 [1, 1, 1, 0],
 [1, 2, 0, 1],
```

```
 [2, 0, 1, 1],
 [2, 1, 0, 2],
 [2, 2, 2, 0]]

sage: pp = designs.DesarguesianProjectivePlaneDesign(16,point_coordinates=False)
sage: _ = projective_plane_to_OA(pp, pt=0)
sage: _ = projective_plane_to_OA(pp, pt=3)
sage: _ = projective_plane_to_OA(pp, pt=7)
```

sage.combinat.designs.block_design.**q3_minus_one_matrix**(K)

Return a companion matrix in $GL(3, K)$ whose multiplicative order is $q^3 - 1$.

This function is used in `HughesPlane()`

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import q3_minus_one_matrix
sage: m = q3_minus_one_matrix(GF(3))
sage: m.multiplicative_order() == 3**3 - 1
True

sage: m = q3_minus_one_matrix(GF(4,'a'))
sage: m.multiplicative_order() == 4**3 - 1
True

sage: m = q3_minus_one_matrix(GF(5))
sage: m.multiplicative_order() == 5**3 - 1
True

sage: m = q3_minus_one_matrix(GF(9,'a'))
sage: m.multiplicative_order() == 9**3 - 1
True
```

sage.combinat.designs.block_design.**tdesign_params**(t, v, k, L)

Return the design's parameters: (t, v, b, r, k, L). Note that t must be given.

EXAMPLES:

```
sage: BD = BlockDesign(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]])
sage: from sage.combinat.designs.block_design import tdesign_params
sage: tdesign_params(2,7,3,1)
(2, 7, 7, 3, 3, 1)
```

5.1.70 Covering designs: coverings of t-element subsets of a v-set by k-sets

A (v, k, t) covering design C is an incidence structure consisting of a set of points P of order v, and a set of blocks B, where each block contains k points of P. Every t-element subset of P must be contained in at least one block.

If every t-set is contained in exactly one block of C, then we have a block design. Following the block design implementation, the standard representation of a covering design uses $P = [0, 1, ..., v - 1]$.

In addition to the parameters and incidence structure for a covering design from this database, we include extra information:

- Best known lower bound on the size of a (v, k, t)-covering design
- Name of the person(s) who produced the design
- Method of construction used

- Date when the design was added to the database

REFERENCES:

AUTHORS:

– Daniel M. Gordon (2008-12-22): initial version

Classes and methods

class sage.combinat.designs.covering_design.**CoveringDesign**(*v=0*, *k=0*, *t=0*, *size=0*, *points=[]*, *blocks=[]*, *low_bd=0*, *method=''*, *creator=''*, *timestamp=''*)

Bases: sage.structure.sage_object.SageObject

Covering design.

INPUT:

- `v,k,t` – integer parameters of the covering design.
- `size` (integer)
- `points` – list of points (default points are $[0, ...v-1]$).
- `blocks`
- `low_bd` (integer) – lower bound for such a design
- `method, creator, timestamp` – database information.

creator()

Return the creator of the covering design

This field is optional, and is used in a database to give attribution for the covering design It can refer to the person who submitted it, or who originally gave a construction

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C=CoveringDesign(7,3,2,7,range(7),[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4,
sage: C.creator()
'Gino Fano'
```

incidence_structure()

Return the incidence structure of a covering design, without all the extra parameters.

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C=CoveringDesign(7,3,2,7,range(7),[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4,
sage: D = C.incidence_structure()
sage: D.ground_set()
[0, 1, 2, 3, 4, 5, 6]
sage: D.blocks()
[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6], [2, 3, 6], [2, 4, 5]]
```

is_covering()

Checks that all t-sets are in fact covered by the blocks of `self`

Note: This is very slow and wasteful of memory.

EXAMPLES:

```
sage: C=CoveringDesign(7,3,2,7,range(7),[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4,
sage: C.is_covering()
True
sage: C=CoveringDesign(7,3,2,7,range(7),[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4,
sage: C.is_covering()
False
```

k()

Return k, the size of blocks of the covering design

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C=CoveringDesign(7,3,2,7,range(7),[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4,
sage: C.k()
3
```

low_bd()

Return a lower bound for the number of blocks a covering design with these parameters could have.

Typically this is the Schonheim bound, but for some parameters better bounds have been shown.

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C=CoveringDesign(7,3,2,7,range(7),[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4,
sage: C.low_bd()
7
```

method()

Return the method used to create the covering design This field is optional, and is used in a database to give information about how coverings were constructed

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C=CoveringDesign(7,3,2,7,range(7),[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4,
sage: C.method()
'Projective Plane'
```

size()

Return the number of blocks in the covering design

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C=CoveringDesign(7,3,2,7,range(7),[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4,
sage: C.size()
7
```

t()

Return t, the size of sets which must be covered by the blocks of the covering design

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C=CoveringDesign(7,3,2,7,range(7),[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4,
sage: C.t()
2
```

timestamp()
Return the time that the covering was submitted to the database

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C=CoveringDesign(7,3,2,7,range(7),[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4,
sage: C.timestamp()  #Fano had an article in 1892, I don't know the date it appeared
'1892-01-01 00:00:00'
```

v()
Return v, the number of points in the covering design.

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import CoveringDesign
sage: C=CoveringDesign(7,3,2,7,range(7),[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4,
sage: C.v()
7
```

sage.combinat.designs.covering_design.**best_known_covering_design_www**(v, k, t, *verbose=False*)
Gives the best known (v, k, t) covering design, using the database available at http://www.ccrwest.org/

INPUeT:

- `v` – integer, the size of the point set for the design
- `k` – integer, the number of points per block
- `t` – integer, the size of sets covered by the blocks
- `verbose` – bool (default=‘‘False‘‘), print verbose message

OUTPUT:

A `CoveringDesign` object representing the `(v,k,t)`-covering design with smallest number of blocks available in the database.

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import best_known_covering_design_www
sage: C = best_known_covering_design_www(7, 3, 2)    # optional - internet
sage: print C                                        # optional - internet
C(7,3,2) = 7
Method: lex covering
Submitted on: 1996-12-01 00:00:00
0  1  2
0  3  4
0  5  6
1  3  5
1  4  6
2  3  6
2  4  5
```

This function raises a ValueError if the `(v,k,t)` parameters are not found in the database.

sage.combinat.designs.covering_design.**schonheim**(v, k, t)
Schonheim lower bound for size of covering design

INPUT:

- `v` – integer, size of point set

- k – integer, cardinality of each block
- t – integer, cardinality of sets being covered

OUTPUT:

The Schonheim lower bound for such a covering design's size: $C(v,k,t) \leq \lceil (\frac{v}{k} \lceil \frac{v-1}{k-1} \cdots \lceil \frac{v-t+1}{k-t+1} \rceil \cdots \rceil \rceil$

EXAMPLES:

```
sage: from sage.combinat.designs.covering_design import schonheim
sage: schonheim(10,3,2)
17
sage: schonheim(32,16,8)
930
```

sage.combinat.designs.covering_design.**trivial_covering_design**(v, k, t)

Construct a trivial covering design.

INPUT:

- v – integer, size of point set
- k – integer, cardinality of each block
- t – integer, cardinality of sets being covered

OUTPUT:

(v,k,t) covering design

EXAMPLE:

```
sage: C = trivial_covering_design(8,3,1)
sage: print C
C(8,3,1) = 3
Method: Trivial
0   1   2
0   6   7
3   4   5
sage: C = trivial_covering_design(5,3,2)
sage: print C
4 <= C(5,3,2) <= 10
Method: Trivial
0   1   2
0   1   3
0   1   4
0   2   3
0   2   4
0   3   4
1   2   3
1   2   4
1   3   4
2   3   4
```

NOTES:

Cases are:

- $t = 0$: This could be empty, but it's a useful convention to have one block (which is empty if $k=0$).
- $t = 1$: This contains $\lceil v/k \rceil$ blocks: $[0, ..., k-1], [k, ..., 2k-1], ...$ The last block wraps around if k does not divide v.
- anything else: Just use every k-subset of $[0, 1, ..., v-1]$.

5.1.71 Database of small combinatorial designs

This module implements combinatorial designs that cannot be obtained by more general constructions. Most of them come from the Handbook of Combinatorial Designs [DesignHandbook].

All this would only be a dream without the mathematical knowledge and help of Julian R. Abel.

These functions can all be obtained through the `designs.<tab>` functions.

This module implements:

- `OA(11,640)`, `OA(15,224)`, `OA(7,66)`, `OA(7,68)`, `OA(8,69)`, `OA(9,135)`, `OA(10,520)`, `OA(14,524)`, `OA(7,74)`, `OA(8,76)`, `OA(10,205)`, `OA(11,80)`, `OA(20,352)`, `OA(7,18)`, `OA(20,416)`, `OA(10,1620)`, `OA(10,469)`, `OA(20,544)`, `OA(10,796)`, `OA(11,160)`, `OA(16,176)`,`OA(15,896)`,`OA(9,40)`,`OA(9,1612)`,`OA(16,208)`,`OA(25,1262)`,`OA(15,112)`, `OA(9,1078)`,`OA(17,560)`,`OA(9,120)`,`OA(11,185)`,`OA(12,522)`,`OA(11,254)`
- `2 MOLS of order 10`, `5 MOLS of order 12`, `3 MOLS of order 18`, `4 MOLS of order 14`, `4 MOLS of order 15`
- $V(m,t)$ **vectors:**
  - $m = 3$ and $t = 2, 4, 6, 10, 12, 14, 20, 24, 26, 32, 34$
  - $m = 4$ and $t = 3, 7, 9, 13, 15, 25$
  - $m = 5$ and $t = 6, 8, 12, 14, 20, 26$
  - $m = 6$ and $t = 5, 7, 11, 13, 17, 21$
  - $m = 7$ and $t = 6, 10, 16, 18$
  - $m = 8$ and $t = 9, 11, 17, 29, 57$
  - $m = 9$ and $t = 12, 14, 18, 20, 22, 30, 34, 42, 44$
  - $m = 10$ and $t = 13, 15, 19, 21, 25, 27, 31, 33, 43, 49, 81, 97, 103, 181, 187, 259, 273, 319, 391, 409$
  - $m = 11$ and $t = 30, 32, 36, 38, 42, 56, 60, 62, 66, 78, 80, 86, 90, 92, 102, 116, 120, 128, 132, 146, 162, 170, 182, 188, 192, 198, 206, 210, 212, 216, 218, 230, 242, 246, 248, 260, 266, 270, 276, 288, 290, 296, 300, 302, 308, 312, 318, 330, 336, 338, 350, 356, 366, 368, 372, 378, 396, 402, 420, 422, 450, 452$
  - $m = 12$ and $t = 33, 35, 45, 51, 55, 59, 61, 63, 69, 71, 73, 83, 85, 89, 91, 93, 101, 103, 115, 119, 121, 129, 133, 135, 139, 141, 145, 149, 155, 161, 169, 171, 185, 189, 191, 195, 199, 203, 213, 223, 229, 233, 243, 253, 255, 259, 265, 269, 271, 275, 281, 289, 293, 295, 301, 303, 309, 311, 321, 323, 335, 341, 355, 363, 379, 383, 385, 399, 401, 405, 409, 411, 413$
- `RBIBD(120,8,1)`
- (v,k,λ)**-BIBD:**
  - $\lambda = 1$**:** $(66,6,1)$, $(76,6,1)$, $(96,6,1)$, $(106,6,1)$, $(111,6,1)$, $(120,8,1)$, $(126,6,1)$, $(136,6,1)$, $(141,6,1)$, $(171,6,1)$, $(196,6,1)$, $(201,6,1)$
  - $\lambda = 8$**:** $(45,9,8)$
  - $\lambda = 14$**:** $(176,50,14)$
- (v,k,λ)**-difference families:**
  - $\lambda = 1$**:** $(15,3,1)$, $(21,3,1)$, $(21,5,1)$, $(25,3,1)$, $(25,4,1)$, $(27,3,1)$, $(33,3,1)$, $(37,4,1)$, $(39,3,1)$, $(40,4,1)$, $(45,3,1)$, $(45,5,1)$, $(49,3,1)$, $(49,4,1)$, $(51,3,1)$, $(52,4,1)$, $(55,3,1)$, $(57,3,1)$, $(63,3,1)$, $(64,4,1)$, $(65,5,1)$, $(69,3,1)$, $(75,3,1)$, $(76,4,1)$, $(81,3,1)$, $(81,5,1)$, $(85,4,1)$,

$(91, 6, 1)$, $(91, 7, 1)$, $(121, 5, 1)$, $(121, 6, 1)$, $(141, 5, 1)$, $(161, 5, 1)$, $(175, 7, 1)$, $(201, 5, 1)$, $(217, 7, 1)$, $(221, 5, 1)$, $(259, 7, 1)$

  - $\lambda = 2$**:** $(16, 3, 2)$, $(19, 4, 2)$, $(22, 4, 2)$, $(28, 3, 2)$, $(31, 4, 2)$, $(31, 5, 2)$, $(34, 4, 2)$, $(35, 5, 2)$, $(40, 3, 2)$, $(43, 4, 2)$, $(43, 7, 2)$, $(46, 4, 2)$, $(46, 6, 2)$, $(51, 5, 2)$, $(61, 6, 2)$, $(64, 7, 2)$, $(71, 5, 2)$, $(75, 5, 2)$, $(85, 7, 2)$, $(85, 8, 2)$, $(153, 9, 2)$, $(181, 10, 2)$
  - $\lambda = 3$**:** $(21, 4, 3)$, $(21, 6, 3)$, $(29, 7, 3)$, $(41, 6, 3)$, $(43, 7, 3)$, $(49, 9, 3)$, $(51, 6, 3)$, $(57, 7, 3)$, $(61, 6, 3)$, $(61, 10, 3)$, $(71, 7, 3)$, $(85, 7, 3)$, $(97, 9, 3)$, $(121, 10, 3)$
  - $\lambda = 4$**:** $(22, 7, 4)$, $(29, 8, 4)$, $(43, 8, 4)$, $(46, 10, 4)$, $(55, 9, 4)$, $(67, 12, 4)$, $(71, 8, 4)$
  - $\lambda = 5$**:** $(13, 5, 5)$, $(17, 5, 5)$, $(21, 6, 5)$, $(22, 6, 5)$, $(28, 6, 5)$, $(33, 5, 5)$, $(33, 6, 5)$, $(37, 10, 5)$, $(39, 6, 5)$, $(45, 11, 5)$, $(46, 10, 5)$, $(55, 10, 5)$, $(67, 11, 5)$, $(73, 10, 5)$
  - $\lambda = 6$**:** $(11, 4, 6)$, $(15, 4, 6)$, $(15, 5, 6)$, $(29, 8, 6)$, $(46, 10, 6)$, $(53, 13, 6)$, $(67, 12, 6)$
  - $\lambda = 7$**:** $(25, 7, 7)$, $(53, 14, 7)$, $(61, 15, 7)$
  - $\lambda = 8$**:** $(22, 8, 8)$, $(34, 12, 8)$, $(133, 33, 8)$
  - $\lambda = 9$**:** $(21, 10, 9)$
  - $\lambda = 10$**:** $(34, 12, 10)$, $(43, 15, 10)$, $(49, 21, 10)$
  - $\lambda = 12$**:** $(22, 8, 12)$
  - $\lambda = 14$**:** $(21, 8, 14)$
  - $\lambda = 56$**:** $(901, 225, 56)$

- (v, k, λ)**-difference matrices:**

  - $\lambda = 1$**:** $(12, 6, 1)$, $(21, 6, 1)$, $(24, 8, 1)$, $(28, 6, 1)$, $(33, 6, 1)$, $(35, 6, 1)$, $(36, 9, 1)$, $(39, 6, 1)$, $(44, 6, 1)$, $(45, 7, 1)$, $(48, 9, 1)$, $(51, 6, 1)$, $(52, 6, 1)$, $(55, 7, 1)$, $(56, 8, 1)$, $(57, 8, 1)$, $(60, 6, 1)$, $(75, 8, 1)$, $(273, 17, 1)$, $(993, 32, 1)$

- $(n, k; \lambda, \mu; u)$**-quasi-difference matrices:** $(19, 6; 1, 1; 1)$, $(21, 5; 1, 1; 1)$, $(21, 6; 1, 1; 5)$, $(25, 6; 1, 1; 5)$, $(33, 6; 1, 1; 1)$, $(35, 7; 1, 1; 7)$, $(37, 6; 1, 1; 1)$, $(45, 7; 1, 1; 9)$, $(54, 7; 1, 1; 8)$, $(57, 9; 1, 1; 8)$

- (q, k) **evenly distributed sets**

  - $k = 4$: 13, 25, 37, 49, 61, 73, 97, 109, 121, 157, 169, 181, 193, 229, 241, 277, 289, 313, 337, 349, 361, 373, 397, 409, 421, 433, 457, 529, 541, 577, 601, 613, 625, 661, 673, 709, 733, 757, 769, 829, 841, 853, 877, 937, 961, 997, 1009, 1021, 1033, 1069, 1093, 1117, 1129, 1153, 1201, 1213, 1237, 1249, 1297, 1321, 1369, 1381, 1429, 1453, 1489, 1549, 1597, 1609, 1621, 1657, 1669, 1681, 1693, 1741, 1753, 1777, 1789, 1801, 1849, 1861, 1873, 1933, 1993, 2017, 2029, 2053, 2089, 2113, 2137, 2161, 2197, 2209, 2221, 2269, 2281, 2293, 2341, 2377, 2389, 2401, 2437, 2473, 2521, 2557, 2593, 2617, 2677, 2689, 2713, 2749, 2797, 2809
  - $k = 5$: 41, 61, 101, 121, 181, 241, 281, 361, 401, 421, 461, 521, 541, 601, 641, 661, 701, 761, 821, 841, 881, 941, 961, 1021, 1061, 1181, 1201, 1301, 1321, 1361, 1381, 1481, 1601, 1621, 1681, 1721, 1741, 1801, 1861, 1901
  - $k = 6$: 31, 151, 181, 211, 241, 271, 331, 361, 421, 541, 571, 601, 631, 661, 691, 751, 811, 841, 961, 991, 1021, 1051, 1171, 1201, 1231, 1291, 1321, 1381, 1471, 1531, 1621, 1681, 1741, 1801, 1831, 1861, 1951
  - $k = 7$: 169, 337, 379, 421, 463, 547, 631, 673, 757, 841, 883, 967, 1009, 1051, 1093, 1303, 1429, 1471, 1597, 1681, 1723, 1849, 1933
  - $k = 8$: 449, 617, 673, 729, 841, 953, 1009, 1289, 1681, 1849
  - $k = 9$: 73, 433, 937, 1009, 1153, 1297, 1369, 1657, 1801, 1873

- $k = 10$: 1171, 1531, 1621, 1801

REFERENCES:

Functions

sage.combinat.designs.database.**BIBD_106_6_1**()
Return a (106,6,1)-BIBD.

This constructions appears in II.3.32 from [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import BIBD_106_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(106, BIBD_106_6_1())
(106,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_111_6_1**()
Return a (111,6,1)-BIBD.

This constructions appears in II.3.32 from [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import BIBD_111_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(111, BIBD_111_6_1())
(111,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_126_6_1**()
Return a (126,6,1)-BIBD.

This constructions appears in VI.16.92 from [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import BIBD_126_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(126, BIBD_126_6_1())
(126,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_136_6_1**()
Return a (136,6,1)-BIBD.

This constructions appears in II.3.32 from [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import BIBD_136_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(136, BIBD_136_6_1())
(136,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_141_6_1**()
Return a (141,6,1)-BIBD.

This constructions appears in II.3.32 from [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import BIBD_141_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(141, BIBD_141_6_1())
(141,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_171_6_1**()

Return a (171,6,1)-BIBD.

This constructions appears in II.3.32 from [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import BIBD_171_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(171, BIBD_171_6_1())
(171,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_196_6_1**()

Return a (196,6,1)-BIBD.

This constructions appears in II.3.32 from [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import BIBD_196_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(196, BIBD_196_6_1())
(196,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_201_6_1**()

Return a (201,6,1)-BIBD.

This constructions appears in II.3.32 from [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import BIBD_201_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(201, BIBD_201_6_1())
(201,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_45_9_8**(*from_code=False*)

Return a $(45, 9, 1)$-BIBD.

This BIBD is obtained from the codewords of minimal weight in the `ExtendedQuadraticResidueCode()` of length 48. This construction appears in VII.11.2 from [DesignHandbook], which cites [HT95].

INPUT:

- `from_code` (boolean) – whether to build the design from hardcoded data (default) or from the code object (much longer).

EXAMPLE:

```
sage: from sage.combinat.designs.database import BIBD_45_9_8
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: B = BalancedIncompleteBlockDesign(45, BIBD_45_9_8(),lambd=8); B
(45,9,8)-Balanced Incomplete Block Design
```

TESTS:

From the definition (takes around 12s):

```
sage: B2 = Hypergraph(BIBD_45_9_8(from_code=True)) # not tested
sage: B2.is_isomorphic(B)                          # not tested
True
```

REFERENCE:

sage.combinat.designs.database.**BIBD_66_6_1**()

Return a (66,6,1)-BIBD.

This BIBD was obtained from La Jolla covering repository (https://www.ccrwest.org/cover.html) where it is attributed to Colin Barker.

EXAMPLE:

```
sage: from sage.combinat.designs.database import BIBD_66_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(66, BIBD_66_6_1())
(66,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_76_6_1**()

Return a (76,6,1)-BIBD.

This BIBD was obtained from La Jolla covering repository (https://www.ccrwest.org/cover.html) where it is attributed to Colin Barker.

EXAMPLE:

```
sage: from sage.combinat.designs.database import BIBD_76_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(76, BIBD_76_6_1())
(76,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**BIBD_96_6_1**()

Return a (96,6,1)-BIBD.

This BIBD was obtained from La Jolla covering repository (https://www.ccrwest.org/cover.html) where it is attributed to Colin Barker.

EXAMPLE:

```
sage: from sage.combinat.designs.database import BIBD_96_6_1
sage: from sage.combinat.designs.bibd import BalancedIncompleteBlockDesign
sage: BalancedIncompleteBlockDesign(96, BIBD_96_6_1())
(96,6,1)-Balanced Incomplete Block Design
```

sage.combinat.designs.database.**DM_12_6_1**()

Return a $(12, 6, 1)$-difference matrix as built in [Hanani75].

This design is Lemma 3.21 from [Hanani75].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_12_6_1
sage: G,M = DM_12_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(12,6)
```

REFERENCES:

sage.combinat.designs.database.**DM_21_6_1**()
Return a $(21, 6, 1)$-difference matrix.

As explained in the Handbook III.3.50 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_21_6_1
sage: G,M = DM_21_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(21,6)
```

sage.combinat.designs.database.**DM_24_8_1**()
Return a $(24, 8, 1)$-difference matrix.

As explained in the Handbook III.3.52 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_24_8_1
sage: G,M = DM_24_8_1()
sage: is_difference_matrix(M,G,8,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(24,8)
```

sage.combinat.designs.database.**DM_273_17_1**()
Return a $(273, 17, 1)$-difference matrix.

Given by Julian R. Abel.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_273_17_1
sage: G,M = DM_273_17_1()
sage: is_difference_matrix(M,G,17,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(273,17)
```

sage.combinat.designs.database.**DM_28_6_1**()
Return a $(28, 6, 1)$-difference matrix.

As explained in the Handbook III.3.54 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_28_6_1
sage: G,M = DM_28_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(28,6)
```

sage.combinat.designs.database.**DM_33_6_1**()

Return a $(33, 6, 1)$-difference matrix.

As explained in the Handbook III.3.56 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_33_6_1
sage: G,M = DM_33_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(33,6)
```

sage.combinat.designs.database.**DM_35_6_1**()

Return a $(35, 6, 1)$-difference matrix.

As explained in the Handbook III.3.58 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_35_6_1
sage: G,M = DM_35_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(35,6)
```

sage.combinat.designs.database.**DM_36_9_1**()

Return a $(36, 9, 1)$-difference matrix.

As explained in the Handbook III.3.59 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_36_9_1
sage: G,M = DM_36_9_1()
sage: is_difference_matrix(M,G,9,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(36,9)
```

sage.combinat.designs.database.**DM_39_6_1**()

Return a $(39, 6, 1)$-difference matrix.

As explained in the Handbook III.3.61 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_39_6_1
sage: G,M = DM_39_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

The design is available from the general constructor:

```
sage: designs.difference_matrix(39,6,existence=True)
True
```

sage.combinat.designs.database.**DM_44_6_1**()

Return a $(44, 6, 1)$-difference matrix.

As explained in the Handbook III.3.64 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_44_6_1
sage: G,M = DM_44_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(44,6)
```

sage.combinat.designs.database.**DM_45_7_1**()

Return a $(45, 7, 1)$-difference matrix.

As explained in the Handbook III.3.65 [DesignHandbook].

... whose description contained a very deadly typo, kindly fixed by Julian R. Abel.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_45_7_1
sage: G,M = DM_45_7_1()
sage: is_difference_matrix(M,G,7,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(45,7)
```

sage.combinat.designs.database.**DM_48_9_1**()

Return a $(48, 9, 1)$-difference matrix.

As explained in the Handbook III.3.67 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_48_9_1
sage: G,M = DM_48_9_1()
sage: is_difference_matrix(M,G,9,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(48,9)
```

sage.combinat.designs.database.**DM_51_6_1**()

Return a $(51, 6, 1)$-difference matrix.

As explained in the Handbook III.3.69 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_51_6_1
sage: G,M = DM_51_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(51,6)
```

sage.combinat.designs.database.**DM_52_6_1**()

Return a $(52, 6, 1)$-difference matrix.

As explained in the Handbook III.3.70 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_52_6_1
sage: G,M = DM_52_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(52,6)
```

sage.combinat.designs.database.**DM_55_7_1**()

Return a $(55, 7, 1)$-difference matrix.

As explained in the Handbook III.3.72 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_55_7_1
sage: G,M = DM_55_7_1()
sage: is_difference_matrix(M,G,7,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(55,7)
```

sage.combinat.designs.database.**DM_56_8_1**()

Return a $(56, 8, 1)$-difference matrix.

As explained in the Handbook III.3.73 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_56_8_1
```

```
sage: G,M = DM_56_8_1()
sage: is_difference_matrix(M,G,8,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(56,8)
```

sage.combinat.designs.database.**DM_57_8_1**()

Return a $(57, 8, 1)$-difference matrix.

Given by Julian R. Abel.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_57_8_1
sage: G,M = DM_57_8_1()
sage: is_difference_matrix(M,G,8,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(57,8)
```

sage.combinat.designs.database.**DM_60_6_1**()

Return a $(60, 6, 1)$-difference matrix.

As explained in [JulianAbel13].

REFERENCES:

http://onlinelibrary.wiley.com/doi/10.1002/jcd.21384/abstract

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_60_6_1
sage: G,M = DM_60_6_1()
sage: is_difference_matrix(M,G,6,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(60,6)
```

sage.combinat.designs.database.**DM_75_8_1**()

Return a $(75, 8, 1)$-difference matrix.

As explained in the Handbook III.3.75 [DesignHandbook].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_75_8_1
sage: G,M = DM_75_8_1()
sage: is_difference_matrix(M,G,8,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(75,8)
```

sage.combinat.designs.database.**DM_993_32_1**()
Return a $(993, 32, 1)$-difference matrix.

Given by Julian R. Abel.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: from sage.combinat.designs.database import DM_993_32_1
sage: G,M = DM_993_32_1()
sage: is_difference_matrix(M,G,32,1)
True
```

Can be obtained from the constructor:

```
sage: _ = designs.difference_matrix(993,32)
```

sage.combinat.designs.database.**HigmanSimsDesign**()
Return the Higman-Sims designs, which is a $(176, 50, 14)$-BIBD.

This design is built from a from the `WittDesign` W on 24 points. We define two points a, b, and consider:

- The collection W_a of all blocks of W containing a but not containing b.
- The collection W_b of all blocks of W containing b but not containing a.

The design is then obtained from the incidence structure produced by the blocks $A \in W_a$ and $B \in W_b$ whose intersection has cardinality 2. This construction, due to M.Smith, can be found in [KY04] or in 10.A.(v) of [BvL84].

EXAMPLE:

```
sage: H = designs.HigmanSimsDesign(); H  # optional - gap_packages
Incidence structure with 176 points and 176 blocks
sage: H.is_t_design(return_parameters=1) # optional - gap_packages
(True, (2, 176, 50, 14))
```

Make sure that the automorphism group of this designs is isomorphic to the automorphism group of the `HigmanSimsGraph()`. Note that the first of those permutation groups acts on 176 points, while the second acts on 100:

```
sage: gH = H.automorphism_group()                          # optional - gap_packages
sage: gG = graphs.HigmanSimsGraph().automorphism_group() # optional - gap_packages
sage: gG.is_isomorphic(gG)                    # long time # optional - gap_packages
True
```

REFERENCE:

sage.combinat.designs.database.**MOLS_10_2**()
Return a pair of MOLS of order 10

Data obtained from http://www.cecm.sfu.ca/organics/papers/lam/paper/html/POLS10/POLS10.html

EXAMPLES:

```
sage: from sage.combinat.designs.latin_squares import are_mutually_orthogonal_latin_squares
sage: from sage.combinat.designs.database import MOLS_10_2
sage: MOLS = MOLS_10_2()
sage: print are_mutually_orthogonal_latin_squares(MOLS)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(2,10)
True
```

sage.combinat.designs.database.**MOLS_12_5**()

Return 5 MOLS of order 12

These MOLS have been found by Brendan McKay.

EXAMPLES:

```
sage: from sage.combinat.designs.latin_squares import are_mutually_orthogonal_latin_squares
sage: from sage.combinat.designs.database import MOLS_12_5
sage: MOLS = MOLS_12_5()
sage: print are_mutually_orthogonal_latin_squares(MOLS)
True
```

sage.combinat.designs.database.**MOLS_14_4**()

Return four MOLS of order 14

These MOLS were shared by Ian Wanless. The first proof of existence was given in [Todorov12].

EXAMPLES:

```
sage: from sage.combinat.designs.latin_squares import are_mutually_orthogonal_latin_squares
sage: from sage.combinat.designs.database import MOLS_14_4
sage: MOLS = MOLS_14_4()
sage: print are_mutually_orthogonal_latin_squares(MOLS)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(4,14)
True
```

REFERENCE:

sage.combinat.designs.database.**MOLS_15_4**()

Return 4 MOLS of order 15.

These MOLS were shared by Ian Wanless.

EXAMPLES:

```
sage: from sage.combinat.designs.latin_squares import are_mutually_orthogonal_latin_squares
sage: from sage.combinat.designs.database import MOLS_15_4
sage: MOLS = MOLS_15_4()
sage: print are_mutually_orthogonal_latin_squares(MOLS)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(4,15)
True
```

sage.combinat.designs.database.**MOLS_18_3**()

Return 3 MOLS of order 18.

These MOLS were shared by Ian Wanless.

EXAMPLES:

```
sage: from sage.combinat.designs.latin_squares import are_mutually_orthogonal_latin_squares
sage: from sage.combinat.designs.database import MOLS_18_3
```

```
sage: MOLS = MOLS_18_3()
sage: print are_mutually_orthogonal_latin_squares(MOLS)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(3,18)
True
```

sage.combinat.designs.database.**OA_10_1620**()

Returns an OA(10,1620)

This is obtained through the generalized Brouwer-van Rees construction. Indeed, $1620 = 144.11 + (36 = 4.9)$ and there exists an $OA(10, 153) - OA(10, 9)$.

Note: This function should be removed once `find_brouwer_van_rees_with_one_truncated_column()` can handle all incomplete orthogonal arrays obtained through `incomplete_orthogonal_array()`.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_10_1620
sage: OA = OA_10_1620()                        # not tested -- ~7s
sage: print is_orthogonal_array(OA,10,1620,2)  # not tested -- ~7s
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(10,1620)
True
```

sage.combinat.designs.database.**OA_10_205**()

Return an OA(10,205)

Julian R. Abel shared the following construction, which originally appeared in Theorem 8.7 of [Greig99], and can in Lemmas 5.14-5.16 of [ColDin01]:

> Consider a $PG(2, 4^2)$ containing a Baer subplane (i.e. a $PG(2, 4)$) B and a point $p \in B$. Among the $4^2 + 1 = 17$ lines of $PG(2, 4^2)$ containing p:
>
> •$4 + 1 = 5$ lines intersect B on 5 points
>
> •$4^2 - 4 = 12$ lines intersect B on 1 point
>
> As those lines are disjoint outside of B we can use them as groups to build a GDD on $16^2 + 16 + 1 - (4^4 + 4 + 1) = 252$ points. By keeping only 9 lines of the second kind, however, we obtain a $(204, \{9, 13, 17\})$-GDD of type 12^5.16^9.
>
> We complete it into a PBD by adding a block $g \cup \{204\}$ for each group g. We then build an OA from this PBD using the fact that all blocks of size 9 are disjoint.

See also:

`sage.combinat.designs.orthogonal_arrays.OA_from_PBD()`

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_10_205
sage: OA = OA_10_205()
sage: print is_orthogonal_array(OA,10,205,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(10,205)
True
```

sage.combinat.designs.database.**OA_10_469**()

Return an OA(10,469)

This construction appears in [Brouwer80]. It is based on the same technique used in brouwer_separable_design().

Julian R. Abel's instructions:

> Brouwer notes that a cyclic $PG(2,37)$ (or $(1407,38,1)$-BIBD) can be obtained with a base block containing $13, 9$, and 16 points in each residue class mod 3. Thus, by reducing the $PG(2,37)$ to its points congruent to $0 \pmod 3$ one obtains a $(469, \{9,13,16\})$-PBD which consists in 3 symmetric designs, i.e. 469 blocks of size 9, 469 blocks of size 13, and 469 blocks of size 16.
>
> For each block size s, one can build a matrix with size $s \times 469$ in which each block is a row, and such that each point of the PBD appears once per column. By multiplying a row of an $OA(9,s) - s.OA(9,1)$ with the rows of the matrix one obtains a parallel class of a resolvable $OA(9,469)$.
>
> Add to this the parallel class of all blocks $(0,0,...),(1,1,...),...$ to obtain a resolvable $OA(9,469)$ equivalent to an $OA(10,469)$.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_10_469
sage: OA = OA_10_469()
sage: print is_orthogonal_array(OA,10,469,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(10,469)
True
```

sage.combinat.designs.database.**OA_10_520**()

Return an OA(10,520).

This design is built by the slightly more general construction OA_520_plus_x().

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_10_520
sage: OA = OA_10_520()
sage: print is_orthogonal_array(OA,10,520,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(10,520)
True
```

sage.combinat.designs.database.**OA_10_796**()

Returns an OA(10,796)

Construction shared by Julian R. Abel, from [AC07]:

> Truncate one block of a $TD(17,47)$ to size 13, then add an extra point. Form a block on each group plus the extra point: we obtain a $(796, \{13,16,17,47,48\})$-PBD in which only the extra point lies in more than one block of size 48 (and each other point lies in exactly 1 such block).

For each block B (of size k say) not containing the extra point, construct a $TD(10,k)-k.TD(k,1)$ on $I(10)XB$. For each block B (of size $k=47$ or 48) containing the extra point, construct a $TD(10,k)-TD(k,1)$ on $I(10)XB$, the size 1 hole being on $I(10)XP$ where P is the extra point. Finally form 1 extra block of size 10 on $I(10)XP$.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_10_796
sage: OA = OA_10_796()
sage: print is_orthogonal_array(OA,10,796,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(10,796)
True
```

sage.combinat.designs.database.**OA_11_160**()

Returns an OA(11,160)

Published by Julian R. Abel in [AbelThesis]. Uses the fact that $160=2^5\times 5$ is a product of a power of 2 and a prime number.

See also:

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_11_160
sage: OA = OA_11_160()
sage: print is_orthogonal_array(OA,11,160,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(11,160)
True
```

sage.combinat.designs.database.**OA_11_185**()

Returns an OA(11,185)

The construction is given in [Greig99]. In Julian R. Abel's words:

> Start with a $PG(2,16)$ with a 7 points Fano subplane; outside this plane there are $7(17-3)=98$ points on a line of the subplane and $273-98-7=168$ other points. Greig notes that the subdesign consisting of these 168 points is a $(168,\{10,12\})-PBD$. Now add the 17 points of a line disjoint from this subdesign (e.g. a line of the Fano subplane). This line will intersect every line of the 168 point subdesign in 1 point. Thus the new line sizes are 11 and 13, plus a unique line of size 17, giving a $(185,\{11,13,17\}$-PBD and an $OA(11,185)$.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_11_185
sage: OA = OA_11_185()
sage: print is_orthogonal_array(OA,11,185,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(11,185)
True
```

sage.combinat.designs.database.**OA_11_254**()
Return an OA(11,254)

This constructions appears in [Greig99].

From a cyclic $PG(2,19)$ whose base blocks contains 7,9, and 4 points in the congruence classes mod 3, build a $(254,11,13,16) - PBD$ by ignoring the points of a congruence class. There exist $OA(12,11), OA(12,13), OA(12,16)$, which gives the $OA(11,254)$.

See also:

sage.combinat.designs.orthogonal_arrays.OA_from_PBD()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_11_254
sage: OA = OA_11_254()
sage: print is_orthogonal_array(OA,11,254,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(11,254)
True
```

sage.combinat.designs.database.**OA_11_640**()
Returns an OA(11,640)

Published by Julian R. Abel in [AbelThesis] (uses the fact that $640 = 2^7 \times 5$ is the product of a power of 2 and a prime number).

See also:

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_11_640
sage: OA = OA_11_640()                          # not tested (too long)
sage: print is_orthogonal_array(OA,11,640,2)   # not tested (too long)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(11,640)
True
```

sage.combinat.designs.database.**OA_11_80**()
Return an OA(11,80)

As explained in the Handbook III.3.76 [DesignHandbook]. Uses the fact that $80 = 2^4 \times 5$ and that 5 is prime.

See also:

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_11_80
sage: OA = OA_11_80()
sage: print is_orthogonal_array(OA,11,80,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(11,80)
True
```

sage.combinat.designs.database.**OA_12_522**()

Return an OA(12,522)

This design is built by the slightly more general construction OA_520_plus_x().

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_12_522
sage: OA = OA_12_522()
sage: print is_orthogonal_array(OA,12,522,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(12,522)
True
```

sage.combinat.designs.database.**OA_14_524**()

Return an OA(14,524)

This design is built by the slightly more general construction OA_520_plus_x().

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_14_524
sage: OA = OA_14_524()
sage: print is_orthogonal_array(OA,14,524,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(14,524)
True
```

sage.combinat.designs.database.**OA_15_112**()

Returns an OA(15,112)

Published by Julian R. Abel in [AbelThesis]. Uses the fact that $112 = 2^4 \times 7$ and that 7 is prime.

See also:

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_15_112
sage: OA = OA_15_112()
sage: print is_orthogonal_array(OA,15,112,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(15,112)
True
```

sage.combinat.designs.database.**OA_15_224**()

Returns an OA(15,224)

Published by Julian R. Abel in [AbelThesis] (uses the fact that $224 = 2^5 \times 7$ is a product of a power of 2 and a prime number).

See also:

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_15_224
sage: OA = OA_15_224()                          # not tested -- too long
sage: print is_orthogonal_array(OA,15,224,2)    # not tested -- too long
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(15,224)
True
```

sage.combinat.designs.database.**OA_15_896**()

Returns an OA(15,896)

Uses the fact that $896 = 2^7 \times 7$ is the product of a power of 2 and a prime number.

See also:

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_15_896
sage: OA = OA_15_896()                          # not tested -- too long (~2min)
sage: print is_orthogonal_array(OA,15,896,2)    # not tested -- too long
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(15,896)
True
```

sage.combinat.designs.database.**OA_16_176**()

Returns an OA(16,176)

Published by Julian R. Abel in [AbelThesis]. Uses the fact that $176 = 2^4 \times 11$ is a product of a power of 2 and a prime number.

See also:

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_16_176
sage: OA = OA_16_176()
```

```
sage: print is_orthogonal_array(OA,16,176,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(16,176)
True
```

sage.combinat.designs.database.**OA_16_208**()

Returns an OA(16,208)

Published by Julian R. Abel in [AbelThesis]. Uses the fact that $208 = 2^4 \times 13$ is a product of 2 and a prime number.

See also:

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_16_208
sage: OA = OA_16_208()                          # not tested -- too long
sage: print is_orthogonal_array(OA,16,208,2)   # not tested -- too long
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(16,208)
True
```

sage.combinat.designs.database.**OA_17_560**()

Returns an OA(17,560)

This OA is built in Corollary 2.2 of [Thwarts].

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_17_560
sage: OA = OA_17_560()
sage: print is_orthogonal_array(OA,17,560,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(17,560)
True
```

sage.combinat.designs.database.**OA_20_352**()

Returns an OA(20,352)

Published by Julian R. Abel in [AbelThesis] (uses the fact that $352 = 2^5 \times 11$ is the product of a power of 2 and a prime number).

See also:

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_20_352
sage: OA = OA_20_352()                          # not tested (~25s)
```

```
sage: print is_orthogonal_array(OA,20,352,2)  # not tested (~25s)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(20,352)
True
```

sage.combinat.designs.database.**OA_20_416**()

Returns an OA(20,416)

Published by Julian R. Abel in [AbelThesis] (uses the fact that $416 = 2^5 \times 13$ is the product of a power of 2 and a prime number).

See also:

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_20_416
sage: OA = OA_20_416()                        # not tested (~35s)
sage: print is_orthogonal_array(OA,20,416,2)  # not tested
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(20,416)
True
```

sage.combinat.designs.database.**OA_20_544**()

Returns an OA(20,544)

Published by Julian R. Abel in [AbelThesis] (uses the fact that $544 = 2^5 \times 17$ is the product of a power of 2 and a prime number).

See also:

sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_20_544
sage: OA = OA_20_544()                        # not tested (too long ~1mn)
sage: print is_orthogonal_array(OA,20,544,2)  # not tested
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(20,544)
True
```

sage.combinat.designs.database.**OA_25_1262**()

Returns an OA(25,1262)

The construction is given in [Greig99]. In Julian R. Abel's words:

> Start with a cyclic $PG(2, 43)$ or $(1893, 44, 1)$-BIBD whose base block contains respectively $12, 13$ and 19 point in the residue classes mod 3. In the resulting BIBD, remove one of the three classes: the result is a $(1262, \{25, 31, 32\})$-PBD, from which the $OA(25, 1262)$ is obtained.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_25_1262
sage: OA = OA_25_1262()                          # not tested -- too long
sage: print is_orthogonal_array(OA,25,1262,2) # not tested -- too long
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(25,1262)
True
```

sage.combinat.designs.database.**OA_520_plus_x**(x)

Return an $OA(10+x, 520+x)$.

The consruction shared by Julian R. Abel works for `OA(10,520)`, `OA(12,522)`, and `OA(14,524)`.

Let $n = 520+x$ and $k = 10+x$. Build a $TD(17, 31)$. Remove $8-x$ points contained in a common block, add a new point p and create a block $g_i \cup \{p\}$ for every (possibly truncated) group g_i. The result is a $(520+x, 9+x, 16, 17, 31, 32) - PBD$. Note that all blocks of size ≥ 30 only intersect on p, and that the unique block B_9 of size 9 intersects all blocks of size 32 on one point. Now:

- Build an $OA(k, 16) - 16.OA(k, 16)$ for each block of size 16
- Build an $OA(k, 17) - 17.OA(k, 17)$ for each block of size 17
- Build an $OA(k, 31) - OA(k, 1)$ for each block of size 31 (with the hole on p).
- Build an $OA(k, 32) - 2.OA(k, 1)$ for each block B of size 32 (with the holes on p and $B \cap B_9$).
- Build an $OA(k, 9)$ on B_9.

Only a row $[p, p, ...]$ is missing from the $OA(10+x, 520+x)$

This construction is used in `OA(10,520)`, `OA(12,522)`, and `OA(14,524)`.

EXAMPLE:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_520_plus_x
sage: OA = OA_520_plus_x(0)                      # not tested (already tested in OA_10_520)
sage: print is_orthogonal_array(OA,10,520,2)   # not tested (already tested in OA_10_520)
True
```

sage.combinat.designs.database.**OA_7_18**()

Return an OA(7,18)

Proved in [JulianAbel13].

See also:

`sage.combinat.designs.orthogonal_arrays.OA_from_quasi_difference_matrix()`

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_7_18
sage: OA = OA_7_18()
sage: print is_orthogonal_array(OA,7,18,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(7,18)
True
```

sage.combinat.designs.database.**OA_7_66**()
Return an OA(7,66)

Construction shared by Julian R. Abel.

See also:

sage.combinat.designs.orthogonal_arrays.OA_from_PBD()

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_7_66
sage: OA = OA_7_66()
sage: print is_orthogonal_array(OA,7,66,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(7,66)
True
```

sage.combinat.designs.database.**OA_7_68**()
Return an OA(7,68)

Construction shared by Julian R. Abel.

See also:

sage.combinat.designs.orthogonal_arrays.OA_from_PBD()

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_7_68
sage: OA = OA_7_68()
sage: print is_orthogonal_array(OA,7,68,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(7,68)
True
```

sage.combinat.designs.database.**OA_7_74**()
Return an OA(7,74)

Construction shared by Julian R. Abel.

See also:

sage.combinat.designs.orthogonal_arrays.OA_from_PBD()

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_7_74
sage: OA = OA_7_74()
sage: print is_orthogonal_array(OA,7,74,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(7,74)
True
```

sage.combinat.designs.database.**OA_8_69**()

Return an OA(8,69)

Construction shared by Julian R. Abel.

See also:

sage.combinat.designs.orthogonal_arrays.OA_from_PBD()

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_8_69
sage: OA = OA_8_69()
sage: print is_orthogonal_array(OA,8,69,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(8,69)
True
```

sage.combinat.designs.database.**OA_8_76**()

Return an OA(8,76)

Construction shared by Julian R. Abel.

See also:

sage.combinat.designs.orthogonal_arrays.OA_from_PBD()

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_8_76
sage: OA = OA_8_76()
sage: print is_orthogonal_array(OA,8,76,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(8,76)
True
```

sage.combinat.designs.database.**OA_9_1078**()

Returns an OA(9,1078)

This is obtained through the generalized Brouwer-van Rees construction. Indeed, $1078 = 89.11 + (99 = 9.11)$ and there exists an $OA(9, 100) - OA(9, 11)$.

Note: This function should be removed once find_brouwer_van_rees_with_one_truncated_column() can handle all incomplete orthogonal arrays obtained through incomplete_orthogonal_array().

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_9_1078
sage: OA = OA_9_1078()                          # not tested -- ~3s
```

```
sage: print is_orthogonal_array(OA,9,1078,2) # not tested -- ~3s
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(9,1078)
True
```

sage.combinat.designs.database.**OA_9_120**()

Return an OA(9,120)

Construction shared by Julian R. Abel:

> From a resolvable $(120, 8, 1) - BIBD$, one can obtain 7 $MOLS(120)$ or a resolvable $TD(8, 120)$ by forming a resolvable $TD(8, 8) - 8.TD(8, 1)$ on $I_8 \times B$ for each block B in the BIBD. This gives a $TD(8, 120) - 120TD(8, 1)$ (which is resolvable as the BIBD is resolvable).

See also:

RBIBD_120_8_1()

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_9_120
sage: OA = OA_9_120()
sage: print is_orthogonal_array(OA,9,120,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(9,120)
True
```

sage.combinat.designs.database.**OA_9_135**()

Return an OA(9,135)

Construction shared by Julian R. Abel:

> This design can be built by Wilson's method ($135 = 8.16 + 7$) applied to an Orthogonal Array $OA(9 + 7, 16)$ with 7 groups truncated to size 1 in such a way that a block contain 0, 1 or 3 points of the truncated groups.
>
> This is possible, because $PG(2, 2)$ (the projective plane over $GF(2)$) is a subdesign in $PG(2, 16)$ (the projective plane over $GF(16)$); in a cyclic $PG(2, 16)$ or $BIBD(273, 17, 1)$ the points $\equiv 0 \pmod{39}$ form such a subdesign (note that $273 = 16^2+16+1$ and $273 = 39 \times 7$ and $7 = 2^2+2+1$).

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_9_135
sage: OA = OA_9_135()
sage: print is_orthogonal_array(OA,9,135,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(9,135)
True
```

As this orthogonal array requires a $(273, 17, 1)$ cyclic difference set, we check that it is available:

```
sage: G,D = designs.difference_family(273,17,1)
sage: G
Ring of integers modulo 273
```

sage.combinat.designs.database.**OA_9_1612**()
Returns an OA(9,1612)

This is obtained through the generalized Brouwer-van Rees construction. Indeed, $1612 = 89.17 + (99 = 9.11)$ and there exists an $OA(9, 100) - OA(9, 11)$.

Note: This function should be removed once `find_brouwer_van_rees_with_one_truncated_column()` can handle all incomplete orthogonal arrays obtained through `incomplete_orthogonal_array()`.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_9_1612
sage: OA = OA_9_1612()                        # not tested -- ~6s
sage: print is_orthogonal_array(OA,9,1612,2) # not tested -- ~6s
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(9,1612)
True
```

sage.combinat.designs.database.**OA_9_40**()
Return an OA(9,40)

As explained in the Handbook III.3.62 [DesignHandbook]. Uses the fact that $40 = 2^3 \times 5$ and that 5 is prime.

See also:

`sage.combinat.designs.orthogonal_arrays.OA_n_times_2_pow_c_from_matrix()`

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.database import OA_9_40
sage: OA = OA_9_40()
sage: print is_orthogonal_array(OA,9,40,2)
True
```

The design is available from the general constructor:

```
sage: designs.orthogonal_arrays.is_available(9,40)
True
```

sage.combinat.designs.database.**QDM_19_6_1_1_1**()
Return a $(19, 6; 1, 1; 1)$-quasi-difference matrix.

Used to build an $OA(6, 20)$

Given in the Handbook III.3.49 [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import QDM_19_6_1_1_1
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_19_6_1_1_1()
sage: is_quasi_difference_matrix(M,G,6,1,1,1)
True
```

sage.combinat.designs.database.**QDM_21_5_1_1_1**()
Return a $(21, 5; 1, 1; 1)$-quasi-difference matrix.

Used to build an $OA(5, 22)$

Given in the Handbook III.3.51 [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import QDM_21_5_1_1_1
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_21_5_1_1_1()
sage: is_quasi_difference_matrix(M,G,5,1,1,1)
True
```

sage.combinat.designs.database.**QDM_21_6_1_1_5**()
Return a $(21, 6; 1, 1; 5)$-quasi-difference matrix.

Used to build an $OA(6, 26)$

Given in the Handbook III.3.53 [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import QDM_21_6_1_1_5
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_21_6_1_1_5()
sage: is_quasi_difference_matrix(M,G,6,1,1,5)
True
```

sage.combinat.designs.database.**QDM_25_6_1_1_5**()
Return a $(25, 6; 1, 1; 5)$-quasi-difference matrix.

Used to build an $OA(6, 30)$

Given in the Handbook III.3.55 [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import QDM_25_6_1_1_5
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_25_6_1_1_5()
sage: is_quasi_difference_matrix(M,G,6,1,1,5)
True
```

sage.combinat.designs.database.**QDM_33_6_1_1_1**()
Return a $(33, 6; 1, 1; 1)$-quasi-difference matrix.

Used to build an $OA(6, 34)$

Given in the Handbook III.3.57 [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import QDM_33_6_1_1_1
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_33_6_1_1_1()
sage: is_quasi_difference_matrix(M,G,6,1,1,1)
True
```

sage.combinat.designs.database.**QDM_35_7_1_1_7**()
Return a $(35, 7; 1, 1; 7)$-quasi-difference matrix.

Used to build an $OA(7, 42)$

As explained in the Handbook III.3.63 [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import QDM_35_7_1_1_7
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_35_7_1_1_7()
sage: is_quasi_difference_matrix(M,G,7,1,1,7)
True
```

sage.combinat.designs.database.**QDM_37_6_1_1_1**()

Return a $(37, 6; 1, 1; 1)$-quasi-difference matrix.

Used to build an $OA(6, 38)$

Given in the Handbook III.3.60 [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import QDM_37_6_1_1_1
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_37_6_1_1_1()
sage: is_quasi_difference_matrix(M,G,6,1,1,1)
True
```

sage.combinat.designs.database.**QDM_45_7_1_1_9**()

Return a $(45, 7; 1, 1; 9)$-quasi-difference matrix.

Used to build an $OA(7, 54)$

As explained in the Handbook III.3.71 [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import QDM_45_7_1_1_9
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_45_7_1_1_9()
sage: is_quasi_difference_matrix(M,G,7,1,1,9)
True
```

sage.combinat.designs.database.**QDM_54_7_1_1_8**()

Return a $(54, 7; 1, 1; 8)$-quasi-difference matrix.

Used to build an $OA(7, 62)$

As explained in the Handbook III.3.74 [DesignHandbook].

EXAMPLE:

```
sage: from sage.combinat.designs.database import QDM_54_7_1_1_8
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_54_7_1_1_8()
sage: is_quasi_difference_matrix(M,G,7,1,1,8)
True
```

sage.combinat.designs.database.**QDM_57_9_1_1_8**()

Return a $(57, 9; 1, 1; 8)$-quasi-difference matrix.

Used to build an $OA(9, 65)$

Construction shared by Julian R. Abel

EXAMPLE:

```
sage: from sage.combinat.designs.database import QDM_57_9_1_1_8
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_57_9_1_1_8()
sage: is_quasi_difference_matrix(M,G,9,1,1,8)
True
```

sage.combinat.designs.database.**RBIBD_120_8_1**()

Return a resolvable $BIBD(120, 8, 1)$

This function output a list `L` of 17×15 blocks such that `L[i*15:(i+1)*15]` is a partition of 120.

Construction shared by Julian R. Abel:

> Seiden's method: Start with a cyclic $(273, 17, 1) - BIBD$ and let B be an hyperoval, i.e. a set of 18 points which intersects any block of the BIBD in either 0 points (153 blocks) or 2 points (120 blocks). Dualise this design and take these last 120 blocks as points in the design; blocks in the design will correspond to the $273 - 18 = 255$ non-hyperoval points.
>
> The design is also resolvable. In the original $PG(2, 16)$ take any point T in the hyperoval and consider a block $B1$ containing T. The 15 points in $B1$ that do not belong to the hyperoval correspond to 15 blocks forming a parallel class in the dualised design. The other 16 parallel classes come in a similar way, by using point T and the other 16 blocks containing T.

See also:

`OA_9_120()`

EXAMPLES:

```
sage: from sage.combinat.designs.database import RBIBD_120_8_1
sage: from sage.combinat.designs.bibd import is_pairwise_balanced_design
sage: RBIBD = RBIBD_120_8_1()
sage: is_pairwise_balanced_design(RBIBD,120,[8])
True
```

It is indeed resolvable, and the parallel classes are given by 17 slices of consecutive 15 blocks:

```
sage: for i in range(17):
....:     assert len(set(sum(RBIBD[i*15:(i+1)*15],[]))) == 120
```

The BIBD is available from the constructor:

```
sage: _ = designs.balanced_incomplete_block_design(120,8)
```

sage.combinat.designs.database.**cyclic_shift**(l, i)

sage.combinat.designs.database.**f**()

Return a $(57, 9; 1, 1; 8)$-quasi-difference matrix.

Used to build an $OA(9, 65)$

Construction shared by Julian R. Abel

EXAMPLE:

```
sage: from sage.combinat.designs.database import QDM_57_9_1_1_8
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: G,M = QDM_57_9_1_1_8()
sage: is_quasi_difference_matrix(M,G,9,1,1,8)
True
```

5.1.72 Catalog of designs

This module gathers all designs that can be reached through `designs.<tab>`. Example with the the Witt design on 24 points:

```
sage: designs.WittDesign(24) # optional - gap_packages
Incidence structure with 24 points and 759 blocks
```

Or a Steiner Triple System on 19 points:

```
sage: designs.steiner_triple_system(19)
(19,3,1)-Balanced Incomplete Block Design
```

La Jolla Covering Repository

The La Jolla Covering Repository (LJCR, see [2]) is an online database of covering designs. As it is frequently updated, it is not included in Sage, but one can query it through `designs.best_known_covering_design_from_LJCR`:

```
sage: C = designs.best_known_covering_design_from_LJCR(7, 3, 2)    # optional - internet
sage: C                                # optional - internet
(7,3,2)-covering design of size 7
Lower bound: 7
Method: lex covering
Submitted on: 1996-12-01 00:00:00
sage: C.incidence_structure()      # optional - internet
Incidence structure with 7 points and 7 blocks
```

Design constructors

This module gathers the following designs :

```
ProjectiveGeometryDesign()
DesarguesianProjectivePlaneDesign()
HughesPlane()
HigmanSimsDesign()
balanced_incomplete_block_design()
resolvable_balanced_incomplete_block_design()
kirkman_triple_system()
AffineGeometryDesign()
CremonaRichmondConfiguration()
WittDesign()
HadamardDesign()
Hadamard3Design()
mutually_orthogonal_latin_squares()
transversal_design()
orthogonal_array()
incomplete_orthogonal_array()
difference_family()
difference_matrix()
steiner_triple_system()
steiner_quadruple_system()
projective_plane()
```

And the `designs.best_known_covering_design_from_LJCR` function which queries the LJCR.

[2] La Jolla Covering Repository, http://www.ccrwest.org/cover.html

Todo

Implement DerivedDesign and ComplementaryDesign.

REFERENCES:

5.1.73 Cython functions for combinatorial designs

This module implements the design methods that need to be somewhat efficient.

Functions

sage.combinat.designs.designs_pyx.**is_difference_matrix**(*M*, *G*, *k*, *lmbda=1*, *verbose=False*)

Test if M is a (G, k, λ)-difference matrix.

A matrix M is a (G, k, λ)-difference matrix if its entries are element of G, and if for any two rows R, R' of M and $x \in G$ there are exactly λ values i such that $R_i - R'_i = x$.

INPUT:

- `M` – a matrix with entries from `G`
- `G` – a group
- `k` – integer
- `lmbda` (integer) – set to 1 by default.
- `verbose` (boolean) – whether to print some information when the answer is `False`.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_difference_matrix
sage: q = 3**3
sage: F = GF(q,'x')
sage: M = [[x*y for y in F] for x in F]
sage: is_difference_matrix(M,F,q,verbose=1)
True

sage: B = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
....:      [0, 1, 2, 3, 4, 2, 3, 4, 0, 1],
....:      [0, 2, 4, 1, 3, 3, 0, 2, 4, 1]]
sage: G = GF(5)
sage: B = [[G(b) for b in R] for R in B]
sage: is_difference_matrix(zip(*B),G,3,2)
True
```

Bad input:

```
sage: for R in M: R.append(None)
sage: is_difference_matrix(M,F,q,verbose=1)
The matrix has 28 columns but k=27
False
sage: for R in M: _=R.pop(-1)
sage: M.append([None]*3**3)
sage: is_difference_matrix(M,F,q,verbose=1)
The matrix has 28 rows instead of lambda(|G|-1+2u)+mu=1(27-1+2.0)+1=27
False
sage: _= M.pop(-1)
```

```
sage: for R in M: R[-1] = 0
sage: is_difference_matrix(M,F,q,verbose=1)
Columns 0 and 26 generate 0 exactly 27 times instead of the expected mu(=1)
False
sage: for R in M: R[-1] = 1
sage: M[-1][-1] = 0
sage: is_difference_matrix(M,F,q,verbose=1)
Columns 0 and 26 do not generate all elements of G exactly lambda(=1) times. The element x appea
False
```

sage.combinat.designs.designs_pyx.**is_group_divisible_design**(*groups*, *blocks*, *v*, *G=None*, *K=None*, *lambd=1*, *verbose=False*)

Checks that input is a Group Divisible Design on $\{0, ..., v-1\}$

For more information on Group Divisible Designs, see `GroupDivisibleDesign`.

INPUT:

- `groups` – a partition of X. If set to `None` the groups are guessed automatically, and the function returns `(True, guessed_groups)` instead of `True`
- `blocks` – collection of blocks
- `v` (integers) – size of the ground set assumed to be $X = \{0, ..., v-1\}$.
- `G` – list of integers of which the sizes of the groups must be elements. Set to `None` (automatic guess) by default.
- `K` – list of integers of which the sizes of the blocks must be elements. Set to `None` (automatic guess) by default.
- `lambd` – value of λ. Set to 1 by default.
- `verbose` (boolean) – whether to display some information when the design is not a GDD.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_group_divisible_design
sage: TD = designs.transversal_design(4,10)
sage: groups = [range(i*10,(i+1)*10) for i in range(4)]
sage: is_group_divisible_design(groups,TD,40,lambd=1)
True
```

TESTS:

```
sage: TD = designs.transversal_design(4,10)
sage: groups = [range(i*10,(i+1)*10) for i in range(4)]
sage: is_group_divisible_design(groups,TD,40,lambd=2,verbose=True)
the pair (0,10) has been seen 1 times but lambda=2
False
sage: is_group_divisible_design([[1,2],[3,4]],[[1,2]],40,lambd=1,verbose=True)
groups is not a partition of [0,...,39]
False
sage: is_group_divisible_design([range(40)],[[1,2]],40,lambd=1,verbose=True)
the pair (1,2) belongs to a group but appears in some block
False
sage: is_group_divisible_design([range(40)],[[2,2]],40,lambd=1,verbose=True)
The following block has repeated elements: [2, 2]
False
sage: is_group_divisible_design([range(40)],[["e",2]],40,lambd=1,verbose=True)
```

```
e does not belong to [0,...,39]
False
sage: is_group_divisible_design([range(40)],[range(40)],40,G=[5],lambd=1,verbose=True)
a group has size 40 while G=[5]
False
sage: is_group_divisible_design([range(40)],[["e",2]],40,K=[1],lambd=1,verbose=True)
a block has size 2 while K=[1]
False

sage: p = designs.projective_plane(3)
sage: is_group_divisible_design(None, p.blocks(), 13)
(True, [[0], [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]])
sage: is_group_divisible_design(None, p.blocks()*2, 13, verbose=True)
the pair (0,1) has been seen 2 times but lambda=1
False
sage: is_group_divisible_design(None, p.blocks()*2, 13, lambd=2)
(True, [[0], [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]])
```

sage.combinat.designs.designs_pyx.**is_orthogonal_array**(*OA*, *k*, *n*, *t=2*, *verbose=False*, *terminology='OA'*)

Check that the integer matrix OA is an $OA(k, n, t)$.

See `orthogonal_array()` for a definition.

INPUT:

- `OA` – the Orthogonal Array to be tested
- `k,n,t` (integers) – only implemented for $t = 2$.
- `verbose` (boolean) – whether to display some information when `OA` is not an orthogonal array $OA(k, n)$.
- `terminology` (string) – how to phrase the information when `verbose = True`. Possible values are $"OA", "MOLS"$.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: OA = designs.orthogonal_arrays.build(8,9)
sage: is_orthogonal_array(OA,8,9)
True
sage: is_orthogonal_array(OA,8,10)
False
sage: OA[4][3] = 1
sage: is_orthogonal_array(OA,8,9)
False
sage: is_orthogonal_array(OA,8,9,verbose=True)
Columns 0 and 3 are not orthogonal
False
sage: is_orthogonal_array(OA,8,9,verbose=True,terminology="MOLS")
Squares 0 and 3 are not orthogonal
False
```

TESTS:

```
sage: is_orthogonal_array(OA,8,9,t=3)
Traceback (most recent call last):
...
NotImplementedError: only implemented for t=2
sage: is_orthogonal_array([[3]*8],8,9,verbose=True)
The number of rows is 1 instead of 9^2=81
```

```
False
sage: is_orthogonal_array([[3]*8],8,9,verbose=True,terminology="MOLS")
All squares do not have dimension n^2=9^2
False
sage: is_orthogonal_array([[3]*7],8,9,verbose=True)
Some row does not have length 8
False
sage: is_orthogonal_array([[3]*7],8,9,verbose=True,terminology="MOLS")
The number of squares is not 6
False
```

Up to relabelling, there is a unique $OA(3,2)$. So their number is just the cardinality of the relabeling group which is $S_2^3 \times S_3$ and has cardinality 48:

```
sage: from itertools import product
sage: n = 0
sage: for a in product(product((0,1), repeat=3), repeat=4):
....:     if is_orthogonal_array(a,3,2):
....:         n += 1
sage: n
48
```

sage.combinat.designs.designs_pyx.**is_pairwise_balanced_design**(*blocks*, *v*, *K=None*, *lambd=1*, *verbose=False*)

Checks that input is a Pairwise Balanced Design (PBD) on $\{0, ..., v-1\}$

For more information on Pairwise Balanced Designs (PBD), see `PairwiseBalancedDesign`.

INPUT:

- `blocks` – collection of blocks
- `v` (integers) – size of the ground set assumed to be $X = \{0, ..., v-1\}$.
- `K` – list of integers of which the sizes of the blocks must be elements. Set to `None` (automatic guess) by default.
- `lambd` – value of λ. Set to 1 by default.
- `verbose` (boolean) – whether to display some information when the design is not a PBD.

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_pairwise_balanced_design
sage: sts = designs.steiner_triple_system(9)
sage: is_pairwise_balanced_design(sts,9,[3],1)
True
sage: TD = designs.transversal_design(4,10).blocks()
sage: groups = [range(i*10,(i+1)*10) for i in range(4)]
sage: is_pairwise_balanced_design(TD+groups,40,[4,10],1,verbose=True)
True
```

TESTS:

```
sage: from sage.combinat.designs.designs_pyx import is_pairwise_balanced_design
sage: is_pairwise_balanced_design(TD+groups,40,[4,10],2,verbose=True)
the pair (0,1) has been seen 1 times but lambda=2
False
sage: is_pairwise_balanced_design(TD+groups,40,[10],1,verbose=True)
a block has size 4 while K=[10]
```

```
False
sage: is_pairwise_balanced_design([[2,2]],40,[2],1,verbose=True)
The following block has repeated elements: [2, 2]
False
sage: is_pairwise_balanced_design([["e",2]],40,[2],1,verbose=True)
e does not belong to [0,...,39]
False
```

sage.combinat.designs.designs_pyx.**is_projective_plane**(*blocks*, *verbose=False*)

Test whether the blocks form a projective plane on $\{0, ..., v-1\}$

A *projective plane* is an incidence structure that has the following properties:

1. Given any two distinct points, there is exactly one line incident with both of them.
2. Given any two distinct lines, there is exactly one point incident with both of them.
3. There are four points such that no line is incident with more than two of them.

For more informations, see Wikipedia article Projective_plane.

`is_t_design()` can also check if an incidence structure is a projective plane with the parameters $v = k^2 + k + 1$, $t = 2$ and $l = 1$.

INPUT:

- `blocks` – collection of blocks
- `verbose` – whether to print additional information

EXAMPLES:

```
sage: from sage.combinat.designs.designs_pyx import is_projective_plane
sage: p = designs.projective_plane(4)
sage: b = p.blocks()
sage: is_projective_plane(b, verbose=True)
True

sage: p = designs.projective_plane(2)
sage: b = p.blocks()
sage: is_projective_plane(b)
True
sage: b[0][2] = 5
sage: is_projective_plane(b, verbose=True)
the pair (0,5) has been seen 2 times but lambda=1
False

sage: is_projective_plane([[0,1,2],[1,2,4]], verbose=True)
the pair (0,3) has been seen 0 times but lambda=1
False

sage: is_projective_plane([[1]], verbose=True)
First block has less than 3 points.
False

sage: p = designs.projective_plane(2)
sage: b = p.blocks()
sage: b[2].append(4)
sage: is_projective_plane(b, verbose=True)
a block has size 4 while K=[3]
False
```

sage.combinat.designs.designs_pyx.**is_quasi_difference_matrix**(*M*, *G*, *k*, *lmbda*, *mu*, *u*, *verbose=False*)

Test if the matrix is a $(G, k; \lambda, \mu; u)$-quasi-difference matrix

Let G be an abelian group of order n. A $(n, k; \lambda, \mu; u)$-quasi-difference matrix (QDM) is a matrix Q_{ij} with $\lambda(n-1+2u)+\mu$ rows and k columns, with each entry either equal to `None` (i.e. the 'missing entries') or to an element of G. Each column contains exactly λu empty entries, and each row contains at most one `None`. Furthermore, for each $1 \leq i < j \leq k$, the multiset

$$\{q_{li} - q_{lj} : 1 \leq l \leq \lambda(n-1+2u)+\mu, \text{ with } q_{li} \text{ and } q_{lj} \text{ not empty}\}$$

contains λ times every nonzero element of G and contains μ times 0.

INPUT:

- `M` – a matrix with entries from `G` (or equal to `None` for missing entries)
- `G` – a group
- `k, lmbda, mu, u` – integers
- `verbose` (boolean) – whether to print some information when the answer is `False`.

EXAMPLES:

Differences matrices:

```
sage: from sage.combinat.designs.designs_pyx import is_quasi_difference_matrix
sage: q = 3**3
sage: F = GF(q,'x')
sage: M = [[x*y for y in F] for x in F]
sage: is_quasi_difference_matrix(M,F,q,1,1,0,verbose=1)
True

sage: B = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
....:      [0, 1, 2, 3, 4, 2, 3, 4, 0, 1],
....:      [0, 2, 4, 1, 3, 3, 0, 2, 4, 1]]
sage: G = GF(5)
sage: B = [[G(b) for b in R] for R in B]
sage: is_quasi_difference_matrix(zip(*B),G,3,2,2,0)
True
```

A quasi-difference matrix from the database:

```
sage: from sage.combinat.designs.database import QDM
sage: G,M = QDM[38,1][37,1,1,1][1]()
sage: is_quasi_difference_matrix(M,G,k=6,lmbda=1,mu=1,u=1)
True
```

Bad input:

```
sage: is_quasi_difference_matrix(M,G,k=6,lmbda=1,mu=1,u=3,verbose=1)
The matrix has 39 rows instead of lambda(|G|-1+2u)+mu=1(37-1+2.3)+1=43
False
sage: is_quasi_difference_matrix(M,G,k=6,lmbda=1,mu=2,u=1,verbose=1)
The matrix has 39 rows instead of lambda(|G|-1+2u)+mu=1(37-1+2.1)+2=40
False
sage: M[3][1] = None
sage: is_quasi_difference_matrix(M,G,k=6,lmbda=1,mu=1,u=1,verbose=1)
Row 3 contains more than one empty entry
False
sage: M[3][1] = 1
sage: M[6][1] = None
```

```
sage: is_quasi_difference_matrix(M,G,k=6,lmbda=1,mu=1,u=1,verbose=1)
Column 1 contains 2 empty entries instead of the expected lambda.u=1.1=1
False
```

5.1.74 Difference families

This module gathers everything related to difference families. One can build a difference family (or check that it can be built) with `difference_family()`:

```
sage: G,F = designs.difference_family(13,4,1)
```

It defines the following functions:

`are_mcfarland_1973_parameters(v,k,lmbda)`	Test whether `(v,k,lmbda)` is a triple that can be obtained from the construction from [McF1973].
`block_stabilizer()`	Compute the left stabilizer of the block `B` under the action of `G`.
`df_q_6_1()`	Return a $(q,6,1)$-difference family over the finite field K.
`difference_family()`	Return a (`k`, `l`)-difference family on an Abelian group of cardinality `v`.
`group_law()`	Return a triple `(identity, operation, inverse)` that define the operations on the group `G`.
`is_difference_family()`	Check wether `D` forms a difference family in the group `G`.
`mcfarland_1973_construction()`	Return a difference set.
`one_cyclic_tiling()`	Given a subset `A` of the cyclic additive group $G = Z/nZ$ return another subset B so that $A + B = G$ and $\|A\|\|B\| = n$ (i.e. any element of G is uniquely expressed as a sum $a + b$ with a in A and b in B).
`one_radical_difference_family()`	Search for a radical difference family on `K` using dancing links algorithm.
`radical_difference_family()`	Return a `(v,k,l)`-radical difference family.
`radical_difference_set()`	Return a difference set made of a cyclotomic coset in the finite field `K` and with paramters `k` and `l`.
`singer_difference_set()`	Return a difference set associated to the set of hyperplanes in a projective space of dimension d over $GF(q)$.
`twin_prime_powers_difference_set()`	Return a difference set on $GF(p) \times GF(p+2)$.

REFERENCES:

Functions

`sage.combinat.designs.difference_family.`**`are_mcfarland_1973_parameters`**(*v*, *k*, *lmbda*, *return_parameters=False*)

Test whether `(v,k,lmbda)` is a triple that can be obtained from the construction from [McF1973].

See `mcfarland_1973_construction()`.

INPUT:

- `v`, `k`, `lmbda` - (integers) parameters of the difference family
- `return_parameters` – (boolean, default `False`) if `True` return a pair `(True, (q, s))` so that `(q,s)` can be used in the function `mcfarland_1973_construction()` to actually build a `(v,k,lmbda)`-difference family. Or `(False, None)` if the construction is not possible.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import are_mcfarland_1973_parameters
sage: are_mcfarland_1973_parameters(64, 28, 12)
True
sage: are_mcfarland_1973_parameters(64, 28, 12, return_parameters=True)
(True, (2, 2))
sage: are_mcfarland_1973_parameters(60, 13, 5)
False
sage: are_mcfarland_1973_parameters(98125, 19500, 3875)
True
sage: are_mcfarland_1973_parameters(98125, 19500, 3875, True)
(True, (5, 3))

sage: from sage.combinat.designs.difference_family import are_mcfarland_1973_parameters
sage: for v in range(1, 100):
....:     for k in range(1,30):
....:         for l in range(1,15):
....:             if are_mcfarland_1973_parameters(v,k,l):
....:                 answer, (q,s) = are_mcfarland_1973_parameters(v,k,l,return_parameters=True
....:                 print v,k,l,q,s
....:                 assert answer is True
....:                 assert designs.difference_family(v,k,l,existence=True) is True
....:                 G,D = designs.difference_family(v,k,l)
16 6 2 2 1
45 12 3 3 1
64 28 12 2 2
96 20 4 4 1
```

sage.combinat.designs.difference_family.**block_stabilizer**(*G*, *B*)

Compute the left stabilizer of the block `B` under the action of `G`.

This function return the list of all $x \in G$ such that $x \cdot B = B$ (as a set).

INPUT:

- `G` – a group (additive or multiplicative).
- `B` – a subset of `G`.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import block_stabilizer

sage: Z8 = Zmod(8)
sage: block_stabilizer(Z8, [Z8(0),Z8(2),Z8(4),Z8(6)])
[0, 2, 4, 6]
sage: block_stabilizer(Z8, [Z8(0),Z8(2)])
[0]

sage: C = cartesian_product([Zmod(4),Zmod(3)])
sage: block_stabilizer(C, [C((0,0)),C((2,0)),C((0,1)),C((2,1))])
[(0, 0), (2, 0)]

sage: b = map(Zmod(45),[1, 3, 7, 10, 22, 25, 30, 35, 37, 38, 44])
sage: block_stabilizer(Zmod(45),b)
[0]
```

sage.combinat.designs.difference_family.**df_q_6_1**(*K*, *existence=False*, *check=True*)

Return a $(q, 6, 1)$-difference family over the finite field K.

The construction uses Theorem 11 of [Wi72].

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import is_difference_family, df_q_6_1
sage: prime_powers = [v for v in xrange(31,500,30) if is_prime_power(v)]
sage: parameters = [v for v in prime_powers if df_q_6_1(GF(v,'a'), existence=True)]
sage: print parameters
[31, 151, 181, 211, 241, 271, 331, 361, 421]
sage: for v in parameters:
....:     K = GF(v, 'a')
....:     df = df_q_6_1(K, check=True)
....:     assert is_difference_family(K, df, v, 6, 1)
```

sage.combinat.designs.difference_family.**difference_family**(*v*, *k*, *l=1*, *existence=False*, *explain_construction=False*, *check=True*)

Return a (k, l)-difference family on an Abelian group of cardinality v.

Let G be a finite Abelian group. For a given subset D of G, we define ΔD to be the multi-set of differences $\Delta D = \{x - y; x \in D, y \in D, x \neq y\}$. A (G, k, λ)-*difference family* is a collection of k-subsets of G, $D = \{D_1, D_2, \ldots, D_b\}$ such that the union of the difference sets ΔD_i for $i = 1, ...b$, seen as a multi-set, contains each element of $G \backslash \{0\}$ exactly λ-times.

When there is only one block, i.e. $\lambda(v-1) = k(k-1)$, then a (G, k, λ)-difference family is also called a *difference set*.

See also Wikipedia article Difference_set.

If there is no such difference family, an EmptySetError is raised and if there is no construction at the moment NotImplementedError is raised.

INPUT:

- v, k, l – parameters of the difference family. If l is not provided it is assumed to be 1.
- existence – if True, then return either True if Sage knows how to build such design, Unknown if it does not and False if it knows that the design does not exist.
- explain_construction – instead of returning a difference family, returns a string that explains the construction used.
- check – boolean (default: True). If True then the result of the computation is checked before being returned. This should not be needed but ensures that the output is correct.

OUTPUT:

A pair (G,D) made of a group G and a difference family D on that group. Or, if existence is True a troolean or if explain_construction is True a string.

EXAMPLES:

```
sage: G,D = designs.difference_family(73,4)
sage: G
Finite Field of size 73
sage: D
[[0, 1, 5, 18],
 [0, 3, 15, 54],
 [0, 9, 45, 16],
 [0, 27, 62, 48],
 [0, 8, 40, 71],
 [0, 24, 47, 67]]

sage: print designs.difference_family(73, 4, explain_construction=True)
```

```
The database contains a (73,4)-evenly distributed set

sage: G,D = designs.difference_family(15,7,3)
sage: G
Ring of integers modulo 15
sage: D
[[0, 1, 2, 4, 5, 8, 10]]
sage: print designs.difference_family(15,7,3,explain_construction=True)
Singer difference set

sage: print designs.difference_family(91,10,1,explain_construction=True)
Singer difference set
sage: print designs.difference_family(64,28,12, explain_construction=True)
McFarland 1973 construction
```

For $k = 6, 7$ we look at the set of small prime powers for which a construction is available:

```
sage: def prime_power_mod(r,m):
....:     k = m+r
....:     while True:
....:         if is_prime_power(k):
....:             yield k
....:         k += m

sage: from itertools import islice
sage: l6 = {True:[], False: [], Unknown: []}
sage: for q in islice(prime_power_mod(1,30), 60):
....:     l6[designs.difference_family(q,6,existence=True)].append(q)
sage: l6[True]
[31, 121, 151, 181, 211, ...,  3061, 3121, 3181]
sage: l6[Unknown]
[61]
sage: l6[False]
[]

sage: l7 = {True: [], False: [], Unknown: []}
sage: for q in islice(prime_power_mod(1,42), 60):
....:     l7[designs.difference_family(q,7,existence=True)].append(q)
sage: l7[True]
[169, 337, 379, 421, 463, 547, 631, 673, 757, 841, 883, 967, ...,  4621, 4957, 5167]
sage: l7[Unknown]
[43, 127, 211, 2017, 2143, 2269, 2311, 2437, 2521, 2647, ..., 4999, 5041, 5209]
sage: l7[False]
[]
```

List available constructions:

```
sage: for v in xrange(2,100):
....:     constructions = []
....:     for k in xrange(2,10):
....:         for l in xrange(1,10):
....:             if designs.difference_family(v,k,l,existence=True):
....:                 constructions.append((k,l))
....:                 _ = designs.difference_family(v,k,l)
....:     if constructions:
....:         print "%2d: %s"%(v, ', '.join('(%d,%d)'%(k,l) for k,l in constructions))
 3: (2,1)
 4: (3,2)
 5: (2,1), (4,3)
```

```
 6: (5,4)
 7: (2,1), (3,1), (3,2), (4,2), (6,5)
 8: (7,6)
 9: (2,1), (4,3), (8,7)
10: (9,8)
11: (2,1), (4,6), (5,2), (5,4), (6,3)
13: (2,1), (3,1), (3,2), (4,1), (4,3), (5,5), (6,5)
15: (3,1), (4,6), (5,6), (7,3)
16: (3,2), (5,4), (6,2)
17: (2,1), (4,3), (5,5), (8,7)
19: (2,1), (3,1), (3,2), (4,2), (6,5), (9,4), (9,8)
21: (3,1), (4,3), (5,1), (6,3), (6,5)
22: (4,2), (6,5), (7,4), (8,8)
23: (2,1)
25: (2,1), (3,1), (3,2), (4,1), (4,3), (6,5), (7,7), (8,7)
27: (2,1), (3,1)
28: (3,2), (6,5)
29: (2,1), (4,3), (7,3), (7,6), (8,4), (8,6)
31: (2,1), (3,1), (3,2), (4,2), (5,2), (5,4), (6,1), (6,5)
33: (3,1), (5,5), (6,5)
34: (4,2)
35: (5,2)
37: (2,1), (3,1), (3,2), (4,1), (4,3), (6,5), (9,2), (9,8)
39: (3,1), (6,5)
40: (3,2), (4,1)
41: (2,1), (4,3), (5,1), (5,4), (6,3), (8,7)
43: (2,1), (3,1), (3,2), (4,2), (6,5), (7,2), (7,3), (7,6), (8,4)
45: (3,1), (5,1)
46: (4,2), (6,2)
47: (2,1)
49: (2,1), (3,1), (3,2), (4,1), (4,3), (6,5), (8,7), (9,3)
51: (3,1), (5,2), (6,3)
52: (4,1)
53: (2,1), (4,3)
55: (3,1), (9,4)
57: (3,1), (7,3), (8,1)
59: (2,1)
61: (2,1), (3,1), (3,2), (4,1), (4,3), (5,1), (5,4), (6,2), (6,3), (6,5)
63: (3,1)
64: (3,2), (4,1), (7,2), (7,6), (9,8)
65: (5,1)
67: (2,1), (3,1), (3,2), (6,5)
69: (3,1)
71: (2,1), (5,2), (5,4), (7,3), (7,6), (8,4)
73: (2,1), (3,1), (3,2), (4,1), (4,3), (6,5), (8,7), (9,1), (9,8)
75: (3,1), (5,2)
76: (4,1)
79: (2,1), (3,1), (3,2), (6,5)
81: (2,1), (3,1), (4,3), (5,1), (5,4), (8,7)
83: (2,1)
85: (4,1), (7,2), (7,3), (8,2)
89: (2,1), (4,3), (8,7)
91: (6,1), (7,1)
97: (2,1), (3,1), (3,2), (4,1), (4,3), (6,5), (8,7), (9,3)
```

TESTS:

Check more of the Wilson constructions from [Wi72]:

```
sage: Q5 = [241, 281,421,601,641, 661, 701, 821,881]
sage: Q9 = [73, 1153, 1873, 2017]
sage: Q15 = [76231]
sage: Q4 = [13, 73, 97, 109, 181, 229, 241, 277, 337, 409, 421, 457]
sage: Q8 = [1009, 3137, 3697]
sage: for Q,k in [(Q4,4),(Q5,5),(Q8,8),(Q9,9),(Q15,15)]:
....:     for q in Q:
....:         assert designs.difference_family(q,k,1,existence=True) is True
....:         _ = designs.difference_family(q,k,1)
```

Check Singer difference sets:

```
sage: sgp = lambda q,d: ((q**(d+1)-1)//(q-1), (q**d-1)//(q-1), (q**(d-1)-1)//(q-1))

sage: for q in range(2,10):
....:     if is_prime_power(q):
....:         for d in [2,3,4]:
....:           v,k,l = sgp(q,d)
....:           assert designs.difference_family(v,k,l,existence=True) is True
....:           _ = designs.difference_family(v,k,l)
```

Check twin primes difference sets:

```
sage: for p in [3,5,7,9,11]:
....:     v = p*(p+2); k = (v-1)/2;  lmbda = (k-1)/2
....:     G,D = designs.difference_family(v,k,lmbda)
```

Check the database:

```
sage: from sage.combinat.designs.database import DF,EDS
sage: for v,k,l in DF:
....:     assert designs.difference_family(v,k,l,existence=True) is True
....:     df = designs.difference_family(v,k,l,check=True)

sage: for k in EDS:
....:     for v in EDS[k]:
....:         assert designs.difference_family(v,k,1,existence=True) is True
....:         df = designs.difference_family(v,k,1,check=True)
```

Check a failing construction (trac ticket #17528):

```
sage: designs.difference_family(9,3)
Traceback (most recent call last):
...
NotImplementedError: No construction available for (9,3,1)-difference family
```

Todo

Implement recursive constructions from Buratti "Recursive for difference matrices and relative difference families" (1998) and Jungnickel "Composition theorems for difference families and regular planes" (1978)

sage.combinat.designs.difference_family.**group_law**(*G*)

Return a triple `(identity, operation, inverse)` that define the operations on the group `G`.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import group_law
sage: group_law(Zmod(3))
(0, <built-in function add>, <built-in function neg>)
```

```
sage: group_law(SymmetricGroup(5))
((), <built-in function mul>, <built-in function inv>)
sage: group_law(VectorSpace(QQ,3))
((0, 0, 0), <built-in function add>, <built-in function neg>)
```

sage.combinat.designs.difference_family.**is_difference_family**(*G*, *D*, *v=None*, *k=None*, *l=None*, *verbose=False*)

Check wether `D` forms a difference family in the group `G`.

INPUT:

- `G` - group of cardinality `v`
- `D` - a set of `k`-subsets of `G`
- `v`, `k` and `l` - optional parameters of the difference family
- `verbose` - whether to print additional information

See also:

`difference_family()`

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import is_difference_family
sage: G = Zmod(21)
sage: D = [[0,1,4,14,16]]
sage: is_difference_family(G, D, 21, 5)
True

sage: G = Zmod(41)
sage: D = [[0,1,4,11,29],[0,2,8,17,21]]
sage: is_difference_family(G, D, verbose=True)
Too few:
  5 is obtained 0 times in blocks []
  14 is obtained 0 times in blocks []
  27 is obtained 0 times in blocks []
  36 is obtained 0 times in blocks []
Too much:
  4 is obtained 2 times in blocks [0, 1]
  13 is obtained 2 times in blocks [0, 1]
  28 is obtained 2 times in blocks [0, 1]
  37 is obtained 2 times in blocks [0, 1]
False
sage: D = [[0,1,4,11,29],[0,2,8,17,22]]
sage: is_difference_family(G, D)
True

sage: G = Zmod(61)
sage: D = [[0,1,3,13,34],[0,4,9,23,45],[0,6,17,24,32]]
sage: is_difference_family(G, D)
True

sage: G = AdditiveAbelianGroup([3]*4)
sage: a,b,c,d = G.gens()
sage: D = [[d, -a+d, -c+d, a-b-d, b+c+d],
....:      [c, a+b-d, -b+c, a-b+d, a+b+c],
....:      [-a-b+c+d, a-b-c-d, -a+c-d, b-c+d, a+b],
....:      [-b-d, a+b+d, a-b+c-d, a-b+c, -b+c+d]]
```

```
sage: is_difference_family(G, D)
True
```

The following example has a third block with a non-trivial stabilizer:

```
sage: G = Zmod(15)
sage: D = [[0,1,4],[0,2,9],[0,5,10]]
sage: is_difference_family(G,D,verbose=True)
It is a (15,3,1)-difference family
True
```

The function also supports multiplicative groups (non necessarily Abelian):

```
sage: G = DihedralGroup(8)
sage: x,y = G.gens()
sage: i = G.one()
sage: D1 = [[i,x,x^4], [i,x^2, y*x], [i,x^5,y], [i,x^6,y*x^2], [i,x^7,y*x^5]]
sage: is_difference_family(G, D1, 16, 3, 2)
True
sage: from sage.combinat.designs.bibd import BIBD_from_difference_family
sage: bibd = BIBD_from_difference_family(G,D1,lambd=2)
```

TESTS:

```
sage: K = GF(3^2,'z')
sage: z = K.gen()
sage: D = [[1,z+1,2]]
sage: _ = is_difference_family(K, D, verbose=True)
the number of differences (=6) must be a multiple of v-1=8
sage: _
False
```

sage.combinat.designs.difference_family.**mcfarland_1973_construction**(q, s)

Return a difference set.

The difference set returned has the following parameters

$$v = \frac{q^{s+1}(q^{s+1}+q-2)}{q-1}, k = \frac{q^s(q^{s+1}-1)}{q-1}, \lambda = \frac{q^s(q^s-1)}{q-1}$$

This construction is due to [McF1973].

INPUT:

- `q`, `s` - (integers) parameters for the difference set (see the above formulas for the expression of `v`, `k`, `l` in terms of `q` and `s`)

See also:

The function `are_mcfarland_1973_parameters()` makes the translation between the parameters (q, s) corresponding to a given triple (v, k, λ).

REFERENCES:

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import (
....:     mcfarland_1973_construction, is_difference_family)

sage: G,D = mcfarland_1973_construction(3, 1)
sage: assert is_difference_family(G, D, 45, 12, 3)
```

```
sage: G,D = mcfarland_1973_construction(2, 2)
sage: assert is_difference_family(G, D, 64, 28, 12)
```

sage.combinat.designs.difference_family.**one_cyclic_tiling**(A, n)

Given a subset A of the cyclic additive group $G = Z/nZ$ return another subset B so that $A + B = G$ and $|A||B| = n$ (i.e. any element of G is uniquely expressed as a sum $a + b$ with a in A and b in B).

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import one_cyclic_tiling
sage: tile = [0,2,4]
sage: m = one_cyclic_tiling(tile,6); m
[0, 3]
sage: sorted((i+j)%6 for i in tile for j in m)
[0, 1, 2, 3, 4, 5]

sage: def print_tiling(tile, translat, n):
....:     for x in translat:
....:         print ''.join('X' if (i-x)%n in tile else '.' for i in range(n))

sage: tile = [0, 1, 2, 7]
sage: m = one_cyclic_tiling(tile, 12)
sage: print_tiling(tile, m, 12)
XXX....X....
....XXX....X
...X....XXX.

sage: tile = [0, 1, 5]
sage: m = one_cyclic_tiling(tile, 12)
sage: print_tiling(tile, m, 12)
XX...X......
...XX...X...
......XX...X
..X......XX.

sage: tile = [0, 2]
sage: m = one_cyclic_tiling(tile, 8)
sage: print_tiling(tile, m, 8)
X.X.....
....X.X.
.X.X....
.....X.X
```

ALGORITHM:

Uses dancing links sage.combinat.dlx

sage.combinat.designs.difference_family.**one_radical_difference_family**(K, k)

Search for a radical difference family on K using dancing links algorithm.

For the definition of radical difference family, see radical_difference_family(). Here, we consider only radical difference family with $\lambda = 1$.

INPUT:

- K – a finite field of cardinality q.
- k – a positive integer so that $k(k-1)$ divides $q-1$.

OUTPUT:

Either a difference family or `None` if it does not exist.

ALGORITHM:

The existence of a radical difference family is equivalent to a one dimensional tiling (or packing) problem in a cyclic group. This subsequent problem is solved by a call to the function `one_cyclic_tiling()`.

Let K^* be the multiplicative group of the finite field K. A radical family has the form $\mathcal{B} = \{x_1B, \ldots, x_kB\}$, where $B = \{x : x^k = 1\}$ (for k odd) or $B = \{x : x^{k-1} = 1\} \cup \{0\}$ (for k even). Equivalently, K^* decomposes as:

$$K^* = \Delta(x_1B) \cup \ldots \cup \Delta(x_kB) = x_1\Delta B \cup \ldots \cup x_k\Delta B$$

We observe that $C = B\backslash 0$ is a subgroup of the (cyclic) group K^*, that can thus be generated by some element r. Furthermore, we observe that ΔB is always a union of cosets of $\pm C$ (which is twice larger than C).

$$\begin{array}{lll} (k \text{ odd}) & \Delta B & = \{r^i - r^j : r^i \neq r^j\} & = \pm C \cdot \{r^i - 1 : 0 < i \leq m\} \\ (k \text{ even}) & \Delta B & = \{r^i - r^j : r^i \neq r^j\} \cup C & = \pm C \cdot \{r^i - 1 : 0 < i < m\} \cup \pm C \end{array}$$

where

$$(k \text{ odd})\ m = (k-1)/2 \quad \text{and} \quad (k \text{ even})\ m = k/2.$$

Consequently, $\mathcal{B} = \{x_1B, \ldots, x_kB\}$ is a radical difference family if and only if $\{x_1(\Delta B/(\pm C)), \ldots, x_k(\Delta B/(\pm C))\}$ is a partition of the cyclic group $K^*/(\pm C)$.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import (
....:     one_radical_difference_family,
....:     is_difference_family)

sage: one_radical_difference_family(GF(13),4)
[[0, 1, 3, 9]]
```

The parameters that appear in [Bu95]:

```
sage: df = one_radical_difference_family(GF(449), 8); df
[[0, 1, 18, 25, 176, 324, 359, 444],
 [0, 9, 88, 162, 222, 225, 237, 404],
 [0, 11, 140, 198, 275, 357, 394, 421],
 [0, 40, 102, 249, 271, 305, 388, 441],
 [0, 49, 80, 93, 161, 204, 327, 433],
 [0, 70, 99, 197, 230, 362, 403, 435],
 [0, 121, 141, 193, 293, 331, 335, 382],
 [0, 191, 285, 295, 321, 371, 390, 392]]
sage: is_difference_family(GF(449), df, 449, 8, 1)
True
```

sage.combinat.designs.difference_family.**radical_difference_family**(*K*, *k*, *l=1*, *existence=False*, *check=True*)

Return a `(v,k,l)`-radical difference family.

Let fix an integer k and a prime power $q = tk(k-1)+1$. Let K be a field of cardinality q. A $(q, k, 1)$-difference family is *radical* if its base blocks are either: a coset of the k-th root of unity for k odd or a coset of $k-1$-th root of unity and 0 if k is even (the number t is the number of blocks of that difference family).

The terminology comes from M. Buratti article [Bu95] but the first constructions go back to R. Wilson [Wi72].

INPUT:

- K - a finite field
- k – positive integer, the size of the blocks
- l – the λ parameter (default to 1)
- existence – if True, then return either True if Sage knows how to build such design, Unknown if it does not and False if it knows that the design does not exist.
- check – boolean (default: True). If True then the result of the computation is checked before being returned. This should not be needed but ensures that the output is correct.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import radical_difference_family

sage: radical_difference_family(GF(73),9)
[[1, 2, 4, 8, 16, 32, 37, 55, 64]]

sage: radical_difference_family(GF(281),5)
[[1, 86, 90, 153, 232],
 [4, 50, 63, 79, 85],
 [5, 36, 149, 169, 203],
 [7, 40, 68, 219, 228],
 [9, 121, 212, 248, 253],
 [29, 81, 222, 246, 265],
 [31, 137, 167, 247, 261],
 [32, 70, 118, 119, 223],
 [39, 56, 66, 138, 263],
 [43, 45, 116, 141, 217],
 [98, 101, 109, 256, 279],
 [106, 124, 145, 201, 267],
 [111, 123, 155, 181, 273],
 [156, 209, 224, 264, 271]]

sage: for k in range(5,10):
....:     print "k = {}".format(k)
....:     for q in range(k*(k-1)+1, 2000, k*(k-1)):
....:         if is_prime_power(q):
....:             K = GF(q,'a')
....:             if radical_difference_family(K, k, existence=True):
....:                 print q,
....:                 _ = radical_difference_family(K,k)
....:     print
k = 5
41 61 81 241 281 401 421 601 641 661 701 761 821 881 1181 1201 1301 1321
1361 1381 1481 1601 1681 1801 1901
k = 6
181 211 241 631 691 1531 1831 1861
k = 7
337 421 463 883 1723
k = 8
449 1009
k = 9
73 1153 1873
```

sage.combinat.designs.difference_family.**radical_difference_set**(*K*, *k*, *l=1*, *existence=False*, *check=True*)

Return a difference set made of a cyclotomic coset in the finite field K and with paramters k and l.

Most of these difference sets appear in chapter VI.18.48 of the Handbook of combinatorial designs.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import radical_difference_set

sage: D = radical_difference_set(GF(7), 3, 1); D
[[1, 2, 4]]
sage: sorted(x-y for x in D[0] for y in D[0] if x != y)
[1, 2, 3, 4, 5, 6]

sage: D = radical_difference_set(GF(16,'a'), 6, 2)
sage: sorted(x-y for x in D[0] for y in D[0] if x != y)
[1,
 1,
 a,
 a,
 a + 1,
 a + 1,
 a^2,
 a^2,
 ...
 a^3 + a^2 + a + 1,
 a^3 + a^2 + a + 1]

sage: for k in range(2,50):
....:     for l in reversed(divisors(k*(k-1))):
....:         v = k*(k-1)//l + 1
....:         if is_prime_power(v) and radical_difference_set(GF(v,'a'),k,l,existence=True):
....:             _ = radical_difference_set(GF(v,'a'),k,l)
....:             print "{:3} {:3} {:3}".format(v,k,l)
  3   2   1
  4   3   2
  7   3   1
  5   4   3
  7   4   2
 13   4   1
 11   5   2
  7   6   5
 11   6   3
 16   6   2
  8   7   6
  9   8   7
 19   9   4
 37   9   2
 73   9   1
 11  10   9
 19  10   5
 23  11   5
 13  12  11
 23  12   6
 27  13   6
 27  14   7
 16  15  14
 31  15   7
...
 41  40  39
 79  40  20
 83  41  20
 43  42  41
```

```
 83  42  21
 47  46  45
 49  48  47
197  49  12
```

sage.combinat.designs.difference_family.**singer_difference_set**(q, d)

Return a difference set associated to the set of hyperplanes in a projective space of dimension d over $GF(q)$.

A Singer difference set has parameters:

$$v = \frac{q^{d+1}-1}{q-1}, \quad k = \frac{q^d-1}{q-1}, \quad \lambda = \frac{q^{d-1}-1}{q-1}.$$

The idea of the construction is as follows. One consider the finite field $GF(q^{d+1})$ as a vector space of dimension $d+1$ over $GF(q)$. The set of $GF(q)$-lines in $GF(q^{d+1})$ is a projective plane and its set of hyperplanes form a balanced incomplete block design.

Now, considering a multiplicative generator z of $GF(q^{d+1})$, we get a transitive action of a cyclic group on our projective plane from which it is possible to build a difference set.

The construction is given in details in [Stinson2004], section 3.3.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import singer_difference_set, is_difference_f
sage: G,D = singer_difference_set(3,2)
sage: is_difference_family(G,D,verbose=True)
It is a (13,4,1)-difference family
True

sage: G,D = singer_difference_set(4,2)
sage: is_difference_family(G,D,verbose=True)
It is a (21,5,1)-difference family
True

sage: G,D = singer_difference_set(3,3)
sage: is_difference_family(G,D,verbose=True)
It is a (40,13,4)-difference family
True

sage: G,D = singer_difference_set(9,3)
sage: is_difference_family(G,D,verbose=True)
It is a (820,91,10)-difference family
True
```

sage.combinat.designs.difference_family.**twin_prime_powers_difference_set**(p, *check=True*)

Return a difference set on $GF(p) \times GF(p+2)$.

The difference set is built from the following element of the Cartesian product of finite fields $GF(p) \times GF(p+2)$:

- $(x, 0)$ with any x
- (x, y) with x and y squares
- (x, y) with x and y non-squares

For more information see Wikipedia article Difference_set.

INPUT:

- `check` – boolean (default: `True`). If `True` then the result of the computation is checked before being returned. This should not be needed but ensures that the output is correct.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_family import twin_prime_powers_difference_set
sage: G,D = twin_prime_powers_difference_set(3)
sage: G
The Cartesian product of (Finite Field of size 3, Finite Field of size 5)
sage: D
[[(1, 1), (1, 4), (2, 2), (2, 3), (0, 0), (1, 0), (2, 0)]]
```

5.1.75 Difference Matrices

This module gathers code related to difference matrices. One can build those objects (or know if they can be built) with `difference_matrix()`:

```
sage: G,DM = designs.difference_matrix(9,5,1)
```

Functions

`sage.combinat.designs.difference_matrices.`**`difference_matrix`**(*g*, *k*, *lmbda=1*, *existence=False*, *check=True*)

Return a (g, k, λ)-difference matrix

A matrix M is a (g, k, λ)-difference matrix if it has size $\lambda g \times k$, its entries belong to the group G of cardinality g, and for any two rows R, R' of M and $x \in G$ there are exactly λ values i such that $R_i - R'_i = x$.

INPUT:

- `k` – (integer) number of columns. If `k=None` it is set to the largest value available.
- `g` – (integer) cardinality of the group G
- `lmbda` – (integer; default: 1) – number of times each element of G appears as a difference.
- `check` – (boolean) Whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.
- `existence` (boolean) – instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`).
  - `False` – meaning that the design does not exist.

Note: When `k=None` and `existence=True` the function returns an integer, i.e. the largest k such that we can build a (g, k, λ)-DM.

EXAMPLES:

```
sage: G,M = designs.difference_matrix(25,10); G
Finite Field in x of size 5^2
sage: designs.difference_matrix(993,None,existence=1)
32
```

Here we print for each g the maximum possible k for which Sage knows how to build a $(g, k, 1)$-difference matrix:

```
sage: for g in range(2,30):
....:     k_max = designs.difference_matrix(g=g,k=None,existence=True)
....:     print "{:2} {}".format(g, k_max)
....:     _ = designs.difference_matrix(g,k_max)
 2 2
 3 3
 4 4
 5 5
 6 2
 7 7
 8 8
 9 9
10 2
11 11
12 6
13 13
14 2
15 3
16 16
17 17
18 2
19 19
20 4
21 6
22 2
23 23
24 8
25 25
26 2
27 27
28 6
29 29
```

TESTS:

```
sage: designs.difference_matrix(10,12,1,existence=True)
False
sage: designs.difference_matrix(10,12,1)
Traceback (most recent call last):
...
EmptySetError: No (10,12,1)-Difference Matrix exists as k(=12)>g(=10)
sage: designs.difference_matrix(10,9,1,existence=True)
Unknown
sage: designs.difference_matrix(10,9,1)
Traceback (most recent call last):
...
NotImplementedError: I don't know how to build a (10,9,1)-Difference Matrix!
```

sage.combinat.designs.difference_matrices.**difference_matrix_product**(*k*, *M1*, *G1*, *lmbda1*, *M2*, *G2*, *lmbda2*, *check=True*)

Return the product of the `(G1,k,lmbda1)` and `(G2,k,lmbda2)` difference matrices `M1` and `M2`.

The result is a $(G1 \times G2, k, \lambda_1\lambda_2)$-difference matrix.

INPUT:

- k, lmbda1, lmbda2 – positive integer
- G1, G2 – groups
- M1, M2 – (G1, k, lmbda1) and (G, k, lmbda2) difference matrices
- check (boolean) – if True (default), the output is checked before being returned.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_matrices import (
....:     difference_matrix_product,
....:     is_difference_matrix)
sage: G1,M1 = designs.difference_matrix(11,6)
sage: G2,M2 = designs.difference_matrix(7,6)
sage: G,M = difference_matrix_product(6,M1,G1,1,M2,G2,1)
sage: G1
Finite Field of size 11
sage: G2
Finite Field of size 7
sage: G
The Cartesian product of (Finite Field of size 11, Finite Field of size 7)
sage: is_difference_matrix(M,G,6,1)
True
```

sage.combinat.designs.difference_matrices.**find_product_decomposition**(*g*, *k*, *lmbda=1*)

Try to find a product decomposition construction for difference matrices.

INPUT:

- g, k, lmbda – integers, parameters of the difference matrix

OUTPUT:

A pair of pairs (g1, lmbda), (g2, lmbda2) if Sage knows how to build $(g1, k, lmbda1)$ and $(g2, k, lmbda2)$ difference matrices and False otherwise.

EXAMPLES:

```
sage: from sage.combinat.designs.difference_matrices import find_product_decomposition
sage: find_product_decomposition(77,6)
((7, 1), (11, 1))
sage: find_product_decomposition(616,7)
((7, 1), (88, 1))
sage: find_product_decomposition(24,10)
False
```

5.1.76 Evenly distributed sets in finite fields

This module consists of a simple class EvenlyDistributedSetsBacktracker. Its main purpose is to iterate through the evenly distributed sets of a given finite field.

The naive backtracker implemented here is not directly used to generate difference family as even for small parameters it already takes time to run. Instead, its output has been stored into a database sage.combinat.designs.database. If the backtracker is improved, then one might want to update this database with more values.

Classes and methods

class sage.combinat.designs.evenly_distributed_sets.**EvenlyDistributedSetsBacktracker**
Bases: object

Set of evenly distributed subsets in finite fields.

Definition: Let K be a finite field of cardinality q and k an integer so that $k(k-1)$ divides $q-1$. Let $H = K^*$ be the multiplicative group of invertible elements in K. A *k-evenly distributed set* in K is a set $B = \{b_1, b_2, \ldots, b_k\}$ of k elements of K so that the $k(k-1)$ differences $\Delta B = \{b_i - b_j; i \neq j\}$ hit each coset modulo $H^{2(q-1)/(k(k-1))}$ exactly twice.

Evenly distributed sets were introduced by Wilson [Wi72] (see also [BJL99-1], Chapter VII.6). He proved that for any k, and for any prime power q large enough such that $k(k-1)$ divides k there exists a k-evenly distributed set in the field of cardinality q. This existence result based on a counting argument (using Dirichlet theorem) does not provide a simple method to generate them.

From a k-evenly distributed set, it is straightforward to build a $(q, k, 1)$-difference family (see `to_difference_family()`). Another approach to generate a difference family, somehow dual to this one, is via radical difference family (see in particular `radical_difference_family()` from the module `difference_family`).

By default, this backtracker only considers evenly distributed sets up to affine automorphisms, i.e. B is considered equivalent to $sB + t$ for any invertible element s and any element t in the field K. Note that the set of differences is just multiplicatively translated by s as $\Delta(sB+t) = s(\Delta B)$, and so that B is an evenly distributed set if and only if sB is one too. This behaviour can be modified via the argument `up_to_isomorphism` (see the input section and the examples below).

INPUT:

- `K` – a finite field of cardinality q
- `k` – a positive integer such that $k(k-1)$ divides $q-1$
- `up_to_isomorphism` - (boolean, default `True`) whether only consider evenly distributed sets up to automorphisms of the field of the form $x \mapsto ax + b$. If set to `False` then the iteration is over all evenly distributed sets that contain `0` and `1`.
- `check` – boolean (default is `False`). Whether you want to check intermediate steps of the iterator. This is mainly intended for debugging purpose. Setting it to `True` will considerably slow the iteration.

EXAMPLES:

The main part of the code is contained in the iterator. To get one evenly distributed set just do:

```
sage: from sage.combinat.designs.evenly_distributed_sets import EvenlyDistributedSetsBacktracker
sage: E = EvenlyDistributedSetsBacktracker(Zmod(151),6)
sage: B = E.an_element()
sage: B
[0, 1, 69, 36, 57, 89]
```

The class has a method to convert it to a difference family:

```
sage: E.to_difference_family(B)
[[0, 1, 69, 36, 57, 89],
 [0, 132, 48, 71, 125, 121],
 [0, 59, 145, 10, 41, 117],
 [0, 87, 114, 112, 127, 42],
 [0, 8, 99, 137, 3, 108]]
```

It is also possible to run over all evenly distributed sets:

```
sage: E = EvenlyDistributedSetsBacktracker(Zmod(13), 4, up_to_isomorphism=False)
sage: for B in E: print B
[0, 1, 11, 5]
[0, 1, 4, 6]
[0, 1, 9, 3]
[0, 1, 8, 10]

sage: E = EvenlyDistributedSetsBacktracker(Zmod(13), 4, up_to_isomorphism=True)
sage: for B in E: print B
[0, 1, 11, 5]
```

Or only count them:

```
sage: for k in range(13, 200, 12):
....:     if is_prime_power(k):
....:         K = GF(k,'a')
....:         E1 = EvenlyDistributedSetsBacktracker(K, 4, False)
....:         E2 = EvenlyDistributedSetsBacktracker(K, 4, True)
....:         print "{:3} {:3} {:3}".format(k, E1.cardinality(), E2.cardinality())
 13   4   1
 25  40   4
 37  12   1
 49  24   2
 61  12   1
 73  48   4
 97  64   6
109  72   6
121 240  20
157  96   8
169 240  20
181 204  17
193 336  28
```

Note that by definition, the number of evenly distributed sets up to isomorphisms is at most $k(k-1)$ times smaller than without isomorphisms. But it might not be exactly $k(k-1)$ as some of them might have symmetries:

```
sage: B = EvenlyDistributedSetsBacktracker(Zmod(13), 4).an_element()
sage: B
[0, 1, 11, 5]
sage: [9*x + 5 for x in B]
[5, 1, 0, 11]
sage: [3*x + 11 for x in B]
[11, 1, 5, 0]
```

an_element()

Return an evenly distributed set.

If there is no such subset raise an `EmptySetError`.

EXAMPLES:

```
sage: from sage.combinat.designs.evenly_distributed_sets import EvenlyDistributedSetsBacktra

sage: E = EvenlyDistributedSetsBacktracker(Zmod(41),5)
sage: E.an_element()
[0, 1, 13, 38, 31]

sage: E = EvenlyDistributedSetsBacktracker(Zmod(61),6)
sage: E.an_element()
Traceback (most recent call last):
```

```
...
EmptySetError: no 6-evenly distributed set in Ring of integers modulo 61
```

cardinality()

Return the number of evenly distributed sets.

Use with precaution as there can be a lot of such sets and this method might be very long to answer!

EXAMPLES:

```
sage: from sage.combinat.designs.evenly_distributed_sets import EvenlyDistributedSetsBacktra

sage: E = EvenlyDistributedSetsBacktracker(GF(25,'a'),4)
sage: E
4-evenly distributed sets (up to isomorphism) in Finite Field in a of size 5^2
sage: E.cardinality()
4

sage: E = EvenlyDistributedSetsBacktracker(GF(25,'a'), 4, up_to_isomorphism=False)
sage: E.cardinality()
40
```

to_difference_family(*B*, *check=True*)

Given an evenly distributed set `B` convert it to a difference family.

As for any $x \in K^* = H$ we have $|\Delta B \cap xH^{2(q-1)/(k(k-1))}| = 2$ (see `EvenlyDistributedSetsBacktracker`), the difference family is produced as $\{xB : x \in H^{2(q-1)/(k(k-1))}\}$

This method is useful if you want to obtain the difference family from the output of the iterator.

INPUT:

- `B` – an evenly distributed set
- `check` – (boolean, default `True`) whether to check the result

EXAMPLES:

```
sage: from sage.combinat.designs.evenly_distributed_sets import EvenlyDistributedSetsBacktra
sage: E = EvenlyDistributedSetsBacktracker(Zmod(41),5)
sage: B = E.an_element(); B
[0, 1, 13, 38, 31]
sage: D = E.to_difference_family(B); D
[[0, 1, 13, 38, 31], [0, 32, 6, 27, 8]]

sage: from sage.combinat.designs.difference_family import is_difference_family
sage: is_difference_family(Zmod(41),D,41,5,1)
True
```

Setting `check` to `False` is much faster:

```
sage: timeit("df = E.to_difference_family(B, check=True)") # random
625 loops, best of 3: 117 µs per loop

sage: timeit("df = E.to_difference_family(B, check=False)")  # random
625 loops, best of 3: 1.83 µs per loop
```

5.1.77 External Representations of Block Designs

The "ext_rep" module is an API to the abstract tree represented by an XML document containing the External Representation of a list of block designs. The module also provides the related I/O operations for reading/writing ext-rep files or data. The parsing is based on expat.

This is a modified form of the module ext_rep.py (version 0.8) written by Peter Dobcsanyi [D2009] peter@designtheory.org.

REFERENCES:

Todo

The XML data from the designtheory.org database contains a wealth of information about things like automorphism groups, transitivity, cycle type representatives, etc, but none of this data is made available through the current implementation.

Functions

class `sage.combinat.designs.ext_rep.`**`XTree`**(*node*)
Bases: `object`

A lazy class to wrap a rooted tree representing an XML document. The tree's nodes are tuples of the structure:

(name, {dictionary of attributes}, [list of children])

Methods and services of an XTree object `t`:

- `t.attribute` – attribute named
- `t.child` – first child named
- `t[i]` – i-th child
- `for child in t:` – iterate over `t`'s children
- `len(t)` – number of `t`'s children

If child is not an empty subtree, return the subtree as an `XTree` object. If child is an empty subtree, return `_name` of the subtree. Otherwise return the child itself.

The lazy tree idea originated from a utility class of the pyRXP 0.9 package by Robin Becker at ReportLab.

class `sage.combinat.designs.ext_rep.`**`XTreeProcessor`**
Bases: `object`

An incremental event-driven parser for ext-rep documents. The processing stages:

- `<list_of_designs ...>` opening element. call-back: `list_of_designs_proc`
- `<list_definition>` subtree. call-back: `list_definition_proc`
- `<info>` subtree. call-back: `info_proc`
- iterating over `<designs>` processing each `<block_design>` separately. call-back: `block_design_proc`
- finishing with closing `</designs>` and `</list_of_designs>`.

`parse`(*xml_source*)
The main parsing function. Given an XML source (either a file handle or a string), parse the entire XML source.

EXAMPLES:

```
sage: from sage.combinat.designs import ext_rep
sage: file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
sage: proc = ext_rep.XTreeProcessor()
sage: proc.save_designs = True
sage: f = ext_rep.open_extrep_file(file_loc)
sage: proc.parse(f)
sage: f.close()
sage: os.remove(file_loc)
sage: proc.list_of_designs[0]
(2, [[0, 1], [0, 1]])
```

sage.combinat.designs.ext_rep.**check_dtrs_protocols**(*input_name*, *input_pv*)

Check that the XML data is in a valid format. We can currently handle version 2.0. For more information see http://designtheory.org/library/extrep/

EXAMPLES:

```
sage: from sage.combinat.designs import ext_rep
sage: ext_rep.check_dtrs_protocols('source', '2.0')
sage: ext_rep.check_dtrs_protocols('source', '3.0')
Traceback (most recent call last):
...
RuntimeError: Incompatible dtrs_protocols: program: 2.0 source: 3.0
```

sage.combinat.designs.ext_rep.**designs_from_XML**(*fname*)

Return a list of designs contained in an XML file fname. The list contains tuples of the form (v, bs) where v is the number of points of the design and bs is the list of blocks.

EXAMPLES:

```
sage: from sage.combinat.designs import ext_rep
sage: file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
sage: ext_rep.designs_from_XML(file_loc)[0]
(2, [[0, 1], [0, 1]])
sage: os.remove(file_loc)

sage: from sage.combinat.designs import ext_rep
sage: from sage.combinat.designs.block_design import BlockDesign
sage: file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
sage: v, blocks = ext_rep.designs_from_XML(file_loc)[0]
sage: d = BlockDesign(v, blocks)
sage: d.blocks()
[[0, 1], [0, 1]]
sage: d.is_t_design(t=2)
True
sage: d.is_t_design(return_parameters=True)
(True, (2, 2, 2, 2))
```

sage.combinat.designs.ext_rep.**designs_from_XML_url**(*url*)

Return a list of designs contained in an XML file named by a URL. The list contains tuples of the form (v, bs) where v is the number of points of the design and bs is the list of blocks.

EXAMPLES:

```
sage: from sage.combinat.designs import ext_rep
sage: file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
sage: ext_rep.designs_from_XML_url("file://" + file_loc)[0]
(2, [[0, 1], [0, 1]])
sage: os.remove(file_loc)
```

```
sage: from sage.combinat.designs import ext_rep
sage: ext_rep.designs_from_XML_url("http://designtheory.org/database/v-b-k/v3-b6-k2.icgsa.txt.bz
[(3, [[0, 1], [0, 1], [0, 1], [0, 1], [0, 1], [0, 2]]),
 (3, [[0, 1], [0, 1], [0, 1], [0, 1], [0, 2], [0, 2]]),
 (3, [[0, 1], [0, 1], [0, 1], [0, 1], [0, 2], [1, 2]]),
 (3, [[0, 1], [0, 1], [0, 1], [0, 2], [0, 2], [0, 2]]),
 (3, [[0, 1], [0, 1], [0, 1], [0, 2], [0, 2], [1, 2]]),
 (3, [[0, 1], [0, 1], [0, 2], [0, 2], [1, 2], [1, 2]])]
```

sage.combinat.designs.ext_rep.**dump_to_tmpfile**(*s*)
Utility function to dump a string to a temporary file.

EXAMPLE:

```
sage: from sage.combinat.designs import ext_rep
sage: file_loc = ext_rep.dump_to_tmpfile("boo")
sage: os.remove(file_loc)
```

sage.combinat.designs.ext_rep.**open_extrep_file**(*fname*)
Try to guess the compression type from extension and open the extrep file.

EXAMPLES:

```
sage: from sage.combinat.designs import ext_rep
sage: file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
sage: proc = ext_rep.XTreeProcessor()
sage: f = ext_rep.open_extrep_file(file_loc)
sage: proc.parse(f)
sage: f.close()
sage: os.remove(file_loc)
```

sage.combinat.designs.ext_rep.**open_extrep_url**(*url*)
Try to guess the compression type from extension and open the extrep file pointed to by the url. This function (unlike open_extrep_file) returns the uncompressed text contained in the file.

EXAMPLES:

```
sage: from sage.combinat.designs import ext_rep
sage: file_loc = ext_rep.dump_to_tmpfile(ext_rep.v2_b2_k2_icgsa)
sage: proc = ext_rep.XTreeProcessor()
sage: s = ext_rep.open_extrep_url("file://" + file_loc)
sage: proc.parse(s)
sage: os.remove(file_loc)

sage: from sage.combinat.designs import ext_rep
sage: s = ext_rep.designs_from_XML_url("http://designtheory.org/database/v-b-k/v3-b6-k2.icgsa.tx
```

5.1.78 Incidence structures (i.e. hypergraphs, i.e. set systems)

An incidence structure is specified by a list of points, blocks, or an incidence matrix ([3], [4]). `IncidenceStructure` instances have the following methods:

[3] Block designs and incidence structures from wikipedia, Wikipedia article Block_design Wikipedia article Incidence_structure
[4]

5. Assmus, J. Key, Designs and their codes, CUP, 1992.

automorphism_group()	Return the subgroup of the automorphism group of the incidence graph which respects the P B partition. It is (isomorphic to) the automorphism group of the block design, although the degrees differ.
block_sizes()	Return the set of block sizes.
blocks()	Return the list of blocks.
canonical_label()	Return a canonical label for the incidence structure.
coloring()	Compute a (weak) k-coloring of the hypergraph
complement()	Return the complement of the incidence structure.
copy()	Return a copy of the incidence structure.
degree()	Return the degree of a point `p` (or a set of points).
degrees()	Return the degree of all sets of given size, or the degree of all points.
dual()	Return the dual of the incidence structure.
edge_coloring()	Compute a proper edge-coloring.
ground_set()	Return the ground set (i.e the list of points).
incidence_graph()	Return the incidence graph of the incidence structure
incidence_matrix()	Return the incidence matrix A of the design. A is a $(v \times b)$ matrix defined by: `A[i,j] = 1` if `i` is in block `B_j` and 0 otherwise.
induced_substructure()	Return the substructure induced by a set of points.
intersection_graph()	Return the intersection graph of the incidence structure.
is_connected()	Test whether the design is connected.
is_generalized_quadrangle()	Test if the incidence structure is a generalized quadrangle.
is_isomorphic()	Return whether the two incidence structures are isomorphic.
is_regular()	Test whether the incidence structure is r-regular.
is_resolvable()	Test whether the hypergraph is resolvable
is_simple()	Test whether this design is simple (i.e. no repeated block).
is_t_design()	Test whether `self` is a $t-(v,k,l)$ design.
is_uniform()	Test whether the incidence structure is k-uniform
isomorphic_substructures_iterator()	Iterates over all copies of `H2` contained in `self`.
num_blocks()	Return the number of blocks.
num_points()	Return the size of the ground set.
packing()	Return a maximum packing
relabel()	Relabel the ground set
trace()	Return the trace of a set of points.

REFERENCES:

AUTHORS:

- Peter Dobcsanyi and David Joyner (2007-2008)

  This is a significantly modified form of part of the module block_design.py (version 0.6) written by Peter Dobcsanyi peter@designtheory.org.

- Vincent Delecroix (2014): major rewrite

Methods

class sage.combinat.designs.incidence_structures.**IncidenceStructure**(*points=None*, *blocks=None*, *incidence_matrix=None*, *name=None*, *check=True*, *copy=True*)

Bases: object

A base class for incidence structures (i.e. hypergraphs, i.e. set systems)

An incidence structure (i.e. hypergraph, i.e. set system) can be defined from a collection of blocks (i.e. sets, i.e. edges), optionally with an explicit ground set (i.e. point set, i.e. vertex set). Alternatively they can be defined from a binary incidence matrix.

INPUT:

- `points` – (i.e. ground set, i.e. vertex set) the underlying set. If `points` is an integer v, then the set is considered to be $\{0, ..., v-1\}$.

Note: The following syntax, where `points` is ommitted, automatically defines the ground set as the union of the blocks:

```
sage: H = IncidenceStructure([['a','b','c'],['c','d','e']])
sage: H.ground_set()
['a', 'b', 'c', 'd', 'e']
```

- `blocks` – (i.e. edges, i.e. sets) the blocks defining the incidence structure. Can be any iterable.
- `incidence_matrix` – a binary incidence matrix. Each column represents a set.
- `name` (a string, such as "Fano plane").
- `check` – whether to check the input
- `copy` – (use with caution) if set to `False` then `blocks` must be a list of lists of integers. The list will not be copied but will be modified in place (each block is sorted, and the whole list is sorted). Your `blocks` object will become the `IncidenceStructure` instance's internal data.

EXAMPLES:

An incidence structure can be constructed by giving the number of points and the list of blocks:

```
sage: IncidenceStructure(7, [[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]])
Incidence structure with 7 points and 7 blocks
```

Only providing the set of blocks is sufficient. In this case, the ground set is defined as the union of the blocks:

```
sage: IncidenceStructure([[1,2,3],[2,3,4]])
Incidence structure with 4 points and 2 blocks
```

Or by its adjacency matrix (a $\{0,1\}$-matrix in which rows are indexed by points and columns by blocks):

```
sage: m = matrix([[0,1,0],[0,0,1],[1,0,1],[1,1,1]])
sage: IncidenceStructure(m)
Incidence structure with 4 points and 3 blocks
```

The points can be any (hashable) object:

```
sage: V = [(0,'a'),(0,'b'),(1,'a'),(1,'b')]
sage: B = [(V[0],V[1],V[2]), (V[1],V[2]), (V[0],V[2])]
sage: I = IncidenceStructure(V, B)
sage: I.ground_set()
[(0, 'a'), (0, 'b'), (1, 'a'), (1, 'b')]
sage: I.blocks()
[[(0, 'a'), (0, 'b'), (1, 'a')], [(0, 'a'), (1, 'a')], [(0, 'b'), (1, 'a')]]
```

The order of the points and blocks does not matter as they are sorted on input (see trac ticket #11333):

```
sage: A = IncidenceStructure([0,1,2], [[0],[0,2]])
sage: B = IncidenceStructure([1,0,2], [[0],[2,0]])
sage: B == A
```

```
True

sage: C = BlockDesign(2, [[0], [1,0]])
sage: D = BlockDesign(2, [[0,1], [0]])
sage: C == D
True
```

If you care for speed, you can set `copy` to `False`, but in that case, your input must be a list of lists and the ground set must be $0, ..., v-1$:

```
sage: blocks = [[0,1],[2,0],[1,2]]  # a list of lists of integers
sage: I = IncidenceStructure(3, blocks, copy=False)
sage: I._blocks is blocks
True
```

automorphism_group()

Return the subgroup of the automorphism group of the incidence graph which respects the P B partition. It is (isomorphic to) the automorphism group of the block design, although the degrees differ.

EXAMPLES:

```
sage: P = designs.DesarguesianProjectivePlaneDesign(2); P
(7,3,1)-Balanced Incomplete Block Design
sage: G = P.automorphism_group()
sage: G.is_isomorphic(PGL(3,2))
True
sage: G
Permutation Group with generators [...]
sage: G.cardinality()
168
```

A non self-dual example:

```
sage: IS = IncidenceStructure(range(4), [[0,1,2,3],[1,2,3]])
sage: IS.automorphism_group().cardinality()
6
sage: IS.dual().automorphism_group().cardinality()
1
```

Examples with non-integer points:

```
sage: I = IncidenceStructure('abc', ('ab','ac','bc'))
sage: I.automorphism_group()
Permutation Group with generators [('b','c'), ('a','b')]
sage: IncidenceStructure([[(1,2),(3,4)]]).automorphism_group()
Permutation Group with generators [((1,2),(3,4))]
```

block_sizes()

Return the set of block sizes.

EXAMPLES:

```
sage: BD = IncidenceStructure(8, [[0,1,3],[1,4,5,6],[1,2],[5,6,7]])
sage: BD.block_sizes()
[3, 2, 4, 3]
sage: BD = IncidenceStructure(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]])
sage: BD.block_sizes()
[3, 3, 3, 3, 3, 3, 3]
```

blocks()

Return the list of blocks.

EXAMPLES:

```
sage: BD = IncidenceStructure(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]])
sage: BD.blocks()
[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6], [2, 3, 6], [2, 4, 5]]
```

canonical_label()

Return a canonical label for the incidence structure.

A canonical label is relabeling of the points into integers $\{0, ..., n-1\}$ such that isomorphic incidence structures are relabelled to equal objects.

EXAMPLE:

```
sage: fano1 = designs.balanced_incomplete_block_design(7,3)
sage: fano2 = designs.projective_plane(2)
sage: fano1 == fano2
False
sage: fano1.relabel(fano1.canonical_label())
sage: fano2.relabel(fano2.canonical_label())
sage: fano1 == fano2
True
```

coloring(*k=None*, *solver=None*, *verbose=0*)

Compute a (weak) k-coloring of the hypergraph

A weak coloring of a hypergraph $\mathcal{H}$ is an assignment of colors to its vertices such that no set is monochromatic.

INPUT:

- `k` (integer) – compute a coloring with k colors if an integer is provided, otherwise returns an optimal coloring (i.e. with the minimum possible number of colors).
- `solver` – (default: `None`) Specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class `MixedIntegerLinearProgram`.
- `verbose` – non-negative integer (default: `0`). Set the level of verbosity you want from the linear program solver. Since the problem is NP-complete, its solving may take some time depending on the graph. A value of 0 means that there will be no message printed by the solver.

EXAMPLES:

The Fano plane has chromatic number 3:

```
sage: len(designs.steiner_triple_system(7).coloring())
3
```

One admissible 3-coloring:

```
sage: designs.steiner_triple_system(7).coloring() # not tested - architecture-dependent
[[0, 2, 5, 1], [4, 3], [6]]
```

The chromatic number of a graph is equal to the chromatic number of its 2-uniform corresponding hypergraph:

```
sage: g = graphs.PetersenGraph()
sage: H = IncidenceStructure(g.edges(labels=False))
sage: len(g.coloring())
3
sage: len(H.coloring())
3
```

complement(*uniform=False*)

Return the complement of the incidence structure.

Two different definitions of "complement" are made available, according to the value of `uniform`.

INPUT:

- `uniform` (boolean) –
  - if set to `False` (default), returns the incidence structure whose blocks are the complements of all blocks of the incidence structure.
  - If set to `True` and the incidence structure is k-uniform, returns the incidence structure whose blocks are all k-sets of the ground set that do not appear in `self`.

EXAMPLES:

The complement of a `BalancedIncompleteBlockDesign` is also a 2-design:

```
sage: bibd = designs.balanced_incomplete_block_design(13,4)
sage: bibd.is_t_design(return_parameters=True)
(True, (2, 13, 4, 1))
sage: bibd.complement().is_t_design(return_parameters=True)
(True, (2, 13, 9, 6))
```

The "uniform" complement of a graph is a graph:

```
sage: g = graphs.PetersenGraph()
sage: G = IncidenceStructure(g.edges(labels=False))
sage: H = G.complement(uniform=True)
sage: h = Graph(H.blocks())
sage: g == h
False
sage: g == h.complement()
True
```

TESTS:

```
sage: bibd.relabel({i:str(i) for i in bibd.ground_set()})
sage: bibd.complement().ground_set()
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12']

sage: I = IncidenceStructure('abc', ['ab','ac','bc'])
sage: I.is_t_design(return_parameters=True)
(True, (2, 3, 2, 1))
```

copy()

Return a copy of the incidence structure.

EXAMPLE:

```
sage: IS = IncidenceStructure([[1,2,3,"e"]],name="Test")
sage: IS
Incidence structure with 4 points and 1 blocks
sage: copy(IS)
Incidence structure with 4 points and 1 blocks
sage: [1, 2, 3, 'e'] in copy(IS)
True
sage: copy(IS)._name
'Test'
```

degree(*p=None*, *subset=False*)

Return the degree of a point `p` (or a set of points).

The degree of a point (or set of points) is the number of blocks that contain it.

INPUT:

- `p` – a point (or a set of points) of the incidence structure.
- `subset` (boolean) – whether to interpret the argument as a set of point (`subset=True`) or as a point (`subset=False`, default).

EXAMPLES:

```
sage: designs.steiner_triple_system(9).degree(3)
4
sage: designs.steiner_triple_system(9).degree({1,2},subset=True)
1
```

TESTS:

```
sage: designs.steiner_triple_system(9).degree()
doctest:...: DeprecationWarning: Please use degrees() instead of degree(None)
See http://trac.sagemath.org/17108 for details.
{0: 4, 1: 4, 2: 4, 3: 4, 4: 4, 5: 4, 6: 4, 7: 4, 8: 4}
sage: designs.steiner_triple_system(9).degree(subset=True)
Traceback (most recent call last):
...
ValueError: subset must be False when p is None
```

degrees (*size=None*)

Return the degree of all sets of given size, or the degree of all points.

The degree of a point (or set of point) is the number of blocks that contain it.

INPUT:

- `size` (integer) – return the degree of all subsets of points of cardinality `size`. When `size=None`, the function outputs the degree of all points.

Note: When `size=None` the output is indexed by the points. When `size=1` it is indexed by tuples of size 1. This is the same information, stored slightly differently.

OUTPUT:

A dictionary whose values are degrees and keys are either:

- the points of the incidence structure if `size=None` (default)
- the subsets of size `size` of the points stored as tuples

EXAMPLES:

```
sage: IncidenceStructure([[1,2,3],[1,4]]).degrees(2)
{(1, 2): 1, (1, 3): 1, (1, 4): 1, (2, 3): 1, (2, 4): 0, (3, 4): 0}
```

In a Steiner triple system, all pairs have degree 1:

```
sage: S13 = designs.steiner_triple_system(13)
sage: all(v == 1 for v in S13.degrees(2).itervalues())
True
```

dual (*algorithm=None*)

Return the dual of the incidence structure.

INPUT:

- `algorithm` – whether to use Sage's implementation (`algorithm=None`, default) or use GAP's (`algorithm="gap"`).

Note: The `algorithm="gap"` option requires GAP's Design package (included in the gap_packages Sage spkg).

EXAMPLES:

The dual of a projective plane is a projective plane:

```
sage: PP = designs.DesarguesianProjectivePlaneDesign(4)
sage: PP.dual().is_t_design(return_parameters=True)
(True, (2, 21, 5, 1))
```

TESTS:

```
sage: D = IncidenceStructure(4, [[0,2],[1,2,3],[2,3]])
sage: D
Incidence structure with 4 points and 3 blocks
sage: D.dual()
Incidence structure with 3 points and 4 blocks
sage: print D.dual(algorithm="gap")        # optional - gap_packages
Incidence structure with 3 points and 4 blocks
sage: blocks = [[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]]
sage: BD = IncidenceStructure(7, blocks, name="FanoPlane");
sage: BD
Incidence structure with 7 points and 7 blocks
sage: print BD.dual(algorithm="gap")          # optional - gap_packages
Incidence structure with 7 points and 7 blocks
sage: BD.dual()
Incidence structure with 7 points and 7 blocks
```

REFERENCE:

- Soicher, Leonard, Design package manual, available at http://www.gap-system.org/Manuals/pkg/design/htm/CHAP003.htm

edge_coloring()

Compute a proper edge-coloring.

A proper edge-coloring is an assignment of colors to the sets of the incidence structure such that two sets with non-empty intersection receive different colors. The coloring returned minimizes the number of colors.

OUTPUT:

A partition of the sets into color classes.

EXAMPLES:

```
sage: H = Hypergraph([{1,2,3},{2,3,4},{3,4,5},{4,5,6}]); H
Incidence structure with 6 points and 4 blocks
sage: C = H.edge_coloring()
sage: C # random
[[[3, 4, 5]], [[2, 3, 4]], [[4, 5, 6], [1, 2, 3]]]
sage: Set(map(Set,sum(C,[]))) == Set(map(Set,H.blocks()))
True
```

ground_set()

Return the ground set (i.e the list of points).

EXAMPLES:

```
sage: IncidenceStructure(3, [[0,1],[0,2]]).ground_set()
[0, 1, 2]
```

incidence_graph (*labels=False*)

Return the incidence graph of the incidence structure

A point and a block are adjacent in this graph whenever they are incident.

INPUT:

- `labels` (boolean) – whether to return a graph whose vertices are integers, or labelled elements.
  - `labels is False` (default) – in this case the first vertices of the graphs are the elements of `ground_set()`, and appear in the same order. Similarly, the following vertices represent the elements of `blocks()`, and appear in the same order.
  - `labels is True`, the points keep their original labels, and the blocks are `Set` objects.

    Note that the labelled incidence graph can be incorrect when blocks are repeated, and on some (rare) occasions when the elements of `ground_set()` mix `Set()` and non-`Set` objects.

EXAMPLE:

```
sage: BD = IncidenceStructure(7, [[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]])
sage: BD.incidence_graph()
Bipartite graph on 14 vertices
sage: A = BD.incidence_matrix()
sage: Graph(block_matrix([[A*0,A],[A.transpose(),A*0]])) == BD.incidence_graph()
True
```

TESTS:

With `labels = True`:

```
sage: BD.incidence_graph(labels=True).has_edge(0,Set([0,1,2]))
True
```

incidence_matrix()

Return the incidence matrix A of the design. A is a $(v \times b)$ matrix defined by: `A[i,j] = 1` if `i` is in block `B_j` and 0 otherwise.

EXAMPLES:

```
sage: BD = IncidenceStructure(7, [[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]])
sage: BD.block_sizes()
[3, 3, 3, 3, 3, 3, 3]
sage: BD.incidence_matrix()
[1 1 1 0 0 0 0]
[1 0 0 1 1 0 0]
[1 0 0 0 0 1 1]
[0 1 0 1 0 1 0]
[0 1 0 0 1 0 1]
[0 0 1 1 0 0 1]
[0 0 1 0 1 1 0]
```

```
sage: I = IncidenceStructure('abc', ('ab','abc','ac','c'))
sage: I.incidence_matrix()
[1 1 1 0]
[1 1 0 0]
[0 1 1 1]
```

induced_substructure(*points*)

Return the substructure induced by a set of points.

The substructure induced in $\mathcal{H}$ by a set $X \subseteq V(\mathcal{H})$ of points is the incidence structure $\mathcal{H}_X$ defined on X whose sets are all $S \in \mathcal{H}$ such that $S \subseteq X$.

INPUT:

- `points` – a set of points.

Note: This method goes over all sets of `self` before building a new `IncidenceStructure` (which involves some relabelling and sorting). It probably should not be called in a performance-critical code.

EXAMPLE:

A Fano plane with one point removed:

```
sage: F = designs.steiner_triple_system(7)
sage: F.induced_substructure([0..5])
Incidence structure with 6 points and 4 blocks
```

TESTS:

```
sage: F.induced_substructure([0..50])
Traceback (most recent call last):
...
ValueError: 7 is not a point of the incidence structure
sage: F.relabel(dict(enumerate("abcdefg")))
sage: F.induced_substructure("abc")
Incidence structure with 3 points and ...
sage: F.induced_substructure("Y")
Traceback (most recent call last):
...
ValueError: 'Y' is not a point of the incidence structure
```

intersection_graph(*sizes=None*)

Return the intersection graph of the incidence structure.

The vertices of this graph are the `blocks()` of the incidence structure. Two of them are adjacent if the size of their intersection belongs to the set `sizes`.

INPUT:

- `sizes` – a list/set of integers. For convenience, setting `sizes` to `5` has the same effect as `sizes=[5]`. When set to `None` (default), behaves as `sizes=PositiveIntegers()`.

EXAMPLE:

The intersection graph of a `balanced_incomplete_block_design()` is a `strongly regular graph` (when it is not trivial):

```
sage: BIBD =  designs.balanced_incomplete_block_design(19,3)
sage: G = BIBD.intersection_graph(1)
sage: G.is_strongly_regular(parameters=True)
(57, 24, 11, 9)
```

is_connected()

Test whether the design is connected.

EXAMPLES:

```
sage: IncidenceStructure(3, [[0,1],[0,2]]).is_connected()
True
```

```
sage: IncidenceStructure(4, [[0,1],[2,3]]).is_connected()
False
```

is_generalized_quadrangle(*verbose=False*, *parameters=False*)

Test if the incidence structure is a generalized quadrangle.

An incidence structure is a generalized quadrangle iff (see [BH12], section 9.6):

- two blocks intersect on at most one point.
- For every point p not in a block B, there is a unique block B' intersecting both $\{p\}$ and B

It is a *regular* generalized quadrangle if furthermore:

- it is $s+1$-`uniform` for some positive integer s.
- it is $t+1$-`regular` for some positive integer t.

For more information, see the Wikipedia article Generalized_quadrangle.

Note: Some references (e.g. [PT09] or [GQwiki]) only allow *regular* generalized quadrangles. To use such a definition, see the `parameters` optional argument described below, or the methods `is_regular()` and `is_uniform()`.

INPUT:

- `verbose` (boolean) – whether to print an explanation when the instance is not a generalized quadrangle.
- `parameters` (boolean; `False`) – if set to `True`, the function returns a pair `(s,t)` instead of `True` answers. In this case, s and t are the integers defined above if they exist (each can be set to `False` otherwise).

EXAMPLE:

```
sage: h = designs.CremonaRichmondConfiguration()
sage: h.is_generalized_quadrangle()
True
```

This is actually a *regular* generalized quadrangle:

```
sage: h.is_generalized_quadrangle(parameters=True)
(2, 2)
```

TESTS:

```
sage: H = IncidenceStructure((2*graphs.CompleteGraph(3)).edges(labels=False))
sage: H.is_generalized_quadrangle(verbose=True)
Some point is at distance >3 from some block.
False

sage: G = graphs.CycleGraph(5)
sage: B = list(G.subgraph_search_iterator(graphs.PathGraph(3)))
sage: H = IncidenceStructure(B)
sage: H.is_generalized_quadrangle(verbose=True)
Two blocks intersect on >1 points.
False

sage: hypergraphs.CompleteUniform(4,2).is_generalized_quadrangle(verbose=1)
Some point has two projections on some line.
False
```

is_isomorphic(*other*, *certificate=False*)

Return whether the two incidence structures are isomorphic.

INPUT:

- `other` – an incidence structure.
- `certificate` (boolean) – whether to return an insomorphism from `self` to `other` instead of a boolean answer.

EXAMPLE:

```
sage: fano1 = designs.balanced_incomplete_block_design(7,3)
sage: fano2 = designs.projective_plane(2)
sage: fano1.is_isomorphic(fano2)
True
sage: fano1.is_isomorphic(fano2,certificate=True)
{0: 0, 1: 1, 2: 2, 3: 6, 4: 4, 5: 3, 6: 5}
```

TESTS:

```
sage: IS  = IncidenceStructure([["A",5,pi],["A",5,"Wouhou"],["A","Wouhou",(9,9)],[pi,12]])
sage: IS2 = IS.copy()
sage: IS2.relabel(IS2.canonical_label())
sage: IS.is_isomorphic(IS2)
True
sage: canon = IS.is_isomorphic(IS2,certificate=True)
sage: IS.relabel(canon)
sage: IS==IS2
True

sage: IS2 = IncidenceStructure([[1,2]])
sage: IS2.is_isomorphic(IS)
False
sage: IS2.is_isomorphic(IS,certificate=True)
{}
```

Checking whether two `IncidenceStructure` are isomorphic incidentally computes their canonical label (if necessary). Thus, subsequent calls to `is_isomorphic()` will be faster:

```
sage: IS1 = designs.projective_plane(3)
sage: IS2 = IS1.relabel(Permutations(IS1.ground_set()).random_element(),inplace=False)
sage: IS2 = IncidenceStructure(IS2.blocks())
sage: IS1._canonical_label is None and IS2._canonical_label is None
True
sage: IS1.is_isomorphic(IS2)
True
sage: IS1._canonical_label is None or IS2._canonical_label is None
False
```

is_regular(*r=None*)

Test whether the incidence structure is r-regular.

An incidence structure is said to be r-regular if all its points are incident with exactly r blocks.

INPUT:

- `r` (integer)

OUTPUT:

If `r` is defined, a boolean is returned. If `r` is set to `None` (default), the method returns either `False` or the integer `r` such that the incidence structure is r-regular.

Warning: In case of 0-regular incidence structure, beware that `if not H.is_regular()` is a satisfied condition.

EXAMPLES:

```
sage: designs.balanced_incomplete_block_design(7,3).is_regular()
3
sage: designs.balanced_incomplete_block_design(7,3).is_regular(r=3)
True
sage: designs.balanced_incomplete_block_design(7,3).is_regular(r=4)
False
```

TESTS:

```
sage: IncidenceStructure([]).is_regular()
Traceback (most recent call last):
...
ValueError: This incidence structure has no points.
```

is_resolvable(*certificate=False*, *solver=None*, *verbose=0*, *check=True*)

Test whether the hypergraph is resolvable

A hypergraph is said to be resolvable if its sets can be partitionned into classes, each of which is a partition of the ground set.

Note: This problem is solved using an Integer Linear Program, and GLPK (the default LP solver) has been reported to be very slow on some instances. If you hit this wall, consider installing a more powerful LP solver (CPLEX, Gurobi, ...).

INPUT:

- `certificate` (boolean) – whether to return the classes along with the binary answer (see examples below).
- `solver` – (default: `None`) Specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`). Sets the level of verbosity. Set to 0 by default, which means quiet.
- `check` (boolean) – whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.

EXAMPLES:

Some resolvable designs:

```
sage: TD = designs.transversal_design(2,2,resolvable=True)
sage: TD.is_resolvable()
True

sage: AG = designs.AffineGeometryDesign(3,1,GF(2))
sage: AG.is_resolvable()
True
```

Their classes:

```
sage: b,cls = TD.is_resolvable(True)
sage: b
True
```

```
sage: cls # random
[[[0, 3], [1, 2]], [[1, 3], [0, 2]]]

sage: b,cls = AG.is_resolvable(True)
sage: b
True
sage: cls # random
[[[6, 7], [4, 5], [0, 1], [2, 3]],
 [[5, 7], [0, 4], [3, 6], [1, 2]],
 [[0, 2], [4, 7], [1, 3], [5, 6]],
 [[3, 4], [0, 7], [1, 5], [2, 6]],
 [[3, 7], [1, 6], [0, 5], [2, 4]],
 [[0, 6], [2, 7], [1, 4], [3, 5]],
 [[4, 6], [0, 3], [2, 5], [1, 7]]]
```

A non-resolvable design:

```
sage: Fano = designs.balanced_incomplete_block_design(7,3)
sage: Fano.is_resolvable()
False
sage: Fano.is_resolvable(True)
(False, [])
```

TESTS:

```
sage: _,cls1 = AG.is_resolvable(certificate=True)
sage: _,cls2 = AG.is_resolvable(certificate=True)
sage: cls1 is cls2
False
```

is_simple()

Test whether this design is simple (i.e. no repeated block).

EXAMPLES:

```
sage: IncidenceStructure(3, [[0,1],[1,2],[0,2]]).is_simple()
True
sage: IncidenceStructure(3, [[0],[0]]).is_simple()
False

sage: V = [(0,'a'),(0,'b'),(1,'a'),(1,'b')]
sage: B = [[V[0],V[1]], [V[1],V[2]]]
sage: I = IncidenceStructure(V, B)
sage: I.is_simple()
True
sage: I2 = IncidenceStructure(V, B*2)
sage: I2.is_simple()
False
```

is_t_design(*t=None*, *v=None*, *k=None*, *l=None*, *return_parameters=False*)

Test whether `self` is a $t-(v,k,l)$ design.

A $t-(v,k,\lambda)$ (sometimes called t-design for short) is a block design in which:

- the underlying set has cardinality v
- the blocks have size k
- each t-subset of points is covered by λ blocks

INPUT:

- `t,v,k,l` (integers) – their value is set to `None` by default. The function tests whether the design is a `t-(v,k,l)` design using the provided values and guesses the others. Note that 'l' cannot be specified if `t` is not.
- `return_parameters` (boolean)– whether to return the parameters of the t-design. If set to `True`, the function returns a pair `(boolean_answer,(t,v,k,l))`.

EXAMPLES:

```
sage: fano_blocks = [[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]]
sage: BD = IncidenceStructure(7, fano_blocks)
sage: BD.is_t_design()
True
sage: BD.is_t_design(return_parameters=True)
(True, (2, 7, 3, 1))
sage: BD.is_t_design(2, 7, 3, 1)
True
sage: BD.is_t_design(1, 7, 3, 3)
True
sage: BD.is_t_design(0, 7, 3, 7)
True

sage: BD.is_t_design(0,6,3,7) or BD.is_t_design(0,7,4,7) or BD.is_t_design(0,7,3,8)
False

sage: BD = designs.AffineGeometryDesign(3, 1, GF(2))
sage: BD.is_t_design(1)
True
sage: BD.is_t_design(2)
True
```

Steiner triple and quadruple systems are other names for $2-(v,3,1)$ and $3-(v,4,1)$ designs:

```
sage: S3_9 = designs.steiner_triple_system(9)
sage: S3_9.is_t_design(2,9,3,1)
True

sage: blocks = designs.steiner_quadruple_system(8)
sage: S4_8 = IncidenceStructure(8, blocks)
sage: S4_8.is_t_design(3,8,4,1)
True

sage: blocks = designs.steiner_quadruple_system(14)
sage: S4_14 = IncidenceStructure(14, blocks)
sage: S4_14.is_t_design(3,14,4,1)
True
```

Some examples of Witt designs that need the gap database:

```
sage: BD = designs.WittDesign(9)          # optional - gap_packages
sage: BD.is_t_design(2,9,3,1)             # optional - gap_packages
True
sage: W12 = designs.WittDesign(12)        # optional - gap_packages
sage: W12.is_t_design(5,12,6,1)           # optional - gap_packages
True
sage: W12.is_t_design(4)                  # optional - gap_packages
True
```

Further examples:

```
sage: D = IncidenceStructure(4,[[],[]])
sage: D.is_t_design(return_parameters=True)
(True,  (0, 4, 0, 2))

sage: D = IncidenceStructure(4, [[0,1],[0,2],[0,3]])
sage: D.is_t_design(return_parameters=True)
(True, (0, 4, 2, 3))

sage: D = IncidenceStructure(4, [[0],[1],[2],[3]])
sage: D.is_t_design(return_parameters=True)
(True, (1, 4, 1, 1))

sage: D = IncidenceStructure(4,[[0,1],[2,3]])
sage: D.is_t_design(return_parameters=True)
(True, (1, 4, 2, 1))

sage: D = IncidenceStructure(4, [range(4)])
sage: D.is_t_design(return_parameters=True)
(True, (4, 4, 4, 1))
```

TESTS:

```
sage: blocks = designs.steiner_quadruple_system(8)
sage: S4_8 = IncidenceStructure(8, blocks)
sage: R = range(15)
sage: [(v,k,l) for v in R for k in R for l in R if S4_8.is_t_design(3,v,k,l)]
[(8, 4, 1)]
sage: [(v,k,l) for v in R for k in R for l in R if S4_8.is_t_design(2,v,k,l)]
[(8, 4, 3)]
sage: [(v,k,l) for v in R for k in R for l in R if S4_8.is_t_design(1,v,k,l)]
[(8, 4, 7)]
sage: [(v,k,l) for v in R for k in R for l in R if S4_8.is_t_design(0,v,k,l)]
[(8, 4, 14)]
sage: A = designs.AffineGeometryDesign(3, 1, GF(2))
sage: A.is_t_design(return_parameters=True)
(True, (2, 8, 2, 1))
sage: A = designs.AffineGeometryDesign(4, 2, GF(2))
sage: A.is_t_design(return_parameters=True)
(True, (3, 16, 4, 1))
sage: I = IncidenceStructure(2, [])
sage: I.is_t_design(return_parameters=True)
(True, (0, 2, 0, 0))
sage: I = IncidenceStructure(2, [[0],[0,1]])
sage: I.is_t_design(return_parameters=True)
(False, (0, 0, 0, 0))
```

is_uniform(*k=None*)

Test whether the incidence structure is k-uniform

An incidence structure is said to be k-uniform if all its blocks have size k.

INPUT:

- k (integer)

OUTPUT:

If k is defined, a boolean is returned. If k is set to None (default), the method returns either False or the integer k such that the incidence structure is r-regular.

Warning: In case of 0-uniform incidence structure, beware that `if not H.is_uniform()` is a satisfied condition.

EXAMPLES:

```
sage: designs.balanced_incomplete_block_design(7,3).is_uniform()
3
sage: designs.balanced_incomplete_block_design(7,3).is_uniform(k=3)
True
sage: designs.balanced_incomplete_block_design(7,3).is_uniform(k=4)
False
```

TESTS:

```
sage: IncidenceStructure([]).is_regular()
Traceback (most recent call last):
...
ValueError: This incidence structure has no points.
```

isomorphic_substructures_iterator(*H2*, *induced=False*)

Iterates over all copies of `H2` contained in `self`.

A hypergraph H_1 contains an isomorphic copy of a hypergraph H_2 if there exists an injection $f : V(H_2) \mapsto V(H_1)$ such that for any set $S_2 \in E(H_2)$ the set $S_1 = f(S2)$ belongs to $E(H_1)$.

It is an *induced* copy if no other set of $E(H_1)$ is contained in $f(V(H_2))$, i.e. $|E(H_2)| = \{S : S \in E(H_1)$ and $f(V(H_2))\}$.

This function lists all such injections. In particular, the number of copies of H in itself is equal to *the size of its automorphism group*.

See `subhypergraph_search` for more information.

INPUT:

- `H2` an `IncidenceStructure` object.
- `induced` (boolean) – whether to require the copies to be induced. Set to `False` by default.

EXAMPLES:

How many distinct C_5 in Petersen's graph ?

```
sage: P = graphs.PetersenGraph()
sage: C = graphs.CycleGraph(5)
sage: IP = IncidenceStructure(P.edges(labels=False))
sage: IC = IncidenceStructure(C.edges(labels=False))
sage: sum(1 for _ in IP.isomorphic_substructures_iterator(IC))
120
```

As the automorphism group of C_5 has size 10, the number of distinct unlabelled copies is 12. Let us check that all functions returned correspond to an actual C_5 subgraph:

```
sage: for f in IP.isomorphic_substructures_iterator(IC):
....:     assert all(P.has_edge(f[x],f[y]) for x,y in C.edges(labels=False))
```

The number of induced copies, in this case, is the same:

```
sage: sum(1 for _ in IP.isomorphic_substructures_iterator(IC,induced=True))
120
```

They begin to differ if we make one vertex universal:

```
sage: P.add_edges([(0,x) for x in P])
sage: IP = IncidenceStructure(P.edges(labels=False))
sage: IC = IncidenceStructure(C.edges(labels=False))
sage: sum(1 for _ in IP.isomorphic_substructures_iterator(IC))
420
sage: sum(1 for _ in IP.isomorphic_substructures_iterator(IC,induced=True))
60
```

The number of copies of H in itself is the size of its automorphism group:

```
sage: H = designs.projective_plane(3)
sage: sum(1 for _ in H.isomorphic_substructures_iterator(H))
5616
sage: H.automorphism_group().cardinality()
5616
```

num_blocks()

Return the number of blocks.

EXAMPLES:

```
sage: designs.DesarguesianProjectivePlaneDesign(2).num_blocks()
7
sage: B = IncidenceStructure(4, [[0,1],[0,2],[0,3],[1,2], [1,2,3]])
sage: B.num_blocks()
5
```

num_points()

Return the size of the ground set.

EXAMPLES:

```
sage: designs.DesarguesianProjectivePlaneDesign(2).num_points()
7
sage: B = IncidenceStructure(4, [[0,1],[0,2],[0,3],[1,2], [1,2,3]])
sage: B.num_points()
4
```

packing(*solver=None*, *verbose=0*)

Return a maximum packing

A maximum packing in a hypergraph is collection of disjoint sets/blocks of maximal cardinality. This problem is NP-complete in general, and in particular on 3-uniform hypergraphs. It is solved here with an Integer Linear Program.

For more information, see the Wikipedia article Packing_in_a_hypergraph.

INPUT:

- `solver` – (default: `None`) Specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`). Sets the level of verbosity. Set to 0 by default, which means quiet.

EXAMPLE:

```
sage; IncidenceStructure([[1,2],[3,"A"],[2,3]]).packing()
[[1, 2], [3, 'A']]
sage: len(designs.steiner_triple_system(9).packing())
3
```

relabel (*perm=None*, *inplace=True*)

Relabel the ground set

INPUT:

- `perm` – can be one of
  - a dictionary – then each point `p` (which should be a key of `d`) is relabeled to `d[p]`
  - a list or a tuple of length `n` – the first point returned by `ground_set()` is relabeled to `l[0]`, the second to `l[1]`, ...
  - `None` – the incidence structure is relabeled to be on $\{0, 1, ..., n-1\}$ in the ordering given by `ground_set()`.
- `inplace` – If `True` then return a relabeled graph and does not touch `self` (default is `False`).

EXAMPLES:

```
sage: TD=designs.transversal_design(5,5)
sage: TD.relabel({i:chr(97+i) for i in range(25)})
sage: print TD.ground_set()
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', '
sage: print TD.blocks()[:3]
[['a', 'f', 'k', 'p', 'u'], ['a', 'g', 'm', 's', 'y'], ['a', 'h', 'o', 'q', 'x']]
```

Relabel to integer points:

```
sage: TD.relabel()
sage: print TD.blocks()[:3]
[[0, 5, 10, 15, 20], [0, 6, 12, 18, 24], [0, 7, 14, 16, 23]]
```

TESTS:

Check that the relabel is consistent on a fixed incidence structure:

```
sage: I = IncidenceStructure([0,1,2,3,4],
....:                [[0,1,3],[0,2,4],[2,3,4],[0,1]])
sage: I.relabel()
sage: from itertools import permutations
sage: for p in permutations([0,1,2,3,4]):
....:     J = I.relabel(p,inplace=False)
....:     if I == J: print p
(0, 1, 2, 3, 4)
(0, 1, 4, 3, 2)
```

And one can also verify that we have exactly two automorphisms:

```
sage: I.automorphism_group()
Permutation Group with generators [(2,4)]
```

trace (*points*, *min_size=1*, *multiset=True*)

Return the trace of a set of points.

Given an hypergraph $\mathcal{H}$, the *trace* of a set X of points in $\mathcal{H}$ is the hypergraph whose blocks are all non-empty $S \cap X$ where $S \in \mathcal{H}$.

INPUT:

- `points` – a set of points.
- `min_size` (integer; default 1) – minimum size of the sets to keep. By default all empty sets are discarded, i.e. `min_size=1`.

- `multiset` (boolean; default `True`) – whether to keep multiple copies of the same set.

Note: This method goes over all sets of `self` before building a new `IncidenceStructure` (which involves some relabelling and sorting). It probably should not be called in a performance-critical code.

EXAMPLE:

A Baer subplane of order 2 (i.e. a Fano plane) in a projective plane of order 4:

```
sage: P4 = designs.projective_plane(4)
sage: F = designs.projective_plane(2)
sage: for x in Subsets(P4.ground_set(),7):
....:     if P4.trace(x,min_size=2).is_isomorphic(F):
....:         break
sage: subplane = P4.trace(x,min_size=2); subplane
Incidence structure with 7 points and 7 blocks
sage: subplane.is_isomorphic(F)
True
```

TESTS:

```
sage: F.trace([0..50])
Traceback (most recent call last):
...
ValueError: 7 is not a point of the incidence structure
sage: F.relabel(dict(enumerate("abcdefg")))
sage: F.trace("abc")
Incidence structure with 3 points and ...
sage: F.trace("Y")
Traceback (most recent call last):
...
ValueError: 'Y' is not a point of the incidence structure
```

5.1.79 Mutually Orthogonal Latin Squares (MOLS)

The main function of this module is `mutually_orthogonal_latin_squares()` and can be can be used to generate MOLS (or check that they exist):

```
sage: MOLS = designs.mutually_orthogonal_latin_squares(4,8)
```

For more information on MOLS, see the Wikipedia entry on MOLS. If you are only interested by latin squares, see `latin`.

The functions defined here are

`mutually_orthogonal_latin_squares()`	Return k Mutually Orthogonal $n \times n$ Latin Squares.
`are_mutually_orthogonal_latin_squares()`	Check that the list `l` of matrices in are MOLS.
`latin_square_product()`	Return the product of two (or more) latin squares.
`MOLS_table()`	Prints the MOLS table.

Table of MOLS

Sage can produce a table of MOLS similar to the one from the Handbook of Combinatorial Designs [DesignHandbook] (available here).

```
sage: from sage.combinat.designs.latin_squares import MOLS_table
sage: MOLS_table(600) # long time
       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
```

```
  0| +oo +oo   1   2   3   4   1   6   7   8   2  10   5  12   4   4  15  16   5  18
 20|   4   5   3  22   7  24   4  26   5  28   4  30  31   5   4   5   8  36   4   5
 40|   7  40   5  42   5   6   4  46   8  48   6   5   5  52   5   6   7   7   5  58
 60|   5  60   5   6  63   7   5  66   5   6   6  70   7  72   5   7   6   6   6  78
 80|   9  80   8  82   6   6   6   6   7  88   6   7   6   6   6   6   7  96   6   8
100|   8 100   6 102   7   7   6 106   6 108   6   6  13 112   6   7   6   8   6   6
120|   7 120   6   6   6 124   6 126 127   7   6 130   6   7   6   7   7 136   6 138
140|   6   7   6  10  10   7   6   7   6 148   6 150   7   8   8   7   6 156   7   6
160|   9   7   6 162   6   7   6 166   7 168   6   8   6 172   6   6  14   9   6 178
180|   6 180   6   6   7   9   6  10   6   8   6 190   7 192   6   7   6 196   6 198
200|   7   7   6   7   6   8   6   8  14  11  10 210   6   7   6   7   7   8   6  10
220|   6  12   6 222  13   8   6 226   6 228   6   7   7 232   6   7   6   7   6 238
240|   7 240   6 242   6   7   6  12   7   7   6 250   6  12   9   7 255 256   6  12
260|   6   8   8 262   7   8   7  10   7 268   7 270  15  16   6  13  10 276   6   9
280|   7 280   6 282   6  12   6   7  15 288   6   6   6 292   6   6   7  10  10  12
300|   7   7   7   7  15  15   6 306   7   7   7 310   7 312   7  10   7 316   7  10
320|  15  15   6  16   8  12   6   7   7   9   6 330   7   8   7   6   7 336   6   7
340|   6  10  10 342   7   7   6 346   6 348   8  12  18 352   6   9   7   9   6 358
360|   7 360   6   7   7   7   6 366  15  15   7  15   7 372   7  15   7  13   7 378
380|   7  12   7 382  15  15   7  15   7 388   7  16   7   7   7   7   8 396   7   7
400|  15 400   7  15  11   8   7  15   8 408   7  13   8  12  10   9  18  15   7 418
420|   7 420   7  15   7  16   6   7   7   7   6 430  15 432   6  15   6  18   7 438
440|   7  15   7 442   7  13   7  11  15 448   7  15   7   7   7  15   7 456   7  16
460|   7 460   7 462  15  15   7 466   8   8   7  15   7  15  10  18   7  15   6 478
480|  15  15   6  15   8   7   6 486   7  15   6 490   6  16   6   7  15  15   6 498
500|   7   8   9 502   7  15   6  15   7 508   6  15 511  18   7  15   8  12   8  15
520|   8 520  10 522  12  15   8  16  15 528   7  15   8  12   7  15   8  15  10  15
540|  12 540   7  15  18   7   7 546   7   8   7  18   7   7   7   7   7 556   7  12
560|  15   7   7 562   7   7   6   7   7 568   6 570   7   7  15  22   8 576   7   7
580|   7   8   7  10   7   8   7 586   7  18  17   7  15 592   8  15   7   7   8 598
```

Comparison with the results from the Handbook of Combinatorial Designs (2ed) [DesignHandbook]:

```
sage: MOLS_table(600,compare=True) # long time
        0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
    ------------------------------------------------------------------------------------
  0|                                                                   +               +
 20|
 40|
 60|   +
 80|
100|
120|
140|
160|
180|
200|       -
220|
240|
260|
280|
300|
320|                                                                               -
340|
360|   -                   -
380|                                                           -
400|
420|                                       -
```

```
440|
460|
480|
500|       -
520|
540|
560|
580|
```

TODO:

- Look at [ColDin01].

REFERENCES:

Functions

sage.combinat.designs.latin_squares.**MOLS_table**(*start*, *stop=None*, *compare=False*, *width=None*)

Prints the MOLS table that Sage can produce.

INPUT:

- `start, stop` (integers) – print the table of MOLS for value of n such that `start<=n<stop`. If only one integer is given as input, it is interpreted as the value of `stop` with `start=0` (same behaviour as `range`).
- `compare` (boolean) – if sets to `True` the MOLS displays with $+$ and $-$ entries its difference with the table from the Handbook of Combinatorial Designs (2ed).
- `width` (integer) – the width of each column of the table. By default, it is computed from range of values determined by the parameters `start` and `stop`.

EXAMPLES:

```
sage: from sage.combinat.designs.latin_squares import MOLS_table
sage: MOLS_table(100)
       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
    ____________________________________________________________________________________
  0| +oo +oo   1   2   3   4   1   6   7   8   2  10   5  12   4   4  15  16   5  18
 20|   4   5   3  22   7  24   4  26   5  28   4  30  31   5   4   5   8  36   4   5
 40|   7  40   5  42   5   6   4  46   8  48   6   5   5  52   5   6   7   7   5  58
 60|   5  60   5   6  63   7   5  66   5   6   6  70   7  72   5   7   6   6   6  78
 80|   9  80   8  82   6   6   6   6   7  88   6   7   6   6   6   6   7  96   6   8
sage: MOLS_table(100, width=4)
          0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17
       ____________________________________________________________________________________________
    0|  +oo  +oo    1    2    3    4    1    6    7    8    2   10    5   12    4    4   15   16
   20|    4    5    3   22    7   24    4   26    5   28    4   30   31    5    4    5    8   36
   40|    7   40    5   42    5    6    4   46    8   48    6    5    5   52    5    6    7    7
   60|    5   60    5    6   63    7    5   66    5    6    6   70    7   72    5    7    6    6
   80|    9   80    8   82    6    6    6    6    7   88    6    7    6    6    6    6    7   96
sage: MOLS_table(100, compare=True)
       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
    ____________________________________________________________________________________
  0|                                                           +               +
 20|
 40|
 60|   +
 80|
```

```
sage: MOLS_table(50, 100, compare=True)
       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
    ___________________________________________________________________________________
 40|
 60|   +
 80|
```

sage.combinat.designs.latin_squares.**are_mutually_orthogonal_latin_squares**(*l*, *verbose=False*)

Check wether the list of matrices in `l` form mutually orthogonal latin squares.

INPUT:

- `verbose` - if `True` then print why the list of matrices provided are not mutually orthogonal latin squares

EXAMPLES:

```
sage: from sage.combinat.designs.latin_squares import are_mutually_orthogonal_latin_squares
sage: m1 = matrix([[0,1,2],[2,0,1],[1,2,0]])
sage: m2 = matrix([[0,1,2],[1,2,0],[2,0,1]])
sage: m3 = matrix([[0,1,2],[2,0,1],[1,2,0]])
sage: are_mutually_orthogonal_latin_squares([m1,m2])
True
sage: are_mutually_orthogonal_latin_squares([m1,m3])
False
sage: are_mutually_orthogonal_latin_squares([m2,m3])
True
sage: are_mutually_orthogonal_latin_squares([m1,m2,m3], verbose=True)
Squares 0 and 2 are not orthogonal
False

sage: m = designs.mutually_orthogonal_latin_squares(7,8)
sage: are_mutually_orthogonal_latin_squares(m)
True
```

TESTS:

Not a latin square:

```
sage: m1 = matrix([[0,1,0],[2,0,1],[1,2,0]])
sage: m2 = matrix([[0,1,2],[1,2,0],[2,0,1]])
sage: are_mutually_orthogonal_latin_squares([m1,m2], verbose=True)
Matrix 0 is not row latin
False
sage: m1 = matrix([[0,1,2],[1,0,2],[1,2,0]])
sage: are_mutually_orthogonal_latin_squares([m1,m2], verbose=True)
Matrix 0 is not column latin
False
sage: m1 = matrix([[0,0,0],[1,1,1],[2,2,2]])
sage: m2 = matrix([[0,1,2],[0,1,2],[0,1,2]])
sage: are_mutually_orthogonal_latin_squares([m1,m2])
False
```

sage.combinat.designs.latin_squares.**latin_square_product**(*M*, *N*, **others*)

Return the product of two (or more) latin squares.

Given two Latin Squares M, N of respective sizes m, n, the direct product $M \times N$ of size mn is defined by $(M \times N)((i_1, i_2), (j_1, j_2)) = (M(i_1, j_1), N(i_2, j_2))$ where $i_1, j_1 \in [m], i_2, j_2 \in [n]$

Each pair of values $(i, j) \in [m] \times [n]$ is then relabeled to $in + j$.

This is Lemma 6.25 of [Stinson2004].

INPUT:

An arbitrary number of latin squares (greater than 2).

EXAMPLES:

```
sage: from sage.combinat.designs.latin_squares import latin_square_product
sage: m=designs.mutually_orthogonal_latin_squares(3,4)[0]
sage: latin_square_product(m,m,m)
64 x 64 sparse matrix over Integer Ring (use the '.str()' method to see the entries)
```

sage.combinat.designs.latin_squares.**mutually_orthogonal_latin_squares**(*k*, *n*, *partitions=False*, *check=True*, *existence=False*)

Return k Mutually Orthogonal $n \times n$ Latin Squares (MOLS).

For more information on Mutually Orthogonal Latin Squares, see `latin_squares`.

INPUT:

- `k` (integer) – number of MOLS. If `k=None` it is set to the largest value available.
- `n` (integer) – size of the latin square.
- `partition` (boolean) – a Latin Square can be seen as 3 partitions of the n^2 cells of the array into n sets of size n, respectively :
  - The partition of rows
  - The partition of columns
  - The partition of number (cells numbered with 0, cells numbered with 1, ...)

  These partitions have the additional property that any two sets from different partitions intersect on exactly one element.

  When `partition` is set to `True`, this function returns a list of $k+2$ partitions satisfying this intersection property instead of the $k+2$ MOLS (though the data is exactly the same in both cases).
- `existence` (boolean) – instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`).
  - `False` – meaning that the design does not exist.

  Note: When `k=None` and `existence=True` the function returns an integer, i.e. the largest k such that we can build a k MOLS of order n.
- `check` – (boolean) Whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.

EXAMPLES:

```
sage: designs.mutually_orthogonal_latin_squares(4,5)
[
[0 2 4 1 3]  [0 3 1 4 2]  [0 4 3 2 1]  [0 1 2 3 4]
```

```
[4 1 3 0 2]  [3 1 4 2 0]  [2 1 0 4 3]  [4 0 1 2 3]
[3 0 2 4 1]  [1 4 2 0 3]  [4 3 2 1 0]  [3 4 0 1 2]
[2 4 1 3 0]  [4 2 0 3 1]  [1 0 4 3 2]  [2 3 4 0 1]
[1 3 0 2 4], [2 0 3 1 4], [3 2 1 0 4], [1 2 3 4 0]
]

sage: designs.mutually_orthogonal_latin_squares(3,7)
[
[0 2 4 6 1 3 5]  [0 3 6 2 5 1 4]  [0 4 1 5 2 6 3]
[6 1 3 5 0 2 4]  [5 1 4 0 3 6 2]  [4 1 5 2 6 3 0]
[5 0 2 4 6 1 3]  [3 6 2 5 1 4 0]  [1 5 2 6 3 0 4]
[4 6 1 3 5 0 2]  [1 4 0 3 6 2 5]  [5 2 6 3 0 4 1]
[3 5 0 2 4 6 1]  [6 2 5 1 4 0 3]  [2 6 3 0 4 1 5]
[2 4 6 1 3 5 0]  [4 0 3 6 2 5 1]  [6 3 0 4 1 5 2]
[1 3 5 0 2 4 6], [2 5 1 4 0 3 6], [3 0 4 1 5 2 6]
]

sage: designs.mutually_orthogonal_latin_squares(2,5,partitions=True)
[[[0, 1, 2, 3, 4],
  [5, 6, 7, 8, 9],
  [10, 11, 12, 13, 14],
  [15, 16, 17, 18, 19],
  [20, 21, 22, 23, 24]],
 [[0, 5, 10, 15, 20],
  [1, 6, 11, 16, 21],
  [2, 7, 12, 17, 22],
  [3, 8, 13, 18, 23],
  [4, 9, 14, 19, 24]],
 [[0, 8, 11, 19, 22],
  [3, 6, 14, 17, 20],
  [1, 9, 12, 15, 23],
  [4, 7, 10, 18, 21],
  [2, 5, 13, 16, 24]],
 [[0, 9, 13, 17, 21],
  [2, 6, 10, 19, 23],
  [4, 8, 12, 16, 20],
  [1, 5, 14, 18, 22],
  [3, 7, 11, 15, 24]]]
```

What is the maximum number of MOLS of size 8 that Sage knows how to build?:

```
sage: designs.orthogonal_arrays.largest_available_k(8)-2
7
```

If you only want to know if Sage is able to build a given set of MOLS, query the `orthogonal_arrays.*` functions:

```
sage: designs.orthogonal_arrays.is_available(5+2, 5) # 5 MOLS of order 5
False
sage: designs.orthogonal_arrays.is_available(4+2,6) # 4 MOLS of order 6
False
```

Sage, however, is not able to prove that the second MOLS do not exist:

```
sage: designs.orthogonal_arrays.exists(4+2,6) # 4 MOLS of order 6
Unknown
```

If you ask for such a MOLS then you will respecively get an informative `EmptySetError` or `NotImplementedError`:

```
sage: designs.mutually_orthogonal_latin_squares(5, 5)
Traceback (most recent call last):
...
EmptySetError: There exist at most n-1 MOLS of size n if n>=2.
sage: designs.mutually_orthogonal_latin_squares(4,6)
Traceback (most recent call last):
...
NotImplementedError: I don't know how to build 4 MOLS of order 6
```

TESTS:

The special case $n = 1$:

```
sage: designs.mutually_orthogonal_latin_squares(3, 1)
[[0], [0], [0]]
sage: designs.mutually_orthogonal_latin_squares(None, 1)
Traceback (most recent call last):
...
ValueError: there are no bound on k when 0<=n<=1
sage: designs.mutually_orthogonal_latin_squares(2,10)
[
[1 8 9 0 2 4 6 3 5 7]  [1 7 6 5 0 9 8 2 3 4]
[7 2 8 9 0 3 5 4 6 1]  [8 2 1 7 6 0 9 3 4 5]
[6 1 3 8 9 0 4 5 7 2]  [9 8 3 2 1 7 0 4 5 6]
[5 7 2 4 8 9 0 6 1 3]  [0 9 8 4 3 2 1 5 6 7]
[0 6 1 3 5 8 9 7 2 4]  [2 0 9 8 5 4 3 6 7 1]
[9 0 7 2 4 6 8 1 3 5]  [4 3 0 9 8 6 5 7 1 2]
[8 9 0 1 3 5 7 2 4 6]  [6 5 4 0 9 8 7 1 2 3]
[2 3 4 5 6 7 1 8 9 0]  [3 4 5 6 7 1 2 8 0 9]
[3 4 5 6 7 1 2 0 8 9]  [5 6 7 1 2 3 4 0 9 8]
[4 5 6 7 1 2 3 9 0 8], [7 1 2 3 4 5 6 9 8 0]
]
```

5.1.80 Orthogonal arrays (OA)

This module gathers some construction related to orthogonal arrays (or transversal designs). One can build an $OA(k, n)$ (or check that it can be built) from the Sage console with `designs.orthogonal_arrays.build`:

```
sage: OA = designs.orthogonal_arrays.build(4,8)
```

See also the modules `orthogonal_arrays_build_recursive` or `orthogonal_arrays_find_recursive` for recursive constructions.

This module defines the following functions:

`orthogonal_array()`	Return an orthogonal array of parameters k, n, t.
`transversal_design()`	Return a transversal design of parameters k, n.
`incomplete_orthogonal_array()`	Return an $OA(k, n) - \sum_{1 \leq i \leq x} OA(k, s_i)$.

is_transversal_design()	Check that a given set of blocks B is a transversal design.
is_orthogonal_array()	Check that the integer matrix OA is an $OA(k,n,t)$.
wilson_construction()	Return a $OA(k, rm+u)$ from a truncated $OA(k+s, r)$ by Wilson's construction.
TD_product()	Return the product of two transversal designs.
OA_find_disjoint_blocks()	Return x disjoint blocks contained in a given $OA(k,n)$.
OA_relabel()	Return a relabelled version of the OA.
OA_from_quasi_difference_matrix()	Return an Orthogonal Array from a Quasi-Difference matrix
OA_from_Vmt()	Return an Orthogonal Array from a $V(m,t)$
OA_from_PBD()	Return an $OA(k,n)$ from a PBD
OA_n_times_2_pow_c_from_matrix()	Return an $OA(k, \lvert G\rvert \cdot 2^c)$ from a constrained $(G, k-1, 2)$-difference matrix.
OA_from_wider_OA()	Return the first k columns of OA.
QDM_from_Vmt()	Return a QDM a $V(m,t)$

REFERENCES:

Functions

class sage.combinat.designs.orthogonal_arrays.**OAMainFunctions**(**args*, ***kwds*)

Functions related to orthogonal arrays.

An orthogonal array of parameters k, n, t is a matrix with k columns filled with integers from $[n]$ in such a way that for any t columns, each of the n^t possible rows occurs exactly once. In particular, the matrix has n^t rows.

For more information on orthogonal arrays, see Wikipedia article Orthogonal_array.

From here you have access to:

- build(k,n,t=2): return an orthogonal array with the given parameters.
- is_available(k,n,t=2): answer whether there is a construction available in Sage for a given set of parameters.
- exists(k,n,t=2): answer whether an orthogonal array with these parameters exist.
- largest_available_k(n,t=2): return the largest integer k such that Sage knows how to build an $OA(k,n)$.
- explain_construction(k,n,t=2): return a string that explains the construction that Sage uses to build an $OA(k,n)$.

EXAMPLES:

```
sage: designs.orthogonal_arrays.build(3,2)
[[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 0]]

sage: designs.orthogonal_arrays.build(5,5)
[[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 1, 3],
 [0, 3, 1, 4, 2], [0, 4, 3, 2, 1], [1, 0, 4, 3, 2],
 [1, 1, 1, 1, 1], [1, 2, 3, 4, 0], [1, 3, 0, 2, 4],
 [1, 4, 2, 0, 3], [2, 0, 3, 1, 4], [2, 1, 0, 4, 3],
 [2, 2, 2, 2, 2], [2, 3, 4, 0, 1], [2, 4, 1, 3, 0],
 [3, 0, 2, 4, 1], [3, 1, 4, 2, 0], [3, 2, 1, 0, 4],
 [3, 3, 3, 3, 3], [3, 4, 0, 1, 2], [4, 0, 1, 2, 3],
 [4, 1, 3, 0, 2], [4, 2, 0, 3, 1], [4, 3, 2, 1, 0],
 [4, 4, 4, 4, 4]]
```

What is the largest value of k for which Sage knows how to compute a $OA(k, 14, 2)$?:

```
sage: designs.orthogonal_arrays.largest_available_k(14)
6
```

If you ask for an orthogonal array that does not exist, then you will either obtain an `EmptySetError` (if it knows that such an orthogonal array does not exist) or a `NotImplementedError`:

```
sage: designs.orthogonal_arrays.build(4,2)
Traceback (most recent call last):
...
EmptySetError: There exists no OA(4,2) as k(=4)>n+t-1=3
sage: designs.orthogonal_arrays.build(12,20)
Traceback (most recent call last):
...
NotImplementedError: I don't know how to build an OA(12,20)!
```

static `build`(*k*, *n*, *t=2*, *resolvable=False*)

Return an $OA(k,n)$ of strength t

An orthogonal array of parameters k, n, t is a matrix with k columns filled with integers from $[n]$ in such a way that for any t columns, each of the n^t possible rows occurs exactly once. In particular, the matrix has n^t rows.

More general definitions sometimes involve a λ parameter, and we assume here that $\lambda = 1$.

For more information on orthogonal arrays, see Wikipedia article Orthogonal_array.

INPUT:

- `k,n,t` (integers) – parameters of the orthogonal array.
- `resolvable` (boolean) – set to `True` if you want the design to be resolvable. The n classes of the resolvable design are obtained as the first n blocks, then the next n blocks, etc ... Set to `False` by default.

EXAMPLES:

```
sage: designs.orthogonal_arrays.build(3,3,resolvable=True) # indirect doctest
[[0, 0, 0],
 [1, 2, 1],
 [2, 1, 2],
 [0, 2, 2],
 [1, 1, 0],
 [2, 0, 1],
 [0, 1, 1],
 [1, 0, 2],
 [2, 2, 0]]
sage: OA_7_50 = designs.orthogonal_arrays.build(7,50)      # indirect doctest
```

static `exists`(*k*, *n*, *t=2*)

Return the existence status of an $OA(k,n)$

INPUT:

- `k,n,t` (integers) – parameters of the orthogonal array.

Warning: The function does not only return booleans, but `True`, `False`, or `Unknown`.

See also:

`is_available()`

EXAMPLE:

```
sage: designs.orthogonal_arrays.exists(3,6) # indirect doctest
True
sage: designs.orthogonal_arrays.exists(4,6) # indirect doctest
Unknown
sage: designs.orthogonal_arrays.exists(7,6) # indirect doctest
False
```

static explain_construction(k, n, t=2)

Return a string describing how to builds an $OA(k,n)$

INPUT:

- k, n, t (integers) – parameters of the orthogonal array.

EXAMPLE:

```
sage: designs.orthogonal_arrays.explain_construction(9,565)
"Wilson's construction n=23.24+13 with master design OA(9+1,23)"
sage: designs.orthogonal_arrays.explain_construction(10,154)
'the database contains a (137,10;1,0;17)-quasi difference matrix'
```

static is_available(k, n, t=2)

Return whether Sage can build an $OA(k,n)$.

INPUT:

- k, n, t (integers) – parameters of the orthogonal array.

See also:

exists()

EXAMPLE:

```
sage: designs.orthogonal_arrays.is_available(3,6) # indirect doctest
True
sage: designs.orthogonal_arrays.is_available(4,6) # indirect doctest
False
```

static largest_available_k(n, t=2)

Return the largest k such that Sage can build an $OA(k,n)$.

INPUT:

- n (integer)
- t – (integer; default: 2) – strength of the array

EXAMPLE:

```
sage: designs.orthogonal_arrays.largest_available_k(0)
+Infinity
sage: designs.orthogonal_arrays.largest_available_k(1)
+Infinity
sage: designs.orthogonal_arrays.largest_available_k(10)
4
sage: designs.orthogonal_arrays.largest_available_k(27)
28
sage: designs.orthogonal_arrays.largest_available_k(100)
10
sage: designs.orthogonal_arrays.largest_available_k(-1)
Traceback (most recent call last):
```

```
...
ValueError: n(=-1) was expected to be >=0
```

sage.combinat.designs.orthogonal_arrays.**OA_find_disjoint_blocks**(*OA*, *k*, *n*, *x*)

Return x disjoint blocks contained in a given $OA(k,n)$.

x blocks of an OA are said to be disjoint if they all have different values for a every given index, i.e. if they correspond to disjoint blocks in the TD assciated with the OA.

INPUT:

- `OA` – an orthogonal array
- `k,n,x` (integers)

See also:

`incomplete_orthogonal_array()`

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import OA_find_disjoint_blocks
sage: k=3;n=4;x=3
sage: Bs = OA_find_disjoint_blocks(designs.orthogonal_arrays.build(k,n),k,n,x)
sage: assert len(Bs) == x
sage: for i in range(k):
....:     assert len(set([B[i] for B in Bs])) == x
sage: OA_find_disjoint_blocks(designs.orthogonal_arrays.build(k,n),k,n,5)
Traceback (most recent call last):
...
ValueError: There does not exist 5 disjoint blocks in this OA(3,4)
```

sage.combinat.designs.orthogonal_arrays.**OA_from_PBD**(*k*, *n*, *PBD*, *check=True*)

Return an $OA(k,n)$ from a PBD

Construction

Let $\mathcal{B}$ be a $(n,K,1)$-PBD. If there exists for every $i \in K$ a $TD(k,i) - i \times TD(k,1)$ (i.e. if there exist k idempotent MOLS), then one can obtain a $OA(k,n)$ by concatenating:

- A $TD(k,i) - i \times TD(k,1)$ defined over the elements of B for every $B \in \mathcal{B}$.
- The rows $(i,...,i)$ of length k for every $i \in [n]$.

Note: This function raises an exception when Sage is unable to build the necessary designs.

INPUT:

- `k,n` (integers)
- `PBD` – a PBD on $0,...,n-1$.

EXAMPLES:

We start from the example VI.1.2 from the [DesignHandbook] to build an $OA(3,10)$:

```
sage: from sage.combinat.designs.orthogonal_arrays import OA_from_PBD
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: pbd = [[0,1,2,3],[0,4,5,6],[0,7,8,9],[1,4,7],[1,5,8],
....: [1,6,9],[2,4,9],[2,5,7],[2,6,8],[3,4,8],[3,5,9],[3,6,7]]
sage: oa = OA_from_PBD(3,10,pbd)
sage: is_orthogonal_array(oa, 3, 10)
True
```

But we cannot build an $OA(4,10)$ for this PBD (although there exists an $OA(4,10)$):

```
sage: OA_from_PBD(4,10,pbd)
Traceback (most recent call last):
...
EmptySetError: There is no OA(n+1,n) - 3.OA(n+1,1) as all blocks intersect in a projective plane
```

Or an $OA(3,6)$ (as the PBD has 10 points):

```
sage: _ = OA_from_PBD(3,6,pbd)
Traceback (most recent call last):
...
RuntimeError: PBD is not a valid Pairwise Balanced Design on [0,...,5]
```

sage.combinat.designs.orthogonal_arrays.**OA_from_Vmt**(*m*, *t*, *V*)

Return an Orthogonal Array from a $V(m,t)$

INPUT:

- `m, t` (integers)
- `V` – the vector $V(m,t)$.

See also:

- `QDM_from_Vmt()`
- `OA_from_quasi_difference_matrix()`

EXAMPLES:

```
sage: _ = designs.orthogonal_arrays.build(6,46) # indirect doctest
```

sage.combinat.designs.orthogonal_arrays.**OA_from_quasi_difference_matrix**(*M*, *G*, *add_col=True*, *fill_hole=True*)

Return an Orthogonal Array from a Quasi-Difference matrix

Difference Matrices

Let G be a group of order g. A *difference matrix* M is a $g \times k$ matrix with entries from G such that for any $1 \leq i < j < k$ the set $\{d_{li} - d_{lj} : 1 \leq l \leq g\}$ is equal to G.

By concatenating the g matrices $M + x$ (where $x \in G$), one obtains a matrix of size $g^2 \times x$ which is also an $OA(k,g)$.

Quasi-difference Matrices

A quasi-difference matrix is a difference matrix with missing entries. The construction above can be applied again in this case, where the missing entries in each column of M are replaced by unique values on which G has a trivial action.

This produces an incomplete orthogonal array with a "hole" (i.e. missing rows) of size 'u' (i.e. the number of missing values per column of M). If there exists an $OA(k,u)$, then adding the rows of this $OA(k,u)$ to the incomplete orthogonal array should lead to an OA...

Formal definition (from the Handbook of Combinatorial Designs [DesignHandbook])

Let G be an abelian group of order n. A $(n,k;\lambda,\mu;u)$-quasi-difference matrix (QDM) is a matrix $Q = (q_{ij})$ with $\lambda(n-1+2u)+\mu$ rows and k columns, with each entry either empty or containing an element of G.

Each column contains exactly λu entries, and each row contains at most one empty entry. Furthermore, for each $1 \leq i < j \leq k$ the multiset

$$\{q_{li} - q_{lj} : 1 \leq l \leq \lambda(n-1+2u)+\mu, \text{ with } q_{li} \text{ and } q_{lj} \text{ not empty}\}$$

contains every nonzero element of G exactly λ times, and contains 0 exactly μ times.

Construction

If a $(n, k; \lambda, \mu; u)$-QDM exists and $\mu \leq \lambda$, then an $ITD_\lambda(k, n+u; u)$ exists. Start with a $(n, k; \lambda, \mu; u)$-QDM A over the group G. Append $\lambda - \mu$ rows of zeroes. Then select u elements $\infty_1, \ldots, \infty_u$ not in G, and replace the empty entries, each by one of these infinite symbols, so that ∞_i appears exactly once in each column. Develop the resulting matrix over the group G (leaving infinite symbols fixed), to obtain a $\lambda(n^2+2nu) \times k$ matrix T. Then T is an orthogonal array with k columns and index λ, having $n+u$ symbols and one hole of size u.

Adding to T an $OA(k, u)$ with elements $\infty_1, \ldots, \infty_u$ yields the $ITD_\lambda(k, n+u; u)$.

For more information, see the Handbook of Combinatorial Designs [DesignHandbook] or http://web.cs.du.edu/~petr/milehigh/2013/Colbourn.pdf.

INPUT:

- `M` – the difference matrix whose entries belong to `G`
- `G` – a group
- `add_col` (boolean) – whether to add a column to the final OA equal to $(x_1, \ldots, x_g, x_1, \ldots, x_g, \ldots)$ where $G = \{x_1, \ldots, x_g\}$.
- `fill_hole` (boolean) – whether to return the incomplete orthogonal array, or complete it with the $OA(k, u)$ (default). When `fill_hole is None`, no block of the incomplete OA contains more than one value $\geq |G|$.

EXAMPLES:

```
sage: _ = designs.orthogonal_arrays.build(6,20) # indirect doctest
```

sage.combinat.designs.orthogonal_arrays.**OA_from_wider_OA**(*OA*, *k*)

Return the first k columns of OA.

If OA has k columns, this function returns OA immediately.

INPUT:

- `OA` – an orthogonal array.
- `k` (integer)

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import OA_from_wider_OA
sage: OA_from_wider_OA(designs.orthogonal_arrays.build(6,20,2),1)[:5]
[(19,), (19,), (19,), (19,), (19,)]
sage: _ = designs.orthogonal_arrays.build(5,46) # indirect doctest
```

sage.combinat.designs.orthogonal_arrays.**OA_n_times_2_pow_c_from_matrix**(*k*, *c*, *G*, *A*, *Y*, *check=True*)

Return an $OA(k, |G| \cdot 2^c)$ from a constrained $(G, k-1, 2)$-difference matrix.

This construction appears in [AbelCheng1994] and [AbelThesis].

Let G be an additive Abelian group. We denote by H a $GF(2)$-hyperplane in $GF(2^c)$.

Let A be a $(k-1) \times 2|G|$ array with entries in $G \times GF(2^c)$ and Y be a vector with $k-1$ entries in $GF(2^c)$. Let B and C be respectively the part of the array that belong to G and $GF(2^c)$.

The input A and Y must satisfy the following conditions. For any $i \neq j$ and $g \in G$:

- there are exactly two values of s such that $B_{i,s} - B_{j,s} = g$ (i.e. B is a $(G, k-1, 2)$-difference matrix),
- let s_1 and s_2 denote the two values of s given above, then exactly one of $C_{i,s_1} - C_{j,s_1}$ and $C_{i,s_2} - C_{j,s_2}$ belongs to the $GF(2)$-hyperplane $(Y_i - Y_j) \cdot H$ (we implicitely assumed that $Y_i \neq Y_j$).

Under these conditions, it is easy to check that the array whose $k-1$ rows of length $|G| \cdot 2^c$ indexed by $1 \leq i \leq k-1$ given by $A_{i,s} + (0, Y_i \cdot v)$ where $1 \leq s \leq 2|G|, v \in H$ is a $(G \times GF(2^c), k-1, 1)$-difference matrix.

INPUT:

- `k, c` (integers) – integers
- `G` – an additive Abelian group
- `A` – a matrix with entries in $G \times GF(2^c)$
- `Y` – a vector with entries in $GF(2^c)$
- `check` – (boolean) Whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.

Note: By convention, a multiplicative generator w of $GF(2^c)^*$ is fixed (inside the function). The hyperplane H is the one spanned by $w^0, w^1, \ldots, w^{c-1}$. The $GF(2^c)$ part of the input matrix A and vector Y are given in the following form: the integer i corresponds to the element w^i and `None` corresponds to 0.

See also:

Several examples use this construction:

- `OA_9_40()`
- `OA_11_80()`
- `OA_15_112()`
- `OA_11_160()`
- `OA_16_176()`
- `OA_16_208()`
- `OA_15_224()`
- `OA_20_352()`
- `OA_20_416()`
- `OA_20_544()`
- `OA_11_640()`
- `OA_15_896()`

EXAMPLE:

```
sage: from sage.combinat.designs.orthogonal_arrays import OA_n_times_2_pow_c_from_matrix
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: A = [
....: [(0,None),(0,None),(0,None),(0,None),(0,None),(0,None),(0,None),(0,None),(0,None),(0,None)
....: [(0,None),(1,None),   (2,2),   (3,2),   (4,2),(2,None),(3,None),(4,None),   (0,2),   (1,2)
```

```
....: [(0,None),   (2,5),   (4,5),   (1,2),   (3,6),   (3,4),   (0,0),   (2,1),   (4,1),   (1,6)
....: [(0,None),   (3,4),   (1,4),   (4,0),   (2,5),(3,None),   (1,0),   (4,1),   (2,2),   (0,3)
....: ]
sage: Y = [None, 0, 1, 6]
sage: OA = OA_n_times_2_pow_c_from_matrix(5,3,GF(5),A,Y)
sage: is_orthogonal_array(OA,5,40,2)
True

sage: A[0][0] = (1,None)
sage: OA_n_times_2_pow_c_from_matrix(5,3,GF(5),A,Y)
Traceback (most recent call last):
...
ValueError: the first part of the matrix A must be a
(G,k-1,2)-difference matrix

sage: A[0][0] = (0,0)
sage: OA_n_times_2_pow_c_from_matrix(5,3,GF(5),A,Y)
Traceback (most recent call last):
...
ValueError: B_2,0 - B_0,0 = B_2,6 - B_0,6 but the associated part of the
matrix C does not satisfies the required condition
```

REFERENCES:

sage.combinat.designs.orthogonal_arrays.**OA_relabel**(*OA*, *k*, *n*, *blocks=()*, *matrix=None*)

Return a relabelled version of the OA.

INPUT:

- `OA` – an OA, or rather a list of blocks of length k, each of which contains integers from 0 to $n-1$.
- `k,n` (integers)
- `blocks` (list of blocks) – relabels the integers of the OA from $[0..n-1]$ into $[0..n-1]$ in such a way that the i blocks from `block` are respectively relabeled as `[n-i,...,n-i]`, ..., `[n-1,...,n-1]`. Thus, the blocks from this list are expected to have disjoint values for each coordinate.

  If set to the empty list (default) no such relabelling is performed.
- `matrix` – a matrix of dimensions k, n such that if the i th coordinate of a block is x, this x will be relabelled with `matrix[i][x]`. This is not necessarily an integer between 0 and $n-1$, and it is not necessarily an integer either. This is performed *after* the previous relabelling.

  If set to `None` (default) no such relabelling is performed.

Note: A `None` coordinate in one block remains a `None` coordinate in the final block.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import OA_relabel
sage: OA = designs.orthogonal_arrays.build(3,2)
sage: OA_relabel(OA,3,2,matrix=[["A","B"],["C","D"],["E","F"]])
[['A', 'C', 'E'], ['A', 'D', 'F'], ['B', 'C', 'F'], ['B', 'D', 'E']]

sage: TD = OA_relabel(OA,3,2,matrix=[[0,1],[2,3],[4,5]]); TD
[[0, 2, 4], [0, 3, 5], [1, 2, 5], [1, 3, 4]]
sage: from sage.combinat.designs.orthogonal_arrays import is_transversal_design
sage: is_transversal_design(TD,3,2)
True
```

Making sure that `[2,2,2,2]` is a block of $OA(4,3)$. We do this by relabelling block `[0,0,0,0]` which belongs to the design:

```
sage: designs.orthogonal_arrays.build(4,3)
[[0, 0, 0, 0], [0, 1, 2, 1], [0, 2, 1, 2], [1, 0, 2, 2], [1, 1, 1, 0], [1, 2, 0, 1], [2, 0, 1, 1
sage: OA_relabel(designs.orthogonal_arrays.build(4,3),4,3,blocks=[[0,0,0,0]])
[[2, 2, 2, 2], [2, 0, 1, 0], [2, 1, 0, 1], [0, 2, 1, 1], [0, 0, 0, 2], [0, 1, 2, 0], [1, 2, 0, 0
```

TESTS:

```
sage: OA_relabel(designs.orthogonal_arrays.build(3,2),3,2,blocks=[[0,1],[0,1]])
Traceback (most recent call last):
...
RuntimeError: Two block have the same coordinate for one of the k dimensions
```

sage.combinat.designs.orthogonal_arrays.**QDM_from_Vmt**(*m*, *t*, *V*)

Return a QDM from a $V(m,t)$

Definition

Let q be a prime power and let $q = mt + 1$ for m, t integers. Let ω be a primitive element of $\mathbb{F}_q$. A $V(m,t)$ vector is a vector $(a_1, \ldots, a_{m+1}$ for which, for each $1 \leq k < m$, the differences

$$\{a_{i+k} - a_i : 1 \leq i \leq m+1, i+k \neq m+2\}$$

represent the m cyclotomic classes of $\mathbb{F}_{mt+1}$ (compute subscripts modulo $m+2$). In other words, for fixed k, is $a_{i+k} - a_i = \omega^{mx+\alpha}$ and $a_{j+k} - a_j = \omega^{my+\beta}$ then $\alpha \not\equiv \beta \mod m$

Construction of a quasi-difference matrix from a 'V(m,t)' vector

Starting with a $V(m,t)$ vector $(a_1, \ldots, a_{m+1})$, form a single row of length $m+2$ whose first entry is empty, and whose remaining entries are $(a_1, \ldots, a_{m+1})$. Form t rows by multiplying this row by the t th roots, i.e. the powers of ω^m. From each of these t rows, form $m+2$ rows by taking the $m+2$ cyclic shifts of the row. The result is a $(a, m+2; 1, 0; t) - QDM$.

For more information, refer to the Handbook of Combinatorial Designs [DesignHandbook].

INPUT:

- `m, t` (integers)
- `V` – the vector $V(m,t)$.

See also:

`OA_from_quasi_difference_matrix()`

EXAMPLES:

```
sage: _ = designs.orthogonal_arrays.build(6,46) # indirect doctest
```

sage.combinat.designs.orthogonal_arrays.**TD_product**(*k*, *TD1*, *n1*, *TD2*, *n2*, *check=True*)

Return the product of two transversal designs.

From a transversal design TD_1 of parameters k, n_1 and a transversal design TD_2 of parameters k, n_2, this function returns a transversal design of parameters k, n where $n = n_1 \times n_2$.

Formally, if the groups of TD_1 are $V_1^1, \ldots, V_k^1$ and the groups of TD_2 are $V_1^2, \ldots, V_k^2$, the groups of the product design are $V_1^1 \times V_1^2, \ldots, V_k^1 \times V_k^2$ and its blocks are the $\{(x_1^1, x_1^2), \ldots, (x_k^1, x_k^2)\}$ where $\{x_1^1, \ldots, x_k^1\}$ is a block of TD_1 and $\{x_1^2, \ldots, x_k^2\}$ is a block of TD_2.

INPUT:

- `TD1, TD2` – transversal designs.

- k, n1, n2 (integers) – see above.
- check (boolean) – Whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to True by default.

Note: This function uses transversal designs with $V_1 = \{0, \ldots, n-1\}, \ldots, V_k = \{(k-1)n, \ldots, kn-1\}$ both as input and ouptut.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import TD_product
sage: TD1 = designs.transversal_design(6,7)
sage: TD2 = designs.transversal_design(6,12)
sage: TD6_84 = TD_product(6,TD1,7,TD2,12)
```

class sage.combinat.designs.orthogonal_arrays.**TransversalDesign**(*blocks*, *k=None*, *n=None*, *check=True*, ***kwds*)

Bases: sage.combinat.designs.group_divisible_designs.GroupDivisibleDesign

Class for Transversal Designs

INPUT:

- blocks – collection of blocks
- k, n (integers) – parameters of the transversal design. They can be set to None (default) in which case their value is determined by the blocks.
- check (boolean) – whether to check that the design is indeed a transversal design with the right parameters. Set to True by default.

EXAMPLES:

```
sage: designs.transversal_design(None,5)
Transversal Design TD(6,5)
sage: designs.transversal_design(None,30)
Transversal Design TD(6,30)
sage: designs.transversal_design(None,36)
Transversal Design TD(10,36)
```

sage.combinat.designs.orthogonal_arrays.**incomplete_orthogonal_array**(*k*, *n*, *holes*, *resolvable=False*, *existence=False*)

Return an $OA(k,n) - \sum_{1 \leq i \leq x} OA(k, s_i)$.

An $OA(k,n) - \sum_{1 \leq i \leq x} OA(k, s_i)$ is an orthogonal array from which have been removed disjoint $OA(k, s_1), ..., OA(k, s_x)$. If there exist $OA(k, s_1), ..., OA(k, s_x)$ they can be used to fill the holes and give rise to an $OA(k, n)$.

A very useful particular case (see e.g. the Wilson construction in wilson_construction()) is when all $s_i = 1$. In that case the incomplete design is a $OA(k,n) - x.OA(k,1)$. Such design is equivalent to transversal design $TD(k,n)$ from which has been removed x disjoint blocks.

INPUT:

- k, n (integers)

- `holes` (list of integers) – respective sizes of the holes to be found.
- `resolvable` (boolean) – set to `True` if you want the design to be resolvable. The classes of the resolvable design are obtained as the first n blocks, then the next n blocks, etc ... Set to `False` by default.
- `existence` (boolean) – instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`).
  - `False` – meaning that the design does not exist.

Note: By convention, the ground set is always $V = \{0, ..., n-1\}$.

If all holes have size 1, in the incomplete orthogonal array returned by this function the holes are $\{n-1, ..., n-s_1\}^k$, $\{n-s_1-1, ..., n-s_1-s_2\}^k$, etc.

More generally, if `holes` is equal to $u1, ..., uk$, the i-th hole is the set of points $\{n - \sum_{j \geq i} u_j, ..., n - \sum_{j \geq i+1} u_j\}^k$.

See also:

`OA_find_disjoint_blocks()`

EXAMPLES:

```
sage: IOA = designs.incomplete_orthogonal_array(3,3,[1,1,1])
sage: IOA
[[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]
sage: missing_blocks = [[0,0,0],[1,1,1],[2,2,2]]
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: is_orthogonal_array(IOA + missing_blocks,3,3,2)
True
```

TESTS:

Affine planes and projective planes:

```
sage: for q in xrange(2,100):
....:     if is_prime_power(q):
....:         assert designs.incomplete_orthogonal_array(q,q,[1]*q,existence=True)
....:         assert not designs.incomplete_orthogonal_array(q+1,q,[1]*2,existence=True)
```

Further tests:

```
sage: designs.incomplete_orthogonal_array(8,4,[1,1,1],existence=True)
False
sage: designs.incomplete_orthogonal_array(5,10,[1,1,1],existence=True)
Unknown
sage: designs.incomplete_orthogonal_array(5,10,[1,1,1])
Traceback (most recent call last):
...
NotImplementedError: I don't know how to build an OA(5,10)!
sage: designs.incomplete_orthogonal_array(4,3,[1,1])
Traceback (most recent call last):
...
EmptySetError: There is no OA(n+1,n) - 2.OA(n+1,1) as all blocks intersect in a projective plane
sage: n=10
sage: k=designs.orthogonal_arrays.largest_available_k(n)
sage: designs.incomplete_orthogonal_array(k,n,[1,1,1],existence=True)
True
```

```
sage: _ = designs.incomplete_orthogonal_array(k,n,[1,1,1])
sage: _ = designs.incomplete_orthogonal_array(k,n,[1])
```

A resolvable $OA(k,n) - n.OA(k,1)$. We check that extending each class and adding the $[i,i,...]$ blocks turns it into an $OA(k+1,n)$.:

```
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: k,n=5,7
sage: OA = designs.incomplete_orthogonal_array(k,n,[1]*n,resolvable=True)
sage: classes = [OA[i*n:(i+1)*n] for i in range(n-1)]
sage: for classs in classes: # The design is resolvable !
....:     assert(len(set(col))==n for col in zip(*classs))
sage: OA.extend([[i]*(k) for i in range(n)])
sage: for i,R in enumerate(OA):
....:     R.append(i//n)
sage: is_orthogonal_array(OA,k+1,n)
True
```

Non-existent resolvable incomplete OA:

```
sage: designs.incomplete_orthogonal_array(9,13,[1]*10,resolvable=True,existence=True)
False
sage: designs.incomplete_orthogonal_array(9,13,[1]*10,resolvable=True)
Traceback (most recent call last):
...
EmptySetError: There is no resolvable incomplete OA(9,13) whose holes' sizes sum to 10<n(=13)
```

Error message for big holes:

```
sage: designs.incomplete_orthogonal_array(6,4*9,[9,9,8])
Traceback (most recent call last):
...
NotImplementedError: I was not able to build this OA(6,36)-OA(6,8)-2.OA(6,9)
```

10 holes of size 9 through the product construction:

```
sage: iOA = designs.incomplete_orthogonal_array(10,153,[9]*10)  # long time
sage: OA9 = designs.orthogonal_arrays.build(10,9)               # long time
sage: for i in range(10):                                       # long time
....:     iOA.extend([[153-9*(i+1)+x for x in B] for B in OA9]) # long time
sage: is_orthogonal_array(iOA,10,153)                           # long time
True
```

An $OA(9,82) - OA(9,9) - OA(9,1)$:

```
sage: ioa = designs.incomplete_orthogonal_array(9,82,[9,1])
sage: ioa.extend([[x+72 for x in B] for B in designs.orthogonal_arrays.build(9,9)])
sage: ioa.extend([[x+81 for x in B] for B in designs.orthogonal_arrays.build(9,1)])
sage: is_orthogonal_array(ioa,9,82,verbose=1)
True
```

An $OA(9,82) - OA(9,9) - 2.OA(9,1)$ in different orders:

```
sage: ioa = designs.incomplete_orthogonal_array(9,82,[1,9,1])
sage: ioa.extend([[x+71 for x in B] for B in designs.orthogonal_arrays.build(9,1)])
sage: ioa.extend([[x+72 for x in B] for B in designs.orthogonal_arrays.build(9,9)])
sage: ioa.extend([[x+81 for x in B] for B in designs.orthogonal_arrays.build(9,1)])
sage: is_orthogonal_array(ioa,9,82,verbose=1)
True
sage: ioa = designs.incomplete_orthogonal_array(9,82,[9,1,1])
```

```
sage: ioa.extend([[x+71 for x in B] for B in designs.orthogonal_arrays.build(9,9)])
sage: ioa.extend([[x+80 for x in B] for B in designs.orthogonal_arrays.build(9,1)])
sage: ioa.extend([[x+81 for x in B] for B in designs.orthogonal_arrays.build(9,1)])
sage: is_orthogonal_array(ioa,9,82,verbose=1)
True
```

Three holes of size 1:

```
sage: ioa = designs.incomplete_orthogonal_array(3,6,[1,1,1])
sage: ioa.extend([[i]*3 for i in [3,4,5]])
sage: is_orthogonal_array(ioa,3,6,verbose=1)
True
```

REFERENCES:

sage.combinat.designs.orthogonal_arrays.**is_transversal_design**(*B*, *k*, *n*, *verbose=False*)

Check that a given set of blocks `B` is a transversal design.

See `transversal_design()` for a definition.

INPUT:

- `B` – the list of blocks
- `k, n` – integers
- `verbose` (boolean) – whether to display information about what is going wrong.

Note: The tranversal design must have $\{0,\dots,kn-1\}$ as a ground set, partitioned as k sets of size n: $\{0,\dots,k-1\} \sqcup \{k,\dots,2k-1\} \sqcup \cdots \sqcup \{k(n-1),\dots,kn-1\}$.

EXAMPLES:

```
sage: TD = designs.transversal_design(5, 5, check=True) # indirect doctest
sage: from sage.combinat.designs.orthogonal_arrays import is_transversal_design
sage: is_transversal_design(TD, 5, 5)
True
sage: is_transversal_design(TD, 4, 4)
False
```

sage.combinat.designs.orthogonal_arrays.**largest_available_k**(*n*, *t=2*)

Return the largest k such that Sage can build an $OA(k,n)$.

INPUT:

- `n` (integer)
- `t` – (integer; default: 2) – strength of the array

EXAMPLE:

```
sage: designs.orthogonal_arrays.largest_available_k(0)
+Infinity
sage: designs.orthogonal_arrays.largest_available_k(1)
+Infinity
sage: designs.orthogonal_arrays.largest_available_k(10)
4
sage: designs.orthogonal_arrays.largest_available_k(27)
28
sage: designs.orthogonal_arrays.largest_available_k(100)
10
```

```
sage: designs.orthogonal_arrays.largest_available_k(-1)
Traceback (most recent call last):
...
ValueError: n(=-1) was expected to be >=0
```

sage.combinat.designs.orthogonal_arrays.**orthogonal_array**(*k*, *n*, *t=2*, *resolvable=False*, *check=True*, *existence=False*, *explain_construction=False*)

Return an orthogonal array of parameters k, n, t.

An orthogonal array of parameters k, n, t is a matrix with k columns filled with integers from $[n]$ in such a way that for any t columns, each of the n^t possible rows occurs exactly once. In particular, the matrix has n^t rows.

More general definitions sometimes involve a λ parameter, and we assume here that $\lambda = 1$.

An orthogonal array is said to be *resolvable* if it corresponds to a resolvable transversal design (see `sage.combinat.designs.incidence_structures.IncidenceStructure.is_resolvable()`).

For more information on orthogonal arrays, see Wikipedia article Orthogonal_array.

INPUT:

- `k` – (integer) number of columns. If `k=None` it is set to the largest value available.
- `n` – (integer) number of symbols
- `t` – (integer; default: 2) – strength of the array
- `resolvable` (boolean) – set to `True` if you want the design to be resolvable. The n classes of the resolvable design are obtained as the first n blocks, then the next n blocks, etc ... Set to `False` by default.
- `check` – (boolean) Whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.
- `existence` (boolean) – instead of building the design, return:
  - `True` – meaning that Sage knows how to build the design
  - `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`).
  - `False` – meaning that the design does not exist.

Note: When `k=None` and `existence=True` the function returns an integer, i.e. the largest k such that we can build a $OA(k, n)$.

- `explain_construction` (boolean) – return a string describing the construction.

OUTPUT:

The kind of output depends on the input:

- if `existence=False` (the default) then the output is a list of lists that represent an orthogonal array with parameters `k` and `n`
- if `existence=True` and `k` is an integer, then the function returns a troolean: either `True`, `Unknown` or `False`
- if `existence=True` and `k=None` then the output is the largest value of `k` for which Sage knows how to compute a $TD(k, n)$.

Note: This method implements theorems from [Stinson2004]. See the code's documentation for details.

See also:

When $t = 2$ an orthogonal array is also a transversal design (see `transversal_design()`) and a family of mutually orthogonal latin squares (see `mutually_orthogonal_latin_squares()`).

TESTS:

The special cases $n = 0, 1$:

```
sage: designs.orthogonal_arrays.build(3,0)
[]
sage: designs.orthogonal_arrays.build(3,1)
[[0, 0, 0]]
sage: designs.orthogonal_arrays.largest_available_k(0)
+Infinity
sage: designs.orthogonal_arrays.largest_available_k(1)
+Infinity
sage: designs.orthogonal_arrays.build(16,0)
[]
sage: designs.orthogonal_arrays.build(16,1)
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

when $t > 2$ and $k = None$:

```
sage: t = 3
sage: designs.orthogonal_arrays.largest_available_k(5,t=t) == t
True
sage: _ = designs.orthogonal_arrays.build(t,5,t)
```

sage.combinat.designs.orthogonal_arrays.**transversal_design**(*k*, *n*, *resolvable=False*, *check=True*, *existence=False*)

Return a transversal design of parameters k, n.

A transversal design of parameters k, n is a collection $\mathcal{S}$ of subsets of $V = V_1 \cup \cdots \cup V_k$ (where the *groups* V_i are disjoint and have cardinality n) such that:

- Any $S \in \mathcal{S}$ has cardinality k and intersects each group on exactly one element.
- Any two elements from distincts groups are contained in exactly one element of $\mathcal{S}$.

More general definitions sometimes involve a λ parameter, and we assume here that $\lambda = 1$.

For more information on transversal designs, see http://mathworld.wolfram.com/TransversalDesign.html.

INPUT:

- n, k – integers. If `k is None` it is set to the largest value available.
- `resolvable` (boolean) – set to `True` if you want the design to be resolvable (see `sage.combinat.designs.incidence_structures.IncidenceStructure.is_resolvable()`). The n classes of the resolvable design are obtained as the first n blocks, then the next n blocks, etc ... Set to `False` by default.
- `check` – (boolean) Whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.
- `existence` (boolean) – instead of building the design, return:

– `True` – meaning that Sage knows how to build the design

– `Unknown` – meaning that Sage does not know how to build the design, but that the design may exist (see `sage.misc.unknown`).

– `False` – meaning that the design does not exist.

Note: When `k=None` and `existence=True` the function returns an integer, i.e. the largest k such that we can build a $TD(k,n)$.

OUTPUT:

The kind of output depends on the input:

- if `existence=False` (the default) then the output is a list of lists that represent a $TD(k,n)$ with $V_1 = \{0,\dots,n-1\},\dots,V_k = \{(k-1)n,\dots,kn-1\}$
- if `existence=True` and `k` is an integer, then the function returns a troolean: either `True`, `Unknown` or `False`
- if `existence=True` and `k=None` then the output is the largest value of `k` for which Sage knows how to compute a $TD(k,n)$.

See also:

`orthogonal_array()` – a tranversal design $TD(k,n)$ is equivalent to an orthogonal array $OA(k,n,2)$.

EXAMPLES:

```
sage: TD = designs.transversal_design(5,5); TD
Transversal Design TD(5,5)
sage: TD.blocks()
[[0, 5, 10, 15, 20], [0, 6, 12, 18, 24], [0, 7, 14, 16, 23],
 [0, 8, 11, 19, 22], [0, 9, 13, 17, 21], [1, 5, 14, 18, 22],
 [1, 6, 11, 16, 21], [1, 7, 13, 19, 20], [1, 8, 10, 17, 24],
 [1, 9, 12, 15, 23], [2, 5, 13, 16, 24], [2, 6, 10, 19, 23],
 [2, 7, 12, 17, 22], [2, 8, 14, 15, 21], [2, 9, 11, 18, 20],
 [3, 5, 12, 19, 21], [3, 6, 14, 17, 20], [3, 7, 11, 15, 24],
 [3, 8, 13, 18, 23], [3, 9, 10, 16, 22], [4, 5, 11, 17, 23],
 [4, 6, 13, 15, 22], [4, 7, 10, 18, 21], [4, 8, 12, 16, 20],
 [4, 9, 14, 19, 24]]
```

Some examples of the maximal number of transversal Sage is able to build:

```
sage: TD_4_10 = designs.transversal_design(4,10)
sage: designs.transversal_design(5,10,existence=True)
Unknown
```

For prime powers, there is an explicit construction which gives a $TD(n+1,n)$:

```
sage: designs.transversal_design(4, 3, existence=True)
True
sage: designs.transversal_design(674, 673, existence=True)
True
```

For other values of `n` it depends:

```
sage: designs.transversal_design(7, 6, existence=True)
False
sage: designs.transversal_design(4, 6, existence=True)
Unknown
sage: designs.transversal_design(3, 6, existence=True)
True
```

```
sage: designs.transversal_design(11, 10, existence=True)
False
sage: designs.transversal_design(4, 10, existence=True)
True
sage: designs.transversal_design(5, 10, existence=True)
Unknown

sage: designs.transversal_design(7, 20, existence=True)
Unknown
sage: designs.transversal_design(6, 12, existence=True)
True
sage: designs.transversal_design(7, 12, existence=True)
True
sage: designs.transversal_design(8, 12, existence=True)
Unknown

sage: designs.transversal_design(6, 20, existence = True)
True
sage: designs.transversal_design(7, 20, existence = True)
Unknown
```

If you ask for a transversal design that Sage is not able to build then an `EmptySetError` or a `NotImplementedError` is raised:

```
sage: designs.transversal_design(47, 100)
Traceback (most recent call last):
...
NotImplementedError: I don't know how to build a TD(47,100)!
sage: designs.transversal_design(55, 54)
Traceback (most recent call last):
...
EmptySetError: There exists no TD(55,54)!
```

Those two errors correspond respectively to the cases where Sage answer `Unknown` or `False` when the parameter `existence` is set to `True`:

```
sage: designs.transversal_design(47, 100, existence=True)
Unknown
sage: designs.transversal_design(55, 54, existence=True)
False
```

If for a given n you want to know the largest k for which Sage is able to build a $TD(k,n)$ just call the function with k set to `None` and `existence` set to `True` as follows:

```
sage: designs.transversal_design(None, 6, existence=True)
3
sage: designs.transversal_design(None, 20, existence=True)
6
sage: designs.transversal_design(None, 30, existence=True)
6
sage: designs.transversal_design(None, 120, existence=True)
9
```

TESTS:

The case when $n = 1$:

```
sage: designs.transversal_design(5,1).blocks()
[[0, 1, 2, 3, 4]]
```

Obtained through Wilson's decomposition:

```
sage: _ = designs.transversal_design(4,38)
```

Obtained through product decomposition:

```
sage: _ = designs.transversal_design(6,60)
sage: _ = designs.transversal_design(5,60) # checks some tricky divisibility error
```

For small values of the parameter n we check the coherence of the function `transversal_design()`:

```
sage: for n in xrange(2,25):                                    # long time -- 15 secs
....:     i = 2
....:     while designs.transversal_design(i, n, existence=True) is True:
....:         i += 1
....:     _ = designs.transversal_design(i-1, n)
....:     assert designs.transversal_design(None, n, existence=True) == i - 1
....:     j = i
....:     while designs.transversal_design(j, n, existence=True) is Unknown:
....:         try:
....:             _ = designs.transversal_design(j, n)
....:             raise AssertionError("no NotImplementedError")
....:         except NotImplementedError:
....:             pass
....:         j += 1
....:     k = j
....:     while k < n+4:
....:         assert designs.transversal_design(k, n, existence=True) is False
....:         try:
....:             _ = designs.transversal_design(k, n)
....:             raise AssertionError("no EmptySetError")
....:         except EmptySetError:
....:             pass
....:         k += 1
....:     print "%2d: (%2d, %2d)"%(n,i,j)
 2: ( 4,  4)
 3: ( 5,  5)
 4: ( 6,  6)
 5: ( 7,  7)
 6: ( 4,  7)
 7: ( 9,  9)
 8: (10, 10)
 9: (11, 11)
10: ( 5, 11)
11: (13, 13)
12: ( 8, 14)
13: (15, 15)
14: ( 7, 15)
15: ( 7, 17)
16: (18, 18)
17: (19, 19)
18: ( 8, 20)
19: (21, 21)
20: ( 7, 22)
21: ( 8, 22)
22: ( 6, 23)
23: (25, 25)
24: (10, 26)
```

The special case $n = 1$:

```
sage: designs.transversal_design(3, 1).blocks()
[[0, 1, 2]]
sage: designs.transversal_design(None, 1, existence=True)
+Infinity
sage: designs.transversal_design(None, 1)
Traceback (most recent call last):
...
ValueError: there is no upper bound on k when 0<=n<=1
```

Resolvable TD:

```
sage: k,n = 5,15
sage: TD = designs.transversal_design(k,n,resolvable=True)
sage: TD.is_resolvable()
True
sage: r     = designs.transversal_design(None,n,resolvable=True,existence=True)
sage: non_r = designs.transversal_design(None,n,existence=True)
sage: r + 1 == non_r
True
```

sage.combinat.designs.orthogonal_arrays.**wilson_construction**(*OA*, *k*, *r*, *m*, *u*, *check=True*, *explain_construction=False*)

Returns a $OA(k, rm + \sum_i u_i)$ from a truncated $OA(k + s, r)$ by Wilson's construction.

Simple form:

Let OA be a truncated $OA(k + s, r)$ with s truncated columns of sizes $u_1, ..., u_s$, whose blocks have sizes in $\{k + b_1, ..., k + b_t\}$. If there exist:

- An $OA(k, m + b_i) - b_i.OA(k, 1)$ for every $1 \leq i \leq t$
- An $OA(k, u_i)$ for every $1 \leq i \leq s$

Then there exists an $OA(k, rm + \sum u_i)$. The construction is a generalization of Lemma 3.16 in [HananiBIBD].

Brouwer-Van Rees form:

Let OA be a truncated $OA(k + s, r)$ with s truncated columns of sizes $u_1, ..., u_s$. Let the set H_i of the u_i points of column $k + i$ be partitionned into $\sum_j H_{ij}$. Let m_{ij} be integers such that:

- For $0 \leq i < l$ there exists an $OA(k, \sum_j m_{ij}|H_{ij}|)$
- For any block $B \in OA$ intersecting the sets $H_{ij(i)}$ there exists an $OA(k, m + \sum_i m_{ij}) - \sum_i OA(k, m_{ij(j)})$.

Then there exists an $OA(k, rm + \sum_{i,j} m_{ij})$. This construction appears in [BvR82].

INPUT:

- `OA` – an incomplete orthogonal array with $k + s$ columns. The elements of a column of size c must belong to $\{0, ..., c\}$. The missing entries of a block are represented by `None` values. If `OA=None`, it is defined as a truncated orthogonal arrays with $k + s$ columns.
- `k, r, m` (integers)
- `u` (list) – two cases depending on the form to use:
  - Simple form: a list of length s such that column `k+i` has size `u[i]`. The untruncated points of column `k+i` are assumed to be `[0, ..., u[i]-1]`.
  - Brouwer-Van Rees form: a list of length s such that `u[i]` is the list of pairs $(m_{i0}, |H_{i0}|), ..., (m_{ip_i}, |H_{ip_i}|)$. The untruncated points of column `k+i` are assumed to be $[0, ..., u_i -$

1] where $u_i = \sum_j |H_{ip_i}|$. Besides, the first $|H_{i0}|$ points represent H_{i0}, the next $|H_{i1}|$ points represent H_{i1}, etc...

- `explain_construction` (boolean) – return a string describing the construction.
- `check` (boolean) – whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.

REFERENCE:

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays import wilson_construction
sage: from sage.combinat.designs.orthogonal_arrays import OA_relabel
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_wilson_decompositi
sage: total = 0
sage: for k in range(3,8):
....:     for n in range(1,30):
....:         if find_wilson_decomposition_with_one_truncated_group(k,n):
....:             total += 1
....:             f, args = find_wilson_decomposition_with_one_truncated_group(k,n)
....:             _ = f(*args)
sage: print total
41

sage: print designs.orthogonal_arrays.explain_construction(7,58)
Wilson's construction n=8.7+1+1 with master design OA(7+2,8)
sage: print designs.orthogonal_arrays.explain_construction(9,115)
Wilson's construction n=13.8+11 with master design OA(9+1,13)
sage: print wilson_construction(None,5,11,21,[[(5,5)]],explain_construction=True)
Brouwer-van Rees construction n=11.21+(5.5) with master design OA(5+1,11)
sage: print wilson_construction(None,71,17,21,[[(4,9),(1,1)],[(9,9),(1,1)]],explain_construction
Brouwer-van Rees construction n=17.21+(9.4+1.1)+(9.9+1.1) with master design OA(71+2,17)
```

An example using the Brouwer-van Rees generalization:

```
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: from sage.combinat.designs.orthogonal_arrays import wilson_construction
sage: OA = designs.orthogonal_arrays.build(6,11)
sage: OA = [[x if (i<5 or x<5) else None for i,x in enumerate(R)] for R in OA]
sage: OAb = wilson_construction(OA,5,11,21,[[(5,5)]])
sage: is_orthogonal_array(OAb,5,256)
True
```

5.1.81 Orthogonal arrays (build recursive constructions)

This module implements several constructions of `Orthogonal Arrays`. As their input can be complex, they all have a counterpart in the `orthogonal_arrays_find_recursive` module that automatically computes it.

All these constructions are automatically queried when the `orthogonal_array()` function is called.

construction_3_3()	Return an $OA(k, nm+i)$.
construction_3_4()	Return a $OA(k, nm+rs)$.
construction_3_5()	Return an $OA(k, nm+r+s+t)$.
construction_3_6()	Return a $OA(k, nm+i)$.
construction_q_x()	Return an $OA(k, (q-1)*(q-x)+x+2)$ using the $q-x$ construction.
OA_and_oval()	Return a $OA(q+1, q)$ whose blocks contains ≤ 2 zeroes in the last q columns.
thwart_lemma_3_5()	Returns an $OA(k, nm+a+b+c+d)$.
thwart_lemma_4_1()	Returns an $OA(k, nm+4(n-2))$.
three_factor_product()	Returns an $OA(k+1, n_1n_2n_3)$.
brouwer_separable_design()	Returns a $OA(k, t(q^2+q+1)+x)$ using Brouwer's result on separable designs.

Functions

sage.combinat.designs.orthogonal_arrays_build_recursive.**OA_and_oval**(*q*)

Return a $OA(q+1, q)$ whose blocks contains ≤ 2 zeroes in the last q columns.

This OA is build from a projective plane of order q, in which there exists an oval O of size $q+1$ (i.e. a set of $q+1$ points no three of which are [colinear/contained in a common set of the projective plane]).

Removing an element $x \in O$ and all sets that contain it, we obtain a $TD(q+1, q)$ in which O intersects all columns except one. As O is an oval, no block of the TD intersects it more than twice.

INPUT:

- q – a prime power

Note: This function is called by construction_3_6(), an implementation of Construction 3.6 from [AC07].

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import OA_and_oval
sage: _ = OA_and_oval
```

sage.combinat.designs.orthogonal_arrays_build_recursive.**brouwer_separable_design**(*k*, *t*, *q*, *x*, *check=False*, *verbose=False*, *explain_constru*

Returns a $OA(k, t(q^2+q+1)+x)$ using Brouwer's result on separable designs.

This method is an implementation of Brouwer's construction presented in [Brouwer80]. It consists in a systematic application of the usual transformation from PBD to OA, applied to a specific PBD.

Baer subplanes

When q is a prime power, the projective plane $PG(2, q^2)$ can be partitionned into subplanes $PG(2, q)$ (called Baer subplanes), giving $PG(2, q^2) = B_1 \cup \cdots \cup B_{q^2-q+1}$. As a result, every line of the $PG(2, q^2)$ intersects one of the subplane on $q+1$ points and all others on 1 point.

The OA are built by considering $B_1 \cup \cdots \cup B_t$, for a total of $t(q^2+q+1)$ points (to which x new points are then added). The blocks of this subdesign belong to two categories:

- The blocks of size t: they come from the lines which intersect a B_i on $q+1$ points for some $i > t$. The blocks of size t can be partitionned into $q^2-q+t-1$ parallel classes according to their associated subplane B_i with $i > t$.
- The blocks of size $q+t$: those blocks form a symmetric design, as every point is incident with $q+t$ of them.

Constructions

In the following, we write $N = t(q^2+q+1)+x$. The code is also heavily commented, and will clear any doubt.

- i) $x = 0$: in that case we build a resolvable $OA(k-1, N)$ that will then be completed into an $OA(k, N)$.

  –*Sets of size t*)

  We take the product of each parallel class with the parallel classes of a resolvable $OA(k-1,t) - t.OA(k-1,t)$, yielding new parallel classes.

  –*Sets of size $q+t$*)

  A $N \times (q+t)$ array is built whose rows are the sets of size $q+t$ such that every value appears once per column. For each block of a $OA(k-1, q+t) - (q+t).OA(k-1,t)$, the product with the rows of the matrix yields a parallel class.

- 2.$x = q+t$

  –*Sets of size t*)

  Each set of size t gives a $OA(k,t) - t.OA(k,1)$, except if there is only one parallel class in which case a $OA(k,t)$ is sufficient.

  –*Sets of size $q+t$*)

  A $(N-x) \times (q+t)$ array M is built whose $N-x$ rows are the sets of size $q+t$ such that every value appears once per column. For each of the new $x = q+t$ points $p_1, \ldots, p_{q+t}$ we build a matrix M_i obtained from M by adding a column equal to $(p_i, p_i, p_i \ldots)$. We add to the OA the product of all rows of the M_i with the block of the $x = q+t$ parallel classes of a resolvable $OA(k, t+q+1) - (t+q+1).OA(k,1)$.

  –*Set of size x*) An $OA(k,x)$

- 3.$x = q^2 - q + 1 - t$

  –*Sets of size t*)

  All blocks of the i-th parallel class are extended with the i-th new point. The blocks are then replaced by a $OA(k, t+1) - (t+1).OA(k,1)$ or, if there is only one parallel class (i.e. $x = 1$) by a $OA(k, t+1) - OA(k,1)$.

  –*Set of size $q+t$*)

  They are replaced by $OA(k, q+t) - (q+t).OA(k,1)$.

  –*Set of size x*) An $OA(k,x)$

- 4.$x = q^2 + 1$

  –*Sets of size t*)

  All blocks of the i-th parallel class are extended with the i-th new point (the other $x-q-t$ new points are not touched at this step). The blocks are then replaced by a $OA(k, t+1) - (t+1).OA(k,1)$ or, if there is only one parallel class (i.e. $x = 1$) by a $OA(k, t+1) - OA(k,1)$.

–*Sets of size* $q+t$) Same as for ii)

–*Set of size* x) An $OA(k,x)$

- $22.0 < x < q^2 - q + 1 - t$

  –*Sets of size* t)

  The blocks of the first x parallel class are extended with the x new points, and replaced with $OA(k.t+1) - (t+1).OA(k,1)$ or, if $x = 1$, by $OA(k.t+1) - .OA(k,1)$

  The blocks of the other parallel classes are replaced by $OA(k,t) - t.OA(k,t)$ or, if there is only one class left, by $OA(k,t) - OA(k,t)$

  –*Sets of size* $q+t$)

  They are replaced with $OA(k,q+t) - (q+t).OA(k,1)$.

  –*Set of size* x) An $OA(k,x)$

- $6.t + q < x < q^2 + 1$

  –*Sets of size* t) Same as in v) with an x equal to $x - q + t$.

  –*Sets of size* t) Same as in vii)

  –*Set of size* x) An $OA(k,x)$

INPUT:

- `k,t,q,x` (integers)
- `check` – (boolean) Whether to check that output is correct before returning it. Set to `False` by default.
- `verbose` (boolean) – whether to print some information on the construction and parameters being used.
- `explain_construction` (boolean) – return a string describing the construction.

See also:

- `find_brouwer_separable_design()`

REFERENCES:

EXAMPLES:

Test all possible cases:

```
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import brouwer_separable_desi
sage: k,q,t=4,4,3; _=brouwer_separable_design(k,q,t,0,verbose=True)
Case i) with k=4,q=3,t=4,x=0
sage: k,q,t=3,3,3; _=brouwer_separable_design(k,t,q,t+q,verbose=True,check=True)
Case ii) with k=3,q=3,t=3,x=6,e3=1
sage: k,q,t=3,3,6; _=brouwer_separable_design(k,t,q,t+q,verbose=True,check=True)
Case ii) with k=3,q=3,t=6,x=9,e3=0
sage: k,q,t=3,3,6; _=brouwer_separable_design(k,t,q,q**2-q+1-t,verbose=True,check=True)
Case iii) with k=3,q=3,t=6,x=1,e2=0
sage: k,q,t=3,4,6; _=brouwer_separable_design(k,t,q,q**2-q+1-t,verbose=True,check=True)
Case iii) with k=3,q=4,t=6,x=7,e2=1
sage: k,q,t=3,4,6; _=brouwer_separable_design(k,t,q,q**2+1,verbose=True,check=True)
Case iv) with k=3,q=4,t=6,x=17,e4=1
sage: k,q,t=3,2,2; _=brouwer_separable_design(k,t,q,q**2+1,verbose=True,check=True)
Case iv) with k=3,q=2,t=2,x=5,e4=0
```

```
sage: k,q,t=3,4,7; _=brouwer_separable_design(k,t,q,3,verbose=True,check=True)
Case v) with k=3,q=4,t=7,x=3,e1=1,e2=1
sage: k,q,t=3,4,7; _=brouwer_separable_design(k,t,q,1,verbose=True,check=True)
Case v) with k=3,q=4,t=7,x=1,e1=1,e2=0
sage: k,q,t=3,4,7; _=brouwer_separable_design(k,t,q,q**2-q-t,verbose=True,check=True)
Case v) with k=3,q=4,t=7,x=5,e1=0,e2=1
sage: k,q,t=5,4,7; _=brouwer_separable_design(k,t,q,t+q+3,verbose=True,check=True)
Case vi) with k=5,q=4,t=7,x=14,e3=1,e4=1
sage: k,q,t=5,4,8; _=brouwer_separable_design(k,t,q,t+q+1,verbose=True,check=True)
Case vi) with k=5,q=4,t=8,x=13,e3=1,e4=0
sage: k,q,t=5,4,8; _=brouwer_separable_design(k,t,q,q**2,verbose=True,check=True)
Case vi) with k=5,q=4,t=8,x=16,e3=0,e4=1

sage: print designs.orthogonal_arrays.explain_construction(10,189)
Brouwer's separable design construction with t=9,q=4,x=0 from:
   Andries E. Brouwer,
   A series of separable designs with application to pairwise orthogonal Latin squares
   Vol. 1, n. 1, pp. 39-41,
   European Journal of Combinatorics, 1980
```

sage.combinat.designs.orthogonal_arrays_build_recursive.**construction_3_3**(*k*, *n*, *m*, *i*, *explain_construction=False*)

Return an $OA(k, nm + i)$.

This is Wilson's construction with i truncated columns of size 1 and such that a block B_0 of the incomplete OA intersects all truncated columns. As a consequence, all other blocks intersect only 0 or 1 of the last i columns. This allow to consider the block B_0 only up to its first k coordinates and then use a $OA(k, i)$ instead of a $OA(k, m + i) - i.OA(k, 1)$.

This is construction 3.3 from [AC07].

INPUT:

- `k,n,m,i` (integers) such that the following designs are available : $OA(k, n)$, $OA(k, m)$, $OA(k, m + 1)$, $OA(k, r)$.
- `explain_construction` (boolean) – return a string describing the construction.

See also:

`find_construction_3_3()`

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_construction_3_3
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import construction_3_3
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: k=11;n=177
sage: is_orthogonal_array(construction_3_3(*find_construction_3_3(k,n)[1]),k,n,2)
True

sage: print designs.orthogonal_arrays.explain_construction(9,91)
Construction 3.3 with n=11,m=8,i=3 from:
   Julian R. Abel, Nicholas Cavenagh
   Concerning eight mutually orthogonal latin squares,
   Vol. 15, n.3, pp. 255-261,
   Journal of Combinatorial Designs, 2007
```

sage.combinat.designs.orthogonal_arrays_build_recursive.**construction_3_4**(*k*, *n*, *m*, *r*, *s*, *explain_construction=False*)

Return a $OA(k, nm + rs)$.

This is Wilson's construction applied to a truncated $OA(k + r + 1, n)$ with r columns of size 1 and one column of size s.

The unique elements of the r truncated columns are picked so that a block B_0 contains them all.

- If there exists an $OA(k, m + r + 1)$ the column of size s is truncated in order to intersect B_0.
- Otherwise, if there exists an $OA(k, m + r)$, the last column must not intersect B_0

This is construction 3.4 from [AC07].

INPUT:

- `k,n,m,r,s` (integers) – we assume that $s < n$ and $1 \leq r, s$

  The following designs must be available: $OA(k, n)$, $OA(k, m)$, $OA(k, m + 1)$, $OA(k, m + 2)$, $OA(k, s)$. Additionnally, it requires either a $OA(k, m + r)$ or a $OA(k, m + r + 1)$.

- `explain_construction` (boolean) – return a string describing the construction.

See also:

`find_construction_3_4()`

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_construction_3_4
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import construction_3_4
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: k=8;n=196
sage: is_orthogonal_array(construction_3_4(*find_construction_3_4(k,n)[1]),k,n,2)
True

sage: print designs.orthogonal_arrays.explain_construction(8,164)
Construction 3.4 with n=23,m=7,r=2,s=1 from:
   Julian R. Abel, Nicholas Cavenagh
   Concerning eight mutually orthogonal latin squares,
   Vol. 15, n.3, pp. 255-261,
   Journal of Combinatorial Designs, 2007
```

sage.combinat.designs.orthogonal_arrays_build_recursive.**construction_3_5**(*k*, *n*, *m*, *r*, *s*, *t*, *explain_construction=False*)

Return an $OA(k, nm + r + s + t)$.

This is exactly Wilson's construction with three truncated groups except we make sure that all blocks have size $> k$, so we don't need a $OA(k, m + 0)$ but only $OA(k, m + 1)$, $OA(k, m + 2)$,'OA(k,m+3)'.

This is construction 3.5 from [AC07].

INPUT:

- `k,n,m` (integers)
- `r,s,t` (integers) – sizes of the three truncated groups, such that $r \leq s$ and $(q - r - 1)(q - s) \geq (q - s - 1) * (q - r)$.
- `explain_construction` (boolean) – return a string describing the construction.

The following designs must be available : $OA(k, n)$, $OA(k, r)$, $OA(k, s)$, $OA(k, t)$, $OA(k, m+1)$, $OA(k, m+2)$, $OA(k, m + 3)$.

See also:

`find_construction_3_5()`

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_construction_3_5
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import construction_3_5
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: k=8;n=111
sage: is_orthogonal_array(construction_3_5(*find_construction_3_5(k,n)[1]),k,n,2)
True

sage: print designs.orthogonal_arrays.explain_construction(8,90)
Construction 3.5 with n=11,m=6,r=8,s=8,t=8 from:
   Julian R. Abel, Nicholas Cavenagh
   Concerning eight mutually orthogonal latin squares,
   Vol. 15, n.3, pp. 255-261,
   Journal of Combinatorial Designs, 2007
```

sage.combinat.designs.orthogonal_arrays_build_recursive.**construction_3_6**(*k*, *n*, *m*, *i*, *explain_construction=False*)

Return a $OA(k, nm + i)$

This is Wilson's construction with r columns of order 1, in which each block intersects at most two truncated columns. Such a design exists when n is a prime power and is returned by `OA_and_oval()`.

INPUT:

- `k,n,m,i` (integers) – n must be a prime power. The following designs must be available: $OA(k + r, q)$, $OA(k, m)$, $OA(k, m + 1)$, $OA(k, m + 2)$.
- `explain_construction` (boolean) – return a string describing the construction.

This is construction 3.6 from [AC07].

See also:

- `find_construction_3_6()`
- `OA_and_oval()`

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_construction_3_6
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import construction_3_6
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: k=8;n=95
sage: is_orthogonal_array(construction_3_6(*find_construction_3_6(k,n)[1]),k,n,2)
True

sage: print designs.orthogonal_arrays.explain_construction(10,756)
Construction 3.6 with n=16,m=47,i=4 from:
   Julian R. Abel, Nicholas Cavenagh
   Concerning eight mutually orthogonal latin squares,
   Vol. 15, n.3, pp. 255-261,
   Journal of Combinatorial Designs, 2007
```

sage.combinat.designs.orthogonal_arrays_build_recursive.**construction_q_x**(*k*, *q*, *x*, *check=True*, *explain_construction=False*

Return an $OA(k,(q-1)*(q-x)+x+2)$ using the $q-x$ construction.

Let $v=(q-1)*(q-x)+x+2$. If there exists a projective plane of order q (e.g. when q is a prime power) and $0<x<q$ then there exists a $(v-1,\{q-x-1,q-x+1\})$-GDD of type $(q-1)^{q-x}(x+1)^1$ (see [Greig99] or Theorem 2.50, section IV.2.3 of [DesignHandbook]). By adding to the ground set one point contained in all groups of the GDD, one obtains a $(v,\{q-x-1,q-x+1,q,x+2\})$-PBD with exactly one set of size $x+2$.

Thus, assuming that we have the following:

- $OA(k,q-x-1)-(q-x-1).OA(k,1)$
- $OA(k,q-x+1)-(q-x+1).OA(k,1)$
- $OA(k,q)-q.OA(k,1)$
- $OA(k,x+2)$

Then we can build from the PBD an $OA(k,v)$.

Construction of the PBD (shared by Julian R. Abel):

> Start with a resolvable $(q^2,q,1)$-BIBD and put the points into a $q\times q$ array so that rows form a parallel class and columns form another.
>
> Now delete:
>
> - All $x(q-1)$ points from the first x columns and not in the first row
> - All $q-x$ points in the last $q-x$ columns AND the first row.
>
> Then add a point p_1 to the blocks that are rows. Add a second point p_2 to the $q-x$ blocks that are columns of size $q-1$, plus the first row of size $x+1$.

INPUT:

- `k, q, x` – integers such that $0<x<q$ and such that Sage can build:
  - A projective plane of order q
  - $OA(k,q-x-1)-(q-x-1).OA(k,1)$
  - $OA(k,q-x+1)-(q-x+1).OA(k,1)$

– $OA(k,q) - q.OA(k,1)$

– $OA(k,x+2)$

- `check` – (boolean) Whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.
- `explain_construction` (boolean) – return a string describing the construction.

See also:

- `find_q_x()`
- `projective_plane()`
- `orthogonal_array()`
- `OA_from_PBD()`

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import construction_q_x
sage: _ = construction_q_x(9,16,6)

sage: print designs.orthogonal_arrays.explain_construction(9,158)
(q-x)-construction with q=16,x=6 from:
   Malcolm Greig,
   Designs from projective planes and PBD bases,
   vol. 7, num. 5, pp. 341--374,
   Journal of Combinatorial Designs, 1999
```

REFERENCES:

sage.combinat.designs.orthogonal_arrays_build_recursive.**three_factor_product**(*k*, *n1*, *n2*, *n3*, *check=False*, *explain_construction=*

Returns an $OA(k+1, n_1n_2n_3)$

The three factor product construction from [DukesLing14] does the following:

> If $n_1 \leq n_2 \leq n_3$ are such that there exists an $OA(k,n_1)$, $OA(k+1,n_2)$ and $OA(k+1,n_3)$, then there exists a $OA(k+1,n_1n_2n_3)$.

It works with a modified product of orthogonal arrays ([Rees93], [Rees00]) which keeps track of parallel classes in the OA (the definition is given for transversal designs).

> A subset of blocks in an $TD(k,n)$ is called a c-parallel class if every point is covered exactly c times. A 1-parallel class is a parallel class.

The modified product:

> If there exists an $OA(k,n_1)$, and if there exists an $OA(k,n_2)$ whose blocks are partitionned into s n_1-parallel classes and $n_2 - sn_1$ parallel classes, then there exists an $OA(k,n_1n_2)$ whose blocks can be partitionned into sn_1^2 parallel classes and $(n_1n_2 - sn_1^2)/n_1 = n_2 - sn_1$ n_1-parallel classes.
>
> Proof:

- The product of the blocks of a parallel class with an $OA(k, n_1)$ yields an n_1-parallel class of an $OA(k, n_1 n_2)$.
- The product of the blocks of a n_1-parallel class of $OA(k, n_2)$ with an $OA(k, n_1)$ can be done in such a way that it yields $n_1 n_2$ parallel classes of $OA(k, n_1 n_2)$. Those classes cover exactly the pairs that woud have been covered with the usual product.

  This can be achieved by simple cyclic permutations. Let us build the product of the n_1-parallel class $\mathcal{P} \subseteq OA(k, n_2)$ with $OA(k, n_1)$: when computing the product of $P \in \mathcal{P}$ with $B^1 \in OA(k, n_1)$ the i-th coordinate should not be (B^1_i, P_i) but $(B^1_i + r, P_i)$ (the sum is mod n_1) where r is the number of blocks of $\mathcal{P}$ we have already processed whose i-th coordinate is equal to P_i (note that $r < n_1$ as $\mathcal{P}$ is n_1-parallel).

With these tools, one can obtain the designs promised by the three factors construction applied to k, n_1, n_2, n_3 (thanks to Julian R. Abel's help):

1. Let s be the largest integer $\leq n_3/n_1$. Apply the product construction to $OA(k, n_1)$ and a resolvable $OA(k, n_3)$ whose blocks are partitionned into s n_1-parallel classes and $n_3 - sn_1$ parallel classes. It results in a $OA(k, n_1 n_3)$ partitionned into sn_1^2 parallel classes plus $(n_1 n_3 - sn_1^2)/n_1 = n_3 - sn_1$ n_1-parallel classes.
2. Add $n_3 - n_1$ parallel classes to every n_1-parallel class to turn them into n_3-parallel classes. Apply the product construction to this partitionned $OA(k, n_1 n_3)$ with a resolvable $OA(k, n_2)$.
3. As $OA(k, n_2)$ is resolvable, the n_2-parallel classes of $OA(k, n_1 n_2 n_3)$ are actually the union of n_2 parallel classes, thus the $OA(k, n_1 n_2 n_3)$ is resolvable and can be turned into an $OA(k + 1, n_1 n_2 n_3)$

INPUT:

- `k,n1,n2,n3` (integers)
- `check` – (boolean) Whether to check that everything is going smoothly while the design is being built. It is disabled by default, as the constructor of orthogonal arrays checks the final design anyway.
- `explain_construction` (boolean) – return a string describing the construction.

See also:

- `find_three_factor_product()`

EXAMPLE:

```
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import three_factor_product

sage: OA = three_factor_product(4,4,4,4)
sage: is_orthogonal_array(OA,5,64)
True

sage: OA = three_factor_product(4,3,4,5)
sage: is_orthogonal_array(OA,5,60)
True

sage: OA = three_factor_product(5,4,5,7)
sage: is_orthogonal_array(OA,6,140)
True

sage: OA = three_factor_product(9,8,9,9) # long time
sage: is_orthogonal_array(OA,10,8*9*9)   # long time
True

sage: print designs.orthogonal_arrays.explain_construction(10,648)
```

```
Three-factor product with n=8.9.9 from:
   Peter J. Dukes, Alan C.H. Ling,
   A three-factor product construction for mutually orthogonal latin squares,
   http://arxiv.org/abs/1401.1466
```

REFERENCE:

sage.combinat.designs.orthogonal_arrays_build_recursive.**thwart_lemma_3_5**(*k*, *n*, *m*, *a*, *b*, *c*, *d=0*, *complement=False*, *explain_construction=False*

Returns an $OA(k, nm + a + b + c + d)$

(When 'd=0')

According to [Thwarts] when n is a prime power and $a + b + c \leq n + 1$, one can build an $OA(k + 3, n)$ with three truncated columns of sizes a, b, c in such a way that all blocks have size $\leq k + 2$.

(in order to build a $OA(k, nm+a+b+c)$ the following designs must also exist: $OA(k, a)$, $OA(k, b)$, $OA(k, c)$, $OA(k, m + 0)$, $OA(k, m + 1)$, $OA(k, m + 2)$)

Considering the complement of each truncated column, it is also possible to build an $OA(k + 3, n)$ with three truncated columns of sizes a, b, c in such a way that all blocks have size $> k$ whenever $(n-a)+(n-b)+(n-c) \leq n + 1$.

(in order to build a $OA(k, nm+a+b+c)$ the following designs must also exist: $OA(k, a)$, $OA(k, b)$, $OA(k, c)$, $OA(k, m + 1)$, $OA(k, m + 2)$, $OA(k, m + 3)$)

Here is the proof of Lemma 3.5 from [Thwarts] enriched with explanations from Julian R. Abel:

> For any prime power n one can build $k - 1$ MOLS by associating to every nonzero $x \in \mathbb{F}_n$ the latin square:
>
> $$M_x(i, j) = i + x * j \text{ where } i, j \in \mathbb{F}_n$$
>
> In particular $M_1(i, j) = i + j$, whose n columns and lines are indexed by the elements of $\mathbb{F}_n$. If we order the elements of $\mathbb{F}_n$ as $0, 1, ..., n - 1, x + 0, ..., x + n - 1, x^2 + 0, ...$ and reorder the columns and lines of M_1 accordingly, the top-left $a \times b$ squares contains at most $a + b - 1$ distinct symbols.

(When $d \neq 0$)

If there exists an $OA(k + 3, n)$ with three truncated columns of sizes a, b, c in such a way that all blocks have size $\leq k + 2$, by truncating arbitrarily another column to size d one obtains an OA with 4 truncated columns whose blocks miss at least one value. Thus, following the proof again one can build an $OA(k + 4)$ with four truncated columns of sizes a, b, c, d with blocks of size $\leq k + 3$.

(in order to build a $OA(k, nm + a + b + c + d)$ the following designs must also exist: $OA(k, a)$, $OA(k, b)$, $OA(k, c)$, $OA(k, d)$, $OA(k, m + 0)$, $OA(k, m + 1)$, $OA(k, m + 2)$, $OA(k, m + 3)$)

As before, this also shows that one can build an $OA(k + 4, n)$ with four truncated columns of sizes a, b, c, d in such a way that all blocks have size $> k$ whenever $(n - a) + (n - b) + (n - c) \leq n + 1$

(in order to build a $OA(k, nm+a+b+c+d)$ the following designs must also exist: $OA(k, n-a)$, $OA(k, n-b)$, $OA(k, n-c)$, $OA(k, d)$, $OA(k, m+1)$, $OA(k, m+2)$, $OA(k, m+3)$, $OA(k, m+4)$)

INPUT:

- `k, n, m, a, b, c, d` – integers which must satisfy the constraints above. In particular, $a+b+c \leq n+1$ must hold. By default, $d = 0$.
- `complement` (boolean) – whether to complement the sets, i.e. follow the $n-a, n-b, n-c$ variant described above.
- `explain_construction` (boolean) – return a string describing the construction.

See also:

- `find_thwart_lemma_3_5()`

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_build_recursive import thwart_lemma_3_5
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array
sage: OA = thwart_lemma_3_5(6,23,7,5,7,8)
sage: is_orthogonal_array(OA,6,23*7+5+7+8,2)
True

sage: print designs.orthogonal_arrays.explain_construction(10,408)
Lemma 4.1 with n=13,m=28 from:
   Charles J.Colbourn, Jeffrey H. Dinitz, Mieczyslaw Wojtas,
   Thwarts in transversal designs,
   Designs, Codes and Cryptography 5, no. 3 (1995): 189-197.
```

With sets of parameters from [Thwarts]:

```
sage: l = [
....:     [11, 27, 78, 16, 17, 25, 0],
....:     [12, 19, 208, 11, 13, 16, 0],
....:     [12, 19, 208, 13, 13, 16, 0],
....:     [10, 13, 78, 9, 9, 13, 1],
....:     [10, 13, 79, 9, 9, 13, 1]]
sage: for k,n,m,a,b,c,d in l:                                    # not tested -- too long
....:     OA = thwart_lemma_3_5(k,n,m,a,b,c,d,complement=True)   # not tested -- too long
....:     assert is_orthogonal_array(OA,k,n*m+a+b+c+d,verbose=True) # not tested -- too long

sage: print designs.orthogonal_arrays.explain_construction(10,1046)
Lemma 3.5 with n=13,m=79,a=9,b=1,c=0,d=9 from:
   Charles J.Colbourn, Jeffrey H. Dinitz, Mieczyslaw Wojtas,
   Thwarts in transversal designs,
   Designs, Codes and Cryptography 5, no. 3 (1995): 189-197.
```

REFERENCE:

sage.combinat.designs.orthogonal_arrays_build_recursive.**thwart_lemma_4_1**(*k*, *n*, *m*, *explain_construction=False*)

Returns an $OA(k, nm + 4(n-2))$.

Implements Lemma 4.1 from [Thwarts].

If $n \equiv 0, 1 \pmod 3$ is a prime power, then there exists a truncated $OA(n+1, n)$ whose last four columns have size $n-2$ and intersect every block on $1, 3$ or 4 values. Consequently, if there exists an $OA(k, m+1)$, $OA(k, m+3)$, $OA(k, m+4)$ and a $OA(k, n-2)$ then there exists an $OA(k, nm + 4(n-2)$

Proof: form the transversal design by removing one point of the $AG(2, 3)$ (Affine Geometry) contained in the Desarguesian Projective Plane $PG(2, n)$.

The affine geometry on 9 points contained in the projective geometry $PG(2, n)$ is given explicitly in [OS64] (Thanks to Julian R. Abel for finding the reference!).

INPUT:

- `k,n,m` (integers)
- `explain_construction` (boolean) – return a string describing the construction.

See also:

- `find_thwart_lemma_4_1()`

EXAMPLE:

```
sage: print designs.orthogonal_arrays.explain_construction(10,408)
Lemma 4.1 with n=13,m=28 from:
   Charles J.Colbourn, Jeffrey H. Dinitz, Mieczyslaw Wojtas,
   Thwarts in transversal designs,
   Designs, Codes and Cryptography 5, no. 3 (1995): 189-197.
```

REFERENCES:

5.1.82 Orthogonal arrays (find recursive constructions)

This module implements several functions to find recursive constructions of `Orthogonal Arrays`.

The main function of this module, i.e. `find_recursive_construction()`, queries all implemented recursive constructions of designs implemented in `orthogonal_arrays_build_recursive` in order to obtain an $OA(k, n)$.

`find_recursive_construction()` is called by the `orthogonal_array()` function.

find_recursive_construction()	Find a recursive construction of an $OA(k,n)$ (calls all others find_* functions)
find_product_decomposition()	Find $n_1n_2 = n$ to obtain an $OA(k,n)$ by the product construction
find_wilson_decomposition_with_one_truncated_column()	Find $rm + u = n$ to obtain an $OA(k,n)$ by Wilson's construction with one truncated column.
find_wilson_decomposition_with_two_truncated_columns()	Find $rm + r_2 = n$ to obtain an $OA(k,n)$ by Wilson's construction with two truncated columns.
find_construction_3_3()	Find a decomposition for construction 3.3 from [AC07].
find_construction_3_4()	Find a decomposition for construction 3.4 from [AC07].
find_construction_3_5()	Find a decomposition for construction 3.5 from [AC07].
find_construction_3_6()	Find a decomposition for construction 3.6 from [AC07].
find_q_x()	Find integers q, x such that the $q - x$ construction yields an $OA(k,n)$.
find_thwart_lemma_3_5()	Find the values on which Lemma 3.5 from [Thwarts] applies.
find_thwart_lemma_4_1()	Find a decomposition for Lemma 4.1 from [Thwarts].
find_three_factor_product()	Find $n_1n_2n_3 = n$ to obtain an $OA(k,n)$ by the three-factor product from [DukesLing14]
find_brouwer_separable_design()	Find $t(q^2+q+1)+x = n$ to obtain an $OA(k,n)$ by Brouwer's separable design construction.
find_brouwer_van_rees_with_one_truncated_column()	Find $rm + x_1 + ... + x_c = n$ such that the Brouwer-van Rees constructions yields a $OA(k,n)$.

REFERENCES:

Functions

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_brouwer_separable_design**(*k*, *n*)

Find $t(q^2+q+1)+x = n$ to obtain an $OA(k,n)$ by Brouwer's separable design construction.

INPUT:

- k, n (integers)

The assumptions made on the parameters t, q, x are explained in the documentation of brouwer_separable_design().

EXAMPLE:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_brouwer_separable_
sage: find_brouwer_separable_design(5,13)[1]
(5, 1, 3, 0)
sage: find_brouwer_separable_design(5,14)
False
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_brouwer_van_rees_with_one_trunc**

Find $rm + x_1 + ... + x_c = n$ such that the Brouwer-van Rees constructions yields a $OA(k,n)$.

Let $n = rm + \sum_{1 \leq i \leq c}$ such that $c \leq r$. The generalization of Wilson's construction found by Brouwer and van Rees (with one truncated column) ensures that an $OA(k,n)$ exists if the following designs exist: $OA(k+1,r)$, $OA(k,m)$, $OA(k, \sum_{1 \leq i \leq c} u_i)$, $OA(k, m+x_1) - OA(k,x_1)$, ..., $OA(k, m+x_c) - OA(k,x_c)$.

For more information, see the documentation of wilson_construction().

INPUT:

- k, n (integers)

EXAMPLE:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_brouwer_van_rees_w
sage: find_brouwer_van_rees_with_one_truncated_column(5,53)[1]
(None, 5, 7, 7, [[(2, 1), (2, 1)]])
sage: find_brouwer_van_rees_with_one_truncated_column(6,96)[1]
(None, 6, 7, 13, [[(3, 1), (1, 1), (1, 1)]])
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_construction_3_3**(*k*, *n*)

Find a decomposition for construction 3.3 from [AC07]

INPUT:

- k, n (integers)

See also:

construction_3_3()

OUTPUT:

A pair f, args such that f(*args) returns the requested OA.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_construction_3_3
sage: find_construction_3_3(11,177)[1]
(11, 11, 16, 1)
sage: find_construction_3_3(12,11)
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_construction_3_4**(*k*, *n*)

Find a decomposition for construction 3.4 from [AC07]

INPUT:

- k, n (integers)

See also:

construction_3_4()

OUTPUT:

A pair f, args such that f(*args) returns the requested OA.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_construction_3_4
sage: find_construction_3_4(8,196)[1]
(8, 25, 7, 12, 9)
sage: find_construction_3_4(9,24)
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_construction_3_5**(*k*, *n*)

Find a decomposition for construction 3.5 from [AC07]

INPUT:

- k, n (integers)

See also:

construction_3_5()

OUTPUT:

A pair `f,args` such that `f(*args)` returns the requested OA.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_construction_3_5
sage: find_construction_3_5(8,111)[1]
(8, 13, 6, 9, 11, 13)
sage: find_construction_3_5(9,24)
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_construction_3_6**(*k*, *n*)

Find a decomposition for construction 3.6 from [AC07]

INPUT:

- `k,n` (integers)

See also:

`construction_3_6()`

OUTPUT:

A pair `f,args` such that `f(*args)` returns the requested OA.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_construction_3_6
sage: find_construction_3_6(8,95)[1]
(8, 13, 7, 4)
sage: find_construction_3_6(8,98)
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_product_decomposition**(*k*, *n*)

Find $n_1 n_2 = n$ to obtain an $OA(k,n)$ by the product construction.

If Sage can build a $OA(k,n_1)$ and a $OA(k,n_2)$ such that $n = n_1 \times n_2$ then a $OA(k,n)$ can be built by a product construction (which correspond to Wilson's construction with no truncated column). This function look for a pair of integers (n_1,n_2) with $n1 \leq n_2$, $n_1 \times n_2 = n$ and such that both an $OA(k,n_1)$ and an $OA(k,n_2)$ are available.

INPUT:

- `k,n` (integers) – see above.

OUTPUT:

A pair `f,args` such that `f(*args)` is an $OA(k,n)$ or `False` if no product decomposition was found.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_product_decomposit
sage: f,args = find_product_decomposition(6, 84)
sage: args
(None, 6, 7, 12, (), False)
sage: _ = f(*args)
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_q_x**(*k*, *n*)

Find integers q,x such that the $q-x$ construction yields an $OA(k,n)$.

See the documentation of `construction_q_x()` to find out what hypotheses the integers q,x must satisfy.

Warning: For efficiency reasons, this function checks that Sage can build an $OA(k+1, q-x-1)$ and an $OA(k+1, q-x+1)$, which is stronger than checking that Sage can build a $OA(k, q-x-1) - (q-x-1).OA(k,1)$ and a $OA(k, q-x+1) - (q-x+1).OA(k,1)$. The latter would trigger a lot of independent set computations in `sage.combinat.designs.orthogonal_arrays.incomplete_orthogonal_array()`.

INPUT:

- `k, n` (integers)

See also:

`construction_q_x()`

EXAMPLE:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_q_x
sage: find_q_x(10,9)
False
sage: find_q_x(9,158)[1]
(9, 16, 6)
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_recursive_construction**(*k*, *n*)

Find a recursive construction of an $OA(k,n)$ (calls all others `find_*` functions)

This determines whether an $OA(k,n)$ can be built through the following constructions:

- `wilson_construction()`
- `construction_3_3()`
- `construction_3_4()`
- `construction_3_5()`
- `construction_3_6()`
- `construction_q_x()`
- `thwart_lemma_3_5()`
- `thwart_lemma_4_1()`
- `three_factor_product()`
- `brouwer_separable_design()`

INPUT:

- `k, n` (integers)

OUTPUT:

Return a pair `f, args` such that `f(*args)` returns the requested OA if possible, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_recursive_construc
sage: from sage.combinat.designs.orthogonal_arrays import is_orthogonal_array
sage: count = 0
sage: for n in range(10,150):
....:     k = designs.orthogonal_arrays.largest_available_k(n)
....:     if find_recursive_construction(k,n):
....:         count = count + 1
....:         f,args = find_recursive_construction(k,n)
```

```
....:         OA = f(*args)
....:         assert is_orthogonal_array(OA,k,n,2,verbose=True)
sage: print count
56
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_three_factor_product**(*k*, *n*)

Find $n_1n_2n_3 = n$ to obtain an $OA(k,n)$ by the three-factor product from [DukesLing14]

INPUT:

- k, n (integers)

See also:

three_factor_product()

OUTPUT:

A pair f, args such that f(*args) returns the requested OA.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_three_factor_produ
sage: find_three_factor_product(10,648)[1]
(9, 8, 9, 9)
sage: find_three_factor_product(10,50)
False
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_thwart_lemma_3_5**(*k*, *N*)

Find the values on which Lemma 3.5 from [Thwarts] applies.

OUTPUT:

A pair (f, args) such that f(*args) returns an $OA(k,n)$ or False if the construction is not available.

See also:

thwart_lemma_3_5()

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_thwart_lemma_3_5
sage: from sage.combinat.designs.designs_pyx import is_orthogonal_array

sage: f,args = find_thwart_lemma_3_5(7,66)
sage: args
(7, 9, 7, 1, 1, 1, 0, False)
sage: OA = f(*args)
sage: is_orthogonal_array(OA,7,66,2)
True

sage: f,args = find_thwart_lemma_3_5(6,100)
sage: args
(6, 8, 10, 8, 7, 5, 0, True)
sage: OA = f(*args)
sage: is_orthogonal_array(OA,6,100,2)
True
```

Some values from [Thwarts]:

```
sage: kn = ((10,1046), (10,1048), (10,1059), (11,1524),
....:       (11,2164), (12,3362), (12,3992),  (12,3994))
```

```
sage: for k,n in kn:
....:     print k,n,find_thwart_lemma_3_5(k,n)[1]
10 1046 (10, 13, 79, 9, 1, 0, 9, False)
10 1048 (10, 13, 79, 9, 1, 0, 11, False)
10 1059 (10, 13, 80, 9, 1, 0, 9, False)
11 1524 (11, 19, 78, 16, 13, 13, 0, True)
11 2164 (11, 27, 78, 23, 19, 16, 0, True)
12 3362 (12, 16, 207, 13, 13, 11, 13, True)
12 3992 (12, 19, 207, 16, 13, 11, 19, True)
12 3994 (12, 19, 207, 16, 13, 13, 19, True)

sage: for k,n in kn:                                                    # not tested -- too lon
....:     assert designs.orthogonal_array(k,n,existence=True) is True   # not tested -- too lon
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_thwart_lemma_4_1**(*k*, *n*)

Find a decomposition for Lemma 4.1 from [Thwarts].

INPUT:

- `k,n` (integers)

See also:

`thwart_lemma_4_1()`

OUTPUT:

A pair `f,args` such that `f(*args)` returns the requested OA.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_thwart_lemma_4_1
sage: find_thwart_lemma_4_1(10,408)[1]
(10, 13, 28)
sage: find_thwart_lemma_4_1(10,50)
False
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_wilson_decomposition_with_one_t**

Find $rm + u = n$ to obtain an $OA(k, n)$ by Wilson's construction with one truncated column.

This function looks for possible integers m, t, u satisfying that $mt + u = n$ and such that Sage knows how to build a $OA(k, m)$, $OA(k, m + 1)$, $OA(k + 1, t)$ and a $OA(k, u)$.

INPUT:

- `k,n` (integers) – see above

OUTPUT:

A pair `f,args` such that `f(*args)` is an $OA(k, n)$ or `False` if no decomposition with one truncated block was found.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_wilson_decompositi
sage: f,args = find_wilson_decomposition_with_one_truncated_group(4,38)
sage: args
(None, 4, 5, 7, (3,), False)
sage: _ = f(*args)

sage: find_wilson_decomposition_with_one_truncated_group(4,20)
False
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**find_wilson_decomposition_with_two_t**

Find $rm + r_1 + r_2 = n$ to obtain an $OA(k, n)$ by Wilson's construction with two truncated columns.

Look for integers r, m, r_1, r_2 satisfying $n = rm + r_1 + r_2$ and $1 \leq r_1, r_2 < r$ and such that the following designs exist : $OA(k + 2, r)$, $OA(k, r1)$, $OA(k, r2)$, $OA(k, m)$, $OA(k, m + 1)$, $OA(k, m + 2)$.

INPUT:

- `k, n` (integers) – see above

OUTPUT:

A pair `f, args` such that `f(*args)` is an $OA(k, n)$ or `False` if no decomposition with two truncated blocks was found.

EXAMPLES:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import find_wilson_decompositi
sage: f,args = find_wilson_decomposition_with_two_truncated_groups(5,58)
sage: args
(None, 5, 7, 7, (4, 5), False)
sage: _ = f(*args)
```

sage.combinat.designs.orthogonal_arrays_find_recursive.**int_as_sum**(*value*, *S*, *k_max*)

Return a tuple $(s_1, s_2, \ldots, s_k)$ of less then k_max elements of S such that $value = s_1 + s_2 + \ldots + s_k$. If there is no such tuples then the function returns `None`.

INPUT:

- `value` (integer)
- `S` – a list of integers
- `k_max` (integer)

EXAMPLE:

```
sage: from sage.combinat.designs.orthogonal_arrays_find_recursive import int_as_sum
sage: D = int_as_sum(21,[5,12],100)
sage: for k in range(20,40):
....:     print k, int_as_sum(k,[5,12],100)
20 (5, 5, 5, 5)
21 None
22 (12, 5, 5)
23 None
24 (12, 12)
25 (5, 5, 5, 5, 5)
26 None
27 (12, 5, 5, 5)
28 None
29 (12, 12, 5)
30 (5, 5, 5, 5, 5, 5)
31 None
32 (12, 5, 5, 5, 5)
33 None
34 (12, 12, 5, 5)
35 (5, 5, 5, 5, 5, 5, 5)
36 (12, 12, 12)
37 (12, 5, 5, 5, 5, 5)
```

```
38 None
39 (12, 12, 5, 5, 5)
```

5.1.83 Steiner Quadruple Systems

A Steiner Quadruple System on n points is a family $SQS_n \subset \binom{[n]}{4}$ of 4-sets, such that any set $S \subset [n]$ of size three is a subset of exactly one member of SQS_n.

This module implements Haim Hanani's constructive proof that a Steiner Quadruple System exists if and only if $n \equiv 2, 4 \pmod 6$. Hanani's proof consists in 6 different constructions that build a large Steiner Quadruple System from a smaller one, and though it does not give a very clear understanding of why it works (to say the least)... it does !

The constructions have been implemented while reading two papers simultaneously, for one of them sometimes provides the informations that the other one does not. The first one is Haim Hanani's original paper [Hanani60], and the other one is a paper from Horan and Hurlbert which goes through all constructions [HH12].

It can be used through the `designs` object:

```
sage: designs.steiner_quadruple_system(8)
Incidence structure with 8 points and 14 blocks
```

REFERENCES:

AUTHORS:

- Nathann Cohen (May 2013, while listening to "*Le Blues Du Pauvre Delahaye*")

Index

This module's main function is the following :

	`steiner_quadruple_system()`	Return a Steiner Quadruple System on n points

This function redistributes its work among 6 constructions :

Construction 1	`two_n()`	Return a Steiner Quadruple System on $2n$ points
Construction 2	`three_n_minus_two()`	Return a Steiner Quadruple System on $3n-2$ points
Construction 3	`three_n_minus_eight()`	Return a Steiner Quadruple System on $3n-8$ points
Construction 4	`three_n_minus_four()`	Return a Steiner Quadruple System on $3n-4$ points
Construction 5	`four_n_minus_six()`	Return a Steiner Quadruple System on $4n-6$ points
Construction 6	`twelve_n_minus_ten()`	Return a Steiner Quadruple System on $12n-10$ points

It also defines two specific Steiner Quadruple Systems that the constructions require, i.e. SQS_{14} and SQS_{38} as well as the systems of pairs $P_\alpha(m)$ and $\overline{P}_\alpha(m)$ (see [Hanani60]).

Functions

`sage.combinat.designs.steiner_quadruple_systems.P`(*alpha*, *m*)

Return the collection of pairs $P_\alpha(m)$

For more information on this system, see [Hanani60].

EXAMPLE:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import P
sage: P(3,4)
[(0, 5), (2, 7), (4, 1), (6, 3)]
```

sage.combinat.designs.steiner_quadruple_systems.**barP**(*eps*, *m*)
Return the collection of pairs $\overline{P}_\alpha(m)$

For more information on this system, see [Hanani60].

EXAMPLE:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import barP
sage: barP(3,4)
[(0, 4), (3, 5), (1, 2)]
```

sage.combinat.designs.steiner_quadruple_systems.**barP_system**(*m*)
Return the 1-factorization of K_{2m} $\overline{P}(m)$

For more information on this system, see [Hanani60].

EXAMPLE:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import barP_system
sage: barP_system(3)
[[(4, 3), (2, 5)],
[(0, 5), (4, 1)],
[(0, 2), (1, 3)],
[(1, 5), (4, 2), (0, 3)],
[(0, 4), (3, 5), (1, 2)],
[(0, 1), (2, 3), (4, 5)]]
```

sage.combinat.designs.steiner_quadruple_systems.**four_n_minus_six**(*B*)
Return a Steiner Quadruple System on $4n-6$ points.

INPUT:

- B – A Steiner Quadruple System on n points.

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import four_n_minus_six
sage: for n in xrange(4, 20):
....:     if (n%6) in [2,4]:
....:         sqs = designs.steiner_quadruple_system(n)
....:         if not four_n_minus_six(sqs).is_t_design(3,4*n-6,4,1):
....:             print "Something is wrong !"
```

sage.combinat.designs.steiner_quadruple_systems.**relabel_system**(*B*)
Relabels the set so that $\{n-4, n-3, n-2, n-1\}$ is in B.

INPUT:

- B – a list of 4-uples on $0, ..., n-1$.

EXAMPLE:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import relabel_system
sage: SQS8 = designs.steiner_quadruple_system(8)
sage: relabel_system(SQS8)
Incidence structure with 8 points and 14 blocks
```

sage.combinat.designs.steiner_quadruple_systems.**steiner_quadruple_system**(*n*, *check=False*)
Return a Steiner Quadruple System on n points.

INPUT:

- n – an integer such that $n \equiv 2, 4 \pmod 6$

- check (boolean) – whether to check that the system is a Steiner Quadruple System before returning it ($False$ by default)

EXAMPLES:

```
sage: sqs4 = designs.steiner_quadruple_system(4)
sage: sqs4
Incidence structure with 4 points and 1 blocks
sage: sqs4.is_t_design(3,4,4,1)
True

sage: sqs8 = designs.steiner_quadruple_system(8)
sage: sqs8
Incidence structure with 8 points and 14 blocks
sage: sqs8.is_t_design(3,8,4,1)
True
```

TESTS:

```
sage: for n in xrange(4, 100):                                        # long time
....:     if (n%6) in [2,4]:                                          # long time
....:         sqs = designs.steiner_quadruple_system(n, check=True) # long time
```

sage.combinat.designs.steiner_quadruple_systems.**three_n_minus_eight**(B)

Return a Steiner Quadruple System on $3n-8$ points.

INPUT:

- B – A Steiner Quadruple System on n points.

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import three_n_minus_eight
sage: for n in xrange(4, 30):
....:     if (n%12) == 2:
....:         sqs = designs.steiner_quadruple_system(n)
....:         if not three_n_minus_eight(sqs).is_t_design(3,3*n-8,4,1):
....:             print "Something is wrong !"
```

sage.combinat.designs.steiner_quadruple_systems.**three_n_minus_four**(B)

Return a Steiner Quadruple System on $3n-4$ points.

INPUT:

- B – A Steiner Quadruple System on n points where $n \equiv 10 \pmod{12}$.

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import three_n_minus_four
sage: for n in xrange(4, 30):
....:     if n%12 == 10:
....:         sqs = designs.steiner_quadruple_system(n)
....:         if not three_n_minus_four(sqs).is_t_design(3,3*n-4,4,1):
....:             print "Something is wrong !"
```

sage.combinat.designs.steiner_quadruple_systems.**three_n_minus_two**(B)

Return a Steiner Quadruple System on $3n-2$ points.

INPUT:

- B – A Steiner Quadruple System on n points.

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import three_n_minus_two
sage: for n in xrange(4, 30):
....:     if (n%6) in [2,4]:
....:         sqs = designs.steiner_quadruple_system(n)
....:         if not three_n_minus_two(sqs).is_t_design(3,3*n-2,4,1):
....:             print "Something is wrong !"
```

sage.combinat.designs.steiner_quadruple_systems.**twelve_n_minus_ten**(*B*)

Return a Steiner Quadruple System on $12n - 6$ points.

INPUT:

- B – A Steiner Quadruple System on n points.

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import twelve_n_minus_ten
sage: for n in xrange(4, 15):
....:     if (n%6) in [2,4]:
....:         sqs = designs.steiner_quadruple_system(n)
....:         if not twelve_n_minus_ten(sqs).is_t_design(3,12*n-10,4,1):
....:             print "Something is wrong !"
```

sage.combinat.designs.steiner_quadruple_systems.**two_n**(*B*)

Return a Steiner Quadruple System on $2n$ points.

INPUT:

- B – A Steiner Quadruple System on n points.

EXAMPLES:

```
sage: from sage.combinat.designs.steiner_quadruple_systems import two_n
sage: for n in xrange(4, 30):
....:     if (n%6) in [2,4]:
....:         sqs = designs.steiner_quadruple_system(n)
....:         if not two_n(sqs).is_t_design(3,2*n,4,1):
....:             print "Something is wrong !"
```

5.1.84 Hypergraph isomorphic copy search

This module implements a code for the following problem:

INPUT: two hypergraphs H_1, H_2

OUTPUT: a copy of H_2 in H_1

It is also possible to enumerate all such copies, and to require that such copies be induced copies. More formally:

A copy of H_2 in H_1 is an injection $f : V(H_2) \mapsto V(H_1)$ such that for any set $S_2 \in E(H_2)$ we have $f(S_2) \in E(H_1)$.

It is an *induced* copy if no other set of $E(H_1)$ is contained in $f(V(H_2))$, i.e. $|E(H_2)| = \{S : S \in E(H_1)$ and $S \subseteq f(V(H_2))\}$.

The functions implemented here lists all such injections. In particular, the number of copies of H in itself is equal to $|Aut(H)|$.

The feature is available through `IncidenceStructure.isomorphic_substructures_iterator()`.

Implementation

A hypergraph is stored as a list of edges, each of which is a "dense" bitset over $|V(H_1)|$ points. In particular, two sets of distinct cardinalities require the same memory space. A hypergraph is a C struct with the following fields:

- `n,m` (`int`) – number of points and edges.
- `limbs` (`int`) – number of 64-bits blocks per set.
- `set_space` (`uint64_t *`) – address of the memory used to store the sets.
- `sets` (`uint64_t **`) – `sets[i]` points toward the `limbs` blocks encoding set i. Note also that `sets[i][limbs]` is equal to the cardinality of `set[i]`, so that `sets` has lenth `m*(limbs+1)*sizeof(uint64_t)`.
- `names` (`int *`) – associates an integer 'name' to each of the `n` points.

The operations used on this data structure are:

- `void permute(hypergraph * h, int n1, int n2)` – exchanges points $n1$ and $n2$ in the data structure. Note that their names are also exchanged so that we still know which is which.
- `int induced_hypergraph(hypergraph * h1, int n, hypergraph * tmp1)` – stores in `tmp1` the hypergraph induced by the first n points, i.e. all sets S such that $S \subseteq \{0, ..., n-1\}$. The function returns the number of such sets.
- `void trace_hypergraph64(hypergraph * h, int n, hypergraph * tmp)` – stores in `tmp1` the trace of h on the first n points, i.e. all sets of the form $S \cap \{0, ..., n-1\}$.

Algorithm

We try all possible assignments of a representant $r_i \in H_1$ for every $i \in H_2$. When we have picked a representant for the first $n <$ points $\{0, ..., n-1\} \subsetneq V(H_2)$, we check that:

- The hypergraph induced by the (ordered) list $0, ..., n-1$ in H_2 is equal to the one induced by $r_0, ..., r_{n-1}$ in H_1.
- If $S \subseteq \{0, ..., n-1\}$ is contained in c sets of size k in H_2, then $\{r_i : i \in S\}$ is contained in $\geq c$ sets of size k in H_1. This is done by comparing the trace of the hypergraphs while remembering the original size of each set.

As we very often need to build the hypergraph obtained by the trace of the first n points (for all possible n), those hypergraphs are cached. The hypergraphs induced by the same points are handled similarly.

Limitations

Number of points For efficiency reason the implementation assumes that H_2 has ≤ 64 points. Making this work for larger values means that calls to `qsort` have to be replaced by calls to `qsort_r` (i.e. to sort the edges you need to know the number of limbs per edge) and that induces a big slowdown for small cases (~50% when this code was implemented). Also, 64 points for H_2 is already very very big considering the problem at hand. Even $|V(H_1)| > 64$ seems too much.

Vertex ordering The order of vertices in H_2 has a huge influence on the performance of the algorithm. If no set of H_2 contains more that one of the first $k < n$ points, then almost all partial assignments of representants are possible for the first k points (though the degree of the vertices is taken into account). For this reason it is best to pick an ordering such that the first vertices are contained in as many sets as possible together. A heuristic is implemented at `relabel_heuristic()`.

AUTHORS:

- Nathann Cohen (November 2014, written in various airports between Nice and Chennai).

Methods

class sage.combinat.designs.subhypergraph_search.**SubHypergraphSearch**
Bases: object

relabel_heuristic()
Relabels H_2 in order to make the algorithm faster.

Objective: we try to pick an ordering $p_1, ..., p_k$ of the points of H_2 that maximizes the number of sets involving the first points in the ordering. One way to formalize the problems indicates that it may be NP-Hard (generalizes the max clique problem for graphs) so we do not try to solve it exactly: we just need a sufficiently good heuristic.

Assuming that the first points are $p_1, ..., p_k$, we determine p_{k+1} as the point x such that the number of sets S with $x \in S$ and $S \cap \{p_1, ..., p_k\} \neq \emptyset$ is maximal. In case of ties, we take a point with maximum degree.

This function is called when an instance of SubHypergraphSearch is created.

EXAMPLE:

```
sage: d = designs.projective_plane(3)
sage: d.isomorphic_substructures_iterator(d).relabel_heuristic()
```

5.1.85 Two-graphs

A two-graph on n points is a family $T \subset \binom{[n]}{3}$ of 3-sets, such that any 4-set $S \subset [n]$ of size four contains an even number of elements of T. Any graph $([n], E)$ gives rise to a two-graph $T(E) = \{t \in \binom{[n]}{3} : \left|\binom{t}{2} \cap E\right| \ odd\}$, and any two graphs with the same two-graph can be obtained one from the other by Seidel switching. This defines an equivalence relation on the graphs on $[n]$, called Seidel switching equivalence. Conversely, given a two-graph T, one can construct a graph Γ in the corresponding Seidel switching class with an isolated vertex w. The graph $\Gamma \setminus w$ is called the descendant of T w.r.t. v.

T is called regular if each two-subset of $[n]$ is contained in the same number alpha of triples of T.

This module implements a direct construction of a two-graph from a list of triples, constrution of descendant graphs, regularity checking, and other things such as constructing the complement two-graph, cf. [BH12].

AUTHORS:

- Dima Pasechnik (Aug 2015)

Index

This module's methods are the following :

is_regular_twograph()	tests if self is a regular two-graph, i.e. a 2-design
complement()	returns the complement of self
descendant()	returns the descendant graph at w

This module's functions are the following :

taylor_twograph()	constructs Taylor's two-graph for $U_3(q)$
is_twograph()	checks that the incidence system is a two-graph
twograph_descendant()	returns the descendant graph w.r.t. a given vertex of the two-graph of a given graph

Methods

class sage.combinat.designs.twographs.**TwoGraph**(*points=None*, *blocks=None*, *incidence_matrix=None*, *name=None*, *check=False*, *copy=True*)

Bases: sage.combinat.designs.incidence_structures.IncidenceStructure

Two-graphs class.

A two-graph on n points is a 3-uniform hypergraph, i.e. a family $T \subset \binom{[n]}{3}$ of 3-sets, such that any 4-set $S \subset [n]$ of size four contains an even number of elements of T. For more information, see the documentation of the twographs module.

complement()

The two-graph which is the complement of self

That is, the two-graph constisting exactly of triples not in self. Note that this is different from complement of the parent class.

EXAMPLES:

```
sage: p=graphs.CompleteGraph(8).line_graph().twograph()
sage: pc = p.complement(); pc
Incidence structure with 28 points and 1260 blocks
```

TESTS:

```
sage: from sage.combinat.designs.twographs import is_twograph
sage: is_twograph(pc)
True
```

descendant(*v*)

The descendant graph at v

The switching class of graphs corresponding to self contains a graph D with v its own connected component; removing v from D, one obtains the descendant graph of self at v, which is constructed by this method.

INPUT:

- v – an element of ground_set()

EXAMPLES:

```
sage: p=graphs.PetersenGraph().twograph().descendant(0)
sage: p.is_strongly_regular(parameters=True)
(9, 4, 1, 2)
```

is_regular_twograph(*alpha=False*)

Tests if the TwoGraph is regular, i.e. is a 2-design.

Namely, each pair of elements of ground_set() is contained in exactly alpha triples.

INPUT:

- alpha – (optional, default is False) return the value of alpha, if possible.

EXAMPLES:

```
sage: p=graphs.PetersenGraph().twograph()
sage: p.is_regular_twograph(alpha=True)
4
sage: p.is_regular_twograph()
True
```

```
sage: p=graphs.PathGraph(5).twograph()
sage: p.is_regular_twograph(alpha=True)
False
sage: p.is_regular_twograph()
False
```

sage.combinat.designs.twographs.**is_twograph**(*T*)

Checks that the incidence system T is a two-graph

INPUT:

- T – an incidence structure

EXAMPLES:

a two-graph from a graph:

```
sage: from sage.combinat.designs.twographs import (is_twograph, TwoGraph)
sage: p=graphs.PetersenGraph().twograph()
sage: is_twograph(p)
True
```

a non-regular 2-uniform hypergraph which is a two-graph:

```
sage: is_twograph(TwoGraph([[1,2,3],[1,2,4]]))
True
```

TESTS:

wrong size of blocks:

```
sage: is_twograph(designs.projective_plane(3))
False
```

a triple system which is not a two-graph:

```
sage: is_twograph(designs.projective_plane(2))
False
```

sage.combinat.designs.twographs.**taylor_twograph**(*q*)

constructing Taylor's two-graph for $U_3(q)$, q odd prime power

The Taylor's two-graph T has the q^3+1 points of the projective plane over F_{q^2} singular w.r.t. the non-degenerate Hermitean form S preserved by $U_3(q)$ as its ground set; the triples are $\{x,y,z\}$ satisfying the condition that $S(x,y)S(y,z)S(z,x)$ is square (respectively non-square) if $q \cong 1 \mod 4$ (respectively if $q \cong 3 \mod 4$). See §7E of [BvL84].

There is also a $2-(q^3+1,q+1,1)$-design on these q^3+1 points, known as the unital of order q, also invariant under $U_3(q)$.

INPUT:

- q – a power of an odd prime

EXAMPLES:

```
sage: from sage.combinat.designs.twographs import taylor_twograph
sage: T=taylor_twograph(3); T
Incidence structure with 28 points and 1260 blocks
```

sage.combinat.designs.twographs.**twograph_descendant**(*G*, *v*, *name=None*)

Returns the descendant graph w.r.t. vertex v of the two-graph of G

In the switching class of G, construct a graph Δ with v an isolated vertex, and return the subgraph $\Delta \setminus v$. It is equivalent to, although much faster than, computing the TwoGraph.descendant() of two-graph of G, as the intermediate two-graph is not constructed.

INPUT:

- G – a graph
- v – a vertex of G
- name – (optional) None - no name, otherwise derive from the construction

EXAMPLES:

one of s.r.g.'s from the database:

```
sage: from sage.combinat.designs.twographs import twograph_descendant
sage: A=graphs.strongly_regular_graph(280,135,70)                    # optional - gap_packages i
sage: twograph_descendant(A, 0).is_strongly_regular(parameters=True) # optional - gap_packages i
(279, 150, 85, 75)
```

TESTS:

```
sage: T8 = graphs.CompleteGraph(8).line_graph()
sage: v = T8.vertices()[0]
sage: twograph_descendant(T8, v)==T8.twograph().descendant(v)
True
sage: twograph_descendant(T8, v).is_strongly_regular(parameters=True)
(27, 16, 10, 8)
sage: p = graphs.PetersenGraph()
sage: twograph_descendant(p,5)
Graph on 9 vertices
sage: twograph_descendant(p,5,name=True)
descendant of Petersen graph at 5: Graph on 9 vertices
```

5.1.86 Diagram and Partition Algebras

AUTHORS:

- Mike Hansen (2007): Initial version
- Stephen Doty, Aaron Lauve, George H. Seelinger (2012): Implementation of partition, Brauer, Temperley–Lieb, and ideal partition algebras
- Stephen Doty, Aaron Lauve, George H. Seelinger (2015): Implementation of *Diagram classes and other methods to improve diagram algebras.

class sage.combinat.diagram_algebras.**AbstractPartitionDiagram**(*parent*, *d*)
Bases: sage.combinat.set_partition.SetPartition

Abstract base class for partition diagrams.

This class represents a single partition diagram, that is used as a basis key for a diagram algebra element. A partition diagram should be a partition of the set $\{1, \dots, k, -1, \dots, -k\}$. Each such set partition is regarded as a graph on nodes $\{1, \dots, k, -1, \dots, -k\}$ arranged in two rows, with nodes $1, \dots, k$ in the top row from left to right and with nodes $-1, \dots, -k$ in the bottom row from left to right, and an edge connecting two nodes if and only if the nodes lie in the same subset of the set partition.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(da.partition_diagrams, 2)
```

```
sage: pd1 = da.AbstractPartitionDiagram(pd, [[1,2],[-1,-2]])
sage: pd2 = da.AbstractPartitionDiagram(pd, [[1,2],[-1,-2]])
sage: pd1
{{-2, -1}, {1, 2}}
sage: pd1 == pd2
True
sage: pd1 == [[1,2],[-1,-2]]
True
sage: pd1 == ((-2,-1),(2,1))
True
sage: pd1 == SetPartition([[1,2],[-1,-2]])
True
sage: pd3 = da.AbstractPartitionDiagram(pd, [[1,-2],[-1,2]])
sage: pd1 == pd3
False
sage: pd4 = da.AbstractPartitionDiagram(pd, [[1,2],[3,4]])
Traceback (most recent call last):
...
ValueError: this does not represent two rows of vertices
```

base_diagram()

Return the underlying implementation of the diagram.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(da.partition_diagrams, 2)
sage: pd([[1,2],[-1,-2]]).base_diagram() == ((-2,-1),(1,2))
True
```

check()

Check the validity of the input for the diagram.

TESTS:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(da.partition_diagrams, 2)
sage: pd1 = da.AbstractPartitionDiagram(pd, [[1,2],[-1,-2]]) # indirect doctest
sage: pd2 = da.AbstractPartitionDiagram(pd, [[1,2],[3,4]]) # indirect doctest
Traceback (most recent call last):
...
ValueError: this does not represent two rows of vertices
```

compose(*other*)

Compose `self` with `other`.

The composition of two diagrams X and Y is given by placing X on top of Y and removing all loops.

OUTPUT:

A tuple where the first entry is the composite diagram and the second entry is how many loop were removed.

Note: This is not really meant to be called directly, but it works to call it this way if desired.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(da.partition_diagrams, 2)
```

```
sage: pd([[1,2],[-1,-2]]).compose(pd([[1,2],[-1,-2]]))
({{-2, -1}, {1, 2}}, 1)
```

diagram()

Return the underlying implementation of the diagram.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(da.partition_diagrams, 2)
sage: pd([[1,2],[-1,-2]]).base_diagram() == pd([[1,2],[-1,-2]]).diagram()
True
```

propagating_number()

Return the propagating number of the diagram.

The propagating number is the number of blocks with both a positive and negative number.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(da.partition_diagrams, 2)
sage: d1 = pd([[1,-2],[2,-1]])
sage: d1.propagating_number()
2
sage: d2 = pd([[1,2],[-2,-1]])
sage: d2.propagating_number()
0
```

class sage.combinat.diagram_algebras.**AbstractPartitionDiagrams**(*diagram_func*, *order*, *category=None*)

Bases: sage.structure.parent.Parent, sage.structure.unique_representation.UniqueRepresenta

This is a class that generates partition diagrams.

Thee primary use of this class is to serve as basis keys for diagram algebras, but diagrams also have properties in their own right. Furthermore, this class is meant to be extended to create more efficient contains methods.

INPUT:

- diagram_func – generator; a function that can create the type of diagram desired
- order – integer or integer $+1/2$; the order of the diagrams

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(da.partition_diagrams, 2)
sage: pd
Partition diagrams of order 2
sage: [i for i in pd]
[{{-2, -1, 1, 2}},
 {{-2, -1, 2}, {1}},
 {{-2, -1, 1}, {2}},
 {{-2}, {-1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2, 1}, {-1, 2}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1}, {1, 2}},
 {{-2, -1}, {1}, {2}},
 {{-2}, {-1, 2}, {1}},
```

```
 {{-2, 2}, {-1}, {1}},
 {{-2}, {-1, 1}, {2}},
 {{-2, 1}, {-1}, {2}},
 {{-2}, {-1}, {1, 2}},
 {{-2}, {-1}, {1}, {2}}]
sage: pd.an_element() in pd
True
sage: elm = pd([[1,2],[-1,-2]])
sage: elm in pd
True
```

Element
alias of AbstractPartitionDiagram

class sage.combinat.diagram_algebras.**BrauerAlgebra**(*k*, *q*, *base_ring*, *prefix*)
Bases: sage.combinat.diagram_algebras.SubPartitionAlgebra

A Brauer algebra.

The Brauer algebra of rank k is an algebra with basis indexed by the collection of set partitions of $\{1, \dots, k, -1, \dots, -k\}$ with block size 2.

This algebra is a subalgebra of the partition algebra. For more information, see PartitionAlgebra.

INPUT:

- k – rank of the algebra
- q – the deformation parameter q

OPTIONAL ARGUMENTS:

- base_ring – (default None) a ring containing q; if None then just takes the parent of q
- prefix – (default "B") a label for the basis elements

EXAMPLES:

We now define the Brauer algebra of rank 2 with parameter x over $\mathbf{Z}$:

```
sage: R.<x> = ZZ[]
sage: B = BrauerAlgebra(2, x, R)
sage: B
Brauer Algebra of rank 2 with parameter x
 over Univariate Polynomial Ring in x over Integer Ring
sage: B.basis()
Lazy family (Term map from Brauer diagrams of order 2 to Brauer Algebra
 of rank 2 with parameter x over Univariate Polynomial Ring in x
 over Integer Ring(i))_{i in Brauer diagrams of order 2}
sage: b = B.basis().list()
sage: b
[B{{-2, 1}, {-1, 2}}, B{{-2, 2}, {-1, 1}}, B{{-2, -1}, {1, 2}}]
sage: b[2]
B{{-2, -1}, {1, 2}}
sage: b[2]^2
x*B{{-2, -1}, {1, 2}}
sage: b[2]^5
x^4*B{{-2, -1}, {1, 2}}
```

Note, also that since the symmetric group algebra is contained in the Brauer algebra, there is also a conversion between the two.

```
sage: R.<x> = ZZ[]
sage: B = BrauerAlgebra(2, x, R)
sage: S = SymmetricGroupAlgebra(R, 2)
sage: S([2,1])*B([[1,-1],[2,-2]])
B{{-2, 1}, {-1, 2}}
```

global_options(*get_value, **set_value)

Set and display the global options for Brauer diagram (algebras). If no parameters are set, then the function returns a copy of the options dictionary.

The options to diagram algebras can be accessed as the method BrauerAlgebra.global_options of BrauerAlgebra and related classes.

OPTIONS:

- display – (default: normal) Specifies how the Brauer diagrams should be printed
  - compact – Using the compact representation
  - normal – Using the normal representation

EXAMPLES:

```
sage: R.<q> = QQ[]
sage: BA = BrauerAlgebra(2, q)
sage: E = BA([[1,2],[-1,-2]])
sage: E
B{{-2, -1}, {1, 2}}
sage: BrauerAlgebra.global_options(display="compact")
sage: E
B[12/12;]
sage: BrauerAlgebra.global_options.reset()
```

See GlobalOptions for more features of these options.

jucys_murphy(*j*)

Return the j-th generalized Jucys-Murphy element of self.

The j-th Jucys-Murphy element of a Brauer algebra is simply the j-th Jucys-Murphy element of the symmetric group algebra with an extra $(z-1)/2$ term, where z is the parameter of the Brauer algebra.

REFERENCES:

EXAMPLES:

```
sage: z = var('z')
sage: B = BrauerAlgebra(3,z)
sage: B.jucys_murphy(1)
(1/2*z-1/2)*B{{-3, 3}, {-2, 2}, {-1, 1}}
sage: B.jucys_murphy(3)
-B{{-3, -2}, {-1, 1}, {2, 3}} - B{{-3, -1}, {-2, 2}, {1, 3}}
 + B{{-3, 1}, {-2, 2}, {-1, 3}} + B{{-3, 2}, {-2, 3}, {-1, 1}}
 + (1/2*z-1/2)*B{{-3, 3}, {-2, 2}, {-1, 1}}
```

class sage.combinat.diagram_algebras.**BrauerDiagram**(*parent, d*)

Bases: sage.combinat.diagram_algebras.AbstractPartitionDiagram

A Brauer diagram.

A Brauer diagram for an integer k is a partition of the set $\{1, \dots, k, -1, \dots, -k\}$ with block size 2.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(2)
sage: bd1 = bd([[1,2],[-1,-2]])
sage: bd2 = bd([[1,2,-1,-2]])
Traceback (most recent call last):
...
ValueError: all blocks must be of size 2
```

bijection_on_free_nodes (*two_line=False*)

Return the induced bijection - as a list of $(x, f(x))$ values - from the free nodes on the top at the Brauer diagram to the free nodes at the bottom of `self`.

OUTPUT:

If `two_line` is `True`, then the output is the induced bijection as a two-row list `(inputs, outputs)`.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1,2],[-2,-3],[3,-1]])
sage: elm.bijection_on_free_nodes()
[[3, -1]]
sage: elm2 = bd([[1,-2],[2,-3],[3,-1]])
sage: elm2.bijection_on_free_nodes(two_line=True)
[[1, 2, 3], [-2, -3, -1]]
```

check ()

Check the validity of the input for `self`.

TESTS:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(2)
sage: bd1 = bd([[1,2],[-1,-2]])  # indirect doctest
sage: bd2 = bd([[1,2,-1,-2]])    # indirect doctest
Traceback (most recent call last):
...
ValueError: all blocks must be of size 2
```

involution_permutation_triple (*curt=True*)

Return the involution permutation triple of `self`.

From Graham-Lehrer (see $class : BrauerDiagrams$), a Brauer diagram is a triple (D_1, D_2, π), where:

- D_1 is a partition of the top nodes;
- D_2 is a partition of the bottom nodes;
- π is the induced permutation on the free nodes.

INPUT:

- `curt` – (default: `True`) if `True`, then return bijection on free nodes as a one-line notation (standardized to look like a permutation), else, return the honest mapping, a list of pairs $(i, -j)$ describing the bijection on free nodes

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1,2],[-2,-3],[3,-1]])
sage: elm.involution_permutation_triple()
```

```
([(1, 2)], [(-3, -2)], [1])
sage: elm.involution_permutation_triple(curt=False)
([(1, 2)], [(-3, -2)], [[3, -1]])
```

is_elementary_symmetric()

Check if is elementary symmetric.

Let (D_1, D_2, π) be the Graham-Lehrer representation of the Brauer diagram d. We say d is *elementary symmetric* if $D_1 = D_2$ and π is the identity.

Todo

Come up with a better name?

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1,2],[-1,-2],[3,-3]])
sage: elm.is_elementary_symmetric()
True
sage: elm2 = bd([[1,2],[-1,-3],[3,-2]])
sage: elm2.is_elementary_symmetric()
False
```

perm()

Return the induced bijection on the free nodes of `self` in one-line notation, re-indexed and treated as a permutation.

See also:

`bijection_on_free_nodes()`

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1,2],[-2,-3],[3,-1]])
sage: elm.perm()
[1]
```

class sage.combinat.diagram_algebras.**BrauerDiagrams**(*order*, *category=None*)

Bases: sage.combinat.diagram_algebras.AbstractPartitionDiagrams

This class represents all Brauer diagrams of integer or integer $+1/2$ order. For more information on Brauer diagrams, see $class : Brauer Algebra$.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: bd.an_element() in bd
True
sage: bd.cardinality() == len(bd.list())
True
```

These diagrams also come equipped with a compact representation based on their bipartition triple representation. See the `from_involution_permutation_triple()` method for more information.

```
sage: bd = da.BrauerDiagrams(3)
sage: bd.global_options(display="compact")
```

```
sage: bd.list()
[[/;321],
 [/;312],
 [23/12;1],
 [/;231],
 [/;132],
 [13/12;1],
 [/;213],
 [/;123],
 [12/12;1],
 [23/23;1],
 [13/23;1],
 [12/23;1],
 [23/13;1],
 [13/13;1],
 [12/13;1]]
sage: bd.global_options.reset()
```

Element
alias of BrauerDiagram

cardinality()
Return the cardinality of self.

The number of Brauer diagrams of integer order k is $(2k-1)!!$.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: bd.cardinality()
15
```

from_involution_permutation_triple(*D1_D2_pi*)
Construct a Bruaer diagram of self from an involution permutation triple.

A Brauer diagram can be represented as a triple where the first entry is a list of arcs on the top row of the diagram, the second entry is a list of arcs on the bottom row of the diagram, and the third entry is a permutation on the remaining nodes. This triple is called the *involution permutation triple*. For more information, see [GL1996].

INPUT:

- D1_D2_pi – a list or tuple where the first entry is a list of arcs on the top of the diagram, the second entry is a list of arcs on the bottom of the diagram, and the third entry is a permutation on the free nodes.

REFERENCES:

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(4)
sage: bd.from_involution_permutation_triple([[[1,2]],[[3,4]],[2,1]])
{{-4, -3}, {-2, 3}, {-1, 4}, {1, 2}}
```

global_options(**get_value*, ***set_value*)
Set and display the global options for Brauer diagram (algebras). If no parameters are set, then the function returns a copy of the options dictionary.

The options to diagram algebras can be accessed as the method BrauerAlgebra.global_options of BrauerAlgebra and related classes.

OPTIONS:

- display – (default: normal) Specifies how the Brauer diagrams should be printed
  - compact – Using the compact representation
  - normal – Using the normal representation

EXAMPLES:

```
sage: R.<q> = QQ[]
sage: BA = BrauerAlgebra(2, q)
sage: E = BA([[1,2],[-1,-2]])
sage: E
B{{-2, -1}, {1, 2}}
sage: BrauerAlgebra.global_options(display="compact")
sage: E
B[12/12;]
sage: BrauerAlgebra.global_options.reset()
```

See GlobalOptions for more features of these options.

symmetric_diagrams(*l=None*, *perm=None*)

Return the list of brauer diagrams with symmetric placement of l arcs, and with free nodes permuted according to $perm$.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(4)
sage: bd.symmetric_diagrams(l=1,perm=[2,1])
[{{-4, -3}, {-2, 1}, {-1, 2}, {3, 4}},
 {{-4, -2}, {-3, 1}, {-1, 3}, {2, 4}},
 {{-4, 1}, {-3, -2}, {-1, 4}, {2, 3}},
 {{-4, -1}, {-3, 2}, {-2, 3}, {1, 4}},
 {{-4, 2}, {-3, -1}, {-2, 4}, {1, 3}},
 {{-4, 3}, {-3, 4}, {-2, -1}, {1, 2}}]
```

class sage.combinat.diagram_algebras.**DiagramAlgebra**(*k*, *q*, *base_ring*, *prefix*, *diagrams*, *category=None*)

Bases: sage.combinat.free_module.CombinatorialFreeModule

Abstract class for diagram algebras and is not designed to be used directly. If used directly, the class could create an "algebra" that is not actually an algebra.

TESTS:

```
sage: import sage.combinat.diagram_algebras as da
sage: R.<x> = QQ[]
sage: D = da.DiagramAlgebra(2, x, R, 'P', da.PartitionDiagrams(2))
sage: sorted(D.basis())
[P{{-2}, {-1}, {1}, {2}},
 P{{-2}, {-1}, {1, 2}},
 P{{-2}, {-1, 1}, {2}},
 P{{-2}, {-1, 1, 2}},
 P{{-2}, {-1, 2}, {1}},
 P{{-2, -1}, {1}, {2}},
 P{{-2, -1}, {1, 2}},
 P{{-2, -1, 1}, {2}},
 P{{-2, -1, 1, 2}},
```

```
P{{-2, -1, 2}, {1}},
P{{-2, 1}, {-1}, {2}},
P{{-2, 1}, {-1, 2}},
P{{-2, 1, 2}, {-1}},
P{{-2, 2}, {-1}, {1}},
P{{-2, 2}, {-1, 1}}]
```

class **Element**(M, x)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

An element of a diagram algebra.

This subclass provides a few additional methods for partition algebra elements. Most element methods are already implemented elsewhere.

diagram()

Return the underlying diagram of `self` if `self` is a basis element. Raises an error if `self` is not a basis element.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: P = PartitionAlgebra(2, x, R)
sage: elt = 3*P([[1,2],[-2,-1]])
sage: elt.diagram()
{{-2, -1}, {1, 2}}
```

diagrams()

Return the diagrams in the support of `self`.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: P = PartitionAlgebra(2, x, R)
sage: elt = 3*P([[1,2],[-2,-1]]) + P([[1,2],[-2], [-1]])
sage: elt.diagrams()
[{{-2}, {-1}, {1, 2}}, {{-2, -1}, {1, 2}}]
```

DiagramAlgebra.**one_basis**()

The following constructs the identity element of `self`.

It is not called directly; instead one should use `DA.one()` if `DA` is a defined diagram algebra.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: R.<x> = QQ[]
sage: D = da.DiagramAlgebra(2, x, R, 'P', da.PartitionDiagrams(2))
sage: D.one_basis()
{{-2, 2}, {-1, 1}}
```

DiagramAlgebra.**order**()

Return the order of `self`.

The order of a partition algebra is defined as half of the number of nodes in the diagrams.

EXAMPLES:

```
sage: q = var('q')
sage: PA = PartitionAlgebra(2, q)
sage: PA.order()
2
```

DiagramAlgebra.**product_on_basis**(*d1*, *d2*)

Return the product $D_{d_1} D_{d_2}$ by two basis diagrams.

TESTS:

```
sage: import sage.combinat.diagram_algebras as da
sage: R.<x> = QQ[]
sage: D = da.DiagramAlgebra(2, x, R, 'P', da.PartitionDiagrams(2))
sage: sp = da.PartitionDiagrams(2)([[1,2],[-1,-2]])
sage: D.product_on_basis(sp, sp)
x*P{{-2, -1}, {1, 2}}
```

DiagramAlgebra.**set_partitions**()

Return the collection of underlying set partitions indexing the basis elements of a given diagram algebra.

Todo

Is this really necessary?

TESTS:

```
sage: import sage.combinat.diagram_algebras as da
sage: R.<x> = QQ[]
sage: D = da.DiagramAlgebra(2, x, R, 'P', da.PartitionDiagrams(2))
sage: list(D.set_partitions()) == list(da.PartitionDiagrams(2))
True
```

class sage.combinat.diagram_algebras.**IdealDiagrams**(*order*)

Bases: sage.combinat.diagram_algebras.AbstractPartitionDiagrams

All "ideal" diagrams of integer or integer $+1/2$ order.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: id = da.IdealDiagrams(3)
sage: id.an_element() in id
True
sage: id.cardinality() == len(id.list())
True
```

class sage.combinat.diagram_algebras.**PartitionAlgebra**(*k*, *q*, *base_ring*, *prefix*)

Bases: sage.combinat.diagram_algebras.DiagramAlgebra

A partition algebra.

A partition algebra of rank k over a given ground ring R is an algebra with (R-module) basis indexed by the collection of set partitions of $\{1, \dots, k, -1, \dots, -k\}$. Each such set partition can be represented by a graph on nodes $\{1, \dots, k, -1, \dots, -k\}$ arranged in two rows, with nodes $1, \dots, k$ in the top row from left to right and with nodes $-1, \dots, -k$ in the bottom row from left to right, and edges drawn such that the connected components of the graph are precisely the parts of the set partition. (This choice of edges is often not unique, and so there are often many graphs representing one and the same set partition; the representation nevertheless is useful and vivid. We often speak of "diagrams" to mean graphs up to such equivalence of choices of edges; of course, we could just as well speak of set partitions.)

There is not just one partition algebra of given rank over a given ground ring, but rather a whole family of them, indexed by the elements of R. More precisely, for every $q \in R$, the partition algebra of rank k over R with parameter q is defined to be the R-algebra with basis the collection of all set partitions of $\{1, \dots, k, -1, \dots, -k\}$, where the product of two basis elements is given by the rule

$$a \cdot b = q^N (a \circ b),$$

where $a \circ b$ is the composite set partition obtained by placing the diagram (i.e., graph) of a above the diagram of b, identifying the bottom row nodes of a with the top row nodes of b, and omitting any closed "loops" in the middle. The number N is the number of connected components formed by the omitted loops.

The parameter q is a deformation parameter. Taking $q = 1$ produces the semigroup algebra (over the base ring) of the partition monoid, in which the product of two set partitions is simply given by their composition.

The Iwahori–Hecke algebra of type A (with a single parameter) is naturally a subalgebra of the partition algebra.

The partition algebra is regarded as an example of a "diagram algebra" due to the fact that its natural basis is given by certain graphs often called diagrams.

An excellent reference for partition algebras and their various subalgebras (Brauer algebra, Temperley–Lieb algebra, etc) is the paper [HR2005].

INPUT:

- `k` – rank of the algebra
- `q` – the deformation parameter q

OPTIONAL ARGUMENTS:

- `base_ring` – (default `None`) a ring containing `q`; if `None`, then Sage automatically chooses the parent of `q`
- `prefix` – (default `"P"`) a label for the basis elements

EXAMPLES:

The following shorthand simultaneously defines the univariate polynomial ring over the rationals as well as the variable `x`:

```
sage: R.<x> = PolynomialRing(QQ)
sage: R
Univariate Polynomial Ring in x over Rational Field
sage: x
x
sage: x.parent() is R
True
```

We now define the partition algebra of rank 2 with parameter `x` over $\mathbf{Z}$:

```
sage: R.<x> = ZZ[]
sage: P = PartitionAlgebra(2, x, R)
sage: P
Partition Algebra of rank 2 with parameter x
 over Univariate Polynomial Ring in x over Integer Ring
sage: P.basis().list()
[P{{-2, -1, 1, 2}}, P{{-2, -1, 2}, {1}},
 P{{-2, -1, 1}, {2}}, P{{-2}, {-1, 1, 2}},
 P{{-2, 1, 2}, {-1}}, P{{-2, 1}, {-1, 2}},
 P{{-2, 2}, {-1, 1}}, P{{-2, -1}, {1, 2}},
 P{{-2, -1}, {1}, {2}}, P{{-2}, {-1, 2}, {1}},
 P{{-2, 2}, {-1}, {1}}, P{{-2}, {-1, 1}, {2}},
 P{{-2, 1}, {-1}, {2}}, P{{-2}, {-1}, {1, 2}},
 P{{-2}, {-1}, {1}, {2}}]
sage: E = P([[1,2],[-2,-1]]); E
P{{-2, -1}, {1, 2}}
sage: E in P.basis().list()
True
sage: E^2
x*P{{-2, -1}, {1, 2}}
```

```
sage: E^5
x^4*P{{-2, -1}, {1, 2}}
sage: (P([[2,-2],[-1,1]]) - 2*P([[1,2],[-1,-2]]))^2
(4*x-4)*P{{-2, -1}, {1, 2}} + P{{-2, 2}, {-1, 1}}
```

One can work with partition algebras using a symbol for the parameter, leaving the base ring unspecified. This implies that the underlying base ring is Sage's symbolic ring.

```
sage: q = var('q')
sage: PA = PartitionAlgebra(2, q); PA
Partition Algebra of rank 2 with parameter q over Symbolic Ring
sage: PA([[1,2],[-2,-1]])^2 == q*PA([[1,2],[-2,-1]])
True
sage: (PA([[2, -2], [1, -1]]) - 2*PA([[-2, -1], [1, 2]]))^2 == (4*q-4)*PA([[1, 2], [-2, -1]]) +
True
```

The identity element of the partition algebra is the set partition $\{\{1,-1\},\{2,-2\},\ldots,\{k,-k\}\}$:

```
sage: P = PA.basis().list()
sage: PA.one()
P{{-2, 2}, {-1, 1}}
sage: PA.one()*P[7] == P[7]
True
sage: P[7]*PA.one() == P[7]
True
```

We now give some further examples of the use of the other arguments. One may wish to "specialize" the parameter to a chosen element of the base ring:

```
sage: R.<q> = RR[]
sage: PA = PartitionAlgebra(2, q, R, prefix='B')
sage: PA
Partition Algebra of rank 2 with parameter q over
 Univariate Polynomial Ring in q over Real Field with 53 bits of precision
sage: PA([[1,2],[-1,-2]])
1.00000000000000*B{{-2, -1}, {1, 2}}
sage: PA = PartitionAlgebra(2, 5, base_ring=ZZ, prefix='B')
sage: PA
Partition Algebra of rank 2 with parameter 5 over Integer Ring
sage: (PA([[2, -2], [1, -1]]) - 2*PA([[-2, -1], [1, 2]]))^2 == 16*PA([[-2, -1], [1, 2]]) + PA([[
True
```

TESTS:

A computation that returned an incorrect result until trac ticket #15958:

```
sage: A = PartitionAlgebra(1,17)
sage: g = SetPartitionsAk(1).list()
sage: a = A[g[1]]
sage: a
P{{-1}, {1}}
sage: a*a
17*P{{-1}, {1}}
```

Symmetric group algebra elements can also be coerced into the partition algebra:

```
sage: S = SymmetricGroupAlgebra(SR, 2)
sage: A = PartitionAlgebra(2, x, SR)
sage: S([2,1])*A([[1,-1],[2,-2]])
P{{-2, 1}, {-1, 2}}
```

REFERENCES:

class sage.combinat.diagram_algebras.**PartitionDiagrams**(*order*, *category=None*)
Bases: sage.combinat.diagram_algebras.AbstractPartitionDiagrams

This class represents all partition diagrams of integer or integer $+1/2$ order.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.PartitionDiagrams(3)
sage: pd.an_element() in pd
True
sage: pd.cardinality() == len(pd.list())
True
```

cardinality()
The cardinality of partition diagrams of integer order n is the $2n$-th Bell number.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.PartitionDiagrams(3)
sage: pd.cardinality()
203
```

class sage.combinat.diagram_algebras.**PlanarAlgebra**(*k*, *q*, *base_ring*, *prefix*)
Bases: sage.combinat.diagram_algebras.SubPartitionAlgebra

A planar algebra.

The planar algebra of rank k is an algebra with basis indexed by the collection of all planar set partitions of $\{1, \ldots, k, -1, \ldots, -k\}$.

This algebra is thus a subalgebra of the partition algebra. For more information, see PartitionAlgebra.

INPUT:

- k – rank of the algebra
- q – the deformation parameter q

OPTIONAL ARGUMENTS:

- base_ring – (default None) a ring containing q; if None then just takes the parent of q
- prefix – (default "Pl") a label for the basis elements

EXAMPLES:

We define the planar algebra of rank 2 with parameter x over $\mathbf{Z}$:

```
sage: R.<x> = ZZ[]
sage: Pl = PlanarAlgebra(2, x, R); Pl
Planar Algebra of rank 2 with parameter x over Univariate Polynomial Ring in x over Integer Ring
sage: Pl.basis().list()
[Pl{{-2, -1, 1, 2}}, Pl{{-2, -1, 2}, {1}},
 Pl{{-2, -1, 1}, {2}}, Pl{{-2}, {-1, 1, 2}},
 Pl{{-2, 1, 2}, {-1}}, Pl{{-2, 2}, {-1, 1}},
 Pl{{-2, -1}, {1, 2}}, Pl{{-2, -1}, {1}, {2}},
 Pl{{-2}, {-1, 2}, {1}}, Pl{{-2, 2}, {-1}, {1}},
 Pl{{-2}, {-1, 1}, {2}}, Pl{{-2, 1}, {-1}, {2}},
 Pl{{-2}, {-1}, {1, 2}}, Pl{{-2}, {-1}, {1}, {2}}]
sage: E = Pl([[1,2],[-1,-2]])
sage: E^2 == x*E
```

```
True
sage: E^5 == x^4*E
True
```

class sage.combinat.diagram_algebras.**PlanarDiagrams**(*order*)
Bases: sage.combinat.diagram_algebras.AbstractPartitionDiagrams

All planar diagrams of integer or integer $+1/2$ order.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pld = da.PlanarDiagrams(3)
sage: pld.an_element() in pld
True
sage: pld.cardinality() == len(pld.list())
True
```

cardinality()
Return the cardinality of `self`.

The number of all planar diagrams of order k is the $2k$-th Catalan number.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pld = da.PlanarDiagrams(3)
sage: pld.cardinality()
132
```

class sage.combinat.diagram_algebras.**PropagatingIdeal**(*k*, *q*, *base_ring*, *prefix*)
Bases: sage.combinat.diagram_algebras.SubPartitionAlgebra

A propagating ideal.

The propagating ideal of rank k is a non-unital algebra with basis indexed by the collection of ideal set partitions of $\{1, \dots, k, -1, \dots, -k\}$. We say a set partition is *ideal* if its propagating number is less than k.

This algebra is a non-unital subalgebra and an ideal of the partition algebra. For more information, see PartitionAlgebra.

EXAMPLES:

We now define the propagating ideal of rank 2 with parameter x over $\mathbf{Z}$:

```
sage: R.<x> = QQ[]
sage: I = PropagatingIdeal(2, x, R); I
Propagating Ideal of rank 2 with parameter x
 over Univariate Polynomial Ring in x over Rational Field
sage: I.basis().list()
[I{{-2, -1, 1, 2}}, I{{-2, -1, 2}, {1}},
 I{{-2, -1, 1}, {2}}, I{{-2}, {-1, 1, 2}},
 I{{-2, 1, 2}, {-1}}, I{{-2, -1}, {1, 2}},
 I{{-2, -1}, {1}, {2}}, I{{-2}, {-1, 2}, {1}},
 I{{-2, 2}, {-1}, {1}}, I{{-2}, {-1, 1}, {2}},
 I{{-2, 1}, {-1}, {2}}, I{{-2}, {-1}, {1, 2}},
 I{{-2}, {-1}, {1}, {2}}]
sage: E = I([[1,2],[-1,-2]])
sage: E^2 == x*E
True
sage: E^5 == x^4*E
True
```

class **Element**(*M*, *x*)

Bases: sage.combinat.diagram_algebras.DiagramAlgebra.Element

An element of a propagating ideal.

We need to take care of exponents since we are not unital.

PropagatingIdeal.**one_basis**()

The propagating ideal is a non-unital algebra, i.e. it does not have a multiplicative identity.

EXAMPLES:

```
sage: R.<q> = QQ[]
sage: I = PropagatingIdeal(2, q, R)
sage: I.one_basis()
Traceback (most recent call last):
...
ValueError: The ideal partition algebra is not unital
sage: I.one()
Traceback (most recent call last):
...
ValueError: The ideal partition algebra is not unital
```

class sage.combinat.diagram_algebras.**SubPartitionAlgebra**(*k*, *q*, *base_ring*, *prefix*, *diagrams*, *category=None*)

Bases: sage.combinat.diagram_algebras.DiagramAlgebra

A subalgebra of the partition algebra indexed by a subset of the diagrams.

ambient()

Return the partition algebra `self` is a sub-algebra of.

EXAMPLES:

```
sage: x = var('x')
sage: BA = BrauerAlgebra(2, x)
sage: BA.ambient()
Partition Algebra of rank 2 with parameter x over Symbolic Ring
```

lift()

Return the lift map from diagram subalgebra to the ambient space.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: BA = BrauerAlgebra(2, x, R)
sage: E = BA([[1,2],[-1,-2]])
sage: lifted = BA.lift(E); lifted
B{{-2, -1}, {1, 2}}
sage: lifted.parent() is BA.ambient()
True
```

retract(*x*)

Retract an appropriate partition algebra element to the corresponding element in the partition subalgebra.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: BA = BrauerAlgebra(2, x, R)
sage: PA = BA.ambient()
sage: E = PA([[1,2], [-1,-2]])
sage: BA.retract(E) in BA
True
```

class sage.combinat.diagram_algebras.**TemperleyLiebAlgebra**(*k*, *q*, *base_ring*, *prefix*)
Bases: sage.combinat.diagram_algebras.SubPartitionAlgebra

A Temperley–Lieb algebra.

The Temperley–Lieb algebra of rank k is an algebra with basis indexed by the collection of planar set partitions of $\{1, \ldots, k, -1, \ldots, -k\}$ with block size 2.

This algebra is thus a subalgebra of the partition algebra. For more information, see PartitionAlgebra.

INPUT:

- k – rank of the algebra
- q – the deformation parameter q

OPTIONAL ARGUMENTS:

- base_ring – (default None) a ring containing q; if None then just takes the parent of q
- prefix – (default "T") a label for the basis elements

EXAMPLES:

We define the Temperley–Lieb algebra of rank 2 with parameter x over $\mathbf{Z}$:

```
sage: R.<x> = ZZ[]
sage: T = TemperleyLiebAlgebra(2, x, R); T
Temperley-Lieb Algebra of rank 2 with parameter x
 over Univariate Polynomial Ring in x over Integer Ring
sage: T.basis()
Lazy family (Term map from Temperleylieb diagrams of order 2
 to Temperley-Lieb Algebra of rank 2 with parameter x
 over Univariate Polynomial Ring in x over
 Integer Ring(i))_{i in Temperleylieb diagrams of order 2}
sage: b = T.basis().list()
sage: b
[T{{-2, 2}, {-1, 1}}, T{{-2, -1}, {1, 2}}]
sage: b[1]
T{{-2, -1}, {1, 2}}
sage: b[1]^2 == x*b[1]
True
sage: b[1]^5 == x^4*b[1]
True
```

class sage.combinat.diagram_algebras.**TemperleyLiebDiagrams**(*order*)
Bases: sage.combinat.diagram_algebras.AbstractPartitionDiagrams

All Temperley-Lieb diagrams of integer or integer $+1/2$ order.

For more information on Temperley-Lieb diagrams, see $class : TemperleyLiebAlgebra$.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: td = da.TemperleyLiebDiagrams(3)
sage: td.an_element() in td
True
sage: td.cardinality() == len(td.list())
True
```

cardinality()
Return the cardinality of self.

The number of Temperley–Lieb diagrams of integer order k is the k-th Catalan number.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: td = da.TemperleyLiebDiagrams(3)
sage: td.cardinality()
5
```

sage.combinat.diagram_algebras.**brauer_diagrams**(*k*)

Return a generator of all Brauer diagrams of order k.

A Brauer diagram of order k is a partition diagram of order k with block size 2.

INPUT:

- k – the order of the Brauer diagrams

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.brauer_diagrams(2)]
[{{-2, 1}, {-1, 2}}, {{-2, 2}, {-1, 1}}, {{-2, -1}, {1, 2}}]
sage: [SetPartition(p) for p in da.brauer_diagrams(5/2)]
[{{-3, 3}, {-2, 1}, {-1, 2}}, {{-3, 3}, {-2, 2}, {-1, 1}}, {{-3, 3}, {-2, -1}, {1, 2}}]
```

sage.combinat.diagram_algebras.**ideal_diagrams**(*k*)

Return a generator of all "ideal" diagrams of order k.

An ideal diagram of order k is a partition diagram of order k with propagating number less than k.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.ideal_diagrams(2)]
[{{-2, -1, 1, 2}}, {{-2, -1, 2}, {1}}, {{-2, -1, 1}, {2}}, {{-2}, {-1, 1, 2}},
 {{-2, 1, 2}, {-1}}, {{-2, -1}, {1, 2}}, {{-2, -1}, {1}, {2}},
 {{-2}, {-1, 2}, {1}}, {{-2, 2}, {-1}, {1}}, {{-2}, {-1, 1}, {2}}, {{-2, 1},
 {-1}, {2}}, {{-2}, {-1}, {1, 2}}, {{-2}, {-1}, {1}, {2}}]
sage: [SetPartition(p) for p in da.ideal_diagrams(3/2)]
[{{-2, -1, 1, 2}}, {{-2, -1, 2}, {1}}, {{-2, 1, 2}, {-1}}, {{-2, 2}, {-1}, {1}}]
```

sage.combinat.diagram_algebras.**identity_set_partition**(*k*)

Return the identity set partition $\{\{1,-1\},\ldots,\{k,-k\}\}$

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: SetPartition(da.identity_set_partition(2))
{{-2, 2}, {-1, 1}}
```

sage.combinat.diagram_algebras.**is_planar**(*sp*)

Return True if the diagram corresponding to the set partition sp is planar; otherwise, return False.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: da.is_planar( da.to_set_partition([[1,-2],[2,-1]]))
False
sage: da.is_planar( da.to_set_partition([[1,-1],[2,-2]]))
True
```

sage.combinat.diagram_algebras.**pair_to_graph**(*sp1*, *sp2*)

Return a graph consisting of the disjoint union of the graphs of set partitions sp1 and sp2 along with edges joining the bottom row (negative numbers) of sp1 to the top row (positive numbers) of sp2.

The vertices of the graph `sp1` appear in the result as pairs `(k, 1)`, whereas the vertices of the graph `sp2` appear as pairs `(k, 2)`.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: sp1 = da.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = da.to_set_partition([[1,-2],[2,-1]])
sage: g = da.pair_to_graph( sp1, sp2 ); g
Graph on 8 vertices

sage: g.vertices()
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
sage: g.edges()
[((-2, 1), (1, 1), None), ((-2, 1), (2, 2), None),
 ((-2, 2), (1, 2), None), ((-1, 1), (1, 2), None),
 ((-1, 1), (2, 1), None), ((-1, 2), (2, 2), None)]
```

Another example which used to be wrong until trac ticket #15958:

```
sage: sp3 = da.to_set_partition([[1, -1], [2], [-2]])
sage: sp4 = da.to_set_partition([[1], [-1], [2], [-2]])
sage: g = da.pair_to_graph( sp3, sp4 ); g
Graph on 8 vertices

sage: g.vertices()
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
sage: g.edges()
[((-2, 1), (2, 2), None), ((-1, 1), (1, 1), None),
 ((-1, 1), (1, 2), None)]
```

sage.combinat.diagram_algebras.**partition_diagrams**(*k*)

Return a generator of all partition diagrams of order `k`.

A partition diagram of order $k \in \mathbf{Z}$ to is a set partition of $\{1, \dots, k, -1, \dots, -k\}$. If we have $k - 1/2 \in ZZ$, then a partition diagram of order $k \in 1/2\mathbf{Z}$ is a set partition of $\{1, \dots, k + 1/2, -1, \dots, -(k + 1/2)\}$ with $k + 1/2$ and $-(k + 1/2)$ in the same block. See [HR2005].

INPUT:

- `k` – the order of the partition diagrams

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.partition_diagrams(2)]
[{{-2, -1, 1, 2}}, {{-2, -1, 2}, {1}}, {{-2, -1, 1}, {2}},
 {{-2}, {-1, 1, 2}}, {{-2, 1, 2}, {-1}}, {{-2, 1}, {-1, 2}},
 {{-2, 2}, {-1, 1}}, {{-2, -1}, {1, 2}}, {{-2, -1}, {1}, {2}},
 {{-2}, {-1, 2}, {1}}, {{-2, 2}, {-1}, {1}}, {{-2}, {-1, 1}, {2}},
 {{-2, 1}, {-1}, {2}}, {{-2}, {-1}, {1, 2}}, {{-2}, {-1}, {1}, {2}}]
sage: [SetPartition(p) for p in da.partition_diagrams(3/2)]
[{{-2, -1, 1, 2}}, {{-2, -1, 2}, {1}}, {{-2, 2}, {-1, 1}},
 {{-2, 1, 2}, {-1}}, {{-2, 2}, {-1}, {1}}]
```

sage.combinat.diagram_algebras.**planar_diagrams**(*k*)

Return a generator of all planar diagrams of order `k`.

A planar diagram of order k is a partition diagram of order k that has no crossings.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.planar_diagrams(2)]
[{{-2, -1, 1, 2}}, {{-2, -1, 2}, {1}}, {{-2, -1, 1}, {2}},
 {{-2}, {-1, 1, 2}}, {{-2, 1, 2}, {-1}}, {{-2, 2}, {-1, 1}},
 {{-2, -1}, {1, 2}}, {{-2, -1}, {1}, {2}}, {{-2}, {-1, 2}, {1}},
 {{-2, 2}, {-1}, {1}}, {{-2}, {-1, 1}, {2}}, {{-2, 1}, {-1}, {2}},
 {{-2}, {-1}, {1, 2}}, {{-2}, {-1}, {1}, {2}}]
sage: [SetPartition(p) for p in da.planar_diagrams(3/2)]
[{{-2, -1, 1, 2}}, {{-2, -1, 2}, {1}}, {{-2, 2}, {-1, 1}},
 {{-2, 1, 2}, {-1}}, {{-2, 2}, {-1}, {1}}]
```

sage.combinat.diagram_algebras.**propagating_number**(*sp*)

Return the propagating number of the set partition `sp`.

The propagating number is the number of blocks with both a positive and negative number.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: sp1 = da.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = da.to_set_partition([[1,2],[-2,-1]])
sage: da.propagating_number(sp1)
2
sage: da.propagating_number(sp2)
0
```

sage.combinat.diagram_algebras.**set_partition_composition**(*sp1*, *sp2*)

Return a tuple consisting of the composition of the set partitions `sp1` and `sp2` and the number of components removed from the middle rows of the graph.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: sp1 = da.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = da.to_set_partition([[1,-2],[2,-1]])
sage: p, c = da.set_partition_composition(sp1, sp2)
sage: (SetPartition(p), c) == (SetPartition(da.identity_set_partition(2)), 0)
True
```

sage.combinat.diagram_algebras.**temperley_lieb_diagrams**(*k*)

Return a generator of all Temperley–Lieb diagrams of order `k`.

A Temperley–Lieb diagram of order k is a partition diagram of order k with block size 2 and is planar.

INPUT:

- `k` – the order of the Temperley–Lieb diagrams

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.temperley_lieb_diagrams(2)]
[{{-2, 2}, {-1, 1}}, {{-2, -1}, {1, 2}}]
sage: [SetPartition(p) for p in da.temperley_lieb_diagrams(5/2)]
[{{-3, 3}, {-2, 2}, {-1, 1}}, {{-3, 3}, {-2, -1}, {1, 2}}]
```

sage.combinat.diagram_algebras.**to_Brauer_partition**(*l*, *k=None*)

Same as `to_set_partition()` but assumes omitted elements are connected straight through.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: f = lambda sp: SetPartition(da.to_Brauer_partition(sp))
sage: f([[1,2],[-1,-2]]) == SetPartition([[1,2],[-1,-2]])
True
sage: f([[1,3],[-1,-3]]) == SetPartition([[1,3],[-3,-1],[2,-2]])
True
sage: f([[1,-4],[-3,-1],[3,4]]) == SetPartition([[-3,-1],[2,-2],[1,-4],[3,4]])
True
sage: p = SetPartition([[1,2],[-1,-2],[3,-3],[4,-4]])
sage: SetPartition(da.to_Brauer_partition([[1,2],[-1,-2]], k=4)) == p
True
```

sage.combinat.diagram_algebras.**to_graph**(*sp*)

Return a graph representing the set partition `sp`.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: g = da.to_graph( da.to_set_partition([[1,-2],[2,-1]])); g
Graph on 4 vertices

sage: g.vertices()
[-2, -1, 1, 2]
sage: g.edges()
[(-2, 1, None), (-1, 2, None)]
```

sage.combinat.diagram_algebras.**to_set_partition**(*l*, *k=None*)

Convert a list of a list of numbers to a set partitions. Each list of numbers in the outer list specifies the numbers contained in one of the blocks in the set partition.

If k is specified, then the set partition will be a set partition of $\{1, \dots, k, -1, \dots, -k\}$. Otherwise, k will default to the minimum number needed to contain all of the specified numbers.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: f = lambda sp: SetPartition(da.to_set_partition(sp))
sage: f([[1,-1],[2,-2]]) == SetPartition(da.identity_set_partition(2))
True
```

5.1.87 Pointwise addition of dictionaries

Provides function to add dictionaries pointwise with values in a common ring and to compute linear combinations

EXAMPLES:

```
sage: from sage.combinat.dict_addition import dict_addition
sage: D1 = { 0:1, 1:1 }; D2 = { 0:-1, 1:1 }
sage: dict_addition( [D1,D2] )
{1: 2}
```

sage.combinat.dict_addition.**dict_addition**(*dict_iter*)

Returns the pointwise addition of dictionaries with coefficients.

Parameters **dict_iter** – iterator of dictionaries with values in a common ring.

OUTPUT:

•a dictionary containing all keys of dictionaries in `dict_list`, with values being the sum of the values in the different dictionaries (keys with zero value are omitted)

EXAMPLES:

```
sage: from sage.combinat.dict_addition import dict_addition
sage: D = { 0:1, 1:1 }; D
{0: 1, 1: 1}
sage: dict_addition( D for _ in range(5) )
{0: 5, 1: 5}

sage: D1 = { 0:1, 1:1 }; D2 = { 0:-1, 1:1 }
sage: dict_addition( [D1,D2] )
{1: 2}
```

sage.combinat.dict_addition.**dict_linear_combination**(*dict_factor_iter*, *factor_on_left=True*)

Returns the pointwise addition of dictionaries with coefficients.

Parameters

- **dict_factor_iter** – iterator of pairs D, coeff, where - the D's are dictionaries with values in a common ring - the coeff's are coefficients in this ring
- **factor_on_left** (boolean; optional, default `True`) – if True, the coefficients are multiplied on the left, otherwise they are multiplied on the right

OUTPUT:

•a dictionary containing all keys of dictionaries in `dict_list`, with values being the sum of the values in the different dictionaries, each one first multiplied by the given factor (keys with zero value are omitted)

EXAMPLES:

```
sage: from sage.combinat.dict_addition import dict_linear_combination
sage: D = { 0:1, 1:1 }; D
{0: 1, 1: 1}
sage: dict_linear_combination( (D,i) for i in range(5) )
{0: 10, 1: 10}
sage: dict_linear_combination( [(D,1),(D,-1)] )
{}
```

5.1.88 Exact Cover Problem via Dancing Links

sage.combinat.dlx.**AllExactCovers**(*M*)

Utilizes A. Ajanki's DLXMatrix class to solve the exact cover problem on the matrix M (treated as a dense binary matrix).

EXAMPLES:

```
sage: M = Matrix([[1,1,0],[1,0,1],[0,1,1]])  #no exact covers
sage: for cover in AllExactCovers(M):
...       print cover
sage: M = Matrix([[1,1,0],[1,0,1],[0,0,1],[0,1,0]]) #two exact covers
sage: for cover in AllExactCovers(M):
...       print cover
[(1, 1, 0), (0, 0, 1)]
[(1, 0, 1), (0, 1, 0)]
```

class sage.combinat.dlx.**DLXMatrix**(*ones*, *initialsolution=None*)

Solves the Exact Cover problem by using the Dancing Links algorithm described by Knuth.

Consider a matrix M with entries of 0 and 1, and compute a subset of the rows of this matrix which sum to the vector of all 1's.

The dancing links algorithm works particularly well for sparse matrices, so the input is a list of lists of the form: (note the 1-index!):

```
[
 [1, [i_11,i_12,...,i_1r]]
 ...
 [m,[i_m1,i_m2,...,i_ms]]
]
```

where M[j][i_jk] = 1.

The first example below corresponds to the matrix:

```
1110
1010
0100
0001
```

which is exactly covered by:

```
1110
0001
```

and

```
1010
0100
0001
```

EXAMPLES:

```
sage: from sage.combinat.dlx import *
sage: ones = [[1,[1,2,3]]]
sage: ones+= [[2,[1,3]]]
sage: ones+= [[3,[2]]]
sage: ones+= [[4,[4]]]
sage: DLXM = DLXMatrix(ones,[4])
sage: for C in DLXM:
...       print C
[4, 1]
[4, 2, 3]
```

Note: The 0 entry is reserved internally for headers in the sparse representation, so rows and columns begin their indexing with 1. Apologies for any heartache this causes. Blame the original author, or fix it yourself.

---

next()

Search for the first solution we can find, and return it.

Knuth describes the Dancing Links algorithm recursively, though actually implementing it as a recursive algorithm is permissible only for highly restricted problems. (for example, the original author implemented this for Sudoku, and it works beautifully there)

What follows is an iterative description of DLX:

```
stack <- [(NULL)]
level <- 0
while level >= 0:
    cur <- stack[level]
```

```
        if cur = NULL:
            if R[h] = h:
                level <- level - 1
                yield solution
            else:
                cover(best_column)
                stack[level] = best_column
        else if D[cur] != C[cur]:
            if cur != C[cur]:
                delete solution[level]
                for j in L[cur], L[L[cur]], ... , while j != cur:
                    uncover(C[j])
            cur <- D[cur]
            solution[level] <- cur
            stack[level] <- cur
            for j in R[cur], R[R[cur]], ... , while j != cur:
                cover(C[j])
            level <- level + 1
            stack[level] <- (NULL)
        else:
            if C[cur] != cur:
                delete solution[level]
                for j in L[cur], L[L[cur]], ... , while j != cur:
                    uncover(C[j])
            uncover(cur)
            level <- level - 1
```

TESTS:

```
sage: from sage.combinat.dlx import *
sage: M = DLXMatrix([[1,[1,2]],[2,[2,3]],[3,[1,3]]])
sage: while 1:
...     try:
...         C = next(M)
...     except StopIteration:
...         print "StopIteration"
...         break
...     print C
StopIteration
sage: M = DLXMatrix([[1,[1,2]],[2,[2,3]],[3,[3]]])
sage: for C in M:
...       print C
[1, 3]
sage: M = DLXMatrix([[1,[1]],[2,[2,3]],[3,[2]],[4,[3]]])
sage: for C in M:
...       print C
[1, 2]
[1, 3, 4]
```

sage.combinat.dlx.**OneExactCover**(*M*)

Utilizes A. Ajanki's DLXMatrix class to solve the exact cover problem on the matrix M (treated as a dense binary matrix).

EXAMPLES:

```
sage: M = Matrix([[1,1,0],[1,0,1],[0,1,1]])  #no exact covers
sage: print OneExactCover(M)
None
sage: M = Matrix([[1,1,0],[1,0,1],[0,0,1],[0,1,0]]) #two exact covers
```

```
sage: print OneExactCover(M)
[(1, 1, 0), (0, 0, 1)]
```

5.1.89 Dyck Words

A class of an object enumerated by the Catalan numbers, see [Sta-EC2], [StaCat98] for details.

AUTHORS:

- Mike Hansen
- Dan Drake (2008–05-30): DyckWordBacktracker support
- Florent Hivert (2009–02-01): Bijections with NonDecreasingParkingFunctions
- Christian Stump (2011–12): added combinatorial maps and statistics
- Mike Zabrocki:
  - (2012–10): added pretty print, characteristic function, more functions
  - (2013–01): added inverse of area/dinv, bounce/area map
- Jean–Baptiste Priez, Travis Scrimshaw (2013–05-17): Added ASCII art
- Travis Scrimshaw (2013–07-09): Removed CombinatorialClass and added global options.

REFERENCES:

class sage.combinat.dyck_word.**CompleteDyckWords**
Bases: sage.combinat.dyck_word.DyckWords

Abstract base class for all complete Dyck words.

Element
alias of DyckWord_complete

from_Catalan_code(*code*)
Return the Dyck word associated to the given Catalan code code.

A Catalan code of length n is a sequence $(a_1, a_2, \ldots, a_n)$ of n integers a_i such that:

•$0 \leq a_i \leq n - i$ for every i;

•if $i < j$ and $a_i > 0$ and $a_j > 0$ and $a_{i+1} = a_{i+2} = \cdots = a_{j-1} = 0$, then $a_i - a_j < j - i$.

It turns out that the Catalan codes of length n are in bijection with Dyck words.

The Catalan code of a Dyck word is example (x) in Richard Stanley's exercises on combinatorial interpretations for Catalan objects. The code in this example is the reverse of the description provided there. See [Sta-EC2] and [StaCat98].

EXAMPLES:

```
sage: DyckWords().from_Catalan_code([])
[]
sage: DyckWords().from_Catalan_code([0])
[1, 0]
sage: DyckWords().from_Catalan_code([0, 1])
[1, 1, 0, 0]
sage: DyckWords().from_Catalan_code([0, 0])
[1, 0, 1, 0]
```

from_area_sequence(*code*)

Return the Dyck word associated to the given area sequence `code`.

See `to_area_sequence()` for a definition of the area sequence of a Dyck word.

See also:

`area()`, `to_area_sequence()`.

INPUT:

- `code` – a list of integers satisfying `code[0] == 0` and `0 <= code[i+1] <= code[i]+1`.

EXAMPLES:

```
sage: DyckWords().from_area_sequence([])
[]
sage: DyckWords().from_area_sequence([0])
[1, 0]
sage: DyckWords().from_area_sequence([0, 1])
[1, 1, 0, 0]
sage: DyckWords().from_area_sequence([0, 0])
[1, 0, 1, 0]
```

from_non_decreasing_parking_function(*pf*)

Bijection from non-decreasing parking functions. See there the method `to_dyck_word()` for more information.

EXAMPLES:

```
sage: D = DyckWords()
sage: D.from_non_decreasing_parking_function([])
[]
sage: D.from_non_decreasing_parking_function([1])
[1, 0]
sage: D.from_non_decreasing_parking_function([1,1])
[1, 1, 0, 0]
sage: D.from_non_decreasing_parking_function([1,2])
[1, 0, 1, 0]
sage: D.from_non_decreasing_parking_function([1,1,1])
[1, 1, 1, 0, 0, 0]
sage: D.from_non_decreasing_parking_function([1,2,3])
[1, 0, 1, 0, 1, 0]
sage: D.from_non_decreasing_parking_function([1,1,3,3,4,6,6])
[1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0]
```

TESTS:

```
sage: D.from_non_decreasing_parking_function(NonDecreasingParkingFunction([]))
[]
sage: D.from_non_decreasing_parking_function(NonDecreasingParkingFunction([1,1,3,3,4,6,6]))
[1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0]
```

from_noncrossing_partition(*ncp*)

Convert a noncrossing partition `ncp` to a Dyck word.

TESTS:

```
sage: DyckWord(noncrossing_partition=[[1,2]]) # indirect doctest
[1, 1, 0, 0]
sage: DyckWord(noncrossing_partition=[[1],[2]])
[1, 0, 1, 0]
```

```
sage: dws = DyckWords(5).list()
sage: ncps = map(lambda x: x.to_noncrossing_partition(), dws)
sage: dws2 = map(lambda x: DyckWord(noncrossing_partition=x), ncps)
sage: dws == dws2
True
```

class sage.combinat.dyck_word.**CompleteDyckWords_all**

Bases: sage.combinat.dyck_word.CompleteDyckWords, sage.combinat.dyck_word.DyckWords_all

All complete Dyck words.

class height_poset

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The poset of complete Dyck words compared componentwise by heights. This is, D is smaller than or equal to D' if it is weakly below D'.

le(*dw1*, *dw2*)

Compare two Dyck words of equal size, and return True if all of the heights of dw1 are less than or equal to the respective heights of dw2 .

See also:

heights

EXAMPLES:

```
sage: poset = DyckWords().height_poset()
sage: poset.le(DyckWord([]), DyckWord([]))
True
sage: poset.le(DyckWord([1,0]), DyckWord([1,0]))
True
sage: poset.le(DyckWord([1,0,1,0]), DyckWord([1,1,0,0]))
True
sage: poset.le(DyckWord([1,1,0,0]), DyckWord([1,0,1,0]))
False
sage: [poset.le(dw1, dw2)
....:     for dw1 in DyckWords(3) for dw2 in DyckWords(3)]
[True, True, True, True, True, False, True, False, True, True,
False, False, True, True, True, False, False, False, True,
True, False, False, False, False, True]
```

class sage.combinat.dyck_word.**CompleteDyckWords_size**(*k*)

Bases: sage.combinat.dyck_word.CompleteDyckWords, sage.combinat.dyck_word.DyckWords_size

All complete Dyck words of a given size.

cardinality()

Return the number of complete Dyck words of semilength n, i.e. the n-th Catalan number.

EXAMPLES:

```
sage: DyckWords(4).cardinality()
14
sage: ns = range(9)
sage: dws = [DyckWords(n) for n in ns]
sage: all([dw.cardinality() == len(dw.list()) for dw in dws])
True
```

random_element()

Return a random complete Dyck word of semilength n

The algorithm is based on a classical combinatorial fact. One chooses at random a word with n 0's and $n+1$ 1's. One then considers every 1 as an ascending step and every 0 as a descending step, and one finds the lowest point of the path (with respect to a slightly tilted slope). One then cuts the path at this point and builds a Dyck word by exchanging the two parts of the word and removing the initial step.

Todo

extend this to m-Dyck words

EXAMPLES:

```
sage: dw = DyckWords(8)
sage: dw.random_element()  # random
[1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0]

sage: D = DyckWords(8)
sage: D.random_element() in D
True
```

class sage.combinat.dyck_word.**DyckWord**(*parent*, *l*, *latex_options={}*)

Bases: sage.combinat.combinat.CombinatorialElement

A Dyck word.

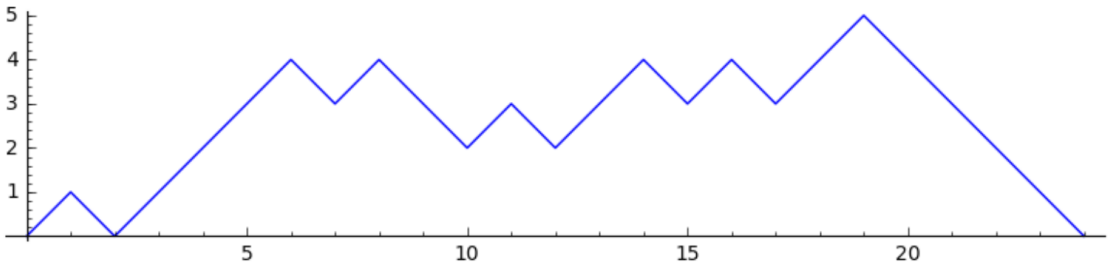
A Dyck word is a sequence of open and close symbols such that every close symbol has a corresponding open symbol preceding it. That is to say, a Dyck word of length n is a list with k entries 1 and $n-k$ entries 0 such that the first i entries always have at least as many 1s among them as 0s. (Here, the 1 serves as the open symbol and the 0 as the close symbol.) Alternatively, the alphabet 1 and 0 can be replaced by other characters such as '(' and ')'.

A Dyck word is *complete* if every open symbol moreover has a corresponding close symbol.

A Dyck word may also be specified by either a noncrossing partition or by an area sequence or the sequence of heights.

A Dyck word may also be thought of as a lattice path in the $\mathbb{Z}^2$ grid, starting at the origin $(0,0)$, and with steps in the North $N=(0,1)$ and east $E=(1,0)$ directions such that it does not pass below the $x=y$ diagonal. The diagonal is referred to as the "main diagonal" in the documentation. A North step is represented by a 1 in the list and an East step is represented by a 0.

Equivalently, the path may be represented with steps in the $NE=(1,1)$ and the $SE=(1,-1)$ direction such that it does not pass below the horizontal axis.



A path representing a Dyck word (either using N and E steps, or using NE and SE steps) is called a Dyck path.

EXAMPLES:

```
sage: dw = DyckWord([1, 0, 1, 0]); dw
[1, 0, 1, 0]
sage: print dw
()()
sage: print dw.height()
1
sage: dw.to_noncrossing_partition()
[[1], [2]]

sage: DyckWord('()()')
[1, 0, 1, 0]
sage: DyckWord('(())')
[1, 1, 0, 0]
sage: DyckWord('((')
[1, 1]
sage: DyckWord('')
[]

sage: DyckWord(noncrossing_partition=[[1],[2]])
[1, 0, 1, 0]
sage: DyckWord(noncrossing_partition=[[1,2]])
[1, 1, 0, 0]
sage: DyckWord(noncrossing_partition=[])
[]
```

```
sage: DyckWord(area_sequence=[0,0])
[1, 0, 1, 0]
sage: DyckWord(area_sequence=[0,1])
[1, 1, 0, 0]
sage: DyckWord(area_sequence=[0,1,2,2,0,1,1,2])
[1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0]
sage: DyckWord(area_sequence=[])
[]

sage: DyckWord(heights_sequence=(0,1,0,1,0))
[1, 0, 1, 0]
sage: DyckWord(heights_sequence=(0,1,2,1,0))
[1, 1, 0, 0]
sage: DyckWord(heights_sequence=(0,))
[]

sage: print DyckWord([1,0,1,1,0,0]).to_path_string()
   /\
/\/  \
sage: DyckWord([1,0,1,1,0,0]).pretty_print()
   ___
  | x
 _|  .
|  . .
```

associated_parenthesis(*pos*)

Report the position for the parenthesis in `self` that matches the one at position `pos`.

The positions in `self` are counted from 0.

INPUT:

- `pos` – the index of the parenthesis in the list

OUTPUT:

- Integer representing the index of the matching parenthesis. If no parenthesis matches, return `None`.

EXAMPLES:

```
sage: DyckWord([1, 0]).associated_parenthesis(0)
1
sage: DyckWord([1, 0, 1, 0]).associated_parenthesis(0)
1
sage: DyckWord([1, 0, 1, 0]).associated_parenthesis(1)
0
sage: DyckWord([1, 0, 1, 0]).associated_parenthesis(2)
3
sage: DyckWord([1, 0, 1, 0]).associated_parenthesis(3)
2
sage: DyckWord([1, 1, 0, 0]).associated_parenthesis(0)
3
sage: DyckWord([1, 1, 0, 0]).associated_parenthesis(2)
1
sage: DyckWord([1, 1, 0]).associated_parenthesis(1)
2
sage: DyckWord([1, 1]).associated_parenthesis(0)
```

height()

Return the height of `self`.

We view the Dyck word as a Dyck path from $(0,0)$ to $(2n,0)$ in the first quadrant by letting `1`'s represent steps in the direction $(1,1)$ and `0`'s represent steps in the direction $(1,-1)$.

The height is the maximum y-coordinate reached.

See also:

`heights()`

EXAMPLES:

```
sage: DyckWord([]).height()
0
sage: DyckWord([1,0]).height()
1
sage: DyckWord([1, 1, 0, 0]).height()
2
sage: DyckWord([1, 1, 0, 1, 0]).height()
2
sage: DyckWord([1, 1, 0, 0, 1, 0]).height()
2
sage: DyckWord([1, 0, 1, 0]).height()
1
sage: DyckWord([1, 1, 0, 0, 1, 1, 1, 0, 0, 0]).height()
3
```

heights()

Return the heights of `self`.

We view the Dyck word as a Dyck path from $(0,0)$ to $(2n,0)$ in the first quadrant by letting `1`'s represent steps in the direction $(1,1)$ and `0`'s represent steps in the direction $(1,-1)$.

The heights is the sequence of the y-coordinates of all $2n+1$ lattice points along the path.

See also:

`from_heights(),min_from_heights()`

EXAMPLES:

```
sage: DyckWord([]).heights()
(0,)
sage: DyckWord([1,0]).heights()
(0, 1, 0)
sage: DyckWord([1, 1, 0, 0]).heights()
(0, 1, 2, 1, 0)
sage: DyckWord([1, 1, 0, 1, 0]).heights()
(0, 1, 2, 1, 2, 1)
sage: DyckWord([1, 1, 0, 0, 1, 0]).heights()
(0, 1, 2, 1, 0, 1, 0)
sage: DyckWord([1, 0, 1, 0]).heights()
(0, 1, 0, 1, 0)
sage: DyckWord([1, 1, 0, 0, 1, 1, 1, 0, 0, 0]).heights()
(0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0)
```

is_complete()

Return `True` if `self` is complete.

A Dyck word d is complete if d contains as many closers as openers.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).is_complete()
True
sage: DyckWord([1, 0, 1, 1, 0]).is_complete()
False
```

TESTS:

```
sage: DyckWord([]).is_complete()
True
```

latex_options()

Return the latex options for use in the _latex_ function as a dictionary. The default values are set using the global options.

- tikz_scale – (default: 1) scale for use with the tikz package.
- diagonal – (default: False) boolean value to draw the diagonal or not.
- line width – (default: 2*‘‘tikz_scale‘‘) value representing the line width.
- color – (default: black) the line color.
- bounce path – (default: False) boolean value to indicate if the bounce path should be drawn.
- peaks – (default: False) boolean value to indicate if the peaks should be displayed.
- valleys – (default: False) boolean value to indicate if the valleys should be displayed.

EXAMPLES:

```
sage: D = DyckWord([1,0,1,0,1,0])
sage: D.latex_options()
{'bounce path': False,
 'color': 'black',
 'diagonal': False,
 'line width': 2,
 'peaks': False,
 'tikz_scale': 1,
 'valleys': False}
```

length()

Return the length of self.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).length()
4
sage: DyckWord([1, 0, 1, 1, 0]).length()
5
```

TESTS:

```
sage: DyckWord([]).length()
0
```

number_of_close_symbols()

Return the number of close symbols in self.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).number_of_close_symbols()
2
sage: DyckWord([1, 0, 1, 1, 0]).number_of_close_symbols()
2
```

TESTS:

```
sage: DyckWord([]).number_of_close_symbols()
0
```

number_of_double_rises()

Return a the number of positions in `self` where there are two consecutive 1's.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0]).number_of_double_rises()
2
sage: DyckWord([1, 1, 0, 0]).number_of_double_rises()
1
sage: DyckWord([1, 0, 1, 0]).number_of_double_rises()
0
```

number_of_initial_rises()

Return the length of the initial run of `self`

OUTPUT:

- a non–negative integer indicating the length of the initial rise

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).number_of_initial_rises()
1
sage: DyckWord([1, 1, 0, 0]).number_of_initial_rises()
2
sage: DyckWord([1, 1, 0, 0, 1, 0]).number_of_initial_rises()
2
sage: DyckWord([1, 0, 1, 1, 0, 0]).number_of_initial_rises()
1
```

TESTS:

```
sage: DyckWord([]).number_of_initial_rises()
0
sage: DyckWord([1, 0]).number_of_initial_rises()
1
```

number_of_open_symbols()

Return the number of open symbols in `self`.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).number_of_open_symbols()
2
sage: DyckWord([1, 0, 1, 1, 0]).number_of_open_symbols()
3
```

TESTS:

```
sage: DyckWord([]).number_of_open_symbols()
0
```

number_of_peaks()

The number of peaks of the Dyck path associated to `self` .

See also:

peaks()

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).number_of_peaks()
2
sage: DyckWord([1, 1, 0, 0]).number_of_peaks()
1
sage: DyckWord([1,1,0,1,0,1,0,0]).number_of_peaks()
3
sage: DyckWord([]).number_of_peaks()
0
```

number_of_touch_points()

Return the number of touches of `self` at the main diagonal.

OUTPUT:

- a non–negative integer

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).number_of_touch_points()
2
sage: DyckWord([1, 1, 0, 0]).number_of_touch_points()
1
sage: DyckWord([1, 1, 0, 0, 1, 0]).number_of_touch_points()
2
sage: DyckWord([1, 0, 1, 1, 0, 0]).number_of_touch_points()
2
```

TESTS:

```
sage: DyckWord([]).number_of_touch_points()
0
```

number_of_valleys()

Return the number of valleys of `self`.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).number_of_valleys()
1
sage: DyckWord([1, 1, 0, 0]).number_of_valleys()
0
sage: DyckWord([1, 1, 0, 0, 1, 0]).number_of_valleys()
1
sage: DyckWord([1, 0, 1, 1, 0, 0]).number_of_valleys()
1
```

TESTS:

```
sage: DyckWord([]).number_of_valleys()
0
sage: DyckWord([1, 0]).number_of_valleys()
0
```

peaks()

Return a list of the positions of the peaks of a Dyck word.

A peak is 1 followed by a 0. Note that this does not agree with the definition given in [Hag2008].

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).peaks()
[0, 2]
sage: DyckWord([1, 1, 0, 0]).peaks()
[1]
sage: DyckWord([1,1,0,1,0,1,0,0]).peaks() # Haglund's def gives 2
[1, 3, 5]
```

plot(***kwds*)

Plot a Dyck word as a continuous path.

EXAMPLES:

```
sage: w = DyckWords(100).random_element()
sage: w.plot()
Graphics object consisting of 1 graphics primitive
```

position_of_first_return()

Return the number of vertical steps before the Dyck path returns to the main diagonal.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0]).position_of_first_return()
1
sage: DyckWord([1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0]).position_of_first_return()
7
sage: DyckWord([1, 1, 0, 0]).position_of_first_return()
2
sage: DyckWord([1, 0, 1, 0]).position_of_first_return()
1
sage: DyckWord([]).position_of_first_return()
0
```

positions_of_double_rises()

Return a list of positions in `self` where there are two consecutive 1's.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0]).positions_of_double_rises()
[2, 5]
sage: DyckWord([1, 1, 0, 0]).positions_of_double_rises()
[0]
sage: DyckWord([1, 0, 1, 0]).positions_of_double_rises()
[]
```

pp(*type=None*, *labelling=None*, *underpath=True*)

Display a DyckWord as a lattice path in the $\mathbf{Z}^2$ grid.

If the `type` is "N-E", then the a cell below the diagonal is indicated by a period, whereas a cell below the path but above the diagonal is indicated by an x. If a list of labels is included, they are displayed along the vertical edges of the Dyck path.

If the `type` is "NE-SE", then the path is simply printed as up steps and down steps.

INPUT:

- `type` – (default: `None`) can either be:
  - `None` to use the global option default
  - "N-E" to show `self` as a path of north and east steps, or
  - "NE-SE" to show `self` as a path of north-east and south-east steps.

- `labelling` – (if type is "N-E") a list of labels assigned to the up steps in `self`.
- `underpath` – (if type is "N-E", default:`True`) If `True`, the labelling is shown under the path; otherwise, it is shown to the right of the path.

EXAMPLES:

```
sage: for D in DyckWords(3): D.pretty_print()
     _
   _|
 _|  .
|  . .
   ___
  | x
 _|  .
|  . .
     _
 ___|
| x  .
|  . .
   ___
 _| x
| x  .
|  . .
 _____
| x x
| x  .
|  . .

sage: for D in DyckWords(3): D.pretty_print(type="NE-SE")
/\/\/\
   /\
/\/  \
 /\
/  \/\
 /\/\
/    \
  /\
 /  \
/    \

sage: D = DyckWord([1,1,1,0,1,0,0,1,1])
sage: D.pretty_print()
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
|  . . . . .

sage: D = DyckWord([1,1,1,0,1,0,0,1,1,0])
sage: D.pretty_print()
       _
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
|  . . . . .
```

```
sage: D = DyckWord([1,1,1,0,1,0,0,1,1,0,0])
sage: D.pretty_print()
       ___
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
|  . . . . .

sage: DyckWord(area_sequence=[0,1,0]).pretty_print(labelling=[1,3,2])
     _
 ___|2
|3x  .
|1 . .

sage: DyckWord(area_sequence=[0,1,0]).pretty_print(labelling=[1,3,2],underpath=False)
     _
 ___|  2
| x  . 3
|  . . 1

sage: DyckWord(area_sequence=[0,1,1,2,3,2,3,3,2,0,1,1,2,3,4,2,3]).pretty_print()
                        _______
                       | x x x
                  _____| x x  .
                 | x x x x  . .
                 | x x x  . . .
                 | x x  . . . .
                _| x  . . . . .
               | x  . . . . . .
           ____|  . . . . . . .
        ___| x x  . . . . . . .
       _| x x x  . . . . . . .
      | x x x  . . . . . . . .
   ___| x x  . . . . . . . . .
  | x x x  . . . . . . . . . .
  | x x  . . . . . . . . . . .
 _| x  . . . . . . . . . . . .
| x  . . . . . . . . . . . . .
|  . . . . . . . . . . . . . .

sage: DyckWord(area_sequence=[0,1,1,2,3,2,3,3,2,0,1,1,2,3,4,2,3]).pretty_print(labelling=rar
                        _______
                       | x x x  16
                  _____| x x  . 15
                 | x x x x  . . 14
                 | x x x  . . . 13
                 | x x  . . . . 12
                _| x  . . . . . 11
               | x  . . . . . . 10
           ____|  . . . . . . .  9
        ___| x x  . . . . . . .  8
       _| x x x  . . . . . . .  7
      | x x x  . . . . . . . .  6
   ___| x x  . . . . . . . . .  5
  | x x x  . . . . . . . . . .  4
  | x x  . . . . . . . . . . .  3
```

```
 _| x  . . . . . . . . . . . . . . . .  2
| x  . . . . . . . . . . . . . . . . .  1
|  . . . . . . . . . . . . . . . . . .  0
```

```
sage: DyckWord([]).pretty_print()
.
```

pretty_print (*type=None*, *labelling=None*, *underpath=True*)

Display a DyckWord as a lattice path in the $\mathbf{Z}^2$ grid.

If the `type` is "N-E", then the a cell below the diagonal is indicated by a period, whereas a cell below the path but above the diagonal is indicated by an x. If a list of labels is included, they are displayed along the vertical edges of the Dyck path.

If the `type` is "NE-SE", then the path is simply printed as up steps and down steps.

INPUT:

- `type` – (default: `None`) can either be:
  - `None` to use the global option default
  - "N-E" to show `self` as a path of north and east steps, or
  - "NE-SE" to show `self` as a path of north-east and south-east steps.
- `labelling` – (if type is "N-E") a list of labels assigned to the up steps in `self`.
- `underpath` – (if type is "N-E", default:`True`) If `True`, the labelling is shown under the path; otherwise, it is shown to the right of the path.

EXAMPLES:

```
sage: for D in DyckWords(3): D.pretty_print()
    _
  _|
 _|  .
|  . .
   ___
  | x
 _|  .
|  . .
    _
 ___|
| x  .
|  . .
   ___
 _| x
| x  .
|  . .
 _____
| x x
| x  .
|  . .
```

```
sage: for D in DyckWords(3): D.pretty_print(type="NE-SE")
/\/\/\
   /\
/\/  \
 /\
/  \/\
```

```
 /\/\
/    \
  /\
 /  \
/    \

sage: D = DyckWord([1,1,1,0,1,0,0,1,1])
sage: D.pretty_print()
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
|  . . . . .

sage: D = DyckWord([1,1,1,0,1,0,0,1,1,0])
sage: D.pretty_print()
       _
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
|  . . . . .

sage: D = DyckWord([1,1,1,0,1,0,0,1,1,0,0])
sage: D.pretty_print()
       ___
      | x x
   ___| x  .
 _| x x  . .
| x x  . . .
| x  . . . .
|  . . . . .

sage: DyckWord(area_sequence=[0,1,0]).pretty_print(labelling=[1,3,2])
     _
 ___|2
|3x  .
|1 . .

sage: DyckWord(area_sequence=[0,1,0]).pretty_print(labelling=[1,3,2],underpath=False)
     _
 ___|  2
| x  . 3
|  . . 1

sage: DyckWord(area_sequence=[0,1,1,2,3,2,3,3,2,0,1,1,2,3,4,2,3]).pretty_print()
                        _______
                       | x x x
                  _____| x x  .
                 | x x x x  . .
                 | x x x  . . .
                 | x x  . . . .
                _| x  . . . . .
               | x  . . . . . .
          _____|  . . . . . . .
         ___| x x  . . . . . . .
```

```
        _| x x x  . . . . . . . . . .
       | x x x  . . . . . . . . . . .
    ___| x x  . . . . . . . . . . . .
   | x x x  . . . . . . . . . . . . .
   | x x  . . . . . . . . . . . . . .
 _| x  . . . . . . . . . . . . . . .
| x  . . . . . . . . . . . . . . . .
|  . . . . . . . . . . . . . . . . .
```

```
sage: DyckWord(area_sequence=[0,1,1,2,3,2,3,3,2,0,1,1,2,3,4,2,3]).pretty_print(labelling=ran
```

```
                          _______
                         | x x x  16
                      ___| x x  . 15
                     | x x x x  . . 14
                     | x x x  . . . 13
                     | x x  . . . . 12
                   _| x  . . . . . 11
                  | x  . . . . . . 10
              ____|  . . . . . . .  9
           ___| x x  . . . . . . .  8
          _| x x x  . . . . . . . .  7
         | x x x  . . . . . . . . .  6
      ___| x x  . . . . . . . . . .  5
     | x x x  . . . . . . . . . . .  4
     | x x  . . . . . . . . . . . .  3
   _| x  . . . . . . . . . . . . .  2
  | x  . . . . . . . . . . . . . .  1
  |  . . . . . . . . . . . . . . .  0
```

```
sage: DyckWord([]).pretty_print()
.
```

returns_to_zero()

Return a list of positions where `self` has height 0, excluding the position 0.

EXAMPLES:

```
sage: DyckWord([]).returns_to_zero()
[]
sage: DyckWord([1, 0]).returns_to_zero()
[2]
sage: DyckWord([1, 0, 1, 0]).returns_to_zero()
[2, 4]
sage: DyckWord([1, 1, 0, 0]).returns_to_zero()
[4]
```

rise_composition()

The sequences of lengths of runs of 1's in `self`. Also equal to the sequence of lengths of vertical segments in the Dyck path.

EXAMPLES:

```
sage: DyckWord([1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]).pretty_print()
```

```
          ___
         | x
  _______|  .
 | x x x  . .
 | x x  . . .
_| x  . . . .
```

```
| x  . . . . .
|  . . . . . .
```

```
sage: DyckWord([1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]).rise_composition()
[2, 3, 2]
sage: DyckWord([1,1,0,0]).rise_composition()
[2]
sage: DyckWord([1,0,1,0]).rise_composition()
[1, 1]
```

set_latex_options(*D*)

Set the latex options for use in the _latex_ function. The default values are set in the __init__ function.

- tikz_scale – (default: 1) scale for use with the tikz package.
- diagonal – (default: False) boolean value to draw the diagonal or not.
- line width – (default: 2*‘‘tikz_scale‘‘) value representing the line width.
- color – (default: black) the line color.
- bounce path – (default: False) boolean value to indicate if the bounce path should be drawn.
- peaks – (default: False) boolean value to indicate if the peaks should be displayed.
- valleys – (default: False) boolean value to indicate if the valleys should be displayed.

INPUT:

- D – a dictionary with a list of latex parameters to change

EXAMPLES:

```
sage: D = DyckWord([1,0,1,0,1,0])
sage: D.set_latex_options({"tikz_scale":2})
sage: D.set_latex_options({"valleys":True, "color":"blue"})
```

tamari_interval(*other*)

Return the Tamari interval between self and other as a TamariIntervalPoset.

A “Tamari interval” means an interval in the Tamari order. The Tamari order on the set of Dyck words of size n is the partial order obtained from the Tamari order on the set of binary trees of size n (see tamari_lequal()) by means of the Tamari bijection between Dyck words and binary trees (to_dyck_word_tamari()).

INPUT:

- other – a Dyck word greater or equal to self in the Tamari order

EXAMPLES:

```
sage: dw = DyckWord([1, 1, 0, 1, 0, 0, 1, 0])
sage: ip = dw.tamari_interval(DyckWord([1, 1, 1, 0, 0, 1, 0, 0])); ip
The Tamari interval of size 4 induced by relations [(2, 4), (3, 4), (3, 1), (2, 1)]
sage: ip.lower_dyck_word()
[1, 1, 0, 1, 0, 0, 1, 0]
sage: ip.upper_dyck_word()
[1, 1, 1, 0, 0, 1, 0, 0]
sage: ip.interval_cardinality()
4
sage: ip.number_of_tamari_inversions()
2
```

```
sage: list(ip.dyck_words())
[[1, 1, 1, 0, 0, 1, 0, 0],
 [1, 1, 1, 0, 0, 0, 1, 0],
 [1, 1, 0, 1, 0, 1, 0, 0],
 [1, 1, 0, 1, 0, 0, 1, 0]]
sage: dw.tamari_interval(DyckWord([1,1,0,0,1,1,0,0]))
Traceback (most recent call last):
...
ValueError: The two Dyck words are not comparable on the Tamari lattice.
```

to_area_sequence()

Return the area sequence of the Dyck word `self`.

The area sequence of a Dyck word w is defined as follows: Representing the Dyck word w as a Dyck path from $(0, 0)$ to (n, n) using N and E steps (this involves padding w by E steps until w reaches the main diagonal if w is not already a complete Dyck path), the area sequence of w is the sequence $(a_1, a_2, \ldots, a_n)$, where a_i is the number of full cells in the i-th row of the rectangle $[0, n] \times [0, n]$ which lie completely above the diagonal. (The cells are the regions into which the rectangle is subdivided by the lines $x = i$ with i integer and the lines $y = j$ with j integer. The i-th row consists of all the cells between the lines $y = i - 1$ and $y = i$.)

An alternative definition: Representing the Dyck word w as a Dyck path consisting of NE and SE steps, the area sequence is the sequence of ordinates of all lattice points on the path which are starting points of NE steps.

A list of integers l is the area sequence of some Dyck path if and only if it satisfies $l_0 = 0$ and $0 \leq l_{i+1} \leq l_i + 1$ for $i > 0$.

EXAMPLES:

```
sage: DyckWord([]).to_area_sequence()
[]
sage: DyckWord([1, 0]).to_area_sequence()
[0]
sage: DyckWord([1, 1, 0, 0]).to_area_sequence()
[0, 1]
sage: DyckWord([1, 0, 1, 0]).to_area_sequence()
[0, 0]
sage: all(dw ==
....:     DyckWords().from_area_sequence(dw.to_area_sequence())
....:     for i in range(6) for dw in DyckWords(i))
True
sage: DyckWord([1,0,1,0,1,0,1,0,1,0]).to_area_sequence()
[0, 0, 0, 0, 0]
sage: DyckWord([1,1,1,1,1,0,0,0,0,0]).to_area_sequence()
[0, 1, 2, 3, 4]
sage: DyckWord([1,1,1,1,0,1,0,0,0,0]).to_area_sequence()
[0, 1, 2, 3, 3]
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).to_area_sequence()
[0, 1, 1, 0, 1, 1, 1]
```

to_binary_tree(*usemap='1L0R'*)

Return a binary tree recursively constructed from the Dyck path `self` by the map `usemap`. The default `usemap` is `'1L0R'` which means:

- an empty Dyck word is a leaf,
- a non empty Dyck word reads $1L0R$ where L and R correspond to respectively its left and right subtrees.

INPUT:

- usemap – a string, either '1L0R', '1R0L', 'L1R0', 'R1L0'

Other valid usemap are '1R0L', 'L1R0', and 'R1L0'. These correspond to different maps from Dyck paths to binary trees, whose recursive definitions are hopefully clear from the names.

EXAMPLES:

```
sage: dw = DyckWord([1,0])
sage: dw.to_binary_tree()
[., .]
sage: dw = DyckWord([])
sage: dw.to_binary_tree()
.
sage: dw = DyckWord([1,0,1,1,0,0])
sage: dw.to_binary_tree()
[., [[., .], .]]
sage: dw.to_binary_tree("L1R0")
[[., .], [., .]]
sage: dw = DyckWord([1,0,1,1,0,0,1,1,1,0,1,0,0,0])
sage: dw.to_binary_tree() == dw.to_binary_tree("1R0L").left_right_symmetry()
True
sage: dw.to_binary_tree() == dw.to_binary_tree("L1R0").left_border_symmetry()
False
sage: dw.to_binary_tree("1R0L") == dw.to_binary_tree("L1R0").left_border_symmetry()
True
sage: dw.to_binary_tree("R1L0") == dw.to_binary_tree("L1R0").left_right_symmetry()
True
sage: dw.to_binary_tree("R10L")
Traceback (most recent call last):
...
ValueError: R10L is not a correct map
```

to_binary_tree_tamari()

Return the binary tree corresponding to self in a way which is consistent with the Tamari orders on the set of Dyck paths and on the set of binary trees.

This is the 'L1R0' map documented in to_binary_tree().

EXAMPLES:

```
sage: DyckWord([1,0]).to_binary_tree_tamari()
[., .]
sage: DyckWord([1,0,1,1,0,0]).to_binary_tree_tamari()
[[., .], [., .]]
sage: DyckWord([1,0,1,0,1,0]).to_binary_tree_tamari()
[[[., .], .], .]
```

to_path_string(*unicode=False*)

A path representation of the Dyck word consisting of steps / and \ .

EXAMPLES:

```
sage: print DyckWord([1, 0, 1, 0]).to_path_string()
/\/\
sage: print DyckWord([1, 1, 0, 0]).to_path_string()
 /\
/  \
sage: print DyckWord([1,1,0,1,1,0,0,1,0,1,0,0]).to_path_string()
    /\
```

```
 /\/  \/\/\
/          \
```

to_standard_tableau()

Return a standard tableau of shape (a, b) where a is the number of open symbols and b is the number of close symbols in `self`.

EXAMPLES:

```
sage: DyckWord([]).to_standard_tableau()
[]
sage: DyckWord([1, 0]).to_standard_tableau()
[[1], [2]]
sage: DyckWord([1, 1, 0, 0]).to_standard_tableau()
[[1, 2], [3, 4]]
sage: DyckWord([1, 0, 1, 0]).to_standard_tableau()
[[1, 3], [2, 4]]
sage: DyckWord([1]).to_standard_tableau()
[[1]]
sage: DyckWord([1, 0, 1]).to_standard_tableau()
[[1, 3], [2]]
```

touch_composition()

Return a composition which indicates the positions where `self` returns to the diagonal.

This assumes `self` to be a complete Dyck word.

OUTPUT:

- a composition of length equal to the length of the Dyck word.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).touch_composition()
[1, 1]
sage: DyckWord([1, 1, 0, 0]).touch_composition()
[2]
sage: DyckWord([1, 1, 0, 0, 1, 0]).touch_composition()
[2, 1]
sage: DyckWord([1, 0, 1, 1, 0, 0]).touch_composition()
[1, 2]
sage: DyckWord([]).touch_composition()
[]
```

touch_points()

Return the abscissae (or, equivalently, ordinates) of the points where the Dyck path corresponding to `self` (comprising NE and SE steps) touches the main diagonal. This includes the last point (if it is on the main diagonal) but excludes the beginning point.

Note that these abscissae are precisely the entries of `returns_to_zero()` divided by 2.

OUTPUT:

- a list of integers indicating where the path touches the diagonal

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).touch_points()
[1, 2]
sage: DyckWord([1, 1, 0, 0]).touch_points()
[2]
sage: DyckWord([1, 1, 0, 0, 1, 0]).touch_points()
```

```
[2, 3]
sage: DyckWord([1, 0, 1, 1, 0, 0]).touch_points()
[1, 3]
```

valleys()

Return a list of the positions of the valleys of a Dyck word.

A valley is 0 followed by a 1.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).valleys()
[1]
sage: DyckWord([1, 1, 0, 0]).valleys()
[]
sage: DyckWord([1,1,0,1,0,1,0,0]).valleys()
[2, 4]
```

class sage.combinat.dyck_word.**DyckWordBacktracker**(*k1*, *k2*)

Bases: sage.combinat.backtrack.GenericBacktracker

This class is an iterator for all Dyck words with n opening parentheses and $n - k$ closing parentheses using the backtracker class. It is used by the DyckWords_size class.

This is not really meant to be called directly, partially because it fails in a couple corner cases: DWB(0) yields [0], not the empty word, and DWB(k, k+1) yields something (it shouldn't yield anything). This could be fixed with a sanity check in _rec(), but then we'd be doing the sanity check *every time* we generate new objects; instead, we do *one* sanity check in DyckWords and assume here that the sanity check has already been made.

AUTHOR:

- Dan Drake (2008-05-30)

class sage.combinat.dyck_word.**DyckWord_complete**(*parent*, *l*, *latex_options={}*)

Bases: sage.combinat.dyck_word.DyckWord

The class of complete Dyck words. A Dyck word is complete, if it contains as many closers as openers.

For further information on Dyck words, see DyckWords_class.

area()

Return the area for self corresponding to the area of the Dyck path.

One can view a balanced Dyck word as a lattice path from $(0, 0)$ to (n, n) in the first quadrant by letting '1's represent steps in the direction $(1, 0)$ and '0's represent steps in the direction $(0, 1)$. The resulting path will remain weakly above the diagonal $y = x$.

The area statistic is the number of complete squares in the integer lattice which are below the path and above the line $y = x$. The 'half-squares' directly above the line $y = x$ do not contribute to this statistic.

EXAMPLES:

```
sage: dw = DyckWord([1,0,1,0])
sage: dw.area() # 2 half-squares, 0 complete squares
0

sage: dw = DyckWord([1,1,1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,0])
sage: dw.area()
19
```

```
sage: DyckWord([1,1,1,1,0,0,0,0]).area()
6
sage: DyckWord([1,1,1,0,1,0,0,0]).area()
5
sage: DyckWord([1,1,1,0,0,1,0,0]).area()
4
sage: DyckWord([1,1,1,0,0,0,1,0]).area()
3
sage: DyckWord([1,0,1,1,0,1,0,0]).area()
2
sage: DyckWord([1,1,0,1,1,0,0,0]).area()
4
sage: DyckWord([1,1,0,0,1,1,0,0]).area()
2
sage: DyckWord([1,0,1,1,1,0,0,0]).area()
3
sage: DyckWord([1,0,1,1,0,0,1,0]).area()
1
sage: DyckWord([1,0,1,0,1,1,0,0]).area()
1
sage: DyckWord([1,1,0,0,1,0,1,0]).area()
1
sage: DyckWord([1,1,0,1,0,0,1,0]).area()
2
sage: DyckWord([1,1,0,1,0,1,0,0]).area()
3
sage: DyckWord([1,0,1,0,1,0,1,0]).area()
0
```

area_dinv_to_bounce_area_map()

Return the image of `self` under the map which sends a Dyck word with `area` equal to r and `dinv` equal to s to a Dyck word with `bounce` equal to r and `area` equal to s .

The inverse of this map is `bounce_area_to_area_dinv_map()`.

For a definition of this map, see [Hag2008] p. 50 where it is called ζ. However, this map differs from Haglund's map by an application of `reverse()` (as does the definition of the `bounce()` statistic).

EXAMPLES:

```
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).area_dinv_to_bounce_area_map()
[1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0]
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).area()
5
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).dinv()
13
sage: DyckWord([1,1,1,1,1,0,0,0,1,0,0,1,0,0]).area()
13
sage: DyckWord([1,1,1,1,1,0,0,0,1,0,0,1,0,0]).bounce()
5
sage: DyckWord([1,1,1,1,1,0,0,0,1,0,0,1,0,0]).area_dinv_to_bounce_area_map()
[1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0]
sage: DyckWord([1,1,0,0]).area_dinv_to_bounce_area_map()
[1, 0, 1, 0]
sage: DyckWord([1,0,1,0]).area_dinv_to_bounce_area_map()
[1, 1, 0, 0]
```

bounce()

Return the bounce statistic of `self` due to J. Haglund, see [Hag2008].

One can view a balanced Dyck word as a lattice path from $(0,0)$ to (n,n) in the first quadrant by letting '1's represent steps in the direction $(0,1)$ and '0's represent steps in the direction $(1,0)$. The resulting path will remain weakly above the diagonal $y = x$.

We describe the bounce statistic of such a path in terms of what is known as the "bounce path".

We can think of our bounce path as describing the trail of a billiard ball shot West from (n,n), which "bounces" down whenever it encounters a vertical step and "bounces" left when it encounters the line $y = x$.

The bouncing ball will strike the diagonal at the places

$$(0,0), (j_1, j_1), (j_2, j_2), \ldots, (j_r - 1, j_r - 1), (j_r, j_r) = (n,n).$$

We define the bounce to be the sum $\sum_{i=1}^{r-1} j_i$.

EXAMPLES:

```
sage: DyckWord([1,1,1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,0]).bounce()
7
sage: DyckWord([1,1,1,1,0,0,0,0]).bounce()
0
sage: DyckWord([1,1,1,0,1,0,0,0]).bounce()
1
sage: DyckWord([1,1,1,0,0,1,0,0]).bounce()
2
sage: DyckWord([1,1,1,0,0,0,1,0]).bounce()
3
sage: DyckWord([1,0,1,1,0,1,0,0]).bounce()
3
sage: DyckWord([1,1,0,1,1,0,0,0]).bounce()
1
sage: DyckWord([1,1,0,0,1,1,0,0]).bounce()
2
sage: DyckWord([1,0,1,1,1,0,0,0]).bounce()
1
sage: DyckWord([1,0,1,1,0,0,1,0]).bounce()
4
sage: DyckWord([1,0,1,0,1,1,0,0]).bounce()
3
sage: DyckWord([1,1,0,0,1,0,1,0]).bounce()
5
sage: DyckWord([1,1,0,1,0,0,1,0]).bounce()
4
sage: DyckWord([1,1,0,1,0,1,0,0]).bounce()
2
sage: DyckWord([1,0,1,0,1,0,1,0]).bounce()
6
```

bounce_area_to_area_dinv_map(D)

Return the image of the Dyck word under the map which sends a Dyck word with `bounce` equal to r and `area` equal to s to a Dyck word with `area` equal to r and `dinv` equal to s .

This implementation uses a recursive method by saying that the last entry in the area sequence of D is equal to the number of touch points of the Dyck path minus 1 of the image of this map.

The inverse of this map is `area_dinv_to_bounce_area_map()`.

For a definition of this map, see [Hag2008] p. 50 where it is called ζ^{-1}. However, this map differs from Haglund's map by an application of `reverse()` (as does the definition of the `bounce()` statistic).

EXAMPLES:

```
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).bounce_area_to_area_dinv_map()
[1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0]
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).area()
5
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).bounce()
9
sage: DyckWord([1,1,0,0,1,1,1,1,0,0,1,0,0,0]).area()
9
sage: DyckWord([1,1,0,0,1,1,1,1,0,0,1,0,0,0]).dinv()
5
sage: all(D==D.bounce_area_to_area_dinv_map().area_dinv_to_bounce_area_map() for D in DyckWo
True
sage: DyckWord([1,1,0,0]).bounce_area_to_area_dinv_map()
[1, 0, 1, 0]
sage: DyckWord([1,0,1,0]).bounce_area_to_area_dinv_map()
[1, 1, 0, 0]
```

bounce_path()

Return the bounce path of `self` formed by starting at (n, n) and traveling West until encountering the first vertical step of `self`, then South until encountering the diagonal, then West again to hit the path, etc. until the $(0, 0)$ point is reached. The path followed by this walk is the bounce path.

See also:

`bounce()`

EXAMPLES:

```
sage: DyckWord([1,1,0,1,0,0]).bounce_path()
[1, 0, 1, 1, 0, 0]
sage: DyckWord([1,1,1,0,0,0]).bounce_path()
[1, 1, 1, 0, 0, 0]
sage: DyckWord([1,0,1,0,1,0]).bounce_path()
[1, 0, 1, 0, 1, 0]
sage: DyckWord([1,1,1,1,0,0,1,0,0,0]).bounce_path()
[1, 1, 0, 0, 1, 1, 1, 0, 0, 0]
```

TESTS:

```
sage: DyckWord([]).bounce_path()
[]
sage: DyckWord([1,0]).bounce_path()
[1, 0]
```

characteristic_symmetric_function(*q=None*, *R=Fraction Field of Multivariate Polynomial Ring in q, t over Rational Field*)

The characteristic function of `self` is the sum of $q^{dinv(D,F)}Q_{ides(read(D,F))}$ over all permutation fillings of the Dyck path representing `self`, where $ides(read(D, F))$ is the descent composition of the inverse of the reading word of the filling.

INPUT:

- `q` – (default: `q = R('q')`) a parameter for the generating function power
- `R` – (default : `R = QQ['q','t'].fraction_field()`) the base ring to do the calculations over

OUTPUT:

- an element of the symmetric functions over the ring `R` (in the Schur basis).

EXAMPLES:

```
sage: R = QQ['q','t'].fraction_field()
sage: (q,t) = R.gens()
sage: f = sum(t**D.area()*D.characteristic_symmetric_function() for D in DyckWords(3)); f
(q^3+q^2*t+q*t^2+t^3+q*t)*s[1, 1, 1] + (q^2+q*t+t^2+q+t)*s[2, 1] + s[3]
sage: f.nabla(power=-1)
s[1, 1, 1]
```

decomposition_reverse()

Return the involution of `self` with a recursive definition.

If a Dyck word D decomposes as $1D_1 0 D_2$ where D_1 and D_2 are complete Dyck words then the decomposition reverse is $1\phi(D_2)0\phi(D_1)$.

EXAMPLES:

```
sage: DyckWord([1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0]).decomposition_reverse()
[1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]
sage: DyckWord([1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]).decomposition_reverse()
[1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0]
sage: DyckWord([1,1,0,0]).decomposition_reverse()
[1, 0, 1, 0]
sage: DyckWord([1,0,1,0]).decomposition_reverse()
[1, 1, 0, 0]
```

dinv(*labeling=None*)

Return the dinv statistic of `self` due to M. Haiman, see [Hag2008].

If a labeling is provided then this function returns the dinv of the labeled Dyck word.

INPUT:

- `labeling` – an optional argument to be viewed as the labelings of the vertical edges of the Dyck path

OUTPUT:

- an integer representing the `dinv` statistic of the Dyck path or the labelled Dyck path.

EXAMPLES:

```
sage: DyckWord([1,0,1,0,1,0,1,0,1,0]).dinv()
10
sage: DyckWord([1,1,1,1,1,0,0,0,0,0]).dinv()
0
sage: DyckWord([1,1,1,1,0,1,0,0,0,0]).dinv()
1
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).dinv()
13
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).dinv([1,2,3,4,5,6,7])
11
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).dinv([6,7,5,3,4,2,1])
2
```

first_return_decomposition()

Decompose a Dyck word into a pair of Dyck words (potentially empty) where the first word consists of the word after the first up step and the corresponding matching closing parenthesis.

EXAMPLES:

```
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).first_return_decomposition()
([1, 0, 1, 0], [1, 1, 0, 1, 0, 1, 0, 0])
```

```
sage: DyckWord([1,1,0,0]).first_return_decomposition()
([1, 0], [])
sage: DyckWord([1,0,1,0]).first_return_decomposition()
([], [1, 0])
```

list_parking_functions()

Return all parking functions whose supporting Dyck path is `self`.

EXAMPLES:

```
sage: DyckWord([1,1,0,0,1,0]).list_parking_functions()
Permutations of the multi-set [1, 1, 3]
sage: DyckWord([1,1,1,0,0,0]).list_parking_functions()
Permutations of the multi-set [1, 1, 1]
sage: DyckWord([1,0,1,0,1,0]).list_parking_functions()
Permutations of the set [1, 2, 3]
```

major_index()

Return the major index of `self` .

The major index of a Dyck word D is the sum of the positions of the valleys of D (when started counting at position `1`).

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).major_index()
2
sage: DyckWord([1, 1, 0, 0]).major_index()
0
sage: DyckWord([1, 1, 0, 0, 1, 0]).major_index()
4
sage: DyckWord([1, 0, 1, 1, 0, 0]).major_index()
2
```

TESTS:

```
sage: DyckWord([]).major_index()
0
sage: DyckWord([1, 0]).major_index()
0
```

number_of_parking_functions()

Return the number of parking functions with `self` as the supporting Dyck path.

One representation of a parking function is as a pair consisting of a Dyck path and a permutation π such that if $[a_0, a_1, \ldots, a_{n-1}]$ is the area_sequence of the Dyck path (see `to_area_sequence`) then the permutation π satisfies $\pi_i < \pi_{i+1}$ whenever $a_i < a_{i+1}$. This function counts the number of permutations π which satisfy this condition.

EXAMPLES:

```
sage: DyckWord(area_sequence=[0,1,2]).number_of_parking_functions()
1
sage: DyckWord(area_sequence=[0,1,1]).number_of_parking_functions()
3
sage: DyckWord(area_sequence=[0,1,0]).number_of_parking_functions()
3
sage: DyckWord(area_sequence=[0,0,0]).number_of_parking_functions()
6
```

number_of_tunnels(*tunnel_type='centered'*)

Return the number of tunnels of `self` of type `tunnel_type`.

A tunnel is a pair (a, b) where `a` is the position of an open parenthesis and `b` is the position of the matching close parenthesis. If $a + b = n$ then the tunnel is called *centered* . If $a + b < n$ then the tunnel is called *left* and if $a + b > n$, then the tunnel is called *right*.

INPUT:

- `tunnel_type` – (default: `'centered'`) can be one of the following: `'left'`, `'right'`, `'centered'`, or `'all'`.

EXAMPLES:

```
sage: DyckWord([1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]).number_of_tunnels()
0
sage: DyckWord([1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]).number_of_tunnels('left')
5
sage: DyckWord([1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]).number_of_tunnels('right')
2
sage: DyckWord([1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]).number_of_tunnels('all')
7
sage: DyckWord([1, 1, 0, 0]).number_of_tunnels('centered')
2
```

pyramid_weight()

A pyramid of `self` is a subsequence of the form 1^h0^h. A pyramid is maximal if it is neither preceded by a 1 nor followed by a 0.

The pyramid weight of a Dyck path is the sum of the lengths of the maximal pyramids and was defined in [DS1992].

EXAMPLES:

```
sage: DyckWord([1,1,0,1,1,1,0,0,1,0,0,0,1,1,0,0]).pyramid_weight()
6
sage: DyckWord([1,1,1,0,0,0]).pyramid_weight()
3
sage: DyckWord([1,0,1,0,1,0]).pyramid_weight()
3
sage: DyckWord([1,1,0,1,0,0]).pyramid_weight()
2
```

REFERENCES:

reading_permutation()

The permutation formed by taking the reading word of the Dyck path representing `self` (with N and E steps) if the vertical edges of the Dyck path are labeled from bottom to top with 1 through n and the diagonals are read from top to bottom starting with the diagonal furthest from the main diagonal.

EXAMPLES:

```
sage: DyckWord([1,0,1,0]).reading_permutation()
[2, 1]
sage: DyckWord([1,1,0,0]).reading_permutation()
[2, 1]
sage: DyckWord([1,1,0,1,0,0]).reading_permutation()
[3, 2, 1]
sage: DyckWord([1,1,0,0,1,0]).reading_permutation()
[2, 3, 1]
sage: DyckWord([1,0,1,1,0,0,1,0]).reading_permutation()
[3, 4, 2, 1]
```

reverse()
Return the reverse and complement of `self`.

This operation corresponds to flipping the Dyck path across the $y = -x$ line.

EXAMPLES:

```
sage: DyckWord([1,1,0,0,1,0]).reverse()
[1, 0, 1, 1, 0, 0]
sage: DyckWord([1,1,1,0,0,0]).reverse()
[1, 1, 1, 0, 0, 0]
sage: len(filter(lambda D: D.reverse() == D, DyckWords(5)))
10
```

TESTS:

```
sage: DyckWord([]).reverse()
[]
```

semilength()
Return the semilength of `self`.

The semilength of a complete Dyck word d is the number of openers and the number of closers.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).semilength()
2
```

TESTS:

```
sage: DyckWord([]).semilength()
0
```

to_132_avoiding_permutation()
Use the bijection by C. Krattenthaler in [Kra2001] to send `self` to a 132-avoiding permutation.

REFERENCES:

EXAMPLES:

```
sage: DyckWord([1,1,0,0]).to_132_avoiding_permutation()
[1, 2]
sage: DyckWord([1,0,1,0]).to_132_avoiding_permutation()
[2, 1]
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).to_132_avoiding_permutation()
[6, 5, 4, 7, 2, 1, 3]
```

TESTS:

```
sage: PD = [D.to_132_avoiding_permutation() for D in DyckWords(5)]
sage: all(pi.avoids([1,3,2]) for pi in PD)
True
```

to_312_avoiding_permutation()
Convert `self` to a 312-avoiding permutation using the bijection by Bandlow and Killpatrick in [BK2001]. Sends the area to the inversion number.

REFERENCES:

EXAMPLES:

```
sage: DyckWord([1,1,0,0]).to_312_avoiding_permutation()
[2, 1]
```

```
sage: DyckWord([1,0,1,0]).to_312_avoiding_permutation()
[1, 2]
sage: p = DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).to_312_avoiding_permutation(); p
[2, 3, 1, 5, 6, 7, 4]
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).area()
5
sage: p.length()
5
```

TESTS:

```
sage: PD = [D.to_312_avoiding_permutation() for D in DyckWords(5)]
sage: all(pi.avoids([3,1,2]) for pi in PD)
True
sage: all(D.area()==D.to_312_avoiding_permutation().length() for D in DyckWords(5))
True
```

to_321_avoiding_permutation()

Use the bijection (pp. 60-61 of [Knu1973] or section 3.1 of [CK2008]) to send `self` to a 321-avoiding permutation.

It is shown in [EP2004] that it sends the number of centered tunnels to the number of fixed points, the number of right tunnels to the number of exceedences, and the semilength plus the height of the middle point to 2 times the length of the longest increasing subsequence.

REFERENCES:

EXAMPLES:

```
sage: DyckWord([1,0,1,0]).to_321_avoiding_permutation()
[2, 1]
sage: DyckWord([1,1,0,0]).to_321_avoiding_permutation()
[1, 2]
sage: D = DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0])
sage: p = D.to_321_avoiding_permutation()
sage: p
[3, 5, 1, 6, 2, 7, 4]
sage: D.number_of_tunnels()
0
sage: p.number_of_fixed_points()
0
sage: D.number_of_tunnels('right')
4
sage: len(p.weak_excedences())-p.number_of_fixed_points()
4
sage: n = D.semilength()
sage: D.heights()[n] + n
8
sage: 2*p.longest_increasing_subsequence_length()
8
```

TESTS:

```
sage: PD = [D.to_321_avoiding_permutation() for D in DyckWords(5)]
sage: all(pi.avoids([3,2,1]) for pi in PD)
True
sage: to_perm = lambda x: x.to_321_avoiding_permutation()
sage: all(D.number_of_tunnels() == to_perm(D).number_of_fixed_points()
....:     for D in DyckWords(5))
True
```

```
sage: all(D.number_of_tunnels('right') == len(to_perm(D).weak_excedences())
....:     -to_perm(D).number_of_fixed_points() for D in DyckWords(5))
True
sage: all(D.heights()[5]+5 == 2*to_perm(D).longest_increasing_subsequence_length()
....:     for D in DyckWords(5))
True
```

to_Catalan_code()

Return the Catalan code associated to `self`.

A Catalan code of length n is a sequence $(a_1, a_2, \ldots, a_n)$ of n integers a_i such that:

- $0 \leq a_i \leq n - i$ for every i;
- if $i < j$ and $a_i > 0$ and $a_j > 0$ and $a_{i+1} = a_{i+2} = \cdots = a_{j-1} = 0$, then $a_i - a_j < j - i$.

It turns out that the Catalan codes of length n are in bijection with Dyck words.

The Catalan code of a Dyck word is example (x) in Richard Stanley's exercises on combinatorial interpretations for Catalan objects. The code in this example is the reverse of the description provided there. See [Sta-EC2] and [StaCat98].

EXAMPLES:

```
sage: DyckWord([]).to_Catalan_code()
[]
sage: DyckWord([1, 0]).to_Catalan_code()
[0]
sage: DyckWord([1, 1, 0, 0]).to_Catalan_code()
[0, 1]
sage: DyckWord([1, 0, 1, 0]).to_Catalan_code()
[0, 0]
sage: all(dw ==
....:     DyckWords().from_Catalan_code(dw.to_Catalan_code())
....:     for i in range(6) for dw in DyckWords(i))
True
```

to_alternating_sign_matrix()

Return `self` as an alternating sign matrix.

This is an inclusion map from Dyck words of length $2n$ to certain $n \times n$ alternating sign matrices.

EXAMPLES:

```
sage: DyckWord([1,1,1,0,1,0,0,0]).to_alternating_sign_matrix()
[ 0  0  1  0]
[ 1  0 -1  1]
[ 0  1  0  0]
[ 0  0  1  0]
sage: DyckWord([1,0,1,0,1,1,0,0]).to_alternating_sign_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 0 1]
[0 0 1 0]
```

to_non_decreasing_parking_function()

Bijection to non-decreasing parking functions. See there the method `to_dyck_word()` for more information.

EXAMPLES:

```
sage: DyckWord([]).to_non_decreasing_parking_function()
[]
sage: DyckWord([1,0]).to_non_decreasing_parking_function()
[1]
sage: DyckWord([1,1,0,0]).to_non_decreasing_parking_function()
[1, 1]
sage: DyckWord([1,0,1,0]).to_non_decreasing_parking_function()
[1, 2]
sage: DyckWord([1,0,1,1,0,1,0,0,1,0]).to_non_decreasing_parking_function()
[1, 2, 2, 3, 5]
```

TESTS:

```
sage: ld=DyckWords(5);
sage: list(ld) == [dw.to_non_decreasing_parking_function().to_dyck_word() for dw in ld]
True
```

to_noncrossing_partition(*bijection=None*)

Bijection of Biane from `self` to a noncrossing partition.

There is an optional parameter `bijection` that indicates if a different bijection from Dyck words to non-crossing partitions should be used (since there are potentially many).

If the parameter `bijection` is "Stump" then the bijection used is from [Stu2008], see also the method `to_noncrossing_permutation()`.

Thanks to Mathieu Dutour for describing the bijection. See also `from_noncrossing_partition()`.

EXAMPLES:

```
sage: DyckWord([]).to_noncrossing_partition()
[]
sage: DyckWord([1, 0]).to_noncrossing_partition()
[[1]]
sage: DyckWord([1, 1, 0, 0]).to_noncrossing_partition()
[[1, 2]]
sage: DyckWord([1, 1, 1, 0, 0, 0]).to_noncrossing_partition()
[[1, 2, 3]]
sage: DyckWord([1, 0, 1, 0, 1, 0]).to_noncrossing_partition()
[[1], [2], [3]]
sage: DyckWord([1, 1, 0, 1, 0, 0]).to_noncrossing_partition()
[[2], [1, 3]]
sage: DyckWord([]).to_noncrossing_partition("Stump")
[]
sage: DyckWord([1, 0]).to_noncrossing_partition("Stump")
[[1]]
sage: DyckWord([1, 1, 0, 0]).to_noncrossing_partition("Stump")
[[1, 2]]
sage: DyckWord([1, 1, 1, 0, 0, 0]).to_noncrossing_partition("Stump")
[[1, 3], [2]]
sage: DyckWord([1, 0, 1, 0, 1, 0]).to_noncrossing_partition("Stump")
[[1], [2], [3]]
sage: DyckWord([1, 1, 0, 1, 0, 0]).to_noncrossing_partition("Stump")
[[1, 2, 3]]
```

to_noncrossing_permutation()

Use the bijection by C. Stump in [Stu2008] to send `self` to a non-crossing permutation.

A non-crossing permutation when written in cyclic notation has cycles which are strictly increasing. Sends

the area to the inversion number and `self.major_index()` to $n(n-1) - maj(\sigma) - maj(\sigma^{-1})$. Uses the function `pealing()`

REFERENCES:

EXAMPLES:

```
sage: DyckWord([1,1,0,0]).to_noncrossing_permutation()
[2, 1]
sage: DyckWord([1,0,1,0]).to_noncrossing_permutation()
[1, 2]
sage: p = DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).to_noncrossing_permutation(); p
[2, 3, 1, 5, 6, 7, 4]
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).area()
5
sage: p.length()
5
```

TESTS:

```
sage: all(D.area()==D.to_noncrossing_permutation().length() for D in DyckWords(5))
True
sage: all(20-D.major_index()==D.to_noncrossing_permutation().major_index()
....:     +D.to_noncrossing_permutation().imajor_index() for D in DyckWords(5))
True
```

to_ordered_tree()

Return the ordered tree corresponding to `self` where the depth of the tree is the maximal height of `self`.

EXAMPLES:

```
sage: D = DyckWord([1,1,0,0])
sage: D.to_ordered_tree()
[[[]]]
sage: D = DyckWord([1,0,1,0])
sage: D.to_ordered_tree()
[[], []]
sage: D = DyckWord([1, 0, 1, 1, 0, 0])
sage: D.to_ordered_tree()
[[], [[]]]
sage: D = DyckWord([1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0])
sage: D.to_ordered_tree()
[[], [[], [[], [[], [[]]]]]]
```

TESTS:

```
sage: D = DyckWord([1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0])
sage: D == D.to_ordered_tree().to_dyck_word()
True
```

to_pair_of_standard_tableaux()

Convert `self` to a pair of standard tableaux of the same shape and of length less than or equal to two.

EXAMPLES:

```
sage: DyckWord([1,0,1,0]).to_pair_of_standard_tableaux()
([[1], [2]], [[1], [2]])
sage: DyckWord([1,1,0,0]).to_pair_of_standard_tableaux()
([[1, 2]], [[1, 2]])
sage: DyckWord([1,1,0,1,0,0,1,1,0,1,0,1,0,0]).to_pair_of_standard_tableaux()
([[1, 2, 4, 7], [3, 5, 6]], [[1, 2, 4, 6], [3, 5, 7]])
```

to_partition()
Return the partition associated to `self` .

This partition is determined by thinking of `self` as a lattice path and considering the cells which are above the path but within the $n \times n$ grid and the partition is formed by reading the sequence of the number of cells in this collection in each row.

OUTPUT:

- a partition representing the rows of cells in the square lattice and above the path

EXAMPLES:

```
sage: DyckWord([]).to_partition()
[]
sage: DyckWord([1,0]).to_partition()
[]
sage: DyckWord([1,1,0,0]).to_partition()
[]
sage: DyckWord([1,0,1,0]).to_partition()
[1]
sage: DyckWord([1,0,1,0,1,0]).to_partition()
[2, 1]
sage: DyckWord([1,1,0,0,1,0]).to_partition()
[2]
sage: DyckWord([1,0,1,1,0,0]).to_partition()
[1, 1]
```

to_permutation(*map*)
This is simply a method collecting all implemented maps from Dyck words to permutations.

INPUT:

- `map` – defines the map from Dyck words to permutations. These are currently:
  - `Bandlow-Killpatrick`: `to_312_avoiding_permutation()`
  - `Knuth`: `to_321_avoiding_permutation()`
  - `Krattenthaler`: `to_132_avoiding_permutation()`
  - `Stump`: `to_noncrossing_permutation()`

EXAMPLES:

```
sage: D = DyckWord([1,1,1,0,1,0,0,0])
sage: D.pretty_print()
   _____
 _| x x
| x x  .
| x  . .
|  . . .

sage: D.to_permutation(map="Bandlow-Killpatrick")
[3, 4, 2, 1]
sage: D.to_permutation(map="Stump")
[4, 2, 3, 1]
sage: D.to_permutation(map="Knuth")
[1, 2, 4, 3]
sage: D.to_permutation(map="Krattenthaler")
[2, 1, 3, 4]
```

to_triangulation()
Map `self` to a triangulation.

The map from complete Dyck words of length $2n$ to triangulations of $n+2$-gon given by this function is a bijection that can be described as follows.

Consider the Dyck word as a path from $(0,0)$ to (n,n) staying above the diagonal, where 1 is an up step and 0 is a right step. Then each horizontal step has a co-height (0 at the top and $n-1$ at most at the bottom). One reads the Dyck word from left to right. At the begining, all vertices from 0 to $n+1$ are available. For each horizontal step, one creates an edge from the vertex indexed by the co-height to the next available vertex. This chops out a triangle from the polygon and one removes the middle vertex of this triangle from the list of available vertices.

This bijection has the property that the set of smallest vertices of the edges in a triangulation is an encoding of the co-heights, from which the Dyck word can be easily recovered.

OUTPUT:

a list of pairs (i,j) that are the edges of the triangulations.

EXAMPLES:

```
sage: DyckWord([1, 1, 0, 0]).to_triangulation()
[(0, 2)]

sage: [t.to_triangulation() for t in DyckWords(3)]
[[(2, 4), (1, 4)],
[(2, 4), (0, 2)],
[(1, 3), (1, 4)],
[(1, 3), (0, 3)],
[(0, 2), (0, 3)]]
```

REFERENCES:

to_triangulation_as_graph()
Map `self` to a triangulation and return the result as a graph.

See `to_triangulation()` for the bijection used to map complete Dyck words to triangulations.

OUTPUT:

- a graph containing both the perimeter edges and the inner edges of a triangulation of a regular polygon.

EXAMPLES:

```
sage: g = DyckWord([1, 1, 0, 0, 1, 0]).to_triangulation_as_graph()
sage: g
Graph on 5 vertices
sage: g.edges(labels=False)
[(0, 1), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (3, 4)]
sage: g.show()        # not tested
```

tunnels()
Return the list of ranges of the matching parentheses in the Dyck word `self`. That is, if `(a,b)` is in `self.tunnels()`, then the matching parenthesis to `self[a]` is `self[b-1]`.

EXAMPLES:

```
sage: DyckWord([1, 1, 0, 1, 1, 0, 0, 1, 0, 0]).tunnels()
[(0, 10), (1, 3), (3, 7), (4, 6), (7, 9)]
```

class sage.combinat.dyck_word.**DyckWords**
Bases: sage.structure.unique_representation.UniqueRepresentation,

sage.structure.parent.Parent

Dyck words.

A Dyck word is a sequence $(w_1, \ldots, w_n)$ consisting of 1 s and 0 s, with the property that for any i with $1 \leq i \leq n$, the sequence $(w_1, \ldots, w_i)$ contains at least as many 1 s as 0 s.

A Dyck word is balanced if the total number of 1 s is equal to the total number of 0 s. The number of balanced Dyck words of length $2k$ is given by the Catalan number C_k.

EXAMPLES:

This class can be called with three keyword parameters k1, k2 and complete.

If neither k1 nor k2 are specified, then DyckWords returns the combinatorial class of all balanced (=complete) Dyck words, unless the keyword complete is set to False (in which case it returns the class of all Dyck words).

```
sage: DW = DyckWords(); DW
Complete Dyck words
sage: [] in DW
True
sage: [1, 0, 1, 0] in DW
True
sage: [1, 1, 0] in DW
False
sage: ADW = DyckWords(complete=False); ADW
Dyck words
sage: [] in ADW
True
sage: [1, 0, 1, 0] in ADW
True
sage: [1, 1, 0] in ADW
True
sage: [1, 0, 0] in ADW
False
```

If just k1 is specified, then it returns the balanced Dyck words with k1 opening parentheses and k1 closing parentheses.

```
sage: DW2 = DyckWords(2); DW2
Dyck words with 2 opening parentheses and 2 closing parentheses
sage: DW2.first()
[1, 0, 1, 0]
sage: DW2.last()
[1, 1, 0, 0]
sage: DW2.cardinality()
2
sage: DyckWords(100).cardinality() == catalan_number(100)
True
```

If k2 is specified in addition to k1, then it returns the Dyck words with k1 opening parentheses and k2 closing parentheses.

```
sage: DW32 = DyckWords(3,2); DW32
Dyck words with 3 opening parentheses and 2 closing parentheses
sage: DW32.list()
[[1, 0, 1, 0, 1],
 [1, 0, 1, 1, 0],
 [1, 1, 0, 0, 1],
 [1, 1, 0, 1, 0],
 [1, 1, 1, 0, 0]]
```

Element
alias of DyckWord

from_heights(*heights*)
Compute a Dyck word knowing its heights.

We view the Dyck word as a Dyck path from $(0, 0)$ to $(2n, 0)$ in the first quadrant by letting `1`'s represent steps in the direction $(1, 1)$ and `0`'s represent steps in the direction $(1, -1)$.

The `heights()` is the sequence of the y-coordinates of the $2n + 1$ lattice points along this path.

EXAMPLES:

```
sage: from sage.combinat.dyck_word import DyckWord
sage: D = DyckWords(complete=False)
sage: D.from_heights((0,))
[]
sage: D.from_heights((0, 1, 0))
[1, 0]
sage: D.from_heights((0, 1, 2, 1, 0))
[1, 1, 0, 0]
```

This also works for incomplete Dyck words:

```
sage: D.from_heights((0, 1, 2, 1, 2, 1))
[1, 1, 0, 1, 0]
sage: D.from_heights((0, 1, 2, 1))
[1, 1, 0]
```

See also:

`heights()`, `min_from_heights()`

TESTS:

```
sage: all(dw == D.from_heights(dw.heights())
....:     for i in range(7) for dw in DyckWords(i))
True

sage: D.from_heights((1, 2, 1))
Traceback (most recent call last):
...
ValueError: heights must start with 0: (1, 2, 1)
sage: D.from_heights((0, 1, 4, 1))
Traceback (most recent call last):
...
ValueError: consecutive heights must differ by exactly 1: (0, 1, 4, 1)
sage: D.from_heights(())
Traceback (most recent call last):
...
ValueError: heights must start with 0: ()
```

global_options(**get_value*, ***set_value*)
Set and display the global options for Dyck words. If no parameters are set, then the function returns a copy of the options dictionary.

The `options` to Dyck words can be accessed as the method `DyckWords.global_options` of `DyckWords` and related parent classes.

OPTIONS:

- `ascii_art` – (default: `path`) Specifies how the ascii art of Dyck words should be printed

– `path` – Using the path string

– `path_string` – alias for `path`

– `pretty_output` – Using pretty printing

– `pretty_print` – alias for `pretty_output`

• `diagram_style` – (default: `grid`)

– `N-E` – alias for `grid`

– `NE-SE` – alias for `line`

– `grid` – printing as paths on a grid using N and E steps

– `line` – printing as paths on a line using NE and SE steps

• `display` – (default: `list`) Specifies how Dyck words should be printed

– `lattice` – displayed on the lattice defined by `diagram_style`

– `list` – displayed as a list

• `latex_bounce_path` – (default: `False`) The default value for displaying the bounce path when latexed

• `latex_color` – (default: `black`) The default value for the color when latexed

• `latex_diagonal` – (default: `False`) The default value for displaying the diagonal when latexed

• `latex_line_width_scalar` – (default: `2`) The default value for the line width as amultiple of the tikz scale when latexed

• `latex_peaks` – (default: `False`) The default value for displaying the peaks when latexed

• `latex_tikz_scale` – (default: `1`) The default value for the tikz scale when latexed

• `latex_valleys` – (default: `False`) The default value for displaying the valleys when latexed

EXAMPLES:

```
sage: D = DyckWord([1, 1, 0, 1, 0, 0])
sage: D
[1, 1, 0, 1, 0, 0]
sage: DyckWords.global_options(display="lattice")
sage: D
   ___
 _| x
| x  .
|  . .
sage: DyckWords.global_options(diagram_style="line")
sage: D
 /\/\
/    \
sage: DyckWords.global_options.reset()
```

See `GlobalOptions` for more features of these options.

min_from_heights(*heights*)

Compute the smallest Dyck word which achieves or surpasses a given sequence of heights.

INPUT:

• `heights` – a nonempty list or iterable consisting of nonnegative integers, the first of which is 0

OUTPUT:

- The smallest Dyck word whose sequence of heights is componentwise higher-or-equal to `heights`. Here, "smaller" can be understood both in the sense of lexicographic order on the Dyck words, and in the sense of every vertex of the path having the smallest possible height.

See also:

- `heights()`
- `from_heights()`

EXAMPLES:

```
sage: D = DyckWords(complete=False)
sage: D.min_from_heights((0,))
[]
sage: D.min_from_heights((0, 1, 0))
[1, 0]
sage: D.min_from_heights((0, 0, 2, 0, 0))
[1, 1, 0, 0]
sage: D.min_from_heights((0, 0, 2, 0, 2, 0))
[1, 1, 0, 1, 0]
sage: D.min_from_heights((0, 0, 1, 0, 1, 0))
[1, 1, 0, 1, 0]
```

TESTS:

```
sage: D.min_from_heights(())
Traceback (most recent call last):
...
ValueError: heights must start with 0: ()
```

class sage.combinat.dyck_word.**DyckWords_all**

Bases: sage.combinat.dyck_word.DyckWords

All Dyck words.

class sage.combinat.dyck_word.**DyckWords_size**(*k1*, *k2*)

Bases: sage.combinat.dyck_word.DyckWords

Dyck words with k_1 openers and k_2 closers.

cardinality()

Return the number of Dyck words with k_1 openers and k_2 closers.

This number is

$$\frac{k_1 - k_2 + 1}{k_1 + 1}\binom{k_1 + k_2}{k_2} = \binom{k_1 + k_2}{k_2} - \binom{k_1 + k_2}{k_2 - 1}$$

if $k_2 \leq k_1 + 1$, and 0 if $k_2 > k_1$ (these numbers are the same if $k_2 = k_1 + 1$).

EXAMPLES:

```
sage: DyckWords(3, 2).cardinality()
5
sage: all(all(DyckWords(p, q).cardinality()
....:           == len(DyckWords(p, q).list()) for q in range(p + 1))
....:       for p in range(7))
True
```

sage.combinat.dyck_word.**from_ordered_tree**(*tree*)

TESTS:

```
sage: sage.combinat.dyck_word.from_ordered_tree(1)
Traceback (most recent call last):
...
NotImplementedError: TODO
```

sage.combinat.dyck_word.**is_a**(*obj*, *k1=None*, *k2=None*)

Test if `obj` is a Dyck word with exactly `k1` open symbols and exactly `k2` close symbols.

If `k1` is not specified, then the number of open symbols can be arbitrary. If `k1` is specified but `k2` is not, then `k2` is set to be `k1`.

EXAMPLES:

```
sage: from sage.combinat.dyck_word import is_a
sage: is_a([1,1,0,0])
True
sage: is_a([1,0,1,0])
True
sage: is_a([1,1,0,0], 2)
True
sage: is_a([1,1,0,0], 3)
False
sage: is_a([1,1,0,0], 3, 2)
False
sage: is_a([1,1,0])
True
sage: is_a([0,1,0])
False
sage: is_a([1,0,0])
False
sage: is_a([1,1,0],2,1)
True
sage: is_a([1,1,0],2)
False
sage: is_a([1,1,0],1,1)
False
```

sage.combinat.dyck_word.**is_area_sequence**(*seq*)

Test if a sequence l of integers satisfies $l_0 = 0$ and $0 \leq l_{i+1} \leq l_i + 1$ for $i > 0$.

EXAMPLES:

```
sage: from sage.combinat.dyck_word import is_area_sequence
sage: is_area_sequence([0,2,0])
False
sage: is_area_sequence([1,2,3])
False
sage: is_area_sequence([0,1,0])
True
sage: is_area_sequence([0,1,2])
True
sage: is_area_sequence([])
True
```

sage.combinat.dyck_word.**pealing**(*D*, *return_touches=False*)

A helper function for computing the bijection from a Dyck word to a 231-avoiding permutation using the bijection "Stump". For details see [Stu2008].

See also:

to_noncrossing_partition()

EXAMPLES:

```
sage: from sage.combinat.dyck_word import pealing
sage: pealing(DyckWord([1,1,0,0]))
[1, 0, 1, 0]
sage: pealing(DyckWord([1,0,1,0]))
[1, 0, 1, 0]
sage: pealing(DyckWord([1, 1, 0, 0, 1, 1, 1, 0, 0, 0]))
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
sage: pealing(DyckWord([1,1,0,0]),return_touches=True)
([1, 0, 1, 0], [[1, 2]])
sage: pealing(DyckWord([1,0,1,0]),return_touches=True)
([1, 0, 1, 0], [])
sage: pealing(DyckWord([1, 1, 0, 0, 1, 1, 1, 0, 0, 0]),return_touches=True)
([1, 0, 1, 0, 1, 0, 1, 0, 1, 0], [[1, 2], [3, 5]])
```

sage.combinat.dyck_word.**replace_parens**(*x*)

A map sending `'('` to open_symbol and `')'` to close_symbol, and raising an error on any input other than `'('` and `')'`. The values of the constants open_symbol and close_symbol are subject to change.

This is the inverse map of replace_symbols().

INPUT:

- x – either an opening or closing parenthesis

OUTPUT:

- If x is an opening parenthesis, replace x with the constant open_symbol.
- If x is a closing parenthesis, replace x with the constant close_symbol.
- Raise a ValueError if x is neither an opening nor a closing parenthesis.

See also:

replace_symbols()

EXAMPLES:

```
sage: from sage.combinat.dyck_word import replace_parens
sage: replace_parens('(')
1
sage: replace_parens(')')
0
sage: replace_parens(1)
Traceback (most recent call last):
...
ValueError
```

sage.combinat.dyck_word.**replace_symbols**(*x*)

A map sending open_symbol to `'('` and close_symbol to `')'`, and raising an error on any input other than open_symbol and close_symbol. The values of the constants open_symbol and close_symbol are subject to change.

This is the inverse map of replace_parens().

INPUT:

- x – either open_symbol or close_symbol.

OUTPUT:

- If x is open_symbol, replace x with '('.
- If x is close_symbol, replace x with ')'.
- If x is neither open_symbol nor close_symbol, a ValueError is raised.

See also:

replace_parens()

EXAMPLES:

```
sage: from sage.combinat.dyck_word import replace_symbols
sage: replace_symbols(1)
'('
sage: replace_symbols(0)
')'
sage: replace_symbols(3)
Traceback (most recent call last):
...
ValueError
```

5.1.90 Substitutions over unit cube faces (Rauzy fractals)

This module implements the $E_1^*(\sigma)$ substitution associated with a one-dimensional substitution σ, that acts on unit faces of dimension $(d-1)$ in $\mathbf{R}^d$.

This module defines the following classes and functions:

- Face - a class to model a face
- Patch - a class to model a finite set of faces
- E1Star - a class to model the $E_1^*(\sigma)$ application defined by the substitution sigma

See the documentation of these objects for more information.

The convention for the choice of the unit faces and the definition of $E_1^*(\sigma)$ varies from article to article. Here, unit faces are defined by

$$\begin{aligned} [x,1]^* &= \{x+\lambda e_2+\mu e_3 : \lambda,\mu\in[0,1]\} \\ [x,2]^* &= \{x+\lambda e_1+\mu e_3 : \lambda,\mu\in[0,1]\} \\ [x,3]^* &= \{x+\lambda e_1+\mu e_2 : \lambda,\mu\in[0,1]\} \end{aligned}$$

and the dual substitution $E_1^*(\sigma)$ is defined by

$$E_1^*(\sigma)([x,i]^*) = \bigcup_{k=1,2,3} \bigcup_{s|\sigma(k)=pis} [M^{-1}(x+\ell(s)),k]^*,$$

where $\ell(s)$ is the abelianized of s, and M is the matrix of σ.

AUTHORS:

- Franco Saliola (2009): initial version
- Vincent Delecroix, Timo Jolivet, Stepan Starosta, Sebastien Labbe (2010-05): redesign
- Timo Jolivet (2010-08, 2010-09, 2011): redesign

REFERENCES:

EXAMPLES:

We start by drawing a simple three-face patch:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: x = [Face((0,0,0),1), Face((0,0,0),2), Face((0,0,0),3)]
sage: P = Patch(x)
sage: P
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*]
sage: P.plot()                    #not tested
```

We apply a substitution to this patch, and draw the result:

```
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: E(P)
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*, [(0, 1, -1), 2]*, [(1, 0, -1), 1]*]
sage: E(P).plot()                 #not tested
```

Note:

- The type of a face is given by an integer in `[1, ..., d]` where `d` is the length of the vector of the face.
- The alphabet of the domain and the codomain of σ must be equal, and they must be of the form `[1, ..., d]`, where `d` is a positive integer corresponding to the length of the vectors of the faces on which $E_1^*(\sigma)$ will act.

```
sage: P = Patch([Face((0,0,0),1), Face((0,0,0),2), Face((0,0,0),3)])
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: E(P)
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*, [(0, 1, -1), 2]*, [(1, 0, -1), 1]*]
```

The application of an `E1Star` substitution assigns to each new face the color of its preimage. The `repaint` method allows us to repaint the faces of a patch. A single color can also be assigned to every face, by specifying a list of a single color:

```
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P = E(P, 5)
sage: P.repaint(['green'])
sage: P.plot()                    #not tested
```

A list of colors allows us to color the faces sequentially:

```
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P = E(P)
sage: P.repaint(['red', 'yellow', 'green', 'blue', 'black'])
sage: P = E(P, 3)
sage: P.plot()                    #not tested
```

All the color schemes from `matplotlib.cm.datad.keys()` can be used:

```
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P.repaint(cmap='summer')
sage: P = E(P, 3)
sage: P.plot()                    #not tested
sage: P.repaint(cmap='hsv')
sage: P = E(P, 2)
sage: P.plot()                    #not tested
```

It is also possible to specify a dictionary to color the faces according to their type:

```
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P = E(P, 5)
sage: P.repaint({1:(0.7, 0.7, 0.7), 2:(0.5,0.5,0.5), 3:(0.3,0.3,0.3)})
sage: P.plot()                    #not tested
sage: P.repaint({1:'red', 2:'yellow', 3:'green'})
sage: P.plot()                    #not tested
```

Let us look at a nice big patch in 3D:

```
sage: sigma = WordMorphism({1:[1,2], 2:[3], 3:[1]})
sage: E = E1Star(sigma)
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P = P + P.translate([-1,1,0])
sage: P = E(P, 11)
sage: P.plot3d()                  #not tested
```

Plotting with TikZ pictures is possible:

```
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: s = P.plot_tikz()
sage: print s                     #not tested
\begin{tikzpicture}
[x={(-0.216506cm,-0.125000cm)}, y={(0.216506cm,-0.125000cm)}, z={(0.000000cm,0.250000cm)}]
\definecolor{facecolor}{rgb}{0.000,1.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 0, 1) -- (1, 0, 1) -- (1, 0, 0) -- cycle;
\definecolor{facecolor}{rgb}{1.000,0.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 1, 0) -- (0, 1, 1) -- (0, 0, 1) -- cycle;
\definecolor{facecolor}{rgb}{0.000,0.000,1.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (1, 0, 0) -- (1, 1, 0) -- (0, 1, 0) -- cycle;
\end{tikzpicture}
```

Plotting patches made of unit segments instead of unit faces:

```
sage: P = Patch([Face([0,0], 1), Face([0,0], 2)])
sage: E = E1Star(WordMorphism({1:[1,2],2:[1]}))
sage: F = E1Star(WordMorphism({1:[1,1,2],2:[2,1]}))
sage: E(P,5).plot()
Graphics object consisting of 21 graphics primitives
sage: F(P,3).plot()
Graphics object consisting of 34 graphics primitives
```

Everything works in any dimension (except for the plotting features which only work in dimension two or three):

```
sage: P = Patch([Face((0,0,0,0),1), Face((0,0,0,0),4)])
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1,4], 4:[1]})
sage: E = E1Star(sigma)
sage: E(P)
Patch: [[(0, 0, 0, 0), 3]*, [(0, 0, 0, 0), 4]*, [(0, 0, 1, -1), 3]*, [(0, 1, 0, -1), 2]*, [(1, 0, 0,

sage: sigma = WordMorphism({1:[1,2],2:[1,3],3:[1,4],4:[1,5],5:[1,6],6:[1,7],7:[1,8],8:[1,9],9:[1,10],
sage: E = E1Star(sigma)
sage: E
E_1^*(1->12, 10->1,11, 11->1,12, 12->1, 2->13, 3->14, 4->15, 5->16, 6->17, 7->18, 8->19, 9->1,10)
sage: P = Patch([Face((0,0,0,0,0,0,0,0,0,0,0),t) for t in [1,2,3]])
sage: for x in sorted(list(E(P)), key=lambda x : (x.vector(),x.type())): print x
```

```
[(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 1]*
[(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 2]*
[(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 12]*
[(0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1), 11]*
[(0, 0, 0, 0, 0, 0, 0, 0, 1, 0, -1), 10]*
[(0, 0, 0, 0, 0, 0, 0, 1, 0, 0, -1), 9]*
[(0, 0, 0, 0, 0, 0, 1, 0, 0, 0, -1), 8]*
[(0, 0, 0, 0, 0, 1, 0, 0, 0, 0, -1), 7]*
[(0, 0, 0, 0, 1, 0, 0, 0, 0, 0, -1), 6]*
[(0, 0, 0, 1, 0, 0, 0, 0, 0, 0, -1), 5]*
[(0, 0, 1, 0, 0, 0, 0, 0, 0, 0, -1), 4]*
[(0, 1, 0, 0, 0, 0, 0, 0, 0, 0, -1), 3]*
[(1, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1), 2]*
```

class sage.combinat.e_one_star.**E1Star**(*sigma*, *method='suffix'*)

Bases: sage.structure.sage_object.SageObject

A class to model the $E_1^*(\sigma)$ map associated with a unimodular substitution σ.

INPUT:

- sigma - unimodular WordMorphism, i.e. such that its incidence matrix has determinant ± 1.
- method - 'prefix' or 'suffix' (optional, default: 'suffix') Enables to use an alternative definition $E_1^*(\sigma)$ substitutions, where the abelianized of the prefix' is used instead of the suffix.

Note: The alphabet of the domain and the codomain of σ must be equal, and they must be of the form [1, ..., d], where d is a positive integer corresponding to the length of the vectors of the faces on which $E_1^*(\sigma)$ will act.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: E(P)
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*, [(0, 1, -1), 2]*, [(1, 0, -1), 1]*]

sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma, method='prefix')
sage: E(P)
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*, [(0, 0, 1), 1]*, [(0, 0, 1), 2]*]

sage: x = [Face((0,0,0,0),1), Face((0,0,0,0),4)]
sage: P = Patch(x)
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1,4], 4:[1]})
sage: E = E1Star(sigma)
sage: E(P)
Patch: [[(0, 0, 0, 0), 3]*, [(0, 0, 0, 0), 4]*, [(0, 0, 1, -1), 3]*, [(0, 1, 0, -1), 2]*, [(1, 0
```

inverse_matrix()

Returns the inverse of the matrix associated with self.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
```

```
sage: E = E1Star(sigma)
sage: E.inverse_matrix()
[ 0  1  0]
[ 0  0  1]
[ 1 -1 -1]
```

matrix()

Returns the matrix associated with self.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: E.matrix()
[1 1 1]
[1 0 0]
[0 1 0]
```

sigma()

Returns the `WordMorphism` associated with self.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: E.sigma()
WordMorphism: 1->12, 2->13, 3->1
```

class sage.combinat.e_one_star.**Face**(*v*, *t*, *color=None*)

Bases: sage.structure.sage_object.SageObject

A class to model a unit face of arbitrary dimension.

A unit face in dimension d is represented by a d-dimensional vector `v` and a type `t` in $\{1, \ldots, d\}$. The type of the face corresponds to the canonical unit vector to which the face is orthogonal. The optional `color` argument is used in plotting functions.

INPUT:

- `v` - tuple of integers
- `t` - integer in `[1, ..., len(v)]`, type of the face. The face of type i is orthogonal to the canonical vector e_i.
- `color` - color (optional, default: `None`) color of the face, used for plotting only. If `None`, its value is guessed from the face type.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face
sage: f = Face((0,2,0), 3)
sage: f.vector()
(0, 2, 0)
sage: f.type()
3

sage: f = Face((0,2,0), 3, color=(0.5, 0.5, 0.5))
sage: f.color()
RGB color (0.5, 0.5, 0.5)
```

color(*color=None*)
Returns or change the color of the face.

INPUT:

- `color` - string, rgb tuple, color (optional, default: `None`) the new color to assign to the face. If `None`, it returns the color of the face.

OUTPUT:

color

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face
sage: f = Face((0,2,0), 3)
sage: f.color()
RGB color (0.0, 0.0, 1.0)
sage: f.color('red')
sage: f.color()
RGB color (1.0, 0.0, 0.0)
```

type()
Returns the type of the face.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face
sage: f = Face((0,2,0), 3)
sage: f.type()
3

sage: f = Face((0,2,0), 3)
sage: f.type()
3
```

vector()
Return the vector of the face.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face
sage: f = Face((0,2,0), 3)
sage: f.vector()
(0, 2, 0)
```

class sage.combinat.e_one_star.**Patch**(*faces*, *face_contour=None*)
Bases: sage.structure.sage_object.SageObject

A class to model a collection of faces. A patch is represented by an immutable set of Faces.

Note: The dimension of a patch is the length of the vectors of the faces in the patch, which is assumed to be the same for every face in the patch.

Note: Since version 4.7.1, Patches are immutable, except for the colors of the faces, which are not taken into account for equality tests and hash functions.

INPUT:

- `faces` - finite iterable of faces

- face_contour - dict (optional, default:None) maps the face type to vectors describing the contour of unit faces. If None, defaults contour are assumed for faces of type 1, 2, 3 or 1, 2, 3. Used in plotting methods only.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*]

sage: face_contour = {}
sage: face_contour[1] = map(vector, [(0,0,0),(0,1,0),(0,1,1),(0,0,1)])
sage: face_contour[2] = map(vector, [(0,0,0),(0,0,1),(1,0,1),(1,0,0)])
sage: face_contour[3] = map(vector, [(0,0,0),(1,0,0),(1,1,0),(0,1,0)])
sage: Patch([Face((0,0,0),t) for t in [1,2,3]], face_contour=face_contour)
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(0, 0, 0), 3]*]
```

difference(*other*)

Returns the difference of self and other.

INPUT:

- other - a finite iterable of faces or a single face

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P.difference(Face([0,0,0],2))
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 3]*]
sage: P.difference(P)
Patch: []
```

dimension()

Returns the dimension of the vectors of the faces of self

It returns None if self is the empty patch.

The dimension of a patch is the length of the vectors of the faces in the patch, which is assumed to be the same for every face in the patch.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P.dimension()
3
```

TESTS:

```
sage: from sage.combinat.e_one_star import Patch
sage: p = Patch([])
sage: p.dimension() is None
True
```

It works when the patch is created from an iterator:

```
sage: p = Patch(Face((0,0,0),t) for t in [1,2,3])
sage: p.dimension()
3
```

faces_of_color(*color*)

Returns a list of the faces that have the given color.

INPUT:

- `color` - color

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),1, 'red'), Face((1,2,0),3, 'blue'), Face((1,2,0),1, 'red')])
sage: P.faces_of_color('red')
[[(0, 0, 0), 1]*, [(1, 2, 0), 1]*]
```

faces_of_type(*t*)

Returns a list of the faces that have type `t`.

INPUT:

- `t` - integer or any other type

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),1), Face((1,2,0),3), Face((1,2,0),1)])
sage: P.faces_of_type(1)
[[(0, 0, 0), 1]*, [(1, 2, 0), 1]*]
```

faces_of_vector(*v*)

Returns a list of the faces whose vector is `v`.

INPUT:

- `v` - a vector

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),1), Face((1,2,0),3), Face((1,2,0),1)])
sage: P.faces_of_vector([1,2,0])
[[(1, 2, 0), 3]*, [(1, 2, 0), 1]*]
```

occurrences_of(*other*)

Returns all positions at which other appears in self, that is, all vectors v such that `set(other.translate(v)) <= set(self)`.

INPUT:

- `other` - a Patch

OUTPUT:

a list of vectors

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch, E1Star
sage: P = Patch([Face([0,0,0], 1), Face([0,0,0], 2), Face([0,0,0], 3)])
sage: Q = Patch([Face([0,0,0], 1), Face([0,0,0], 2)])
sage: P.occurrences_of(Q)
[(0, 0, 0)]
sage: Q = Q.translate([1,2,3])
sage: P.occurrences_of(Q)
[(-1, -2, -3)]
```

```
sage: E = E1Star(WordMorphism({1:[1,2], 2:[1,3], 3:[1]}))
sage: P = Patch([Face([0,0,0], 1), Face([0,0,0], 2), Face([0,0,0], 3)])
sage: P = E(P,4)
sage: Q = Patch([Face([0,0,0], 1), Face([0,0,0], 2)])
sage: L = P.occurrences_of(Q)
sage: sorted(L)
[(0, 0, 0), (0, 0, 1), (0, 1, -1), (1, 0, -1), (1, 1, -3), (1, 1, -2)]
```

plot (*projmat=None*, *opacity=0.75*)

Returns a 2D graphic object depicting the patch.

INPUT:

- `projmat` - matrix (optional, default: `None`) the projection matrix. Its number of lines must be two. Its number of columns must equal the dimension of the ambient space of the faces. If `None`, the isometric projection is used by default.
- `opacity` - float between `0` and `1` (optional, default: `0.75`) opacity of the the face

Warning: Plotting is implemented only for patches in two or three dimensions.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P.plot()
Graphics object consisting of 3 graphics primitives

sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P = E(P, 5)
sage: P.plot()
Graphics object consisting of 57 graphics primitives
```

Plot with a different projection matrix:

```
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: M = matrix(2, 3, [1,0,-1,0.3,1,-3])
sage: P = E(P, 3)
sage: P.plot(projmat=M)
Graphics object consisting of 17 graphics primitives
```

Plot patches made of unit segments:

```
sage: P = Patch([Face([0,0], 1), Face([0,0], 2)])
sage: E = E1Star(WordMorphism({1:[1,2],2:[1]}))
sage: F = E1Star(WordMorphism({1:[1,1,2],2:[2,1]}))
sage: E(P,5).plot()
Graphics object consisting of 21 graphics primitives
sage: F(P,3).plot()
Graphics object consisting of 34 graphics primitives
```

plot3d()

Returns a 3D graphics object depicting the patch.

Warning: 3D plotting is implemented only for patches in three dimensions.

EXAMPLES:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P.plot3d()                #not tested

sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P = E(P, 5)
sage: P.repaint()
sage: P.plot3d()                #not tested
```

plot_tikz (*projmat=None*, *print_tikz_env=True*, *edgecolor='black'*, *scale=0.25*, *drawzero=False*, *extra_code_before=''*, *extra_code_after=''*)

Returns a string containing some TikZ code to be included into a LaTeX document, depicting the patch.

Warning: Tikz Plotting is implemented only for patches in three dimensions.

INPUT:

- `projmat` - matrix (optional, default: `None`) the projection matrix. Its number of lines must be two. Its number of columns must equal the dimension of the ambient space of the faces. If `None`, the isometric projection is used by default.
- `print_tikz_env` - bool (optional, default: `True`) if `True`, the tikzpicture environment are printed
- `edgecolor` - string (optional, default: `'black'`) either `'black'` or `'facecolor'` (color of unit face edges)
- `scale` - real number (optional, default: `0.25`) scaling constant for the whole figure
- `drawzero` - bool (optional, default: `False`) if `True`, mark the origin by a black dot
- `extra_code_before` - string (optional, default: `''`) extra code to include in the tikz picture
- `extra_code_after` - string (optional, default: `''`) extra code to include in the tikz picture

EXAMPLES:

```
sage: from sage.combinat.e_one_star import E1Star, Face, Patch
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: s = P.plot_tikz()
sage: len(s)
602
sage: print s       #not tested
\begin{tikzpicture}
[x={(-0.216506cm,-0.125000cm)}, y={(0.216506cm,-0.125000cm)}, z={(0.000000cm,0.250000cm)}]
\definecolor{facecolor}{rgb}{0.000,1.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 0, 1) -- (1, 0, 1) -- (1, 0, 0) -- cycle;
\definecolor{facecolor}{rgb}{1.000,0.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 1, 0) -- (0, 1, 1) -- (0, 0, 1) -- cycle;
\definecolor{facecolor}{rgb}{0.000,0.000,1.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (1, 0, 0) -- (1, 1, 0) -- (0, 1, 0) -- cycle;
\end{tikzpicture}
```

```
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P = E(P, 4)
sage: from sage.misc.latex import latex              #not tested
sage: latex.add_to_preamble('\\usepackage{tikz}')    #not tested
sage: view(P, tightpage=true)                        #not tested
```

Plot using shades of gray (useful for article figures):

```
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P.repaint([(0.9, 0.9, 0.9), (0.65,0.65,0.65), (0.4,0.4,0.4)])
sage: P = E(P, 4)
sage: s = P.plot_tikz()
```

Plotting with various options:

```
sage: sigma = WordMorphism({1:[1,2], 2:[1,3], 3:[1]})
sage: E = E1Star(sigma)
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: M = matrix(2, 3, map(float, [1,0,-0.7071,0,1,-0.7071]))
sage: P = E(P, 3)
sage: s = P.plot_tikz(projmat=M, edgecolor='facecolor', scale=0.6, drawzero=True)
```

Adding X, Y, Z axes using the extra code feature:

```
sage: length = 1.5
sage: space = 0.3
sage: axes = ''
sage: axes += "\\draw[->, thick, black] (0,0,0) -- (%s, 0, 0);\n" % length
sage: axes += "\\draw[->, thick, black] (0,0,0) -- (0, %s, 0);\n" % length
sage: axes += "\\node at (%s,0,0) {$x$};\n" % (length + space)
sage: axes += "\\node at (0,%s,0) {$y$};\n" % (length + space)
sage: axes += "\\node at (0,0,%s) {$z$};\n" % (length + space)
sage: axes += "\\draw[->, thick, black] (0,0,0) -- (0, 0, %s);\n" % length
sage: cube = Patch([Face((0,0,0),1), Face((0,0,0),2), Face((0,0,0),3)])
sage: options = dict(scale=0.5,drawzero=True,extra_code_before=axes)
sage: s = cube.plot_tikz(**options)
sage: len(s)
986
sage: print s   #not tested
\begin{tikzpicture}
[x={(-0.433013cm,-0.250000cm)}, y={(0.433013cm,-0.250000cm)}, z={(0.000000cm,0.500000cm)}]
\draw[->, thick, black] (0,0,0) -- (1.50000000000000, 0, 0);
\draw[->, thick, black] (0,0,0) -- (0, 1.50000000000000, 0);
\node at (1.80000000000000,0,0) {$x$};
\node at (0,1.80000000000000,0) {$y$};
\node at (0,0,1.80000000000000) {$z$};
\draw[->, thick, black] (0,0,0) -- (0, 0, 1.50000000000000);
\definecolor{facecolor}{rgb}{0.000,1.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 0, 1) -- (1, 0, 1) -- (1, 0, 0) -- cycle;
\definecolor{facecolor}{rgb}{1.000,0.000,0.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
(0, 0, 0) -- (0, 1, 0) -- (0, 1, 1) -- (0, 0, 1) -- cycle;
\definecolor{facecolor}{rgb}{0.000,0.000,1.000}
\fill[fill=facecolor, draw=black, shift={(0,0,0)}]
```

```
(0, 0, 0) -- (1, 0, 0) -- (1, 1, 0) -- (0, 1, 0) -- cycle;
\node[circle,fill=black,draw=black,minimum size=1.5mm,inner sep=0pt] at (0,0,0) {};
\end{tikzpicture}
```

repaint (*cmap='Set1'*)

Repaints all the faces of self from the given color map.

This only changes the colors of the faces of self.

INPUT:

- `cmap` - color map (default: `'Set1'`). It can be one of the following :
  - string - A coloring map. For available coloring map names type: `sorted(colormaps)`
  - list - a list of colors to assign cyclically to the faces. A list of a single color colors all the faces with the same color.
  - dict - a dict of face types mapped to colors, to color the faces according to their type.
  - `{}`, the empty dict - shorcut for `{1:'red', 2:'green', 3:'blue'}`.

EXAMPLES:

Using a color map:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: color = (0, 0, 0)
sage: P = Patch([Face((0,0,0),t,color) for t in [1,2,3]])
sage: for f in P: f.color()
RGB color (0.0, 0.0, 0.0)
RGB color (0.0, 0.0, 0.0)
RGB color (0.0, 0.0, 0.0)
sage: P.repaint()
sage: next(iter(P)).color()     #random
RGB color (0.498..., 0.432..., 0.522...)
```

Using a list of colors:

```
sage: P = Patch([Face((0,0,0),t,color) for t in [1,2,3]])
sage: P.repaint([(0.9, 0.9, 0.9), (0.65,0.65,0.65), (0.4,0.4,0.4)])
sage: for f in P: f.color()
RGB color (0.9, 0.9, 0.9)
RGB color (0.65, 0.65, 0.65)
RGB color (0.4, 0.4, 0.4)
```

Using a dictionary to color faces according to their type:

```
sage: P = Patch([Face((0,0,0),t) for t in [1,2,3]])
sage: P.repaint({1:'black', 2:'yellow', 3:'green'})
sage: P.plot()                    #not tested
sage: P.repaint({})
sage: P.plot()                    #not tested
```

translate (*v*)

Returns a translated copy of self by vector `v`.

INPUT:

- `v` - vector or tuple

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),1), Face((1,2,0),3), Face((1,2,0),1)])
sage: P.translate([-1,-2,0])
Patch: [[(-1, -2, 0), 1]*, [(0, 0, 0), 1]*, [(0, 0, 0), 3]*]
```

union(*other*)

Returns a Patch consisting of the union of self and other.

INPUT:

- `other` - a Patch or a Face or a finite iterable of faces

EXAMPLES:

```
sage: from sage.combinat.e_one_star import Face, Patch
sage: P = Patch([Face((0,0,0),1), Face((0,0,0),2)])
sage: P.union(Face((1,2,3), 3))
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(1, 2, 3), 3]*]
sage: P.union([Face((1,2,3), 3), Face((2,3,3), 2)])
Patch: [[(0, 0, 0), 1]*, [(0, 0, 0), 2]*, [(1, 2, 3), 3]*, [(2, 3, 3), 2]*]
```

5.1.91 Enumerated sets and combinatorial objects

Todo

Proofread / point to the main classes rather than the modules

Categories

- `EnumeratedSets`, `FiniteEnumeratedSets`

Basic enumerated sets

- `Subsets`, `Combinations`
- `Arrangements`, `Tuples`
- `FiniteEnumeratedSet`
- `DisjointUnionEnumeratedSets`

Integer lists

- *Integer partitions* (see also: *Enumerated sets of partitions, tableaux, ...*)
- *Integer compositions*
- `SignedCompositions`
- `IntegerListsLex`
- `IntegerVectors`
- `WeightedIntegerVectors()`

- IntegerVectorsModPermutationGroup
- *Parking Functions*
- *Non-Decreasing Parking Functions*
- *Sidon sets and their generalizations, Sidon -sets*

Words

- Words
- *Subwords*
- *Necklaces*
- *Lyndon words*
- *Dyck Words*
- *De Bruijn sequences*
- *Shuffle product of iterables*

Permutations, ...

- *Permutations*
- *Permutations (Cython file)*
- *Affine Permutations*
- Arrangements
- *Derangements*
- *Baxter permutations*

See also:

- SymmetricGroup, PermutationGroup(), *Catalog of permutation groups*
- FiniteSetMaps
- *Integer vectors modulo the action of a permutation group*
- *Robinson-Schensted-Knuth correspondence*

Partitions, tableaux, ...

See: *Enumerated sets of partitions, tableaux, ...*

Integer matrices, ...

- *Counting, generating, and manipulating non-negative integer matrices*
- *Hadamard matrices*
- *Latin Squares*
- *Alternating Sign Matrices*
- *Six Vertex Model*

- *Similarity class types of matrices with entries in a finite field*
- *Restricted growth arrays*
- *Vector Partitions*

See also:

- `MatrixSpace`
- *Library of Interesting Groups*

Subsets and set partitions

- `Subsets`, `Combinations`
- `PairwiseCompatibleSubsets`
- *Subsets satisfying a hereditary property*
- *Ordered Set Partitions*
- *Set Partitions*

Trees

- *Abstract Recursive Trees*
- *Ordered Rooted Trees*
- *Binary Trees*
- *Rooted (Unordered) Trees*

Enumerated sets related to graphs

- *Degree sequences*
- *Paths in Directed Acyclic Graphs*
- *Perfect matchings*

Backtracking solvers and generic enumerated sets

Todo

Do we want a separate section, possibly more proeminent, for backtracking solvers?

- `RecursivelyEnumeratedSet()`
- `GenericBacktracker`
- *Tiling Solver*
- *Exact Cover Problem via Dancing Links*
- *Dancing links C++ wrapper*
- *Combinatorial Species*
- `IntegerListsLex`

- IntegerVectorsModPermutationGroup

Low level enumerated sets

- *Low-level multichoose*
- *Gray codes*

Misc enumerated sets

- GelfandTsetlinPattern, GelfandTsetlinPatterns
- KnutsonTaoPuzzleSolver
- LatticePolytope()

5.1.92 Tools for enumeration modulo the action of a permutation group

sage.combinat.enumeration_mod_permgroup.**all_children**(*v*, *max_part*)

Returns all the children of an integer vector (ClonableIntArray) v in the tree of enumeration by lexicographic order. The children of an integer vector v whose entries have the sum n are all integer vectors of sum $n+1$ which follow v in the lexicographic order.

That means this function adds 1 on the last non zero entries and the following ones. For an integer vector v such that

$$v = [\ldots, a, 0, 0] \text{ with } a \neq 0,$$

then, the list of children is

$$[[\ldots, a+1, 0, 0], [\ldots, a, 1, 0], [\ldots, a, 0, 1]].$$

EXAMPLES:

```
sage: from sage.combinat.enumeration_mod_permgroup import all_children
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: v = IncreasingIntArrays()([1,2,3,4])
sage: all_children(v, -1)
[[1, 2, 3, 5]]
```

sage.combinat.enumeration_mod_permgroup.**canonical_children**(*sgs*, *v*, *max_part*)

Returns the canonical children of the integer vector v. This function computes all children of the integer vector v via the function all_children() and returns from this list only these which are canonicals identified via the function is_canonical().

EXAMPLES:

```
sage: from sage.combinat.enumeration_mod_permgroup import canonical_children
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: sgs = G.strong_generating_system()
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: IA = IncreasingIntArrays()
sage: canonical_children(sgs, IA([1,2,3,5]), -1)
[]
```

sage.combinat.enumeration_mod_permgroup.**canonical_representative_of_orbit_of**(*sgs*, *v*)

Returns the maximal vector for the lexicographic order living in the orbit of v under the action of the permutation group whose strong generating system is `sgs`. The maximal vector is also called "canonical". Hence, this method returns the canonical vector inside the orbit of v. If v is already canonical, the method returns v.

Let G to be the permutation group which admits `sgs` as a strong generating system. An integer vector v is said to be canonical under the action of G if and only if:

$$v = \max_{\text{lex order}} \{g \cdot v | g \in G\}$$

EXAMPLES:

```
sage: from sage.combinat.enumeration_mod_permgroup import canonical_representative_of_orbit_of
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: sgs = G.strong_generating_system()
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: IA = IncreasingIntArrays()
sage: canonical_representative_of_orbit_of(sgs, IA([1,2,3,5]))
[5, 1, 2, 3]
```

sage.combinat.enumeration_mod_permgroup.**is_canonical**(*sgs*, *v*)

Returns `True` if the integer vector v is maximal with respect to the lexicographic order in its orbit under the action of the permutation group whose strong generating system is `sgs`. Such vectors are said to be canonical.

Let G to be the permutation group which admit `sgs` as a strong generating system. An integer vector v is said to be canonical under the action of G if and only if:

$$v = \max_{\text{lex order}} \{g \cdot v | g \in G\}$$

EXAMPLES:

```
sage: from sage.combinat.enumeration_mod_permgroup import is_canonical
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: sgs = G.strong_generating_system()
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: IA = IncreasingIntArrays()
sage: is_canonical(sgs, IA([1,2,3,6]))
False
```

sage.combinat.enumeration_mod_permgroup.**lex_cmp**(*v1*, *v2*)

Lexicographic comparison of `ClonableIntArray`.

INPUT:

Two instances v_1, v_2 of `ClonableIntArray`

OUPUT:

`-1,0,1`, depending on whether v_1 is lexicographically smaller, equal, or greater than v_2.

EXAMPLES:

```
sage: I =  IntegerVectorsModPermutationGroup(SymmetricGroup(5),5)
sage: I =  IntegerVectorsModPermutationGroup(SymmetricGroup(5))
sage: J =  IntegerVectorsModPermutationGroup(SymmetricGroup(6))
sage: v1 = I([2,3,1,2,3], check=False)
sage: v2 = I([2,3,2,3,2], check=False)
sage: v3 = J([2,3,1,2,3,1], check=False)
sage: from sage.combinat.enumeration_mod_permgroup import lex_cmp
sage: lex_cmp(v1, v1)
0
```

```
sage: lex_cmp(v1, v2)
-1
sage: lex_cmp(v2, v1)
1
sage: lex_cmp(v1, v3)
-1
sage: lex_cmp(v3, v1)
1
```

sage.combinat.enumeration_mod_permgroup.**lex_cmp_partial**(*v1*, *v2*, *step*)

Partial comparison of the two lists according the lexicographic order. It compares the `step`-th first entries.

EXAMPLES:

```
sage: from sage.combinat.enumeration_mod_permgroup import lex_cmp_partial
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: IA = IncreasingIntArrays()
sage: lex_cmp_partial(IA([0,1,2,3]),IA([0,1,2,4]),3)
0
sage: lex_cmp_partial(IA([0,1,2,3]),IA([0,1,2,4]),4)
-1
```

sage.combinat.enumeration_mod_permgroup.**orbit**(*sgs*, *v*)

Returns the orbit of the integer vector `v` under the action of the permutation group whose strong generating system is `sgs`.

NOTE:

The returned orbit is a set. In the doctests, we convert it into a sorted list.

EXAMPLES:

```
sage: from sage.combinat.enumeration_mod_permgroup import orbit
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: sgs = G.strong_generating_system()
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: IA = IncreasingIntArrays()
sage: sorted(orbit(sgs, IA([1,2,3,4])))
[[1, 2, 3, 4], [2, 3, 4, 1], [3, 4, 1, 2], [4, 1, 2, 3]]
```

5.1.93 Compute Bell and Uppuluri-Carpenter numbers

AUTHORS:

- Nick Alexander

sage.combinat.expnums.**expnums**(*n*, *aa*)

Compute the first n exponential numbers around aa, starting with the zero-th.

INPUT:

- `n` - C machine int
- `aa` - C machine int

OUTPUT: A list of length n.

ALGORITHM: We use the same integer addition algorithm as GAP. This is an extension of Bell's triangle to the general case of exponential numbers. The recursion performs $O(n^2)$ additions, but the running time is dominated by the cost of the last integer addition, because the growth of the integer results of partial computations is exponential in n. The algorithm stores $O(n)$ integers, but each is exponential in n.

EXAMPLES:

```
sage: expnums(10, 1)
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147]

sage: expnums(10, -1)
[1, -1, 0, 1, 1, -2, -9, -9, 50, 267]

sage: expnums(1, 1)
[1]
sage: expnums(0, 1)
[]
sage: expnums(-1, 0)
[]
```

AUTHORS:

- Nick Alexander

sage.combinat.expnums.**expnums2**(*n*, *aa*)

A vanilla python (but compiled via Cython) implementation of expnums.

We Compute the first n exponential numbers around aa, starting with the zero-th.

EXAMPLES:

```
sage: from sage.combinat.expnums import expnums2
sage: expnums2(10, 1)
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147]
```

5.1.94 Families

This is a backward compatibility stub. Use sage.sets.family instead.

5.1.95 Finite combinatorial classes

class sage.combinat.finite_class.**FiniteCombinatorialClass**(*l*)

Bases: sage.combinat.combinat.CombinatorialClass

INPUT:

- l a list or iterable

Returns l, wrapped as a combinatorial class

EXAMPLES:

```
sage: F = FiniteCombinatorialClass([1,2,3])
sage: F.list()
[1, 2, 3]
sage: F.cardinality()
3
sage: F.random_element()
1
sage: F.first()
1
sage: F.last()
3
```

cardinality()
EXAMPLES:

```
sage: F = FiniteCombinatorialClass([1,2,3])
sage: F.cardinality()
3
```

keys()
EXAMPLES:

```
sage: F = FiniteCombinatorialClass([1,2,3])
sage: F.keys()
[0, 1, 2]
```

list()
TESTS:

```
sage: F = FiniteCombinatorialClass([1,2,3])
sage: F.list()
[1, 2, 3]
```

sage.combinat.finite_class.**FiniteCombinatorialClass_l**
alias of FiniteCombinatorialClass

5.1.96 Finite State Machines, Automata, Transducers

This module adds support for finite state machines, automata and transducers.

For creating automata and transducers you can use classes

- Automaton and Transducer (or the more general class FiniteStateMachine)

or the generators

- automata and transducers

which contain *preconstructed and commonly used automata and transducers*. See also the *examples* below.

Contents

FiniteStateMachine and derived classes Transducer and Automaton

Accessing parts of a finite state machine

state()	Get a state by its label
states()	List of states
iter_states()	Iterator over the states
initial_states()	List of initial states
iter_initial_states()	Iterator over initial states
final_states()	List of final states
iter_final_states()	Iterator over final states
transition()	Get a transition by its states and labels
transitions()	List of transitions
iter_transitions()	Iterator over the transitions
predecessors()	List of predecessors of a state
induced_sub_finite_state_machine()	Induced sub-machine
accessible_components()	Accessible components
coaccessible_components()	Coaccessible components
final_components()	Final components (connected componer again)

(Modified) Copies

empty_copy()	Returns an empty deep copy
deepcopy()	Returns a deep copy
relabeled()	Returns a relabeled deep copy
Automaton.with_output()	Extends an automaton to a transducer

Manipulation

add_state()	Add a state
add_states()	Add states
delete_state()	Delete a state
add_transition()	Add a transition
add_transitions_from_functi	Add transitions
input_alphabet	Input alphabet
output_alphabet	Output alphabet
on_duplicate_transition	Hook for handling duplicate transitions
add_from_transition_functio	Add transitions by a transition function
delete_transition()	Delete a transition
remove_epsilon_transitions()	Remove epsilon transitions (not implemented)
split_transitions()	Split transitions with input words of length > 1
determine_alphabets()	Determine input and output alphabets
determine_input_alphabet()	Determine input alphabet
determine_output_alphabet()	Determine output alphabet
construct_final_word_out()	Construct final output by implicitly reading trailing letters; cf. with_final_word_out()

Properties

has_state()	Checks for a state
has_initial_state()	Checks for an initial state
has_initial_states()	Checks for initial states
has_final_state()	Checks for an final state
has_final_states()	Checks for final states
has_transition()	Checks for a transition
is_deterministic()	Checks for a deterministic machine
is_complete()	Checks for a complete machine
is_connected()	Checks for a connected machine
Automaton.is_equivalent()	Checks for equivalent automata
is_Markov_chain()	Checks for a Markov chain
is_monochromatic()	Checks whether the colors of all states are equal
number_of_words()	Determine the number of successful paths
asymptotic_moments()	Main terms of expectation and variance of sums of labels
moments_waiting_time()	Moments of the waiting time for first true output
epsilon_successors()	Epsilon successors of a state
Automaton.shannon_parry_markov_chain()	Compute Markov chain with Parry measure

Operations

disjoint_union()	Disjoint union
concatenation()	Concatenation
kleene_star()	Kleene star
Automaton.complement()	Complement of an automaton
Automaton.intersection()	Intersection of automata
Transducer.intersection()	Intersection of transducers
Transducer.cartesian_product()	Cartesian product of a transducer with another finite state machine
product_FiniteStateMachine()	Product of finite state machines
composition()	Composition (output of other is input of self)
__call__()	Composition with other finite state machine
input_projection()	Input projection (output is deleted)
output_projection()	Output projection (old output is new input)
projection()	Input or output projection
transposition()	Transposition (all transitions are reversed)
with_final_word_out()	Machine with final output constructed by implicitly reading trailing letters, cf. construct_final_word_out() for inplace version
Automaton.determinisation()	Determinisation of an automaton
completion()	Completion of a finite state machine
process()	Process input
__call__()	Process input with shortened output
Automaton.process()	Process input of an automaton (output differs from general case)
Transducer.process()	Process input of a transducer (output differs from general case)
iter_process()	Return process iterator
language()	Return all possible output words
Automaton.language()	Return all possible accepted words

Simplification

prepone_output()	Prepone output where possible
equivalence_classes()	List of equivalent states
quotient()	Quotient with respect to equivalence classes
merged_transitions()	Merge transitions while adding input
markov_chain_simplification()	Simplification of a Markov chain
Automaton.minimization()	Minimization of an automaton
Transducer.simplification()	Simplification of a transducer

Conversion	`adjacency_matrix()`	(Weighted) adjacency `matrix`
	`graph()`	Underlying `DiGraph`
	`plot()`	Plot

LaTeX output	`latex_options()`	Set options
	`set_coordinates()`	Set coordinates of the states
	`default_format_transition_label()`	Default formatting of words in transition labels
	`format_letter_negative()`	Format negative numbers as overlined number
	`format_transition_label_reversed()`	Format words in transition labels in reversed order

See also:

LaTeX output

`FSMState`

`final_word_out`	Final output of a state
`is_final`	Describes whether a state is final or not
`is_initial`	Describes whether a state is initial or not
`initial_probability`	Probability of starting in this state as part of a Markov chain
`label()`	Label of a state
`relabeled()`	Returns a relabeled deep copy of a state
`fully_equal()`	Checks whether two states are fully equal (including all attributes)

`FSMTransition`

`from_state`	State in which transition starts
`to_state`	State in which transition ends
`word_in`	Input word of the transition
`word_out`	Output word of the transition
`deepcopy()`	Returns a deep copy of the transition

`FSMProcessIterator`

`next()`	Makes one step in processing the input tape
`preview_word()`	Reads a word from the input tape
`result()`	Returns the finished branches during process

Helper Functions

equal()	Checks whether all elements of iterator are equal
full_group_by()	Group iterable by values of some key
startswith()	Determine whether list starts with the given prefix
FSMLetterSymbol()	Returns a string associated to the input letter
FSMWordSymbol()	Returns a string associated to a word
is_FSMState()	Tests whether an object inherits from FSMState
is_FSMTransition()	Tests whether an object inherits from FSMTransition
is_FiniteStateMachine()	Tests whether an object inherits from FiniteStateMachine
duplicate_transition_ignore()	Default function for handling duplicate transitions
duplicate_transition_raise_error()	Raise error when inserting a duplicate transition
duplicate_transition_add_input()	Add input when inserting a duplicate transition

Examples

We start with a general FiniteStateMachine. Later there will be also an Automaton and a Transducer.

A simple finite state machine

We can easily create a finite state machine by

```
sage: fsm = FiniteStateMachine()
sage: fsm
Empty finite state machine
```

By default this is the empty finite state machine, so not very interesting. Let's create and add some states and transitions:

```
sage: day = fsm.add_state('day')
sage: night = fsm.add_state('night')
sage: sunrise = fsm.add_transition(night, day)
sage: sunset = fsm.add_transition(day, night)
```

Let us look at sunset more closely:

```
sage: sunset
Transition from 'day' to 'night': -|-
```

Note that could also have created and added the transitions directly by:

```
sage: fsm.add_transition('day', 'night')
Transition from 'day' to 'night': -|-
```

This would have had added the states automatically, since they are present in the transitions.

Anyhow, we got the following finite state machine:

```
sage: fsm
Finite state machine with 2 states
```

We can also obtain the underlying directed graph by

```
sage: fsm.graph()
Looped multi-digraph on 2 vertices
```

To visualize a finite state machine, we can use `latex()` and run the result through LaTeX, see the section on *LaTeX output* below.

Alternatively, we could have created the finite state machine above simply by

```
sage: FiniteStateMachine([('night', 'day'), ('day', 'night')])
Finite state machine with 2 states
```

See `FiniteStateMachine` for a lot of possibilities to create finite state machines.

A simple Automaton (recognizing NAFs)

We want to build an automaton which recognizes non-adjacent forms (NAFs), i.e., sequences which have no adjacent non-zeros. We use 0, 1, and -1 as digits:

```
sage: NAF = Automaton(
....:     {'A': [('A', 0), ('B', 1), ('B', -1)], 'B': [('A', 0)]})
sage: NAF.state('A').is_initial = True
sage: NAF.state('A').is_final = True
sage: NAF.state('B').is_final = True
sage: NAF
Automaton with 2 states
```

Of course, we could have specified the initial and final states directly in the definition of `NAF` by `initial_states=['A']` and `final_states=['A', 'B']`.

So let's test the automaton with some input:

```
sage: NAF([0])
True
sage: NAF([0, 1])
True
sage: NAF([1, -1])
False
sage: NAF([0, -1, 0, 1])
True
sage: NAF([0, -1, -1, -1, 0])
False
sage: NAF([-1, 0, 0, 1, 1])
False
```

Alternatively, we could call that by

```
sage: NAF.process([0, -1, 0, 1])
(True, 'B')
```

which gives additionally the state in which we arrived.

We can also let an automaton act on a *word*:

```
sage: W = Words([-1, 0, 1]); W
Finite and infinite words over {-1, 0, 1}
sage: w = W([1, 0, 1, 0, -1]); w
word: 1,0,1,0,-1
```

```
sage: NAF(w)
True
```

Recognizing NAFs via Automata Operations Alternatively, we can use automata operations to recognize NAFs; for simplicity, we only use the input alphabet `[0, 1]`. On the one hand, we can construct such an automaton by forbidding the word `11`:

```
sage: forbidden = automata.ContainsWord([1, 1], input_alphabet=[0, 1])
sage: NAF_negative = forbidden.complement()
sage: NAF_negative([1, 1, 0, 1])
False
sage: NAF_negative([1, 0, 1, 0, 1])
True
```

On the other hand, we can write this as a regular expression and translate that into automata operations:

```
sage: zero = automata.Word([0])
sage: one = automata.Word([1])
sage: epsilon = automata.EmptyWord(input_alphabet=[0, 1])
sage: NAF_positive = (zero + one*zero).kleene_star() * (epsilon + one)
```

We check that the two approaches are equivalent:

```
sage: NAF_negative.is_equivalent(NAF_positive)
True
```

See also:

`ContainsWord()`, `Word()`, `complement()`, `kleene_star()`, `EmptyWord()`, `is_equivalent()`.

LaTeX output

We can visualize a finite state machine by converting it to LaTeX by using the usual function `latex()`. Within LaTeX, TikZ is used for typesetting the graphics, see the Wikipedia article PGF/TikZ.

```
sage: print latex(NAF)
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state, accepting, initial] (v0) at (3.000000, 0.000000) {$\text{\texttt{A}}$};
\node[state, accepting] (v1) at (-3.000000, 0.000000) {$\text{\texttt{B}}$};
\path[->] (v0) edge[loop above] node {$0$} ();
\path[->] (v0.185.00) edge node[rotate=360.00, anchor=north] {$1, -1$} (v1.355.00);
\path[->] (v1.5.00) edge node[rotate=0.00, anchor=south] {$0$} (v0.175.00);
\end{tikzpicture}
```

We can turn this into a graphical representation.

```
sage: view(NAF) # not tested
```

To actually see this, use the live documentation in the Sage notebook and execute the cells in this and the previous section.

Several options can be set to customize the output, see `latex_options()` for details. In particular, we use `format_letter_negative()` to format -1 as $\overline{1}$.

```
sage: NAF.latex_options(
....:     coordinates={'A': (0, 0),
....:                  'B': (6, 0)},
....:     initial_where={'A': 'below'},
....:     format_letter=NAF.format_letter_negative,
....:     format_state_label=lambda x:
....:         r'\mathcal{%s}' % x.label()
....: )
sage: print latex(NAF)
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state, accepting, initial, initial where=below] (v0) at (0.000000, 0.000000) {$\mathcal{A}$};
\node[state, accepting] (v1) at (6.000000, 0.000000) {$\mathcal{B}$};
\path[->] (v0) edge[loop above] node {$0$} ();
\path[->] (v0.5.00) edge node[rotate=0.00, anchor=south] {$1, \overline{1}$} (v1.175.00);
\path[->] (v1.185.00) edge node[rotate=360.00, anchor=north] {$0$} (v0.355.00);
\end{tikzpicture}
sage: view(NAF) # not tested
```

To use the output of `latex()` in your own LaTeX file, you have to include

```
\usepackage{tikz}
\usetikzlibrary{automata}
```

into the preamble of your file.

A simple transducer (binary inverter)

Let's build a simple transducer, which rewrites a binary word by iverting each bit:

```
sage: inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:     initial_states=['A'], final_states=['A'])
```

We can look at the states and transitions:

```
sage: inverter.states()
['A']
sage: for t in inverter.transitions():
....:     print t
Transition from 'A' to 'A': 0|1
Transition from 'A' to 'A': 1|0
```

Now we apply a word to it and see what the transducer does:

```
sage: inverter([0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1])
[1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0]
```

`True` means, that we landed in a final state, that state is labeled `'A'`, and we also got an output.

Transducers and (in)finite Words

A transducer can also act on everything iterable, in particular, on Sage's *words*.

```
sage: W = Words([0, 1]); W
Finite and infinite words over {0, 1}
```

Let us take the inverter from the previous section and feed some finite word into it:

```
sage: w = W([1, 1, 0, 1]); w
word: 1101
sage: inverter(w)
word: 0010
```

We see that the output is again a word (this is a consequence of calling `process()` with `automatic_output_type`).

We can even input something infinite like an infinite word:

```
sage: tm = words.ThueMorseWord(); tm
word: 01101001100101101001011001101001100101 10...
sage: inverter(tm)
word: 10010110011010010110100110010110011010 01...
```

A transducer which performs division by 3 in binary

Now we build a transducer, which divides a binary number by 3. The labels of the states are the remainder of the division. The transition function is

```
sage: def f(state_from, read):
....:     if state_from + read <= 1:
....:         state_to = 2*state_from + read
....:         write = 0
....:     else:
....:         state_to = 2*state_from + read - 3
....:         write = 1
....:     return (state_to, write)
```

which assumes reading a binary number from left to right. We get the transducer with

```
sage: D = Transducer(f, initial_states=[0], final_states=[0],
....:                input_alphabet=[0, 1])
```

Let us try to divide 12 by 3:

```
sage: D([1, 1, 0, 0])
[0, 1, 0, 0]
```

Now we want to divide 13 by 3:

```
sage: D([1, 1, 0, 1])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
```

The raised `ValueError` means 13 is not divisible by 3.

Gray Code

The Gray code is a binary numeral system where two successive values differ in only one bit, cf. the Wikipedia article Gray_code. The Gray code of an integer n is obtained by a bitwise xor between the binary expansion of n and the binary expansion of $\lfloor n/2 \rfloor$; the latter corresponds to a shift by one position in binary.

The purpose of this example is to construct a transducer converting the standard binary expansion to the Gray code by translating this construction into operations with transducers.

For this construction, the least significant digit is at the left-most position. Note that it is easier to shift everything to the right first, i.e., multiply by 2 instead of building $\lfloor n/2 \rfloor$. Then, we take the input xor with the right shift of the input and forget the first letter.

We first construct a transducer shifting the binary expansion to the right. This requires storing the previously read digit in a state.

```
sage: def shift_right_transition(state, digit):
....:     if state == 'I':
....:         return (digit, None)
....:     else:
....:         return (digit, state)
sage: shift_right_transducer = Transducer(
....:     shift_right_transition,
....:     initial_states=['I'],
....:     input_alphabet=[0, 1],
....:     final_states=[0])
sage: shift_right_transducer.transitions()
[Transition from 'I' to 0: 0|-,
 Transition from 'I' to 1: 1|-,
 Transition from 0 to 0: 0|0,
 Transition from 0 to 1: 1|0,
 Transition from 1 to 0: 0|1,
 Transition from 1 to 1: 1|1]
sage: shift_right_transducer([0, 1, 1, 0])
[0, 1, 1]
sage: shift_right_transducer([1, 0, 0])
[1, 0]
```

The output of the shifts above look a bit weird (from a right-shift transducer, we would expect, for example, that `[1, 0, 0]` was mapped to `[0, 1, 0]`), since we write `None` instead of the zero at the left. Further, note that only 0 is listed as a final state as we have to enforce that a most significant zero is read as the last input letter in order to flush the last digit:

```
sage: shift_right_transducer([0, 1, 0, 1])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
```

Next, we construct the transducer performing the xor operation. We also have to take `None` into account as our `shift_right_transducer` waits one iteration until it starts writing output. This corresponds with our intention to forget the first letter.

```
sage: def xor_transition(state, digits):
....:     if digits[0] is None or digits[1] is None:
....:         return (0, None)
....:     else:
....:         return (0, digits[0].__xor__(digits[1]))
sage: from itertools import product
sage: xor_transducer = Transducer(
....:     xor_transition,
....:     initial_states=[0],
....:     final_states=[0],
....:     input_alphabet=list(product([None, 0, 1], [0, 1])))
sage: xor_transducer.transitions()
[Transition from 0 to 0: (None, 0)|-,
```

```
 Transition from 0 to 0: (None, 1)|-,
 Transition from 0 to 0: (0, 0)|0,
 Transition from 0 to 0: (0, 1)|1,
 Transition from 0 to 0: (1, 0)|1,
 Transition from 0 to 0: (1, 1)|0]
sage: xor_transducer([(None, 0), (None, 1), (0, 0), (0, 1), (1, 0), (1, 1)])
[0, 1, 1, 0]
sage: xor_transducer([(0, None)])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
```

The transducer computing the Gray code is then constructed as a Cartesian product between the shifted version and the original input (represented here by the `shift_right_transducer` and the identity transducer, respectively). This Cartesian product is then fed into the `xor_transducer` as a composition of transducers.

```
sage: product_transducer = shift_right_transducer.cartesian_product(transducers.Identity([0, 1]))
sage: Gray_transducer = xor_transducer(product_transducer)
```

We use `construct_final_word_out()` to make sure that all output is written; otherwise, we would have to make sure that a sufficient number of trailing zeros is read.

```
sage: Gray_transducer.construct_final_word_out([0])
sage: Gray_transducer.transitions()
[Transition from (('I', 0), 0) to ((0, 0), 0): 0|-,
 Transition from (('I', 0), 0) to ((1, 0), 0): 1|-,
 Transition from ((0, 0), 0) to ((0, 0), 0): 0|0,
 Transition from ((0, 0), 0) to ((1, 0), 0): 1|1,
 Transition from ((1, 0), 0) to ((0, 0), 0): 0|1,
 Transition from ((1, 0), 0) to ((1, 0), 0): 1|0]
```

There is a prepackaged transducer for Gray code, let's see whether they agree. We have to use `relabeled()` to relabel our states with integers.

```
sage: constructed = Gray_transducer.relabeled()
sage: packaged = transducers.GrayCode()
sage: constructed == packaged
True
```

Finally, we check that this indeed computes the Gray code of the first 10 non-negative integers.

```
sage: for n in srange(10):
....:     Gray_transducer(n.bits())
[]
[1]
[1, 1]
[0, 1]
[0, 1, 1]
[1, 1, 1]
[1, 0, 1]
[0, 0, 1]
[0, 0, 1, 1]
[1, 0, 1, 1]
```

Using the hook-functions

Let's use the *previous example "divison by 3"* to demonstrate the optional state and transition parameters `hook`.

First, we define what those functions should do. In our case, this is just saying in which state we are and which transition we take

```
sage: def state_hook(process, state, output):
....:     print "We are now in State %s." % (state.label(),)
sage: from sage.combinat.finite_state_machine import FSMWordSymbol
sage: def transition_hook(transition, process):
....:     print ("Currently we go from %s to %s, "
....:            "reading %s and writing %s." % (
....:                transition.from_state, transition.to_state,
....:                FSMWordSymbol(transition.word_in),
....:                FSMWordSymbol(transition.word_out)))
```

Now, let's add these hook-functions to the existing transducer:

```
sage: for s in D.iter_states():
....:     s.hook = state_hook
sage: for t in D.iter_transitions():
....:     t.hook = transition_hook
```

Rerunning the process again now gives the following output:

```
sage: D.process([1, 1, 0, 1], check_epsilon_transitions=False)
We are now in State 0.
Currently we go from 0 to 1, reading 1 and writing 0.
We are now in State 1.
Currently we go from 1 to 0, reading 1 and writing 1.
We are now in State 0.
Currently we go from 0 to 0, reading 0 and writing 0.
We are now in State 0.
Currently we go from 0 to 1, reading 1 and writing 0.
We are now in State 1.
(False, 1, [0, 1, 0, 0])
```

The example above just explains the basic idea of using hook-functions. In the following, we will use those hooks more seriously.

Warning: The hooks of the states are also called while exploring the epsilon successors of a state (during processing). In the example above, we used `check_epsilon_transitions=False` to avoid this (and also therefore got a cleaner output).

Warning: The arguments used when calling a hook have changed in trac ticket #16538 from `hook(state, process)` to `hook(process, state, output)`. If you are using an old-style hook, a deprecation warning is displayed.

Detecting sequences with same number of 0 and 1

Suppose we have a binary input and want to accept all sequences with the same number of 0 and 1. This cannot be done with a finite automaton. Anyhow, we can make usage of the hook functions to extend our finite automaton by a counter:

```
sage: from sage.combinat.finite_state_machine import FSMState, FSMTransition
sage: C = FiniteStateMachine()
sage: def update_counter(process, state, output):
....:     l = process.preview_word()
....:     process.fsm.counter += 1 if l == 1 else -1
....:     if process.fsm.counter > 0:
....:         next_state = 'positive'
....:     elif process.fsm.counter < 0:
....:         next_state = 'negative'
....:     else:
....:         next_state = 'zero'
....:     return FSMTransition(state, process.fsm.state(next_state),
....:                          l, process.fsm.counter)
sage: C.add_state(FSMState('zero', hook=update_counter,
....:             is_initial=True, is_final=True))
'zero'
sage: C.add_state(FSMState('positive', hook=update_counter))
'positive'
sage: C.add_state(FSMState('negative', hook=update_counter))
'negative'
```

Now, let's input some sequence:

```
sage: C.counter = 0; C([1, 1, 1, 1, 0, 0])
(False, 'positive', [1, 2, 3, 4, 3, 2])
```

The result is False, since there are four 1 but only two 0. We land in the state `positive` and we can also see the values of the counter in each step.

Let's try some other examples:

```
sage: C.counter = 0; C([1, 1, 0, 0])
(True, 'zero', [1, 2, 1, 0])
sage: C.counter = 0; C([0, 1, 0, 0])
(False, 'negative', [-1, 0, -1, -2])
```

See also methods `Automaton.process()` and `Transducer.process()` (or even `FiniteStateMachine.process()`), the explanation of the parameter `hook` and the examples in `FSMState` and `FSMTransition`, and the description and examples in `FSMProcessIterator` for more information on processing and hooks.

REFERENCES:

AUTHORS:

- Daniel Krenn (2012-03-27): initial version
- Clemens Heuberger (2012-04-05): initial version
- Sara Kropf (2012-04-17): initial version
- Clemens Heuberger (2013-08-21): release candidate for Sage patch
- Daniel Krenn (2013-08-21): release candidate for Sage patch
- Sara Kropf (2013-08-21): release candidate for Sage patch
- Clemens Heuberger (2013-09-02): documentation improved
- Daniel Krenn (2013-09-13): comments from trac worked in

- **Clemens Heuberger (2013-11-03): output (labels) of determinisation,** product, composition, etc. changed (for consistency), representation of state changed, documentation improved
- Daniel Krenn (2013-11-04): whitespaces in documentation corrected
- Clemens Heuberger (2013-11-04): full_group_by added
- Daniel Krenn (2013-11-04): next release candidate for Sage patch
- Sara Kropf (2013-11-08): fix for adjacency matrix
- Clemens Heuberger (2013-11-11): fix for prepone_output
- **Daniel Krenn (2013-11-11): comments from trac ticket #15078 included:** docstring of FiniteStateMachine rewritten, Automaton and Transducer inherited from FiniteStateMachine
- **Daniel Krenn (2013-11-25): documentation improved according to** comments from trac ticket #15078
- Clemens Heuberger, Daniel Krenn, Sara Kropf (2014-02-21–2014-07-18): A huge bunch of improvements. Details see trac ticket #15841, trac ticket #15847, trac ticket #15848, trac ticket #15849, trac ticket #15850, trac ticket #15922, trac ticket #15923, trac ticket #15924, trac ticket #15925, trac ticket #15928, trac ticket #15960, trac ticket #15961, trac ticket #15962, trac ticket #15963, trac ticket #15975, trac ticket #16016, trac ticket #16024, trac ticket #16061, trac ticket #16128, trac ticket #16132, trac ticket #16138, trac ticket #16139, trac ticket #16140, trac ticket #16143, trac ticket #16144, trac ticket #16145, trac ticket #16146, trac ticket #16191, trac ticket #16200, trac ticket #16205, trac ticket #16206, trac ticket #16207, trac ticket #16229, trac ticket #16253, trac ticket #16254, trac ticket #16255, trac ticket #16266, trac ticket #16355, trac ticket #16357, trac ticket #16387, trac ticket #16425, trac ticket #16539, trac ticket #16555, trac ticket #16557, trac ticket #16588, trac ticket #16589, trac ticket #16666, trac ticket #16668, trac ticket #16674, trac ticket #16675, trac ticket #16677.
- Daniel Krenn (2015-09-14): cleanup trac ticket #18227

ACKNOWLEDGEMENT:

- Clemens Heuberger, Daniel Krenn and Sara Kropf are supported by the Austrian Science Fund (FWF): P 24644-N26.

Methods

class sage.combinat.finite_state_machine.**Automaton**(**args*, ***kwargs*)

Bases: sage.combinat.finite_state_machine.FiniteStateMachine

This creates an automaton, which is a finite state machine, whose transitions have input labels.

An automaton has additional features like creating a deterministic and a minimized automaton.

See class FiniteStateMachine for more information.

EXAMPLES:

We can create an automaton recognizing even numbers (given in binary and read from left to right) in the following way:

```
sage: A = Automaton([('P', 'Q', 0), ('P', 'P', 1),
....:                ('Q', 'P', 1), ('Q', 'Q', 0)],
....:               initial_states=['P'], final_states=['Q'])
sage: A
Automaton with 2 states
sage: A([0])
True
sage: A([1, 1, 0])
True
```

```
sage: A([1, 0, 1])
False
```

Note that the full output of the commands can be obtained by calling `process()` and looks like this:

```
sage: A.process([1, 0, 1])
(False, 'P')
```

TESTS:

```
sage: Automaton()
Empty automaton
```

cartesian_product(*other*, *only_accessible_components=True*)

Returns a new automaton which accepts an input if it is accepted by both given automata.

INPUT:

- `other` – an automaton
- `only_accessible_components` – If `True` (default), then the result is piped through `accessible_components()`. If no `new_input_alphabet` is given, it is determined by `determine_alphabets()`.

OUTPUT:

A new automaton which computes the intersection (see below) of the languages of `self` and `other`.

The set of states of the new automaton is the Cartesian product of the set of states of both given automata. There is a transition $((A, B), (C, D), a)$ in the new automaton if there are transitions (A, C, a) and (B, D, a) in the old automata.

The methods `intersection()` and `cartesian_product()` are the same (for automata).

EXAMPLES:

```
sage: aut1 = Automaton([('1', '2', 1),
....:                   ('2', '2', 1),
....:                   ('2', '2', 0)],
....:                  initial_states=['1'],
....:                  final_states=['2'],
....:                  determine_alphabets=True)
sage: aut2 = Automaton([('A', 'A', 1),
....:                   ('A', 'B', 0),
....:                   ('B', 'B', 0),
....:                   ('B', 'A', 1)],
....:                  initial_states=['A'],
....:                  final_states=['B'],
....:                  determine_alphabets=True)
sage: res = aut1.intersection(aut2)
sage: (aut1([1, 1]), aut2([1, 1]), res([1, 1]))
(True, False, False)
sage: (aut1([1, 0]), aut2([1, 0]), res([1, 0]))
(True, True, True)
sage: res.transitions()
[Transition from ('1', 'A') to ('2', 'A'): 1|-,
 Transition from ('2', 'A') to ('2', 'B'): 0|-,
 Transition from ('2', 'A') to ('2', 'A'): 1|-,
 Transition from ('2', 'B') to ('2', 'B'): 0|-,
 Transition from ('2', 'B') to ('2', 'A'): 1|-]
```

For automata with epsilon-transitions, intersection is not well defined. But for any finite state machine, epsilon-transitions can be removed by remove_epsilon_transitions().

```
sage: a1 = Automaton([(0, 0, 0),
....:                 (0, 1, None),
....:                 (1, 1, 1),
....:                 (1, 2, 1)],
....:                 initial_states=[0],
....:                 final_states=[1],
....:                 determine_alphabets=True)
sage: a2 = Automaton([(0, 0, 0), (0, 1, 1), (1, 1, 1)],
....:                 initial_states=[0],
....:                 final_states=[1],
....:                 determine_alphabets=True)
sage: a1.intersection(a2)
Traceback (most recent call last):
...
ValueError: An epsilon-transition (with empty input)
was found.
sage: a1.remove_epsilon_transitions()  # not tested (since not implemented yet)
sage: a1.intersection(a2)  # not tested
```

complement()

Return the complement of this automaton.

OUTPUT:

An Automaton.

If this automaton recognizes language $\mathcal{L}$ over an input alphabet $\mathcal{A}$, then the complement recognizes $\mathcal{A} \setminus \mathcal{L}$.

EXAMPLES:

```
sage: A = automata.Word([0, 1])
sage: [w for w in
....:  [], [0], [1], [0, 0], [0, 1], [1, 0], [1, 1]
....:  if A(w)]
[[0, 1]]
sage: Ac = A.complement()
sage: Ac.transitions()
[Transition from 0 to 1: 0|-,
 Transition from 0 to 3: 1|-,
 Transition from 2 to 3: 0|-,
 Transition from 2 to 3: 1|-,
 Transition from 1 to 2: 1|-,
 Transition from 1 to 3: 0|-,
 Transition from 3 to 3: 0|-,
 Transition from 3 to 3: 1|-]
sage: [w for w in
....:  [], [0], [1], [0, 0], [0, 1], [1, 0], [1, 1]
....:  if Ac(w)]
[[], [0], [1], [0, 0], [1, 0], [1, 1]]
```

The automaton must be deterministic:

```
sage: A = automata.Word([0]) * automata.Word([1])
sage: A.complement()
Traceback (most recent call last):
...
ValueError: The finite state machine must be deterministic.
sage: Ac = A.determinisation().complement()
```

```
sage: [w for w in
....:  [], [0], [1], [0, 0], [0, 1], [1, 0], [1, 1]
....:  if Ac(w)]
[[], [0], [1], [0, 0], [1, 0], [1, 1]]
```

determinisation()

Returns a deterministic automaton which accepts the same input words as the original one.

INPUT:

Nothing.

OUTPUT:

A new automaton, which is deterministic.

The labels of the states of the new automaton are frozensets of states of `self`. The color of a new state is the frozenset of colors of the constituent states of `self`. Therefore, the colors of the constituent states have to be hashable. However, if all constituent states have color `None`, then the resulting color is `None`, too.

The input alphabet must be specified.

EXAMPLES:

```
sage: aut = Automaton([('A', 'A', 0), ('A', 'B', 1), ('B', 'B', 1)],
....:                 initial_states=['A'], final_states=['B'])
sage: aut.determinisation().transitions()
[Transition from frozenset(['A'])
             to frozenset(['A']): 0|-,
 Transition from frozenset(['A'])
             to frozenset(['B']): 1|-,
 Transition from frozenset(['B'])
             to frozenset([]): 0|-,
 Transition from frozenset(['B'])
             to frozenset(['B']): 1|-,
 Transition from frozenset([])
             to frozenset([]): 0|-,
 Transition from frozenset([])
             to frozenset([]): 1|-]

sage: A = Automaton([('A', 'A', 1), ('A', 'A', 0), ('A', 'B', 1),
....:                ('B', 'C', 0), ('C', 'C', 1), ('C', 'C', 0)],
....:               initial_states=['A'], final_states=['C'])
sage: A.determinisation().states()
[frozenset(['A']), frozenset(['A', 'B']),
frozenset(['A', 'C']), frozenset(['A', 'C', 'B'])]

sage: A = Automaton([(0, 1, 1), (0, 2, [1, 1]), (0, 3, [1, 1, 1]),
....:                (1, 0, -1), (2, 0, -2), (3, 0, -3)],
....:               initial_states=[0], final_states=[0, 1, 2, 3])
sage: B = A.determinisation().relabeled().coaccessible_components()
sage: sorted(B.transitions())
[Transition from 0 to 1: 1|-,
 Transition from 1 to 0: -1|-,
 Transition from 1 to 3: 1|-,
 Transition from 3 to 0: -2|-,
 Transition from 3 to 4: 1|-,
 Transition from 4 to 0: -3|-]
```

Note that colors of states have to be hashable:

```
sage: A = Automaton([[0, 0, 0]], initial_states=[0])
sage: A.state(0).color = []
sage: A.determinisation()
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
sage: A.state(0).color = ()
sage: A.determinisation()
Automaton with 1 state
```

If the colors of all constituent states are `None`, the resulting color is `None`, too (trac ticket #19199):

```
sage: A = Automaton([(0, 0, 0)],
....:               initial_states=[0],
....:               final_states=[0])
sage: [s.color for s in A.determinisation().iter_states()]
[None]
```

TESTS:

This is from trac ticket #15078, comment 13.

```
sage: D = {'A': [('A', 'a'), ('B', 'a'), ('A', 'b')],
....:      'C': [], 'B': [('C', 'b')]}
sage: auto = Automaton(D, initial_states=['A'], final_states=['C'])
sage: auto.is_deterministic()
False
sage: auto.process(list('aaab'))
[(False, 'A'), (True, 'C')]
sage: auto.states()
['A', 'C', 'B']
sage: Ddet = auto.determinisation()
sage: Ddet
Automaton with 3 states
sage: Ddet.is_deterministic()
True
sage: sorted(Ddet.transitions())
[Transition from frozenset(['A']) to frozenset(['A', 'B']): 'a'|-,
 Transition from frozenset(['A']) to frozenset(['A']): 'b'|-,
 Transition from frozenset(['A', 'B']) to frozenset(['A', 'B']): 'a'|-,
 Transition from frozenset(['A', 'B']) to frozenset(['A', 'C']): 'b'|-,
 Transition from frozenset(['A', 'C']) to frozenset(['A', 'B']): 'a'|-,
 Transition from frozenset(['A', 'C']) to frozenset(['A']): 'b'|-]
sage: Ddet.initial_states()
[frozenset(['A'])]
sage: Ddet.final_states()
[frozenset(['A', 'C'])]
sage: Ddet.process(list('aaab'))
(True, frozenset(['A', 'C']))
```

Test that trac ticket #18992 is fixed:

```
sage: A = Automaton([(0, 1, []), (1, 1, 0)],
....:               initial_states=[0], final_states=[1])
sage: B = A.determinisation()
sage: B.initial_states()
[frozenset([0, 1])]
sage: B.final_states()
[frozenset([0, 1]), frozenset([1])]
```

```
sage: B.transitions()
[Transition from frozenset([0, 1]) to frozenset([1]): 0|-,
Transition from frozenset([1]) to frozenset([1]): 0|-]
sage: C = B.minimization().relabeled()
sage: C.initial_states()
[0]
sage: C.final_states()
[0]
sage: C.transitions()
[Transition from 0 to 0: 0|-]
```

intersection(*other*, *only_accessible_components=True*)

Returns a new automaton which accepts an input if it is accepted by both given automata.

INPUT:

- `other` – an automaton
- `only_accessible_components` – If `True` (default), then the result is piped through `accessible_components()`. If no `new_input_alphabet` is given, it is determined by `determine_alphabets()`.

OUTPUT:

A new automaton which computes the intersection (see below) of the languages of `self` and `other`.

The set of states of the new automaton is the Cartesian product of the set of states of both given automata. There is a transition $((A, B), (C, D), a)$ in the new automaton if there are transitions (A, C, a) and (B, D, a) in the old automata.

The methods `intersection()` and `cartesian_product()` are the same (for automata).

EXAMPLES:

```
sage: aut1 = Automaton([('1', '2', 1),
....:                   ('2', '2', 1),
....:                   ('2', '2', 0)],
....:                  initial_states=['1'],
....:                  final_states=['2'],
....:                  determine_alphabets=True)
sage: aut2 = Automaton([('A', 'A', 1),
....:                   ('A', 'B', 0),
....:                   ('B', 'B', 0),
....:                   ('B', 'A', 1)],
....:                  initial_states=['A'],
....:                  final_states=['B'],
....:                  determine_alphabets=True)
sage: res = aut1.intersection(aut2)
sage: (aut1([1, 1]), aut2([1, 1]), res([1, 1]))
(True, False, False)
sage: (aut1([1, 0]), aut2([1, 0]), res([1, 0]))
(True, True, True)
sage: res.transitions()
[Transition from ('1', 'A') to ('2', 'A'): 1|-,
 Transition from ('2', 'A') to ('2', 'B'): 0|-,
 Transition from ('2', 'A') to ('2', 'A'): 1|-,
 Transition from ('2', 'B') to ('2', 'B'): 0|-,
 Transition from ('2', 'B') to ('2', 'A'): 1|-]
```

For automata with epsilon-transitions, intersection is not well defined. But for any finite state machine,

epsilon-transitions can be removed by remove_epsilon_transitions().

```
sage: a1 = Automaton([(0, 0, 0),
....:                 (0, 1, None),
....:                 (1, 1, 1),
....:                 (1, 2, 1)],
....:                 initial_states=[0],
....:                 final_states=[1],
....:                 determine_alphabets=True)
sage: a2 = Automaton([(0, 0, 0), (0, 1, 1), (1, 1, 1)],
....:                 initial_states=[0],
....:                 final_states=[1],
....:                 determine_alphabets=True)
sage: a1.intersection(a2)
Traceback (most recent call last):
...
ValueError: An epsilon-transition (with empty input)
was found.
sage: a1.remove_epsilon_transitions()  # not tested (since not implemented yet)
sage: a1.intersection(a2)  # not tested
```

is_equivalent(*other*)

Test whether two automata are equivalent, i.e., accept the same language.

INPUT:

- other – an Automaton.

EXAMPLES:

```
sage: A = Automaton([(0, 0, 0), (0, 1, 1), (1, 0, 1)],
....:               initial_states=[0],
....:               final_states=[0])
sage: B = Automaton([('a', 'a', 0), ('a', 'b', 1), ('b', 'a', 1)],
....:               initial_states=['a'],
....:               final_states=['a'])
sage: A.is_equivalent(B)
True
sage: B.add_transition('b', 'a', 0)
Transition from 'b' to 'a': 0|-
sage: A.is_equivalent(B)
False
```

language(*max_length=None*, ***kwargs*)

Return all words accepted by this automaton.

INPUT:

- max_length – an integer or None (default). Only inputs of length at most max_length will be considered. If None, then this iterates over all possible words without length restrictions.
- kwargs – will be passed on to to the process iterator. See process() for a description.

OUTPUT:

An iterator.

EXAMPLES:

```
sage: NAF = Automaton(
....:     {'A': [('A', 0), ('B', 1), ('B', -1)],
....:      'B': [('A', 0)]},
....:     initial_states=['A'], final_states=['A', 'B'])
```

```
sage: list(NAF.language(3))
[[],
 [0], [-1], [1],
 [-1, 0], [0, 0], [1, 0], [0, -1], [0, 1],
 [-1, 0, 0], [0, -1, 0], [0, 0, 0], [0, 1, 0], [1, 0, 0],
 [-1, 0, -1], [-1, 0, 1], [0, 0, -1],
 [0, 0, 1], [1, 0, -1], [1, 0, 1]]
```

See also:

FiniteStateMachine.language(),process().

TESTS:

```
sage: def R(ell):
....:     return (2^(ell+2)-(-1)^ell)/3
sage: import itertools
sage: all(len(list(NAFs)) == R(ell) for ell, NAFs in
....:     itertools.groupby(NAF.language(5), key=len))
True
```

minimization(*algorithm=None*)

Returns the minimization of the input automaton as a new automaton.

INPUT:

- algorithm – Either Moore's algorithm (by algorithm='Moore' or as default for deterministic automata) or Brzozowski's algorithm (when algorithm='Brzozowski' or when the automaton is not deterministic) is used.

OUTPUT:

A new automaton.

The resulting automaton is deterministic and has a minimal number of states.

EXAMPLES:

```
sage: A = Automaton([('A', 'A', 1), ('A', 'A', 0), ('A', 'B', 1),
....:                ('B', 'C', 0), ('C', 'C', 1), ('C', 'C', 0)],
....:               initial_states=['A'], final_states=['C'])
sage: B = A.minimization(algorithm='Brzozowski')
sage: B.transitions(B.states()[1])
[Transition from frozenset([frozenset(['A', 'C', 'B']),
frozenset(['C', 'B']), frozenset(['A', 'C'])]) to
frozenset([frozenset(['A', 'C', 'B']), frozenset(['C', 'B']),
frozenset(['A', 'C']), frozenset(['C'])]): 0|-,
Transition from frozenset([frozenset(['A', 'C', 'B']),
frozenset(['C', 'B']), frozenset(['A', 'C'])]) to
frozenset([frozenset(['A', 'C', 'B']), frozenset(['C', 'B']),
frozenset(['A', 'C'])]): 1|-]
sage: len(B.states())
3
sage: C = A.minimization(algorithm='Brzozowski')
sage: C.transitions(C.states()[1])
[Transition from frozenset([frozenset(['A', 'C', 'B']),
frozenset(['C', 'B']), frozenset(['A', 'C'])]) to
frozenset([frozenset(['A', 'C', 'B']), frozenset(['C', 'B']),
frozenset(['A', 'C']), frozenset(['C'])]): 0|-,
Transition from frozenset([frozenset(['A', 'C', 'B']),
frozenset(['C', 'B']), frozenset(['A', 'C'])]) to
frozenset([frozenset(['A', 'C', 'B']), frozenset(['C', 'B']),
```

```
frozenset(['A', 'C'])]): 1|-]
sage: len(C.states())
3

sage: aut = Automaton([('1', '2', 'a'), ('2', '3', 'b'),
....:                  ('3', '2', 'a'), ('2', '1', 'b'),
....:                  ('3', '4', 'a'), ('4', '3', 'b')],
....:                 initial_states=['1'], final_states=['1'])
sage: min = aut.minimization(algorithm='Brzozowski')
sage: [len(min.states()), len(aut.states())]
[3, 4]
sage: min = aut.minimization(algorithm='Moore')
Traceback (most recent call last):
...
NotImplementedError: Minimization via Moore's Algorithm is only
implemented for deterministic finite state machines
```

process (**args*, ***kwargs*)

Return whether the automaton accepts the input and the state where the computation stops.

INPUT:

- `input_tape` – the input tape can be a list or an iterable with entries from the input alphabet. If we are working with a multi-tape machine (see parameter `use_multitape_input` and notes below), then the tape is a list or tuple of tracks, each of which can be a list or an iterable with entries from the input alphabet.
- `initial_state` or `initial_states` – the initial state(s) in which the machine starts. Either specify a single one with `initial_state` or a list of them with `initial_states`. If both are given, `initial_state` will be appended to `initial_states`. If neither is specified, the initial states of the finite state machine are taken.
- `list_of_outputs` – (default: `None`) a boolean or `None`. If `True`, then the outputs are given in list form (even if we have no or only one single output). If `False`, then the result is never a list (an exception is raised if the result cannot be returned). If `list_of_outputs=None` the method determines automatically what to do (e.g. if a non-deterministic machine returns more than one path, then the output is returned in list form).
- `only_accepted` – (default: `False`) a boolean. If set, then the first argument in the output is guaranteed to be `True` (if the output is a list, then the first argument of each element will be `True`).
- `full_output` – (default: `True`) a boolean. If set, then the full output is given, otherwise only whether the sequence is accepted or not (the first entry below only).
- `always_include_output` – if set (not by default), always return a triple containing the (non-existing) output. This is in order to obtain output compatible with that of `FiniteStateMachine.process()`. If this parameter is set, `full_output` has no effect.
- `format_output` – a function that translates the written output (which is in form of a list) to something more readable. By default (`None`) identity is used here.
- `check_epsilon_transitions` – (default: `True`) a boolean. If `False`, then epsilon transitions are not taken into consideration during process.
- `write_final_word_out` – (default: `True`) a boolean specifying whether the final output words should be written or not.
- `use_multitape_input` – (default: `False`) a boolean. If `True`, then the multi-tape mode of the process iterator is activated. See also the notes below for multi-tape machines.

- process_all_prefixes_of_input – (default: False) a boolean. If True, then each prefix of the input word is processed (instead of processing the whole input word at once). Consequently, there is an output generated for each of these prefixes.
- process_iterator_class – (default: None) a class inherited from FSMProcessIterator. If None, then FSMProcessIterator is taken. An instance of this class is created and is used during the processing.

OUTPUT:

The full output is a pair (or a list of pairs, cf. parameter list_of_outputs), where

- the first entry is True if the input string is accepted and
- the second gives the state reached after processing the input tape (This is a state with label None if the input could not be processed, i.e., if at one point no transition to go on could be found.).

If full_output is False, then only the first entry is returned.

If always_include_output is set, an additional third entry [] is included.

Note that in the case the automaton is not deterministic, all possible paths are taken into account. You can use determinisation() to get a deterministic automaton machine.

This function uses an iterator which, in its simplest form, goes from one state to another in each step. To decide which way to go, it uses the input words of the outgoing transitions and compares them to the input tape. More precisely, in each step, the iterator takes an outgoing transition of the current state, whose input label equals the input letter of the tape.

If the choice of the outgoing transition is not unique (i.e., we have a non-deterministic finite state machine), all possibilites are followed. This is done by splitting the process into several branches, one for each of the possible outgoing transitions.

The process (iteration) stops if all branches are finished, i.e., for no branch, there is any transition whose input word coincides with the processed input tape. This can simply happen when the entire tape was read.

Also see __call__() for a version of process() with shortened output.

Internally this function creates and works with an instance of FSMProcessIterator. This iterator can also be obtained with iter_process().

If working with multi-tape finite state machines, all input words of transitions are words of k-tuples of letters. Moreover, the input tape has to consist of k tracks, i.e., be a list or tuple of k iterators, one for each track.

Warning: Working with multi-tape finite state machines is still experimental and can lead to wrong outputs.

EXAMPLES:

In the following examples, we construct an automaton which accepts non-adjacent forms (see also the example on *non-adjacent forms* in the documentation of the module *Finite State Machines, Automata, Transducers*) and then test it by feeding it with several binary digit expansions.

```
sage: NAF = Automaton(
....:     {'_': [('_', 0), ('1', 1)], '1': [('_', 0)]},
....:     initial_states=['_'], final_states=['_', '1'])
sage: [NAF.process(w) for w in [[0], [0, 1], [1, 1], [0, 1, 0, 1],
....:                           [0, 1, 1, 1, 0], [1, 0, 0, 1, 1]]]
[(True, '_'), (True, '1'), (False, None),
 (True, '1'), (False, None), (False, None)]
```

If we just want a condensed output, we use:

```
sage: [NAF.process(w, full_output=False)
....:     for w in [[0], [0, 1], [1, 1], [0, 1, 0, 1],
....:               [0, 1, 1, 1, 0], [1, 0, 0, 1, 1]]]
[True, True, False, True, False, False]
```

It is equivalent to:

```
sage: [NAF(w) for w in [[0], [0, 1], [1, 1], [0, 1, 0, 1],
....:                   [0, 1, 1, 1, 0], [1, 0, 0, 1, 1]]]
[True, True, False, True, False, False]
```

The following example illustrates the difference between non-existing paths and reaching a non-final state:

```
sage: NAF.process([2])
(False, None)
sage: NAF.add_transition(('_', 's', 2))
Transition from '_' to 's': 2|-
sage: NAF.process([2])
(False, 's')
```

A simple example of a (non-deterministic) multi-tape automaton is the following: It checks whether the two input tapes have the same number of ones:

```
sage: M = Automaton([('=', '=', (1, 1)),
....:                ('=', '=', (None, 0)),
....:                ('=', '=', (0, None)),
....:                ('=', '<', (None, 1)),
....:                ('<', '<', (None, 1)),
....:                ('<', '<', (None, 0)),
....:                ('=', '>', (1, None)),
....:                ('>', '>', (1, None)),
....:                ('>', '>', (0, None))],
....:               initial_states=['='],
....:               final_states=['='])
sage: M.process(([1, 0, 1], [1, 0]), use_multitape_input=True)
(False, '>')
sage: M.process(([0, 1, 0], [0, 1, 1]), use_multitape_input=True)
(False, '<')
sage: M.process(([1, 1, 0, 1], [0, 0, 1, 0, 1, 1]),
....:           use_multitape_input=True)
(True, '=')
```

Alternatively, we can use the following (non-deterministic) multi-tape automaton for the same check:

```
sage: N = Automaton([('=', '=', (0, 0)),
....:                ('=', '<', (None, 1)),
....:                ('<', '<', (0, None)),
....:                ('<', '=', (1, None)),
....:                ('=', '>', (1, None)),
....:                ('>', '>', (None, 0)),
....:                ('>', '=', (None, 1))],
....:               initial_states=['='],
....:               final_states=['='])
sage: N.process(([1, 0, 1], [1, 0]), use_multitape_input=True)
(False, '>')
sage: N.process(([0, 1, 0], [0, 1, 1]), use_multitape_input=True)
(False, '<')
sage: N.process(([1, 1, 0, 1], [0, 0, 1, 0, 1, 1]),
....:           use_multitape_input=True)
```

```
(True, '=')
```

See also:

FiniteStateMachine.process(), Transducer.process(), iter_process(), __call__(), FSMProcessIterator.

shannon_parry_markov_chain()

Compute a time homogeneous Markov chain such that all words of a given length recognized by the original automaton occur as the output with the same weight; the transition probabilities correspond to the Parry measure.

OUTPUT:

A Markov chain. Its input labels are the transition probabilities, the output labels the labels of the original automaton. In order to obtain equal weight for all words of the same length, an "exit weight" is needed. It is stored in the attribute `color` of the states of the Markov chain. The weights of the words of the same length sum up to one up to an exponentially small error.

The stationary distribution of this Markov chain is saved as the initial probabilities of the states.

The transition probabilities correspond to the Parry measure (see [S1948] and [P1964]).

The automaton is assumed to be deterministic, irreducible and aperiodic. All states must be final.

EXAMPLES:

```
sage: NAF = Automaton([(0, 0, 0), (0, 1, 1), (0, 1, -1),
....:                  (1, 0, 0)], initial_states=[0],
....:                 final_states=[0, 1])
sage: P_NAF = NAF.shannon_parry_markov_chain()
sage: P_NAF.transitions()
[Transition from 0 to 0: 1/2|0,
 Transition from 0 to 1: 1/4|1,
 Transition from 0 to 1: 1/4|-1,
 Transition from 1 to 0: 1|0]
sage: for s in P_NAF.iter_states():
....:     print s.color
3/4
3/2
```

The stationary distribution is also computed and saved as the initial probabilities of the returned Markov chain:

```
sage: for s in P_NAF.states():
....:     print s, s.initial_probability
0 2/3
1 1/3
```

The automaton is assumed to be deterministic, irreducible and aperiodic:

```
sage: A = Automaton([(0, 0, 0), (0, 1, 1), (1, 1, 1), (1, 1, 0)],
....:               initial_states=[0])
sage: A.shannon_parry_markov_chain()
Traceback (most recent call last):
...
NotImplementedError: Automaton must be strongly connected.
sage: A = Automaton([(0, 0, 0), (0, 1, 0)],
....:               initial_states=[0])
sage: A.shannon_parry_markov_chain()
Traceback (most recent call last):
```

```
...
NotImplementedError: Automaton must be deterministic.
sage: A = Automaton([(0, 1, 0), (1, 0, 0)],
....:               initial_states=[0])
sage: A.shannon_parry_markov_chain()
Traceback (most recent call last):
...
NotImplementedError: Automaton must be aperiodic.
```

All states must be final:

```
sage: A = Automaton([(0, 1, 0), (0, 0, 1), (1, 0, 0)],
....:               initial_states=[0])
sage: A.shannon_parry_markov_chain()
Traceback (most recent call last):
...
NotImplementedError: All states must be final.
```

ALGORITHM:

See [HKP2015a], Lemma 4.1.

REFERENCES:

with_output(*word_out_function=None*)

Construct a transducer out of this automaton.

INPUT:

- `word_out_function` – (default: `None`) a function. It transforms a `transition` to the output word for this transition.

  If this is `None`, then the output word will be equal to the input word of each transition.

OUTPUT:

A transducer.

EXAMPLES:

```
sage: A = Automaton([(0, 0, 'A'), (0, 1, 'B'), (1, 2, 'C')])
sage: T = A.with_output(); T
Transducer with 3 states
sage: T.transitions()
[Transition from 0 to 0: 'A'|'A',
 Transition from 0 to 1: 'B'|'B',
 Transition from 1 to 2: 'C'|'C']
```

This result is in contrast to:

```
sage: Transducer(A).transitions()
[Transition from 0 to 0: 'A'|-,
 Transition from 0 to 1: 'B'|-,
 Transition from 1 to 2: 'C'|-]
```

where no output labels are created.

Here is another example:

```
sage: T2 = A.with_output(lambda t: [c.lower() for c in t.word_in])
sage: T2.transitions()
[Transition from 0 to 0: 'A'|'a',
```

```
 Transition from 0 to 1: 'B'|'b',
 Transition from 1 to 2: 'C'|'c']
```

We can obtain the same result by composing two transducers. As inner transducer of the composition, we use `with_output()` without the optional argument `word_out_function` (which makes the output of each transition equal to its input); as outer transducer we use a `map-transducer` (for converting to lower case). This gives

```
sage: L = transducers.map(lambda x: x.lower(), ['A', 'B', 'C'])
sage: L.composition(A.with_output()).relabeled().transitions()
[Transition from 0 to 0: 'A'|'a',
 Transition from 0 to 1: 'B'|'b',
 Transition from 1 to 2: 'C'|'c']
```

See also:

`input_projection()`, `output_projection()`, `Transducer`, `transducers.map()`.

TESTS:

```
sage: A.with_output().input_projection() == A
True
sage: NAF = Automaton(
....:     {'A': [('A', 0), ('B', 1), ('B', -1)], 'B': [('A', 0)]})
sage: NAF.with_output().input_projection() == NAF
True
sage: B = Automaton(
....:     {0: [(0, 'a'), (1, ['b', 'c']), (2, ['d', 'e'])],
....:      1: [(0, ['f', 'g']), (1, 'h'), (2, None)],
....:      2: [(0, None), (1, None), (2, ['i', 'j'])]},
....:     initial_states=[1, 2], final_states=[0])
sage: B.with_output(lambda t: [c.upper() for c in t.word_in]).input_projection() == B
True
```

`sage.combinat.finite_state_machine.`**`FSMLetterSymbol`**(*letter*)

Returns a string associated to the input letter.

INPUT:

- `letter` – the input letter or `None` (representing the empty word).

OUTPUT:

If `letter` is `None` the symbol for the empty word `FSMEmptyWordSymbol` is returned, otherwise the string associated to the letter.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMLetterSymbol
sage: FSMLetterSymbol(0)
'0'
sage: FSMLetterSymbol(None)
'_'
```

class sage.combinat.finite_state_machine.**FSMProcessIterator**(*fsm*, *input_tape=None*, *initial_state=None*, *initial_states=[]*, *use_multitape_input=False*, *check_epsilon_transitions=True*, *write_final_word_out=True*, *format_output=None*, *process_all_prefixes_of_input=False*, ***kwargs*)

Bases: sage.structure.sage_object.SageObject, _abcoll.Iterator

This class takes an input, feeds it into a finite state machine (automaton or transducer, in particular), tests whether this was successful and calculates the written output.

INPUT:

- `fsm` – the finite state machine on which the input should be processed.
- `input_tape` – the input tape can be a list or an iterable with entries from the input alphabet. If we are working with a multi-tape machine (see parameter `use_multitape_input` and notes below), then the tape is a list or tuple of tracks, each of which can be a list or an iterable with entries from the input alphabet.
- `initial_state` or `initial_states` – the initial state(s) in which the machine starts. Either specify a single one with `initial_state` or a list of them with `initial_states`. If both are given, `initial_state` will be appended to `initial_states`. If neither is specified, the initial states of the finite state machine are taken.
- `format_output` – a function that translates the written output (which is in form of a list) to something more readable. By default (`None`) identity is used here.
- `check_epsilon_transitions` – (default: `True`) a boolean. If `False`, then epsilon transitions are not taken into consideration during process.
- `write_final_word_out` – (default: `True`) a boolean specifying whether the final output words should be written or not.
- `use_multitape_input` – (default: `False`) a boolean. If `True`, then the multi-tape mode of the process iterator is activated. See also the notes below for multi-tape machines.
- `process_all_prefixes_of_input` – (default: `False`) a boolean. If `True`, then each prefix of the input word is processed (instead of processing the whole input word at once). Consequently, there is an output generated for each of these prefixes.

OUTPUT:

An iterator.

In its simplest form, it behaves like an iterator which, in each step, goes from one state to another. To decide which way to go, it uses the input words of the outgoing transitions and compares them to the input tape. More precisely, in each step, the process iterator takes an outgoing transition of the current state, whose input label equals the input letter of the tape. The output label of the transition, if present, is written on the output tape.

If the choice of the outgoing transition is not unique (i.e., we have a non-deterministic finite state machine), all possibilites are followed. This is done by splitting the process into several branches, one for each of the possible outgoing transitions.

The process (iteration) stops if all branches are finished, i.e., for no branch, there is any transition whose input word coincides with the processed input tape. This can simply happen when the entire tape was read. When the process stops, a `StopIteration` exception is thrown.

Warning: Processing an input tape of length n usually takes at least $n + 1$ iterations, since there will be $n + 1$ states visited (in the case the taken transitions have input words consisting of single letters).

An instance of this class is generated when `FiniteStateMachine.process()` or `FiniteStateMachine.iter_process()` of a finite state machine, an automaton, or a transducer is invoked.

When working with multi-tape finite state machines, all input words of transitions are words of k-tuples of letters. Moreover, the input tape has to consist of k tracks, i.e., be a list or tuple of k iterators, one for each track.

Warning: Working with multi-tape finite state machines is still experimental and can lead to wrong outputs.

EXAMPLES:

The following transducer reads binary words and outputs a word, where blocks of ones are replaced by just a single one. Further only words that end with a zero are accepted.

```
sage: T = Transducer({'A': [('A', 0, 0), ('B', 1, None)],
....:                 'B': [('B', 1, None), ('A', 0, [1, 0])]},
....:     initial_states=['A'], final_states=['A'])
sage: input = [1, 1, 0, 0, 1, 0, 1, 1, 1, 0]
sage: T.process(input)
(True, 'A', [1, 0, 0, 1, 0, 1, 0])
```

The function `FiniteStateMachine.process()` (internally) uses a `FSMProcessIterator`. We can do that manually, too, and get full access to the iteration process:

```
sage: from sage.combinat.finite_state_machine import FSMProcessIterator
sage: it = FSMProcessIterator(T, input_tape=input)
sage: for current in it:
....:     print current
process (1 branch)
+ at state 'B'
+-- tape at 1, [[]]
process (1 branch)
+ at state 'B'
+-- tape at 2, [[]]
process (1 branch)
+ at state 'A'
+-- tape at 3, [[1, 0]]
process (1 branch)
+ at state 'A'
+-- tape at 4, [[1, 0, 0]]
process (1 branch)
+ at state 'B'
+-- tape at 5, [[1, 0, 0]]
process (1 branch)
+ at state 'A'
+-- tape at 6, [[1, 0, 0, 1, 0]]
process (1 branch)
+ at state 'B'
+-- tape at 7, [[1, 0, 0, 1, 0]]
process (1 branch)
+ at state 'B'
+-- tape at 8, [[1, 0, 0, 1, 0]]
process (1 branch)
+ at state 'B'
+-- tape at 9, [[1, 0, 0, 1, 0]]
```

```
process (1 branch)
+ at state 'A'
+-- tape at 10, [[1, 0, 0, 1, 0, 1, 0]]
process (0 branches)
sage: it.result()
[Branch(accept=True, state='A', output=[1, 0, 0, 1, 0, 1, 0])]

sage: T = Transducer([(0, 0, 0, 'a'), (0, 1, 0, 'b'),
....:                 (1, 2, 1, 'c'), (2, 0, 0, 'd'),
....:                 (2, 1, None, 'd')],
....:                initial_states=[0], final_states=[2])
sage: T.process([0, 0, 1], format_output=lambda o: ''.join(o))
[(False, 1, 'abcd'), (True, 2, 'abc')]
sage: it = FSMProcessIterator(T, input_tape=[0, 0, 1],
....:                         format_output=lambda o: ''.join(o))
sage: for current in it:
....:     print current
process (2 branches)
+ at state 0
+-- tape at 1, [['a']]
+ at state 1
+-- tape at 1, [['b']]
process (2 branches)
+ at state 0
+-- tape at 2, [['a', 'a']]
+ at state 1
+-- tape at 2, [['a', 'b']]
process (2 branches)
+ at state 1
+-- tape at 3, [['a', 'b', 'c', 'd']]
+ at state 2
+-- tape at 3, [['a', 'b', 'c']]
process (0 branches)
sage: it.result()
[Branch(accept=False, state=1, output='abcd'),
 Branch(accept=True, state=2, output='abc')]
```

See also:

FiniteStateMachine.process(), Automaton.process(), Transducer.process(), FiniteStateMachine.iter_process(), FiniteStateMachine.__call__(), next().

TESTS:

```
sage: T = Transducer([[0, 0, 0, 0]])
sage: T.process([])
Traceback (most recent call last):
...
ValueError: No state is initial.

sage: T = Transducer([[0, 1, 0, 0]], initial_states=[0, 1])
sage: T.process([])
[(False, 0, []), (False, 1, [])]

sage: T = Transducer([[0, 0, 0, 0]],
....:                initial_states=[0], final_states=[0])
sage: T.state(0).final_word_out = [42]
sage: T.process([0])
(True, 0, [0, 42])
```

```
sage: T.process([0], write_final_word_out=False)
(True, 0, [0])
```

class Current

Bases: dict

This class stores the branches which have to be processed during iteration and provides a nicer formatting of them.

This class is derived from dict. It is returned by the next-function during iteration.

EXAMPLES:

In the following example you can see the dict directly and then the nicer output provided by this class:

```
sage: from sage.combinat.finite_state_machine import FSMProcessIterator
sage: inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:     initial_states=['A'], final_states=['A'])
sage: it = FSMProcessIterator(inverter, input_tape=[0, 1])
sage: for current in it:
....:     print dict(current)
....:     print current
{((1, 0),): {'A': Branch(tape_cache=tape at 1, outputs=[[1]])}}
process (1 branch)
+ at state 'A'
+-- tape at 1, [[1]]
{((2, 0),): {'A': Branch(tape_cache=tape at 2, outputs=[[1, 0]])}}
process (1 branch)
+ at state 'A'
+-- tape at 2, [[1, 0]]
{}
process (0 branches)
```

class FSMProcessIterator.**FinishedBranch**

Bases: tuple

A named tuple representing the attributes of a branch, once it is fully processed.

accept

Alias for field number 0

output

Alias for field number 2

state

Alias for field number 1

FSMProcessIterator.**accept_input**

Is True if the reached state is accepted. This is only available at the end of the iteration process.

> **Warning:** This attribute is deprecated and should not be used any longer (it may return a wrong result for non-deterministic finite state machines).

TESTS:

```
sage: inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:     initial_states=['A'], final_states=['A'])
sage: it = inverter.iter_process(input_tape=[0, 1, 1])
sage: for _ in it:
....:     pass
sage: it.result()
```

```
[Branch(accept=True, state='A', output=[1, 0, 0])]
sage: it.accept_input
doctest:...: DeprecationWarning: This attribute will be removed
in future releases. Use result() at the end of our iteration
or the output of next().
See http://trac.sagemath.org/16538 for details.
True
```

FSMProcessIterator.**current_state**

The current/reached state in the process.

Warning: This attribute is deprecated and should not be used any longer (it may return a wrong result for non-deterministic finite state machines).

TESTS:

```
sage: inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:     initial_states=['A'], final_states=['A'])
sage: it = inverter.iter_process(input_tape=[0, 1, 1])
sage: for current in it:
....:     s = it.current_state
....:     print current
....:     print 'current state:', s
doctest:...: DeprecationWarning: This attribute will be
removed in future releases. Use result() at the end of our
iteration or the output of next().
See http://trac.sagemath.org/16538 for details.
process (1 branch)
+ at state 'A'
+-- tape at 1, [[1]]
current state: 'A'
process (1 branch)
+ at state 'A'
+-- tape at 2, [[1, 0]]
current state: 'A'
process (1 branch)
+ at state 'A'
+-- tape at 3, [[1, 0, 0]]
current state: 'A'
process (0 branches)
current state: None
```

FSMProcessIterator.**next**()

Makes one step in processing the input tape.

INPUT:

Nothing.

OUTPUT:

It returns the current status of the iterator (see below). A `StopIteration` exception is thrown when there is/was nothing to do (i.e. all branches ended with previous call of `next()`).

The current status is a dictionary (encapsulated into an instance of `Current`). The keys are positions on the tape. The value corresponding to such a position is again a dictionary, where each entry represents a branch of the process. This dictionary maps the current state of a branch to a pair consisting of a tape cache and a list of output words, which were written during reaching this current state.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMProcessIterator
sage: inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:     initial_states=['A'], final_states=['A'])
sage: it = FSMProcessIterator(inverter, input_tape=[0, 1])
sage: next(it)
process (1 branch)
+ at state 'A'
+-- tape at 1, [[1]]
sage: next(it)
process (1 branch)
+ at state 'A'
+-- tape at 2, [[1, 0]]
sage: next(it)
process (0 branches)
sage: next(it)
Traceback (most recent call last):
...
StopIteration
```

See also:

FiniteStateMachine.process(), Automaton.process(), Transducer.process(), FiniteStateMachine.iter_process(), FiniteStateMachine.__call__(), FSMProcessIterator.

TESTS:

```
sage: Z = Transducer()
sage: s = Z.add_state(0)
sage: s.is_initial = True
sage: s.is_final = True
sage: s.final_word_out = [1, 2]
sage: Z.process([])
(True, 0, [1, 2])
sage: it = FSMProcessIterator(Z, input_tape=[])
sage: next(it)
process (0 branches)
sage: next(it)
Traceback (most recent call last):
...
StopIteration

sage: N = Transducer([(0, 0, 0, 1)], initial_states=[0])
sage: def h_old(state, process):
....:     print state, process
sage: N.state(0).hook = h_old
sage: N.process([0, 0])
doctest:...: DeprecationWarning: The hook of state 0 cannot
be processed: It seems that you are using an old-style hook,
which is deprecated.
See http://trac.sagemath.org/16538 for details.
(False, 0, [1, 1])
sage: def h_new(process, state, outputs):
....:     print state, outputs
sage: N.state(0).hook = h_new
sage: N.process([0, 0], check_epsilon_transitions=False)
0 [[]]
0 [[1]]
```

```
0 [[1, 1]]
(False, 0, [1, 1])
```

FSMProcessIterator.**output_tape**

The written output.

Warning: This attribute is deprecated and should not be used any longer (it may return a wrong result for non-deterministic finite state machines).

TESTS:

```
sage: inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:     initial_states=['A'], final_states=['A'])
sage: it = inverter.iter_process(input_tape=[0, 1, 1])
sage: for current in it:
....:     t = it.output_tape
....:     print current
....:     print 'output:', t
doctest:...: DeprecationWarning: This attribute will be removed
in future releases. Use result() at the end of our iteration
or the output of next().
See http://trac.sagemath.org/16538 for details.
process (1 branch)
+ at state 'A'
+-- tape at 1, [[1]]
output: [1]
process (1 branch)
+ at state 'A'
+-- tape at 2, [[1, 0]]
output: [1, 0]
process (1 branch)
+ at state 'A'
+-- tape at 3, [[1, 0, 0]]
output: [1, 0, 0]
process (0 branches)
output: None
```

FSMProcessIterator.**preview_word**(*track_number=None*, *length=1*, *return_word=False*)

Reads a word from the input tape.

INPUT:

- `track_number` – an integer or `None`. If `None`, then a tuple of words (one from each track) is returned.
- `length` – (default: `1`) the length of the word(s).
- `return_word` – (default: `False`) a boolean. If set, then a word is returned, otherwise a single letter (in which case `length` has to be `1`).

OUTPUT:

A single letter or a word.

An exception `StopIteration` is thrown if the tape (at least one track) has reached its end.

Typically, this method is called from a hook-function of a state.

EXAMPLES:

```
sage: inverter = Transducer({'A': [('A', 0, 'one'),
....:                              ('A', 1, 'zero')]},
....:     initial_states=['A'], final_states=['A'])
sage: def state_hook(process, state, output):
....:     print "We are now in state %s." % (state.label(),)
....:     print "Next on the tape is a %s." % (
....:         process.preview_word(),)
sage: inverter.state('A').hook = state_hook
sage: it = inverter.iter_process(
....:     input_tape=[0, 1, 1],
....:     check_epsilon_transitions=False)
sage: for _ in it:
....:     pass
We are now in state A.
Next on the tape is a 0.
We are now in state A.
Next on the tape is a 1.
We are now in state A.
Next on the tape is a 1.
We are now in state A.
sage: it.result()
[Branch(accept=True, state='A', output=['one', 'zero', 'zero'])]
```

FSMProcessIterator.**result**(*format_output=None*)

Returns the already finished branches during process.

INPUT:

- format_output – a function converting the output from list form to something more readable (default: output the list directly).

OUTPUT:

A list of triples (accepted, state, output).

See also the parameter format_output of FSMProcessIterator.

EXAMPLES:

```
sage: inverter = Transducer({'A': [('A', 0, 'one'), ('A', 1, 'zero')]},
....:     initial_states=['A'], final_states=['A'])
sage: it = inverter.iter_process(input_tape=[0, 1, 1])
sage: for _ in it:
....:     pass
sage: it.result()
[Branch(accept=True, state='A', output=['one', 'zero', 'zero'])]
sage: it.result(lambda L: ', '.join(L))
[(True, 'A', 'one, zero, zero')]
```

Using both the parameter format_output of FSMProcessIterator and the parameter format_output of result() leads to concatenation of the two functions:

```
sage: it = inverter.iter_process(input_tape=[0, 1, 1],
....:                            format_output=lambda L: ', '.join(L))
sage: for _ in it:
....:     pass
sage: it.result()
[Branch(accept=True, state='A', output='one, zero, zero')]
sage: it.result(lambda L: ', '.join(L))
[(True, 'A', 'o, n, e, ,  , z, e, r, o, ,  , z, e, r, o')]
```

class sage.combinat.finite_state_machine.**FSMState**(*label*, *word_out=None*, *is_initial=False*, *is_final=False*, *final_word_out=None*, *initial_probability=None*, *hook=None*, *color=None*, *allow_label_None=False*)

Bases: sage.structure.sage_object.SageObject

Class for a state of a finite state machine.

INPUT:

- label – the label of the state.
- word_out – (default: None) a word that is written when the state is reached.
- is_initial – (default: False)
- is_final – (default: False)
- final_word_out – (default: None) a word that is written when the state is reached as the last state of some input; only for final states.
- initial_probability – (default: None) The probability of starting in this state if it is a state of a Markov chain.
- hook – (default: None) A function which is called when the state is reached during processing input. It takes two input parameters: the first is the current state (to allow using the same hook for several states), the second is the current process iterator object (to have full access to everything; e.g. the next letter from the input tape can be read in). It can output the next transition, i.e. the transition to take next. If it returns None the process iterator chooses. Moreover, this function can raise a StopIteration exception to stop processing of a finite state machine the input immediately. See also the example below.
- color – (default: None) In order to distinguish states, they can be given an arbitrary "color" (an arbitrary object). This is used in FiniteStateMachine.equivalence_classes(): states of different colors are never considered to be equivalent. Note that Automaton.determinisation() requires that color is hashable.
- allow_label_None – (default: False) If True allows also None as label. Note that a state with label None is used in FSMProcessIterator.

OUTPUT:

Returns a state of a finite state machine.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('state 1', word_out=0, is_initial=True)
sage: A
'state 1'
sage: A.label()
'state 1'
sage: B = FSMState('state 2')
sage: A == B
False
```

We can also define a final output word of a final state which is used if the input of a transducer leads to this state. Such final output words are used in subsequential transducers.

```
sage: C = FSMState('state 3', is_final=True, final_word_out='end')
sage: C.final_word_out
['end']
```

The final output word can be a single letter, None or a list of letters:

```
sage: A = FSMState('A')
sage: A.is_final = True
sage: A.final_word_out = 2
sage: A.final_word_out
[2]
sage: A.final_word_out = [2, 3]
sage: A.final_word_out
[2, 3]
```

Only final states can have a final output word which is not None:

```
sage: B = FSMState('B')
sage: B.final_word_out is None
True
sage: B.final_word_out = 2
Traceback (most recent call last):
...
ValueError: Only final states can have a final output word,
but state B is not final.
```

Setting the final_word_out of a final state to None is the same as setting it to [] and is also the default for a final state:

```
sage: C = FSMState('C', is_final=True)
sage: C.final_word_out
[]
sage: C.final_word_out = None
sage: C.final_word_out
[]
sage: C.final_word_out = []
sage: C.final_word_out
[]
```

It is not allowed to use None as a label:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: FSMState(None)
Traceback (most recent call last):
...
ValueError: Label None reserved for a special state,
choose another label.
```

This can be overridden by:

```
sage: FSMState(None, allow_label_None=True)
None
```

Note that Automaton.determinisation() requires that color is hashable:

```
sage: A = Automaton([[0, 0, 0]], initial_states=[0])
sage: A.state(0).color = []
sage: A.determinisation()
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
sage: A.state(0).color = ()
sage: A.determinisation()
Automaton with 1 state
```

We can use a hook function of a state to stop processing. This is done by raising a `StopIteration` exception. The following code demonstrates this:

```
sage: T = Transducer([(0, 1, 9, 'a'), (1, 2, 9, 'b'),
....:                 (2, 3, 9, 'c'), (3, 4, 9, 'd')],
....:                initial_states=[0],
....:                final_states=[4],
....:                input_alphabet=[9])
sage: def stop(process, state, output):
....:     raise StopIteration()
sage: T.state(3).hook = stop
sage: T.process([9, 9, 9, 9])
(False, 3, ['a', 'b', 'c'])
```

copy()

Returns a (shallow) copy of the state.

INPUT:

Nothing.

OUTPUT:

A new state.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A')
sage: A.is_initial = True
sage: A.is_final = True
sage: A.final_word_out = [1]
sage: A.color = 'green'
sage: A.initial_probability = 1/2
sage: B = copy(A)
sage: B.fully_equal(A)
True
sage: A.label() is B.label()
True
sage: A.is_initial is B.is_initial
True
sage: A.is_final is B.is_final
True
sage: A.final_word_out is B.final_word_out
True
sage: A.color is B.color
True
sage: A.initial_probability is B.initial_probability
True
```

deepcopy(*memo=None*)

Returns a deep copy of the state.

INPUT:

- `memo` – (default: `None`) a dictionary storing already processed elements.

OUTPUT:

A new state.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState((1, 3), color=[1, 2],
....:              is_final=True, final_word_out=3,
....:              initial_probability=1/3)
sage: B = deepcopy(A)
sage: B
(1, 3)
sage: B.label == A.label
True
sage: B.label is A.label
False
sage: B.color == A.color
True
sage: B.color is A.color
False
sage: B.is_final == A.is_final
True
sage: B.is_final is A.is_final
True
sage: B.final_word_out == A.final_word_out
True
sage: B.final_word_out is A.final_word_out
False
sage: B.initial_probability == A.initial_probability
True
sage: B.initial_probability is A.initial_probability
False
```

final_word_out

The final output word of a final state which is written if the state is reached as the last state of the input of the finite state machine. For a non-final state, the value is `None`.

`final_word_out` can be a single letter, a list or `None`, but for a final-state, it is always saved as a list.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A', is_final=True, final_word_out=2)
sage: A.final_word_out
[2]
sage: A.final_word_out = 3
sage: A.final_word_out
[3]
sage: A.final_word_out = [3, 4]
sage: A.final_word_out
[3, 4]
sage: A.final_word_out = None
sage: A.final_word_out
[]
sage: B = FSMState('B')
sage: B.final_word_out is None
True
```

A non-final state cannot have a final output word:

```
sage: B.final_word_out = [3, 4]
Traceback (most recent call last):
...
ValueError: Only final states can have a final
output word, but state B is not final.
```

fully_equal(*left*, *right*, *compare_color=True*)

Checks whether two states are fully equal, i.e., including all attributes except `hook`.

INPUT:

- `left` – a state.
- `right` – a state.
- `compare_color` – If `True` (default) colors are compared as well, otherwise not.

OUTPUT:

`True` or `False`.

Note that usual comparison by == does only compare the labels.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A')
sage: B = FSMState('A', is_initial=True)
sage: A.fully_equal(B)
False
sage: A == B
True
sage: A.is_initial = True; A.color = 'green'
sage: A.fully_equal(B)
False
sage: A.fully_equal(B, compare_color=False)
True
```

initial_probability

The probability of starting in this state if it is part of a Markov chain.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: S = FSMState('state', initial_probability=1/3)
sage: S.initial_probability
1/3
```

is_final

Describes whether the state is final or not.

`True` if the state is final and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A', is_final=True, final_word_out=3)
sage: A.is_final
True
sage: A.is_final = False
Traceback (most recent call last):
...
ValueError: State A cannot be non-final, because it has a
final output word. Only final states can have a final output
word.
sage: A.final_word_out = None
sage: A.is_final = False
```

```
sage: A.is_final
False
```

is_initial

Describes whether the state is initial.

EXAMPLES:

```
sage: T = Automaton([(0,0,0)])
sage: T.initial_states()
[]
sage: T.state(0).is_initial = True
sage: T.initial_states()
[0]
```

label()

Returns the label of the state.

INPUT:

Nothing.

OUTPUT:

The label of the state.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('state')
sage: A.label()
'state'
```

relabeled(*label*, *memo=None*)

Returns a deep copy of the state with a new label.

INPUT:

- `label` – the label of new state.
- `memo` – (default: `None`) a dictionary storing already processed elements.

OUTPUT:

A new state.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A')
sage: A.relabeled('B')
'B'
```

class sage.combinat.finite_state_machine.**FSMTransition**(*from_state*, *to_state*, *word_in=None*, *word_out=None*, *hook=None*)

Bases: `sage.structure.sage_object.SageObject`

Class for a transition of a finite state machine.

INPUT:

- `from_state` – state from which transition starts.
- `to_state` – state in which transition ends.

- word_in – the input word of the transitions (when the finite state machine is used as automaton)
- word_out – the output word of the transitions (when the finite state machine is used as transducer)

OUTPUT:

A transition of a finite state machine.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState, FSMTransition
sage: A = FSMState('A')
sage: B = FSMState('B')
sage: S = FSMTransition(A, B, 0, 1)
sage: T = FSMTransition('A', 'B', 0, 1)
sage: T == S
True
sage: U = FSMTransition('A', 'B', 0)
sage: U == T
False
```

copy()

Returns a (shallow) copy of the transition.

INPUT:

Nothing.

OUTPUT:

A new transition.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: t = FSMTransition('A', 'B', 0)
sage: copy(t)
Transition from 'A' to 'B': 0|-
```

deepcopy(*memo=None*)

Returns a deep copy of the transition.

INPUT:

- memo – (default: None) a dictionary storing already processed elements.

OUTPUT:

A new transition.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: t = FSMTransition('A', 'B', 0)
sage: deepcopy(t)
Transition from 'A' to 'B': 0|-
```

from_state

State from which the transition starts. Read-only.

to_state

State in which the transition ends. Read-only.

word_in

Input word of the transition. Read-only.

word_out
Output word of the transition. Read-only.

sage.combinat.finite_state_machine.**FSMWordSymbol**(*word*)
Returns a string of word. It may returns the symbol of the empty word FSMEmptyWordSymbol.

INPUT:

- word – the input word.

OUTPUT:

A string of word.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMWordSymbol
sage: FSMWordSymbol([0, 1, 1])
'0,1,1'
```

class sage.combinat.finite_state_machine.**FiniteStateMachine**(*data=None*, *initial_states=None*, *final_states=None*, *input_alphabet=None*, *output_alphabet=None*, *determine_alphabets=None*, *with_final_word_out=None*, *store_states_dict=True*, *on_duplicate_transition=None*)
Bases: sage.structure.sage_object.SageObject

Class for a finite state machine.

A finite state machine is a finite set of states connected by transitions.

INPUT:

- data – can be any of the following:
  1. a dictionary of dictionaries (of transitions),
  2. a dictionary of lists (of states or transitions),
  3. a list (of transitions),
  4. a function (transition function),
  5. an other instance of a finite state machine.
- initial_states and final_states – the initial and final states of this machine
- input_alphabet and output_alphabet – the input and output alphabets of this machine
- determine_alphabets – If True, then the function determine_alphabets() is called after data was read and processed, if False, then not. If it is None, then it is decided during the construction of the finite state machine whether determine_alphabets() should be called.
- with_final_word_out – If given (not None), then the function with_final_word_out() (more precisely, its inplace pendant construct_final_word_out()) is called with input letters=with_final_word_out at the end of the creation process.
- store_states_dict – If True, then additionally the states are stored in an interal dictionary for speed up.

- on_duplicate_transition – A function which is called when a transition is inserted into self which already existed (same from_state, same to_state, same word_in, same word_out).

  This function is assumed to take two arguments, the first being the already existing transition, the second being the new transition (as an FSMTransition). The function must return the (possibly modified) original transition.

  By default, we have on_duplicate_transition=None, which is interpreted as on_duplicate_transition=duplicate_transition_ignore, where duplicate_transition_ignore is a predefined function ignoring the occurrence. Other such predefined functions are duplicate_transition_raise_error and duplicate_transition_add_input.

OUTPUT:

A finite state machine.

The object creation of Automaton and Transducer is the same as the one described here (i.e. just replace the word FiniteStateMachine by Automaton or Transducer).

Each transition of an automaton has an input label. Automata can, for example, be determinised (see Automaton.determinisation()) and minimized (see Automaton.minimization()). Each transition of a transducer has an input and an output label. Transducers can, for example, be simplified (see Transducer.simplification()).

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState, FSMTransition
```

See documentation for more examples.

We illustrate the different input formats:

1. The input-data can be a dictionary of dictionaries, where

   - the keys of the outer dictionary are state-labels (from-states of transitions),
   - the keys of the inner dictionaries are state-labels (to-states of transitions),
   - the values of the inner dictionaries specify the transition more precisely.

   The easiest is to use a tuple consisting of an input and an output word:

   ```
   sage: FiniteStateMachine({'a':{'b':(0, 1), 'c':(1, 1)}})
   Finite state machine with 3 states
   ```

   Instead of the tuple anything iterable (e.g. a list) can be used as well.

   If you want to use the arguments of FSMTransition directly, you can use a dictionary:

   ```
   sage: FiniteStateMachine({'a':{'b':{'word_in':0, 'word_out':1},
   ....:                          'c':{'word_in':1, 'word_out':1}}})
   Finite state machine with 3 states
   ```

   In the case you already have instances of FSMTransition, it is possible to use them directly:

   ```
   sage: FiniteStateMachine({'a':{'b':FSMTransition('a', 'b', 0, 1),
   ....:                          'c':FSMTransition('a', 'c', 1, 1)}})
   Finite state machine with 3 states
   ```

2.The input-data can be a dictionary of lists, where the keys are states or label of states.

The list-elements can be states:

```
sage: a = FSMState('a')
sage: b = FSMState('b')
sage: c = FSMState('c')
sage: FiniteStateMachine({a:[b, c]})
Finite state machine with 3 states
```

Or the list-elements can simply be labels of states:

```
sage: FiniteStateMachine({'a':['b', 'c']})
Finite state machine with 3 states
```

The list-elements can also be transitions:

```
sage: FiniteStateMachine({'a':[FSMTransition('a', 'b', 0, 1),
....:                          FSMTransition('a', 'c', 1, 1)]})
Finite state machine with 3 states
```

Or they can be tuples of a label, an input word and an output word specifying a transition:

```
sage: FiniteStateMachine({'a':[('b', 0, 1), ('c', 1, 1)]})
Finite state machine with 3 states
```

3.The input-data can be a list, where its elements specify transitions:

```
sage: FiniteStateMachine([FSMTransition('a', 'b', 0, 1),
....:                     FSMTransition('a', 'c', 1, 1)])
Finite state machine with 3 states
```

It is possible to skip `FSMTransition` in the example above:

```
sage: FiniteStateMachine([('a', 'b', 0, 1), ('a', 'c', 1, 1)])
Finite state machine with 3 states
```

The parameters of the transition are given in tuples. Anyhow, anything iterable (e.g. a list) is possible.

You can also name the parameters of the transition. For this purpose you take a dictionary:

```
sage: FiniteStateMachine([{'from_state':'a', 'to_state':'b',
....:                      'word_in':0, 'word_out':1},
....:                     {'from_state':'a', 'to_state':'c',
....:                      'word_in':1, 'word_out':1}])
Finite state machine with 3 states
```

Other arguments, which `FSMTransition` accepts, can be added, too.

4.The input-data can also be function acting as transition function:

This function has two input arguments:

(a)a label of a state (from which the transition starts),

(b)a letter of the (input-)alphabet (as input-label of the transition).

It returns a tuple with the following entries:

(a)a label of a state (to which state the transition goes),

(b)a letter of or a word over the (output-)alphabet (as output-label of the transition).

It may also output a list of such tuples if several transitions from the from-state and the input letter exist (this means that the finite state machine is non-deterministic).

If the transition does not exist, the function should raise a `LookupError` or return an empty list.

When constructing a finite state machine in this way, some inital states and an input alphabet have to be specified.

```
sage: def f(state_from, read):
....:     if int(state_from) + read <= 2:
....:         state_to = 2*int(state_from)+read
....:         write = 0
....:     else:
....:         state_to = 2*int(state_from) + read - 5
....:         write = 1
....:     return (str(state_to), write)
sage: F = FiniteStateMachine(f, input_alphabet=[0, 1],
....:                        initial_states=['0'],
....:                        final_states=['0'])
sage: F([1, 0, 1])
(True, '0', [0, 0, 1])
```

5.The input-data can be an other instance of a finite state machine:

```
sage: F = FiniteStateMachine()
sage: G = Transducer(F)
sage: G == F
True
```

The other parameters cannot be specified in that case. If you want to change these, use the attributes `FSMState.is_initial`, `FSMState.is_final`, `input_alphabet`, `output_alphabet`, `on_duplicate_transition` and methods `determine_alphabets()`, `construct_final_word_out()` on the new machine, respectively.

The following examples demonstrate the use of `on_duplicate_transition`:

```
sage: F = FiniteStateMachine([['a', 'a', 1/2], ['a', 'a', 1/2]])
sage: F.transitions()
[Transition from 'a' to 'a': 1/2|-]

sage: from sage.combinat.finite_state_machine import duplicate_transition_raise_error
sage: F1 = FiniteStateMachine([['a', 'a', 1/2], ['a', 'a', 1/2]],
....:                         on_duplicate_transition=duplicate_transition_raise_error)
Traceback (most recent call last):
...
ValueError: Attempting to re-insert transition Transition from 'a' to 'a': 1/2|-
```

Use `duplicate_transition_add_input` to emulate a Markov chain, the input labels are considered as transition probabilities:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_add_input
sage: F = FiniteStateMachine([['a', 'a', 1/2], ['a', 'a', 1/2]],
....:                        on_duplicate_transition=duplicate_transition_add_input)
sage: F.transitions()
[Transition from 'a' to 'a': 1|-]
```

Use `with_final_word_out` to construct final output:

```
sage: T = Transducer([(0, 1, 0, 0), (1, 0, 0, 0)],
....:                initial_states=[0],
....:                final_states=[0],
....:                with_final_word_out=0)
sage: for s in T.iter_final_states():
....:     print s, s.final_word_out
0 []
1 [0]
```

TESTS:

```
sage: a = FSMState('S_a', 'a')
sage: b = FSMState('S_b', 'b')
sage: c = FSMState('S_c', 'c')
sage: d = FSMState('S_d', 'd')
sage: FiniteStateMachine({a:[b, c], b:[b, c, d],
....:                     c:[a, b], d:[a, c]})
Finite state machine with 4 states
```

We have several constructions which lead to the same finite state machine:

```
sage: A = FSMState('A')
sage: B = FSMState('B')
sage: C = FSMState('C')
sage: FSM1 = FiniteStateMachine(
....:  {A:{B:{'word_in':0, 'word_out':1},
....:   C:{'word_in':1, 'word_out':1}}})
sage: FSM2 = FiniteStateMachine({A:{B:(0, 1), C:(1, 1)}})
sage: FSM3 = FiniteStateMachine(
....:  {A:{B:FSMTransition(A, B, 0, 1),
....:      C:FSMTransition(A, C, 1, 1)}})
sage: FSM4 = FiniteStateMachine({A:[(B, 0, 1), (C, 1, 1)]})
sage: FSM5 = FiniteStateMachine(
....:  {A:[FSMTransition(A, B, 0, 1), FSMTransition(A, C, 1, 1)]})
sage: FSM6 = FiniteStateMachine(
....:  [{'from_state':A, 'to_state':B, 'word_in':0, 'word_out':1},
....:   {'from_state':A, 'to_state':C, 'word_in':1, 'word_out':1}])
sage: FSM7 = FiniteStateMachine([(A, B, 0, 1), (A, C, 1, 1)])
sage: FSM8 = FiniteStateMachine(
....:  [FSMTransition(A, B, 0, 1), FSMTransition(A, C, 1, 1)])

sage: FSM1 == FSM2 == FSM3 == FSM4 == FSM5 == FSM6 == FSM7 == FSM8
True
```

It is possible to skip `FSMTransition` in the example above.

Some more tests for different input-data:

```
sage: FiniteStateMachine({'a':{'a':[0, 0], 'b':[1, 1]},
....:                     'b':{'b':[1, 0]}})
Finite state machine with 2 states
```

```
sage: a = FSMState('S_a', 'a')
sage: b = FSMState('S_b', 'b')
sage: c = FSMState('S_c', 'c')
sage: d = FSMState('S_d', 'd')
sage: t1 = FSMTransition(a, b)
sage: t2 = FSMTransition(b, c)
sage: t3 = FSMTransition(b, d)
sage: t4 = FSMTransition(c, d)
sage: FiniteStateMachine([t1, t2, t3, t4])
Finite state machine with 4 states
```

We test that no input parameter is allowed when creating a finite state machine from an existing instance:

```
sage: F = FiniteStateMachine()
sage: FiniteStateMachine(F, initial_states=[1])
Traceback (most recent call last):
...
ValueError: initial_states cannot be specified when
copying another finite state machine.
sage: FiniteStateMachine(F, final_states=[1])
Traceback (most recent call last):
...
ValueError: final_states cannot be specified when
copying another finite state machine.
sage: FiniteStateMachine(F, input_alphabet=[1])
Traceback (most recent call last):
...
ValueError: input_alphabet cannot be specified when
copying another finite state machine.
sage: FiniteStateMachine(F, output_alphabet=[1])
Traceback (most recent call last):
...
ValueError: output_alphabet cannot be specified when
copying another finite state machine.
sage: from sage.combinat.finite_state_machine import (
....:     duplicate_transition_add_input)
sage: FiniteStateMachine(F,
....:     on_duplicate_transition=duplicate_transition_add_input)
Traceback (most recent call last):
...
ValueError: on_duplicate_transition cannot be specified when
copying another finite state machine.
sage: FiniteStateMachine(F, determine_alphabets=False)
Traceback (most recent call last):
...
ValueError: determine_alphabets cannot be specified when
copying another finite state machine.
sage: FiniteStateMachine(F, with_final_word_out=[1])
Traceback (most recent call last):
...
ValueError: with_final_word_out cannot be specified when
copying another finite state machine.
```

trac ticket #19454 rewrote automatic detection of the alphabets:

```
sage: def transition_function(state, letter):
....:     return (0, 3 + letter)
sage: T1 = Transducer(transition_function,
....:     input_alphabet=[0, 1],
```

```
....:     initial_states=[0],
....:     final_states=[0])
sage: T1.output_alphabet
[3, 4]
sage: T2 = Transducer([(0, 0, 0, 3), (0, 0, 0, 4)],
....:     initial_states=[0],
....:     final_states=[0])
sage: T2.output_alphabet
[3, 4]
sage: T = Transducer([(0, 0, 1, 2)])
sage: (T.input_alphabet, T.output_alphabet)
([1], [2])
sage: T = Transducer([(0, 0, 1, 2)], determine_alphabets=False)
sage: (T.input_alphabet, T.output_alphabet)
(None, None)
sage: T = Transducer([(0, 0, 1, 2)], input_alphabet=[0, 1])
sage: (T.input_alphabet, T.output_alphabet)
([0, 1], [2])
sage: T = Transducer([(0, 0, 1, 2)], output_alphabet=[2, 3])
sage: (T.input_alphabet, T.output_alphabet)
([1], [2, 3])
sage: T = Transducer([(0, 0, 1, 2)], input_alphabet=[0, 1],
....:     output_alphabet=[2, 3])
sage: (T.input_alphabet, T.output_alphabet)
([0, 1], [2, 3])
```

__call__(**args*, ***kwargs*)

Call either method `composition()` or `process()` (with `full_output=False`). If the input is not finite (`is_finite` of input is `False`), then `iter_process()` (with `iterator_type='simple'`) is called. Moreover, the flag `automatic_output_type` is set (unless `format_output` is specified). See the documentation of these functions for possible parameters.

EXAMPLES:

The following code performs a `composition()`:

```
sage: F = Transducer([('A', 'B', 1, 0), ('B', 'B', 1, 1),
....:                 ('B', 'B', 0, 0)],
....:                initial_states=['A'], final_states=['B'])
sage: G = Transducer([(1, 1, 0, 0), (1, 2, 1, 0),
....:                 (2, 2, 0, 1), (2, 1, 1, 1)],
....:                initial_states=[1], final_states=[1])
sage: H = G(F)
sage: H.states()
[('A', 1), ('B', 1), ('B', 2)]
```

An automaton or transducer can also act on an input (an list or other iterable of letters):

```
sage: binary_inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:                              initial_states=['A'], final_states=['A'])
sage: binary_inverter([0, 1, 0, 0, 1, 1])
[1, 0, 1, 1, 0, 0]
```

We can also let them act on *words*:

```
sage: W = Words([0, 1]); W
Finite and infinite words over {0, 1}
sage: binary_inverter(W([0, 1, 1, 0, 1, 1]))
word: 100100
```

Infinite words work as well:

```
sage: words.FibonacciWord()
word: 0100101001001010010100100101001001010010...
sage: binary_inverter(words.FibonacciWord())
word: 1011010110110101101011011010110110101101...
```

When only one successful path is found in a non-deterministic transducer, the result of that path is returned.

```
sage: T = Transducer([(0, 1, 0, 1), (0, 2, 0, 2)],
....:                initial_states=[0], final_states=[1])
sage: T.process([0])
[(True, 1, [1]), (False, 2, [2])]
sage: T([0])
[1]
```

See also:

composition(), process(), iter_process(), Automaton.process(), Transducer.process().

TESTS:

```
sage: F = FiniteStateMachine([(0, 1, 1, 'a'), (0, 2, 2, 'b')],
....:                        initial_states=[0],
....:                        final_states=[1])
sage: A = Automaton([(0, 1, 1), (0, 2, 2)],
....:               initial_states=[0],
....:               final_states=[1])
sage: T = Transducer([(0, 1, 1, 'a'), (0, 2, 2, 'b')],
....:                initial_states=[0],
....:                final_states=[1])
sage: F([1])
(True, 1, ['a'])
sage: A([1])
True
sage: T([1])
['a']
sage: F([2])
(False, 2, ['b'])
sage: A([2])
False
sage: T([2])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
sage: F([3])
(False, None, None)
sage: A([3])
False
sage: T([3])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.

sage: F = FiniteStateMachine([(11, 11, 1, 'a'), (11, 12, 2, 'b'),
....:                         (11, 13, 3, 'c'), (11, 14, 4, 'd'),
....:                         (12, 13, 3, 'e'), (12, 13, 3, 'f'),
....:                         (12, 14, 4, 'g'), (12, 14, 4, 'h'),
....:                         (12, 13, 2, 'i'), (12, 14, 2, 'j')],
....:                        initial_states=[11],
```

```
....:                          final_states=[13])
sage: def f(o):
....:     return ''.join(o)
sage: F([0], format_output=f)
(False, None, None)
sage: F([3], format_output=f)
(True, 13, 'c')
sage: F([4], format_output=f)
(False, 14, 'd')
sage: F([2, 2], format_output=f)
Traceback (most recent call last):
...
ValueError: Got more than one output, but only allowed to show
one. Change list_of_outputs option.
sage: F([2, 2], format_output=f, list_of_outputs=True)
[(True, 13, 'bi'), (False, 14, 'bj')]
sage: F([2, 3], format_output=f)
Traceback (most recent call last):
...
ValueError: Got more than one output, but only allowed to show
one. Change list_of_outputs option.
sage: F([2, 3], format_output=f, list_of_outputs=True)
[(True, 13, 'be'), (True, 13, 'bf')]
sage: F([2, 4], format_output=f)
Traceback (most recent call last):
...
ValueError: Got more than one output, but only allowed to show
one. Change list_of_outputs option.
sage: F([2, 4], format_output=f, list_of_outputs=True)
[(False, 14, 'bg'), (False, 14, 'bh')]

sage: A = Automaton([(11, 11, 1), (11, 12, 2),
....:                (11, 13, 3), (11, 14, 4),
....:                (12, 13, 3), (12, 14, 4),
....:                (12, 32, 3), (12, 42, 4),
....:                (12, 13, 2), (12, 14, 2)],
....:               initial_states=[11],
....:               final_states=[13, 32])
sage: def f(o):
....:     return ''.join(o)
sage: A([0], format_output=f)
False
sage: A([3], format_output=f)
True
sage: A([4], format_output=f)
False
sage: A([2, 2], format_output=f)
True
sage: A([2, 2], format_output=f, list_of_outputs=True)
[True, False]
sage: A([2, 3], format_output=f)
True
sage: A([2, 3], format_output=f, list_of_outputs=True)
[True, True]
sage: A([2, 4], format_output=f)
False
sage: A([2, 4], format_output=f, list_of_outputs=True)
[False, False]
```

```
sage: T = Transducer([(11, 11, 1, 'a'), (11, 12, 2, 'b'),
....:                 (11, 13, 3, 'c'), (11, 14, 4, 'd'),
....:                 (12, 13, 3, 'e'), (12, 13, 3, 'f'),
....:                 (12, 14, 4, 'g'), (12, 14, 4, 'h'),
....:                 (12, 13, 2, 'i'), (12, 14, 2, 'j')],
....:                initial_states=[11],
....:                final_states=[13])
sage: def f(o):
....:     return ''.join(o)
sage: T([0], format_output=f)
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
sage: T([3], format_output=f)
'c'
sage: T([4], format_output=f)
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
sage: T([2, 2], format_output=f)
'bi'
sage: T([2, 2], format_output=f, list_of_outputs=True)
['bi', None]
sage: T([2, 2], format_output=f,
....:   list_of_outputs=True, only_accepted=True)
['bi']
sage: T.process([2, 2], format_output=f, list_of_outputs=True)
[(True, 13, 'bi'), (False, 14, 'bj')]
sage: T([2, 3], format_output=f)
Traceback (most recent call last):
...
ValueError: Found more than one accepting path.
sage: T([2, 3], format_output=f, list_of_outputs=True)
['be', 'bf']
sage: T([2, 4], format_output=f)
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
sage: T([2, 4], format_output=f, list_of_outputs=True)
[None, None]

sage: from itertools import islice
sage: inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:     initial_states=['A'], final_states=['A'])
sage: inverter(words.FibonacciWord())
word: 1011010110110101101101101101101101101101101101...
sage: inverter(words.FibonacciWord(), automatic_output_type=True)
word: 1011010110110101101101101101101101101101101101...
sage: tuple(islice(inverter(words.FibonacciWord(),
....:                       automatic_output_type=False), 10))
(1, 0, 1, 1, 0, 1, 0, 1, 1, 0)
sage: type(inverter((1, 0, 1, 1, 0, 1, 0, 1, 1, 0),
....:               automatic_output_type=False))
<type 'list'>
sage: type(inverter((1, 0, 1, 1, 0, 1, 0, 1, 1, 0),
....:               automatic_output_type=True))
<type 'tuple'>
```

accessible_components()

Return a new finite state machine with the accessible states of self and all transitions between those states.

INPUT:

Nothing.

OUTPUT:

A finite state machine with the accessible states of self and all transitions between those states.

A state is accessible if there is a directed path from an initial state to the state. If self has no initial states then a copy of the finite state machine self is returned.

EXAMPLES:

```
sage: F = Automaton([(0, 0, 0), (0, 1, 1), (1, 1, 0), (1, 0, 1)],
....:               initial_states=[0])
sage: F.accessible_components()
Automaton with 2 states

sage: F = Automaton([(0, 0, 1), (0, 0, 1), (1, 1, 0), (1, 0, 1)],
....:               initial_states=[0])
sage: F.accessible_components()
Automaton with 1 state
```

See also:

coaccessible_components()

TESTS:

Check whether input of length > 1 works:

```
sage: F = Automaton([(0, 1, [0, 1]), (0, 2, 0)],
....:               initial_states=[0])
sage: F.accessible_components()
Automaton with 3 states
```

add_from_transition_function(*function*, *initial_states=None*, *explore_existing_states=True*)

Constructs a finite state machine from a transition function.

INPUT:

- function may return a tuple (new_state, output_word) or a list of such tuples.
- initial_states – If no initial states are given, the already existing initial states of self are taken.
- If explore_existing_states is True (default), then already existing states in self (e.g. already given final states) will also be processed if they are reachable from the initial states.

OUTPUT:

Nothing.

EXAMPLES:

```
sage: F = FiniteStateMachine(initial_states=['A'],
....:                        input_alphabet=[0, 1])
sage: def f(state, input):
....:     return [('A', input), ('B', 1-input)]
sage: F.add_from_transition_function(f)
sage: F.transitions()
[Transition from 'A' to 'A': 0|0,
```

```
Transition from 'A' to 'B': 0|1,
Transition from 'A' to 'A': 1|1,
Transition from 'A' to 'B': 1|0,
Transition from 'B' to 'A': 0|0,
Transition from 'B' to 'B': 0|1,
Transition from 'B' to 'A': 1|1,
Transition from 'B' to 'B': 1|0]
```

Initial states can also be given as a parameter:

```
sage: F = FiniteStateMachine(input_alphabet=[0,1])
sage: def f(state, input):
....:     return [('A', input), ('B', 1-input)]
sage: F.add_from_transition_function(f,initial_states=['A'])
sage: F.initial_states()
['A']
```

Already existing states in the finite state machine (the final states in the example below) are also explored:

```
sage: F = FiniteStateMachine(initial_states=[0],
....:                        final_states=[1],
....:                        input_alphabet=[0])
sage: def transition_function(state, letter):
....:     return(1-state, [])
sage: F.add_from_transition_function(transition_function)
sage: F.transitions()
[Transition from 0 to 1: 0|-,
 Transition from 1 to 0: 0|-]
```

If `explore_existing_states=False`, however, this behavior is turned off, i.e., already existing states are not explored:

```
sage: F = FiniteStateMachine(initial_states=[0],
....:                        final_states=[1],
....:                        input_alphabet=[0])
sage: def transition_function(state, letter):
....:     return(1-state, [])
sage: F.add_from_transition_function(transition_function,
....:                                explore_existing_states=False)
sage: F.transitions()
[Transition from 0 to 1: 0|-]
```

TEST:

```
sage: F = FiniteStateMachine(initial_states=['A'])
sage: def f(state, input):
....:     return [('A', input), ('B', 1-input)]
sage: F.add_from_transition_function(f)
Traceback (most recent call last):
...
ValueError: No input alphabet is given.
Try calling determine_alphabets().

sage: def transition(state, where):
....:     return (vector([0, 0]), 1)
sage: Transducer(transition, input_alphabet=[0], initial_states=[0])
Traceback (most recent call last):
...
TypeError: mutable vectors are unhashable
```

add_state(*state*)

Adds a state to the finite state machine and returns the new state. If the state already exists, that existing state is returned.

INPUT:

- `state` is either an instance of `FSMState` or, otherwise, a label of a state.

OUTPUT:

The new or existing state.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: F = FiniteStateMachine()
sage: A = FSMState('A', is_initial=True)
sage: F.add_state(A)
'A'
```

add_states(*states*)

Adds several states. See add_state for more information.

INPUT:

- `states` – a list of states or iterator over states.

OUTPUT:

Nothing.

EXAMPLES:

```
sage: F = FiniteStateMachine()
sage: F.add_states(['A', 'B'])
sage: F.states()
['A', 'B']
```

add_transition(**args*, ***kwargs*)

Adds a transition to the finite state machine and returns the new transition.

If the transition already exists, the return value of `self.on_duplicate_transition` is returned. See the documentation of `FiniteStateMachine`.

INPUT:

The following forms are all accepted:

```
sage: from sage.combinat.finite_state_machine import FSMState, FSMTransition
sage: A = FSMState('A')
sage: B = FSMState('B')

sage: FSM = FiniteStateMachine()
sage: FSM.add_transition(FSMTransition(A, B, 0, 1))
Transition from 'A' to 'B': 0|1

sage: FSM = FiniteStateMachine()
sage: FSM.add_transition(A, B, 0, 1)
Transition from 'A' to 'B': 0|1

sage: FSM = FiniteStateMachine()
sage: FSM.add_transition(A, B, word_in=0, word_out=1)
```

```
Transition from 'A' to 'B': 0|1

sage: FSM = FiniteStateMachine()
sage: FSM.add_transition('A', 'B', {'word_in': 0, 'word_out': 1})
Transition from 'A' to 'B': {'word_in': 0, 'word_out': 1}|-

sage: FSM = FiniteStateMachine()
sage: FSM.add_transition(from_state=A, to_state=B,
....:                    word_in=0, word_out=1)
Transition from 'A' to 'B': 0|1

sage: FSM = FiniteStateMachine()
sage: FSM.add_transition({'from_state': A, 'to_state': B,
....:                    'word_in': 0, 'word_out': 1})
Transition from 'A' to 'B': 0|1

sage: FSM = FiniteStateMachine()
sage: FSM.add_transition((A, B, 0, 1))
Transition from 'A' to 'B': 0|1

sage: FSM = FiniteStateMachine()
sage: FSM.add_transition([A, B, 0, 1])
Transition from 'A' to 'B': 0|1
```

If the states `A` and `B` are not instances of `FSMState`, then it is assumed that they are labels of states.

OUTPUT:

The new transition.

add_transitions_from_function (*function*, *labels_as_input=True*)

Adds one or more transitions if `function(state, state)` says that there are some.

INPUT:

- `function` – a transition function. Given two states `from_state` and `to_state` (or their labels if `label_as_input` is true), this function shall return a tuple `(word_in, word_out)` to add a transition from `from_state` to `to_state` with input and output labels `word_in` and `word_out`, respectively. If no such addition is to be added, the transition function shall return `None`. The transition function may also return a list of such tuples in order to add multiple transitions between the pair of states.
- `label_as_input` – (default: `True`)

OUTPUT:

Nothing.

EXAMPLES:

```
sage: F = FiniteStateMachine()
sage: F.add_states(['A', 'B', 'C'])
sage: def f(state1, state2):
....:     if state1 == 'C':
....:         return None
....:     return (0, 1)
sage: F.add_transitions_from_function(f)
sage: len(F.transitions())
6
```

Multiple transitions are also possible:

```
sage: F = FiniteStateMachine()
sage: F.add_states([0, 1])
sage: def f(state1, state2):
....:     if state1 != state2:
....:         return [(0, 1), (1, 0)]
....:     else:
....:         return None
sage: F.add_transitions_from_function(f)
sage: F.transitions()
[Transition from 0 to 1: 0|1,
 Transition from 0 to 1: 1|0,
 Transition from 1 to 0: 0|1,
 Transition from 1 to 0: 1|0]
```

TESTS:

```
sage: F = FiniteStateMachine()
sage: F.add_state(0)
0
sage: def f(state1, state2):
....:     return 1
sage: F.add_transitions_from_function(f)
Traceback (most recent call last):
...
ValueError: The callback function for add_transitions_from_function
is expected to return a pair (word_in, word_out) or a list of such
pairs. For states 0 and 0 however, it returned 1,
which is not acceptable.
```

adjacency_matrix(*input=None*, *entry=None*)

Returns the adjacency matrix of the underlying graph.

INPUT:

- `input` – Only transitions with input label `input` are respected.
- `entry` – The function `entry` takes a transition and the return value is written in the matrix as the entry `(transition.from_state, transition.to_state)`. The default value (`None`) of entry takes the variable `x` to the power of the sum of the output word of the transition.

OUTPUT:

A matrix.

If any label of a state is not an integer, the finite state machine is relabeled at the beginning. If there are more than one transitions between two states, then the different return values of `entry` are added up.

EXAMPLES:

```
sage: B = FiniteStateMachine({0:{0:(0, 0), 'a':(1, 0)},
....:                         'a':{2:(0, 0), 3:(1, 0)},
....:                         2:{0:(1, 1), 4:(0, 0)},
....:                         3:{'a':(0, 1), 2:(1, 1)},
....:                         4:{4:(1, 1), 3:(0, 1)}},
....:                        initial_states=[0])
sage: B.adjacency_matrix()
[1 1 0 0 0]
[0 0 1 1 0]
[x 0 0 0 1]
[0 x x 0 0]
[0 0 0 x x]
```

This is equivalent to:

```
sage: matrix(B)
[1 1 0 0 0]
[0 0 1 1 0]
[x 0 0 0 1]
[0 x x 0 0]
[0 0 0 x x]
```

It is also possible to use other entries in the adjacency matrix:

```
sage: B.adjacency_matrix(entry=(lambda transition: 1))
[1 1 0 0 0]
[0 0 1 1 0]
[1 0 0 0 1]
[0 1 1 0 0]
[0 0 0 1 1]
sage: B.adjacency_matrix(1, entry=(lambda transition:
....:     exp(I*transition.word_out[0]*var('t'))))
[      0       1       0       0       0]
[      0       0       0       1       0]
[e^(I*t)       0       0       0       0]
[      0       0 e^(I*t)       0       0]
[      0       0       0       0 e^(I*t)]
sage: a = Automaton([(0, 1, 0),
....:                (1, 2, 0),
....:                (2, 0, 1),
....:                (2, 1, 0)],
....:               initial_states=[0],
....:               final_states=[0])
sage: a.adjacency_matrix()
[0 1 0]
[0 0 1]
[1 1 0]
```

asymptotic_moments(*variable=n*)

Returns the main terms of expectation and variance of the sum of output labels and its covariance with the sum of input labels.

INPUT:

- `variable` – a symbol denoting the length of the input, by default n.

OUTPUT:

A dictionary consisting of

- `expectation` – $en + \mathrm{Order}(1)$,
- `variance` – $vn + \mathrm{Order}(1)$,
- `covariance` – $cn + \mathrm{Order}(1)$

for suitable constants e, v and c.

Assume that all input and output labels are numbers and that `self` is complete and has only one final component. Assume further that this final component is aperiodic. Furthermore, assume that there is exactly one initial state and that all states are final.

Denote by X_n the sum of output labels written by the finite state machine when reading a random input word of length n over the input alphabet (assuming equidistribution).

Then the expectation of X_n is $en + O(1)$, the variance of X_n is $vn + O(1)$ and the covariance of X_n and the sum of input labels is $cn + O(1)$, cf. [HKW2015], Theorem 3.9.

In the case of non-integer input or output labels, performance degrades significantly. For rational input and output labels, consider rescaling to integers. This limitation comes from the fact that determinants over polynomial rings can be computed much more efficiently than over the symbolic ring. In fact, we compute (parts) of a trivariate generating function where the input and output labels are exponents of some indeterminates, see [HKW2015], Theorem 3.9 for details. If those exponents are integers, we can use a polynomial ring.

EXAMPLES:

1. A trivial example: write the negative of the input:

```
sage: T = Transducer([(0, 0, 0, 0), (0, 0, 1, -1)],
....:                initial_states=[0],
....:                final_states=[0])
sage: T([0, 1, 1])
[0, -1, -1]
sage: moments = T.asymptotic_moments()
sage: moments['expectation']
-1/2*n + Order(1)
sage: moments['variance']
1/4*n + Order(1)
sage: moments['covariance']
-1/4*n + Order(1)
```

2. For the case of the Hamming weight of the non-adjacent-form (NAF) of integers, cf. the Wikipedia article Non-adjacent_form and the *example on recognizing NAFs*, the following agrees with the results in [HP2007].

   We first use the transducer to convert the standard binary expansion to the NAF given in [HP2007]. We use the parameter `with_final_word_out` such that we do not have to add sufficiently many trailing zeros:

```
sage: NAF = Transducer([(0, 0, 0, 0),
....:                   (0, '.1', 1, None),
....:                   ('.1', 0, 0, [1, 0]),
....:                   ('.1', 1, 1, [-1, 0]),
....:                   (1, 1, 1, 0),
....:                   (1, '.1', 0, None)],
....:                  initial_states=[0],
....:                  final_states=[0],
....:                  with_final_word_out=[0])
```

   As an example, we compute the NAF of 27 by this transducer.

```
sage: binary_27 = 27.bits()
sage: binary_27
[1, 1, 0, 1, 1]
sage: NAF_27 = NAF(binary_27)
sage: NAF_27
[-1, 0, -1, 0, 0, 1, 0]
sage: ZZ(NAF_27, base=2)
27
```

Next, we are only interested in the Hamming weight:

```
sage: def weight(state, input):
....:     if input is None:
....:         result = 0
....:     else:
....:         result = ZZ(input != 0)
....:     return (0, result)
sage: weight_transducer = Transducer(weight,
....:                                input_alphabet=[-1, 0, 1],
....:                                initial_states=[0],
....:                                final_states=[0])
sage: NAFweight = weight_transducer.composition(NAF)
sage: NAFweight.transitions()
[Transition from (0, 0) to (0, 0): 0|0,
 Transition from (0, 0) to ('.1', 0): 1|-,
 Transition from ('.1', 0) to (0, 0): 0|1,0,
 Transition from ('.1', 0) to (1, 0): 1|1,0,
 Transition from (1, 0) to ('.1', 0): 0|-,
 Transition from (1, 0) to (1, 0): 1|0]
sage: NAFweight(binary_27)
[1, 0, 1, 0, 0, 1, 0]
```

Now, we actually compute the asymptotic moments:

```
sage: moments = NAFweight.asymptotic_moments()
sage: moments['expectation']
1/3*n + Order(1)
sage: moments['variance']
2/27*n + Order(1)
sage: moments['covariance']
Order(1)
```

3. This is Example 3.16 in [HKW2015], where a transducer with variable output labels is given. There, the aim was to choose the output labels of this very simple transducer such that the input and output sum are asymptotically independent, i.e., the constant c vanishes.

```
sage: var('a_1, a_2, a_3, a_4')
(a_1, a_2, a_3, a_4)
sage: T = Transducer([[0, 0, 0, a_1], [0, 1, 1, a_3],
....:                 [1, 0, 0, a_4], [1, 1, 1, a_2]],
....:                initial_states=[0], final_states=[0, 1])
sage: moments = T.asymptotic_moments()
verbose 0 (...) Non-integer output weights lead to
significant performance degradation.
sage: moments['expectation']
1/4*(a_1 + a_2 + a_3 + a_4)*n + Order(1)
sage: moments['covariance']
-1/4*(a_1 - a_2)*n + Order(1)
```

Therefore, the asymptotic covariance vanishes if and only if $a_2 = a_1$.

4. This is Example 4.3 in [HKW2015], dealing with the transducer converting the binary expansion of an integer into Gray code (cf. the Wikipedia article Gray_code and the *example on Gray code*):

```
sage: moments = transducers.GrayCode().asymptotic_moments()
sage: moments['expectation']
1/2*n + Order(1)
sage: moments['variance']
1/4*n + Order(1)
sage: moments['covariance']
Order(1)
```

5. This is the first part of Example 4.4 in [HKW2015], counting the number of 10 blocks in the standard binary expansion. The least significant digit is at the left-most position:

```
sage: block10 = transducers.CountSubblockOccurrences(
....:     [1, 0],
....:     input_alphabet=[0, 1])
sage: sorted(block10.transitions())
[Transition from () to (): 0|0,
 Transition from () to (1,): 1|0,
 Transition from (1,) to (): 0|1,
 Transition from (1,) to (1,): 1|0]
sage: moments = block10.asymptotic_moments()
sage: moments['expectation']
1/4*n + Order(1)
sage: moments['variance']
1/16*n + Order(1)
sage: moments['covariance']
Order(1)
```

6. This is the second part of Example 4.4 in [HKW2015], counting the number of 11 blocks in the standard binary expansion. The least significant digit is at the left-most position:

```
sage: block11 = transducers.CountSubblockOccurrences(
....:     [1, 1],
....:     input_alphabet=[0, 1])
sage: sorted(block11.transitions())
[Transition from () to (): 0|0,
 Transition from () to (1,): 1|0,
 Transition from (1,) to (): 0|0,
 Transition from (1,) to (1,): 1|1]
sage: var('N')
N
sage: moments = block11.asymptotic_moments(N)
sage: moments['expectation']
1/4*N + Order(1)
sage: moments['variance']
5/16*N + Order(1)
sage: correlation = (moments['covariance'].coefficient(N) /
....:                (1/2 * sqrt(moments['variance'].coefficient(N))))
sage: correlation
2/5*sqrt(5)
```

7. This is Example 4.5 in [HKW2015], counting the number of 01 blocks minus the number of 10 blocks in the standard binary expansion. The least significant digit is at the left-most position:

```
sage: block01 = transducers.CountSubblockOccurrences(
....:     [0, 1],
....:     input_alphabet=[0, 1])
sage: product_01x10 = block01.cartesian_product(block10)
sage: block_difference = transducers.sub([0, 1])(product_01x10)
sage: T = block_difference.simplification().relabeled()
sage: T.transitions()
[Transition from 0 to 1: 0|-1,
 Transition from 0 to 0: 1|0,
 Transition from 1 to 1: 0|0,
 Transition from 1 to 0: 1|1,
 Transition from 2 to 1: 0|0,
 Transition from 2 to 0: 1|0]
sage: moments = T.asymptotic_moments()
sage: moments['expectation']
Order(1)
sage: moments['variance']
Order(1)
sage: moments['covariance']
Order(1)
```

8. The finite state machine must have a unique final component:

```
sage: T = Transducer([(0, -1, -1, -1), (0, 1, 1, 1),
....:                 (-1, -1, -1, -1), (-1, -1, 1, -1),
....:                 (1, 1, -1, 1), (1, 1, 1, 1)],
....:                initial_states=[0],
....:                final_states=[0, 1, -1])
sage: T.asymptotic_moments()
Traceback (most recent call last):
...
NotImplementedError: asymptotic_moments is only
implemented for finite state machines with one final
component.
```

In this particular example, the first letter of the input decides whether we reach the loop at -1 or the loop at 1. In the first case, we have $X_n = -n$, while we have $X_n = n$ in the second case. Therefore, the expectation $E(X_n)$ of X_n is $E(X_n) = 0$. We get $(X_n - E(X_n))^2 = n^2$ in all cases, which results in a variance of n^2.

So this example shows that the variance may be non-linear if there is more than one final component.

TESTS:

1. An input alphabet must be given:

```
sage: T = Transducer([[0, 0, 0, 0]],
....:                initial_states=[0], final_states=[0],
....:                determine_alphabets=False)
sage: T.asymptotic_moments()
Traceback (most recent call last):
...
ValueError: No input alphabet is given.
Try calling determine_alphabets().
```

2. The finite state machine must have a unique initial state:

```
sage: T = Transducer([(0, 0, 0, 0)])
sage: T.asymptotic_moments()
Traceback (most recent call last):
...
ValueError: A unique initial state is required.
```

3.The finite state machine must be complete:

```
sage: T = Transducer([[0, 0, 0, 0]],
....:                initial_states=[0], final_states=[0],
....:                input_alphabet=[0, 1])
sage: T.asymptotic_moments()
Traceback (most recent call last):
...
NotImplementedError: This finite state machine is
not complete.
```

4.The final component of the finite state machine must be aperiodic:

```
sage: T = Transducer([(0, 1, 0, 0), (1, 0, 0, 0)],
....:                initial_states=[0], final_states=[0, 1])
sage: T.asymptotic_moments()
Traceback (most recent call last):
...
NotImplementedError: asymptotic_moments is only
implemented for finite state machines whose unique final
component is aperiodic.
```

5.Non-integer input or output labels lead to a warning:

```
sage: T = Transducer([[0, 0, 0, 0], [0, 0, 1, -1/2]],
....:                initial_states=[0], final_states=[0])
sage: moments = T.asymptotic_moments()
verbose 0 (...) Non-integer output weights lead to
significant performance degradation.
sage: moments['expectation']
-1/4*n + Order(1)
sage: moments['variance']
1/16*n + Order(1)
sage: moments['covariance']
-1/8*n + Order(1)
```

This warning can be silenced by `set_verbose()`:

```
sage: set_verbose(-1, "finite_state_machine.py")
sage: moments = T.asymptotic_moments()
sage: moments['expectation']
-1/4*n + Order(1)
sage: moments['variance']
1/16*n + Order(1)
sage: moments['covariance']
-1/8*n + Order(1)
sage: set_verbose(0, "finite_state_machine.py")
```

6.Check whether `word_out` of `FSMState` are correctly dealt with:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: s = FSMState(0, word_out=2,
....:              is_initial=True,
....:              is_final=True)
sage: T = Transducer([(s, s, 0, 1)],
....:                initial_states=[s], final_states=[s])
sage: T([0, 0])
[2, 1, 2, 1, 2]
sage: T.asymptotic_moments()['expectation']
3*n + Order(1)
```

The same test for non-integer output:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: s = FSMState(0, word_out=2/3)
sage: T = Transducer([(s, s, 0, 1/2)],
....:                initial_states=[s], final_states=[s])
sage: T.asymptotic_moments()['expectation']
verbose 0 (...) Non-integer output weights lead to
significant performance degradation.
7/6*n + Order(1)
```

7.All states of `self` have to be final:

```
sage: T = Transducer([(0, 1, 1, 4)], initial_states=[0])
sage: T.asymptotic_moments()
Traceback (most recent call last):
...
ValueError: Not all states are final.
```

ALGORITHM:

See [HKW2015], Theorem 3.9.

REFERENCES:

coaccessible_components()

Return the sub-machine induced by the coaccessible states of this finite state machine.

OUTPUT:

A finite state machine of the same type as this finite state machine.

EXAMPLES:

```
sage: A = automata.ContainsWord([1, 1],
....:     input_alphabet=[0, 1]).complement().minimization().relabeled()
sage: A.transitions()
[Transition from 0 to 0: 0|-,
 Transition from 0 to 0: 1|-,
 Transition from 1 to 1: 0|-,
 Transition from 1 to 2: 1|-,
 Transition from 2 to 1: 0|-,
```

```
 Transition from 2 to 0: 1|-]
sage: A.initial_states()
[1]
sage: A.final_states()
[1, 2]
sage: C = A.coaccessible_components()
sage: C.transitions()
[Transition from 1 to 1: 0|-,
 Transition from 1 to 2: 1|-,
 Transition from 2 to 1: 0|-]
```

See also:

accessible_components(), induced_sub_finite_state_machine()

completion(*sink=None*)

Return a completion of this finite state machine.

INPUT:

- sink – either an instance of FSMState or a label for the sink (default: None). If None, the least available non-zero integer is used.

OUTPUT:

A FiniteStateMachine of the same type as this finite state machine.

The resulting finite state machine is a complete version of this finite state machine. A finite state machine is considered to be complete if each transition has an input label of length one and for each pair (q, a) where q is a state and a is an element of the input alphabet, there is exactly one transition from q with input label a.

If this finite state machine is already complete, a deep copy is returned. Otherwise, a new non-final state (usually called a sink) is created and transitions to this sink are introduced as appropriate.

EXAMPLES:

```
sage: F = FiniteStateMachine([(0, 0, 0, 0),
....:                         (0, 1, 1, 1),
....:                         (1, 1, 0, 0)])
sage: F.is_complete()
False
sage: G1 = F.completion()
sage: G1.is_complete()
True
sage: G1.transitions()
[Transition from 0 to 0: 0|0,
 Transition from 0 to 1: 1|1,
 Transition from 1 to 1: 0|0,
 Transition from 1 to 2: 1|-,
 Transition from 2 to 2: 0|-,
 Transition from 2 to 2: 1|-]
sage: G2 = F.completion('Sink')
sage: G2.is_complete()
True
sage: G2.transitions()
[Transition from 0 to 0: 0|0,
 Transition from 0 to 1: 1|1,
 Transition from 1 to 1: 0|0,
 Transition from 1 to 'Sink': 1|-,
 Transition from 'Sink' to 'Sink': 0|-,
```

```
 Transition from 'Sink' to 'Sink': 1|-]
sage: F.completion(1)
Traceback (most recent call last):
...
ValueError: The finite state machine already contains a state
'1'.
```

An input alphabet must be given:

```
sage: F = FiniteStateMachine([(0, 0, 0, 0),
....:                         (0, 1, 1, 1),
....:                         (1, 1, 0, 0)],
....:                        determine_alphabets=False)
sage: F.is_complete()
Traceback (most recent call last):
...
ValueError: No input alphabet is given. Try calling
determine_alphabets().
```

Non-deterministic machines are not allowed.

```
sage: F = FiniteStateMachine([(0, 0, 0, 0), (0, 1, 0, 0)])
sage: F.is_complete()
False
sage: F.completion()
Traceback (most recent call last):
...
ValueError: The finite state machine must be deterministic.
sage: F = FiniteStateMachine([(0, 0, [0, 0], 0)])
sage: F.is_complete()
False
sage: F.completion()
Traceback (most recent call last):
...
ValueError: The finite state machine must be deterministic.
```

See also:

`is_complete()`, `split_transitions()`, `determine_alphabets()`, `is_deterministic()`.

TESTS:

Test the use of an `FSMState` as sink:

```
sage: F = FiniteStateMachine([(0, 0, 0, 0),
....:                         (0, 1, 1, 1),
....:                         (1, 1, 0, 0)])
sage: from sage.combinat.finite_state_machine import FSMState
sage: F.completion(FSMState(1))
Traceback (most recent call last):
...
ValueError: The finite state machine already contains a state
'1'.
sage: s = FSMState(2)
sage: G = F.completion(s)
sage: G.state(2) is s
True
```

composition(*other*, *algorithm=None*, *only_accessible_components=True*)

Returns a new transducer which is the composition of `self` and `other`.

INPUT:

- `other` – a transducer
- `algorithm` – can be one of the following
  - `direct` – The composition is calculated directly.

    There can be arbitrarily many initial and final states, but the input and output labels must have length 1.

    Warning: The output of `other` is fed into `self`.

  - `explorative` – An explorative algorithm is used.

    The input alphabet of self has to be specified.

    Warning: The output of `other` is fed into `self`.

  If algorithm is `None`, then the algorithm is chosen automatically (at the moment always `direct`, except when there are output words of `other` or input words of `self` of length greater than 1).

OUTPUT:

A new transducer.

The labels of the new finite state machine are pairs of states of the original finite state machines. The color of a new state is the tuple of colors of the constituent states.

EXAMPLES:

```
sage: F = Transducer([('A', 'B', 1, 0), ('B', 'A', 0, 1)],
....:                initial_states=['A', 'B'], final_states=['B'],
....:                determine_alphabets=True)
sage: G = Transducer([(1, 1, 1, 0), (1, 2, 0, 1),
....:                 (2, 2, 1, 1), (2, 2, 0, 0)],
....:                initial_states=[1], final_states=[2],
....:                determine_alphabets=True)
sage: Hd = F.composition(G, algorithm='direct')
sage: Hd.initial_states()
[(1, 'B'), (1, 'A')]
sage: Hd.transitions()
[Transition from (1, 'B') to (1, 'A'): 1|1,
 Transition from (1, 'A') to (2, 'B'): 0|0,
 Transition from (2, 'B') to (2, 'A'): 0|1,
 Transition from (2, 'A') to (2, 'B'): 1|0]
sage: He = F.composition(G, algorithm='explorative')
sage: He.initial_states()
[(1, 'A'), (1, 'B')]
sage: He.transitions()
[Transition from (1, 'A') to (2, 'B'): 0|0,
 Transition from (1, 'B') to (1, 'A'): 1|1,
 Transition from (2, 'B') to (2, 'A'): 0|1,
 Transition from (2, 'A') to (2, 'B'): 1|0]
sage: Hd == He
True
```

The following example has output of length > 1, so the explorative algorithm has to be used (and is selected automatically).

```
sage: F = Transducer([('A', 'B', 1, [1, 0]), ('B', 'B', 1, 1),
....:                 ('B', 'B', 0, 0)],
....:                initial_states=['A'], final_states=['B'])
sage: G = Transducer([(1, 1, 0, 0), (1, 2, 1, 0),
....:                 (2, 2, 0, 1), (2, 1, 1, 1)],
....:                initial_states=[1], final_states=[1])
sage: He = G.composition(F, algorithm='explorative')
sage: He.transitions()
[Transition from ('A', 1) to ('B', 2): 1|0,1,
 Transition from ('B', 2) to ('B', 2): 0|1,
 Transition from ('B', 2) to ('B', 1): 1|1,
 Transition from ('B', 1) to ('B', 1): 0|0,
 Transition from ('B', 1) to ('B', 2): 1|0]
sage: Ha = G.composition(F)
sage: Ha == He
True
```

Final output words are also considered:

```
sage: F = Transducer([('A', 'B', 1, 0), ('B', 'A', 0, 1)],
....:                initial_states=['A', 'B'],
....:                final_states=['A', 'B'])
sage: F.state('A').final_word_out = 0
sage: F.state('B').final_word_out = 1
sage: G = Transducer([(1, 1, 1, 0), (1, 2, 0, 1),
....:                 (2, 2, 1, 1), (2, 2, 0, 0)],
....:                initial_states=[1], final_states=[2])
sage: G.state(2).final_word_out = 0
sage: Hd = F.composition(G, algorithm='direct')
sage: Hd.final_states()
[(2, 'B')]
sage: He = F.composition(G, algorithm='explorative')
sage: He.final_states()
[(2, 'B')]
```

Note that `(2, 'A')` is not final, as the final output 0 of state 2 of G cannot be processed in state `'A'` of F.

```
sage: [s.final_word_out for s in Hd.final_states()]
[[1, 0]]
sage: [s.final_word_out for s in He.final_states()]
[[1, 0]]
sage: Hd == He
True
```

Here is a non-deterministic example with intermediate output length > 1.

```
sage: F = Transducer([(1, 1, 1, ['a', 'a']), (1, 2, 1, 'b'),
....:                 (2, 1, 2, 'a'), (2, 2, 2, 'b')],
....:                initial_states=[1, 2])
sage: G = Transducer([('A', 'A', 'a', 'i'),
....:                 ('A', 'B', 'a', 'l'),
....:                 ('B', 'B', 'b', 'e')],
....:                initial_states=['A', 'B'])
sage: G(F).transitions()
[Transition from (1, 'A') to (1, 'A'): 1|'i','i',
 Transition from (1, 'A') to (1, 'B'): 1|'i','l',
 Transition from (1, 'B') to (2, 'B'): 1|'e',
 Transition from (2, 'A') to (1, 'A'): 2|'i',
```

```
 Transition from (2, 'A') to (1, 'B'): 2|'l',
 Transition from (2, 'B') to (2, 'B'): 2|'e']
```

Be aware that after composition, different transitions may share the same output label (same python object):

```
sage: F = Transducer([ ('A','B',0,0), ('B','A',0,0)],
....:                initial_states=['A'],
....:                final_states=['A'])
sage: F.transitions()[0].word_out is F.transitions()[1].word_out
False
sage: G = Transducer([('C','C',0,1)],)
....:                initial_states=['C'],
....:                final_states=['C'])
sage: H = G.composition(F)
sage: H.transitions()[0].word_out is H.transitions()[1].word_out
True
```

TESTS:

In the explorative algorithm, transducers with non-empty final output words are implemented in trac ticket #16548:

```
sage: A = transducers.GrayCode()
sage: B = transducers.abs([0, 1])
sage: A.composition(B, algorithm='explorative').transitions()
[Transition from (0, 0) to (0, 1): 0|-,
 Transition from (0, 0) to (0, 2): 1|-,
 Transition from (0, 1) to (0, 1): 0|0,
 Transition from (0, 1) to (0, 2): 1|1,
 Transition from (0, 2) to (0, 1): 0|1,
 Transition from (0, 2) to (0, 2): 1|0]
```

Similarly, the explorative algorithm can handle non-deterministic finite state machines as of trac ticket #16548:

```
sage: A = Transducer([(0, 0, 0, 0), (0, 1, 0, 0)],
....:                initial_states=[0])
sage: B = transducers.Identity([0])
sage: A.composition(B, algorithm='explorative').transitions()
[Transition from (0, 0) to (0, 0): 0|0,
 Transition from (0, 0) to (0, 1): 0|0]
sage: B.composition(A, algorithm='explorative').transitions()
[Transition from (0, 0) to (0, 0): 0|0,
 Transition from (0, 0) to (1, 0): 0|0]
```

In the following example, `algorithm='direct'` is inappropriate as there are edges with output labels of length greater than 1:

```
sage: F = Transducer([('A', 'B', 1, [1, 0]), ('B', 'B', 1, 1),
....:                 ('B', 'B', 0, 0)],
....:                initial_states=['A'], final_states=['B'])
sage: G = Transducer([(1, 1, 0, 0), (1, 2, 1, 0),
....:                 (2, 2, 0, 1), (2, 1, 1, 1)],
....:                initial_states=[1], final_states=[1])
sage: Hd = G.composition(F, algorithm='direct')
```

In the following examples, we compose transducers and automata and check whether the types are correct.

```
sage: from sage.combinat.finite_state_machine import (
....:     is_Automaton, is_Transducer)
sage: T = Transducer([(0, 0, 0, 0)], initial_states=[0])
sage: A = Automaton([(0, 0, 0)], initial_states=[0])
sage: is_Transducer(T.composition(T, algorithm='direct'))
True
sage: is_Transducer(T.composition(T, algorithm='explorative'))
True
sage: T.composition(A, algorithm='direct')
Traceback (most recent call last):
...
TypeError: Composition with automaton is not possible.
sage: T.composition(A, algorithm='explorative')
Traceback (most recent call last):
...
TypeError: Composition with automaton is not possible.
sage: A.composition(A, algorithm='direct')
Traceback (most recent call last):
...
TypeError: Composition with automaton is not possible.
sage: A.composition(A, algorithm='explorative')
Traceback (most recent call last):
...
TypeError: Composition with automaton is not possible.
sage: is_Automaton(A.composition(T, algorithm='direct'))
True
sage: is_Automaton(A.composition(T, algorithm='explorative'))
True
```

Non-deterministic final output cannot be handeled:

```
sage: F = Transducer([('I', 'A', 0, 42), ('I', 'B', 0, 42)],
....:                initial_states=['I'],
....:                final_states=['A', 'B'])
sage: G = Transducer(initial_states=[0],
....:                final_states=[0],
....:                input_alphabet=[0])
sage: G.state(0).final_word_out = 0
sage: H = F.composition(G, algorithm='explorative')
sage: for s in H.final_states():
....:     print s, s.final_word_out
(0, 'I') [42]
sage: F.state('A').final_word_out = 'a'
sage: F.state('B').final_word_out = 'b'
sage: F.composition(G, algorithm='explorative')
Traceback (most recent call last):
...
NotImplementedError: Stopping in state (0, 'I') leads to
non-deterministic final output.
```

Check that the output and input alphabets are set correctly:

```
sage: F = Transducer([(0, 0, 1, 'A')],
....:                initial_states=[0],
....:                determine_alphabets=False)
sage: G = Transducer([(2, 2, 'A', 'a')],
....:                initial_states=[2],
....:                determine_alphabets=False)
sage: Hd = G(F, algorithm='direct')
```

```
sage: Hd.input_alphabet, Hd.output_alphabet
([1], ['a'])
sage: He = G(F, algorithm='explorative')
Traceback (most recent call last):
...
ValueError: No input alphabet is given. Try calling
determine_alphabets().
sage: F.input_alphabet = [1]
sage: Hd = G(F, algorithm='direct')
sage: Hd.input_alphabet, Hd.output_alphabet
([1], ['a'])
sage: He = G(F, algorithm='explorative')
sage: He.input_alphabet, He.output_alphabet
([1], None)
sage: G.output_alphabet = ['a']
sage: Hd = G(F, algorithm='direct')
sage: Hd.input_alphabet, Hd.output_alphabet
([1], ['a'])
sage: He = G(F, algorithm='explorative')
sage: He.input_alphabet, He.output_alphabet
([1], ['a'])
sage: Hd == He
True
sage: F.input_alphabet = None
sage: Hd = G(F, algorithm='direct')
sage: Hd.input_alphabet, Hd.output_alphabet
([1], ['a'])
sage: He = G(F, algorithm='explorative')
Traceback (most recent call last):
...
ValueError: No input alphabet is given. Try calling
determine_alphabets().
```

concatenation(*other*)

Concatenate this finite state machine with another finite state machine.

INPUT:

- `other` – a `FiniteStateMachine`.

OUTPUT:

A `FiniteStateMachine` of the same type as this finite state machine.

Assume that both finite state machines are automata. If $\mathcal{L}_1$ is the language accepted by this automaton and $\mathcal{L}_2$ is the language accepted by the other automaton, then the language accepted by the concatenated automaton is $\{w_1w_2 \mid w_1 \in \mathcal{L}_1, w_2 \in \mathcal{L}_2\}$ where w_1w_2 denotes the concatenation of the words w_1 and w_2.

Assume that both finite state machines are transducers and that this transducer maps words $w_1 \in \mathcal{L}_1$ to words $f_1(w_1)$ and that the other transducer maps words $w_2 \in \mathcal{L}_2$ to words $f_2(w_2)$. Then the concatenated transducer maps words w_1w_2 with $w_1 \in \mathcal{L}_1$ and $w_2 \in \mathcal{L}_2$ to $f_1(w_1)f_2(w_2)$. Here, w_1w_2 and $f_1(w_1)f_2(w_2)$ again denote concatenation of words.

The input alphabet is the union of the input alphabets (if possible) and `None` otherwise. In the latter case, try calling `determine_alphabets()`.

Instead of `A.concatenation(B)`, the notation `A * B` can be used.

EXAMPLES:

Concatenation of two automata:

```
sage: A = automata.Word([0])
sage: B = automata.Word([1])
sage: C = A.concatenation(B)
sage: C.transitions()
[Transition from (0, 0) to (0, 1): 0|-,
 Transition from (0, 1) to (1, 0): -|-,
 Transition from (1, 0) to (1, 1): 1|-]
sage: [w
....:  for w in ([0, 0], [0, 1], [1, 0], [1, 1])
....:  if C(w)]
[[0, 1]]
sage: from sage.combinat.finite_state_machine import (
....:     is_Automaton, is_Transducer)
sage: is_Automaton(C)
True
```

Concatenation of two transducers:

```
sage: A = Transducer([(0, 1, 0, 1), (0, 1, 1, 2)],
....:                initial_states=[0],
....:                final_states=[1])
sage: B = Transducer([(0, 1, 0, 1), (0, 1, 1, 0)],
....:                initial_states=[0],
....:                final_states=[1])
sage: C = A.concatenation(B)
sage: C.transitions()
[Transition from (0, 0) to (0, 1): 0|1,
 Transition from (0, 0) to (0, 1): 1|2,
 Transition from (0, 1) to (1, 0): -|-,
 Transition from (1, 0) to (1, 1): 0|1,
 Transition from (1, 0) to (1, 1): 1|0]
sage: [(w, C(w)) for w in ([0, 0], [0, 1], [1, 0], [1, 1])]
[([0, 0], [1, 1]),
 ([0, 1], [1, 0]),
 ([1, 0], [2, 1]),
 ([1, 1], [2, 0])]
sage: is_Transducer(C)
True
```

Alternative notation as multiplication:

```
sage: C == A * B
True
```

Final output words are taken into account:

```
sage: A = Transducer([(0, 1, 0, 1)],
....:                initial_states=[0],
....:                final_states=[1])
sage: A.state(1).final_word_out = 2
sage: B = Transducer([(0, 1, 0, 3)],
....:                initial_states=[0],
....:                final_states=[1])
sage: B.state(1).final_word_out = 4
sage: C = A * B
sage: C([0, 0])
[1, 2, 3, 4]
```

Handling of the input alphabet:

```
sage: A = Automaton([(0, 0, 0)])
sage: B = Automaton([(0, 0, 1)], input_alphabet=[1, 2])
sage: C = Automaton([(0, 0, 2)], determine_alphabets=False)
sage: D = Automaton([(0, 0, [[0, 0]])], input_alphabet=[[0, 0]])
sage: A.input_alphabet
[0]
sage: B.input_alphabet
[1, 2]
sage: C.input_alphabet is None
True
sage: D.input_alphabet
[[0, 0]]
sage: (A * B).input_alphabet
[0, 1, 2]
sage: (A * C).input_alphabet is None
True
sage: (A * D).input_alphabet is None
True
```

See also:

disjoint_union(), determine_alphabets().

TESTS:

```
sage: A = Automaton()
sage: F = FiniteStateMachine()
sage: A * F
Traceback (most recent call last):
...
TypeError: Cannot concatenate finite state machines of
different types.
sage: F * A
Traceback (most recent call last):
...
TypeError: Cannot concatenate finite state machines of
different types.
sage: F * 5
Traceback (most recent call last):
...
TypeError: A finite state machine can only be concatenated
with a another finite state machine.
```

construct_final_word_out(*letters*, *allow_non_final=True*)

This is an inplace version of with_final_word_out(). See with_final_word_out() for documentation and examples.

TESTS:

```
sage: T = Transducer([(0, 1, 0, 0), (1, 0, 0, 1)],
....:                initial_states=[0],
....:                final_states=[0])
sage: F = T.with_final_word_out(0)
sage: T.construct_final_word_out(0)
sage: T == F  # indirect doctest
True
sage: T = Transducer([(0, 1, 0, None)],
....:                final_states=[1])
sage: F = T.with_final_word_out(0)
sage: F.state(0).final_word_out
```

```
[]
```

copy()

Returns a (shallow) copy of the finite state machine.

INPUT:

Nothing.

OUTPUT:

A new finite state machine.

TESTS:

```
sage: copy(FiniteStateMachine())
Traceback (most recent call last):
...
NotImplementedError
```

deepcopy(*memo=None*)

Returns a deep copy of the finite state machine.

INPUT:

- memo – (default: None) a dictionary storing already processed elements.

OUTPUT:

A new finite state machine.

EXAMPLES:

```
sage: F = FiniteStateMachine([('A', 'A', 0, 1), ('A', 'A', 1, 0)])
sage: deepcopy(F)
Finite state machine with 1 state
```

TESTS:

Make sure that the links between transitions and states are still intact:

```
sage: C = deepcopy(F)
sage: C.transitions()[0].from_state is C.state('A')
True
sage: C.transitions()[0].to_state is C.state('A')
True
```

default_format_letter(*x*, *combine_all=False*)

Return a LatexExpr built out of the argument x.

INPUT:

- x – a Sage object
- combine_all – boolean (Default: False) If combine_all is True and the input is a tuple, then it does not return a tuple and instead returns a string with all the elements separated by a single space.

OUTPUT:

A LatexExpr built from x

EXAMPLES:

```
sage: latex(Integer(3))  # indirect doctest
3
sage: latex(1==0)
\mathrm{False}
sage: print latex([x,2])
\left[x, 2\right]
```

Check that trac ticket #11775 is fixed:

```
sage: latex((x,2), combine_all=True)
x 2
```

default_format_transition_label(*word*)

Default formatting of words in transition labels for LaTeX output.

INPUT:

`word` – list of letters

OUTPUT:

String representation of `word` suitable to be typeset in mathematical mode.

- For a non-empty word: Concatenation of the letters, piped through `self.format_letter` and separated by blanks.
- For an empty word: `sage.combinat.finite_state_machine.EmptyWordLaTeX`.

There is also a variant `format_transition_label_reversed()` writing the words in reversed order.

EXAMPLES:

1. Example of a non-empty word:

```
sage: T = Transducer()
sage: print T.default_format_transition_label(
....:     ['a', 'alpha', 'a_1', '0', 0, (0, 1)])
\text{\texttt{a}} \text{\texttt{alpha}}
\text{\texttt{a{\char`\_}1}} 0 0 \left(0, 1\right)
```

2. In the example above, `'a'` and `'alpha'` should perhaps be symbols:

```
sage: var('a alpha a_1')
(a, alpha, a_1)
sage: print T.default_format_transition_label([a, alpha, a_1])
a \alpha a_{1}
```

3. Example of an empty word:

```
sage: print T.default_format_transition_label([])
\varepsilon
```

We can change this by setting `sage.combinat.finite_state_machine.EmptyWordLaTeX`:

```
sage: sage.combinat.finite_state_machine.EmptyWordLaTeX = ''
sage: T.default_format_transition_label([])
''
```

Finally, we restore the default value:

```
sage: sage.combinat.finite_state_machine.EmptyWordLaTeX = r'\varepsilon'
```

4. This method is the default value for `FiniteStateMachine.format_transition_label`. That can be changed to be any other function:

```
sage: A = Automaton([(0, 1, 0)])
sage: def custom_format_transition_label(word):
....:     return "t"
sage: A.latex_options(format_transition_label=custom_format_transition_label)
sage: print latex(A)
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state] (v0) at (3.000000, 0.000000) {$0$};
\node[state] (v1) at (-3.000000, 0.000000) {$1$};
\path[->] (v0) edge node[rotate=360.00, anchor=south] {$t$} (v1);
\end{tikzpicture}
```

TEST:

Check that trac ticket #16357 is fixed:

```
sage: T = Transducer()
sage: T.default_format_transition_label([])
'\\varepsilon'
sage: T.default_format_transition_label(iter([]))
'\\varepsilon'
```

delete_state(*s*)

Deletes a state and all transitions coming or going to this state.

INPUT:

- `s` – a label of a state or an `FSMState`.

OUTPUT:

Nothing.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: t1 = FSMTransition('A', 'B', 0)
sage: t2 = FSMTransition('B', 'B', 1)
sage: F = FiniteStateMachine([t1, t2])
sage: F.delete_state('A')
sage: F.transitions()
[Transition from 'B' to 'B': 1|-]
```

TESTS:

This shows that trac ticket #16024 is fixed.

```
sage: F._states_
['B']
sage: F._states_dict_
{'B': 'B'}
```

delete_transition(*t*)

Deletes a transition by removing it from the list of transitions of the state, where the transition starts.

INPUT:

- `t` – a transition.

OUTPUT:

Nothing.

EXAMPLES:

```
sage: F = FiniteStateMachine([('A', 'B', 0), ('B', 'A', 1)])
sage: F.delete_transition(('A', 'B', 0))
sage: F.transitions()
[Transition from 'B' to 'A': 1|-]
```

determine_alphabets(*reset=True*)

Determine the input and output alphabet according to the transitions in this finite state machine.

INPUT:

- `reset` – If reset is `True`, then the existing input and output alphabets are erased, otherwise new letters are appended to the existing alphabets.

OUTPUT:

Nothing.

After this operation the input alphabet and the output alphabet of this finite state machine are a list of letters.

Todo

At the moment, the letters of the alphabets need to be hashable.

EXAMPLES:

```
sage: T = Transducer([(1, 1, 1, 0), (1, 2, 2, 1),
....:                 (2, 2, 1, 1), (2, 2, 0, 0)],
....:                final_states=[1],
....:                determine_alphabets=False)
sage: T.state(1).final_word_out = [1, 4]
sage: (T.input_alphabet, T.output_alphabet)
(None, None)
sage: T.determine_alphabets()
sage: (T.input_alphabet, T.output_alphabet)
([0, 1, 2], [0, 1, 4])
```

See also:

`determine_input_alphabet()`, `determine_output_alphabet()`.

determine_input_alphabet(*reset=True*)

Determine the input alphabet according to the transitions of this finite state machine.

INPUT:

- `reset` – a boolean (default: `True`). If `True`, then the existing input alphabet is erased, otherwise new letters are appended to the existing alphabet.

OUTPUT:

Nothing.

After this operation the input alphabet of this finite state machine is a list of letters.

Todo

At the moment, the letters of the alphabet need to be hashable.

EXAMPLES:

```
sage: T = Transducer([(1, 1, 1, 0), (1, 2, 2, 1),
....:                 (2, 2, 1, 1), (2, 2, 0, 0)],
....:                final_states=[1],
....:                determine_alphabets=False)
sage: (T.input_alphabet, T.output_alphabet)
(None, None)
sage: T.determine_input_alphabet()
sage: (T.input_alphabet, T.output_alphabet)
([0, 1, 2], None)
```

See also:

`determine_output_alphabet()`, `determine_alphabets()`.

determine_output_alphabet(*reset=True*)

Determine the output alphabet according to the transitions of this finite state machine.

INPUT:

- `reset` – a boolean (default: `True`). If `True`, then the existing output alphabet is erased, otherwise new letters are appended to the existing alphabet.

OUTPUT:

Nothing.

After this operation the output alphabet of this finite state machine is a list of letters.

Todo

At the moment, the letters of the alphabet need to be hashable.

EXAMPLES:

```
sage: T = Transducer([(1, 1, 1, 0), (1, 2, 2, 1),
....:                 (2, 2, 1, 1), (2, 2, 0, 0)],
....:                final_states=[1],
....:                determine_alphabets=False)
sage: T.state(1).final_word_out = [1, 4]
sage: (T.input_alphabet, T.output_alphabet)
(None, None)
sage: T.determine_output_alphabet()
sage: (T.input_alphabet, T.output_alphabet)
(None, [0, 1, 4])
```

See also:

`determine_input_alphabet()`, `determine_alphabets()`.

digraph(*edge_labels='words_in_out'*)

Returns the graph of the finite state machine with labeled vertices and labeled edges.

INPUT:

- **edge_label: (default: 'words_in_out') can be**
  - 'words_in_out' (labels will be strings 'i|o')
  - a function with which takes as input a transition and outputs (returns) the label

OUTPUT:

A directed graph.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A')
sage: T = Transducer()
sage: T.graph()
Looped multi-digraph on 0 vertices
sage: T.add_state(A)
'A'
sage: T.graph()
Looped multi-digraph on 1 vertex
sage: T.add_transition(('A', 'A', 0, 1))
Transition from 'A' to 'A': 0|1
sage: T.graph()
Looped multi-digraph on 1 vertex
```

See also:

DiGraph

disjoint_union(*other*)

Return the disjoint union of this and another finite state machine.

INPUT:

- other – a FiniteStateMachine.

OUTPUT:

A finite state machine of the same type as this finite state machine.

In general, the disjoint union of two finite state machines is non-deterministic. In the case of a automata, the language accepted by the disjoint union is the union of the languages accepted by the constituent automata. In the case of transducer, for each successful path in one of the constituent transducers, there will be one successful path with the same input and output labels in the disjoint union.

The labels of the states of the disjoint union are pairs (i, s): for each state s of this finite state machine, there is a state (0, s) in the disjoint union; for each state s of the other finite state machine, there is a state (1, s) in the disjoint union.

The input alphabet is the union of the input alphabets (if possible) and None otherwise. In the latter case, try calling determine_alphabets().

The disjoint union can also be written as A + B or A | B.

EXAMPLES:

```
sage: A = Automaton([(0, 1, 0), (1, 0, 1)],
....:               initial_states=[0],
....:               final_states=[0])
```

```
sage: A([0, 1, 0, 1])
True
sage: B = Automaton([(0, 1, 0), (1, 2, 0), (2, 0, 1)],
....:               initial_states=[0],
....:               final_states=[0])
sage: B([0, 0, 1])
True
sage: C = A.disjoint_union(B)
sage: C
Automaton with 5 states
sage: C.transitions()
[Transition from (0, 0) to (0, 1): 0|-,
 Transition from (0, 1) to (0, 0): 1|-,
 Transition from (1, 0) to (1, 1): 0|-,
 Transition from (1, 1) to (1, 2): 0|-,
 Transition from (1, 2) to (1, 0): 1|-]
sage: C([0, 0, 1])
True
sage: C([0, 1, 0, 1])
True
sage: C([1])
False
sage: C.initial_states()
[(0, 0), (1, 0)]
```

Instead of `.disjoint_union`, alternative notations are available:

```
sage: C1 = A + B
sage: C1 == C
True
sage: C2 = A | B
sage: C2 == C
True
```

In general, the disjoint union is not deterministic.:

```
sage: C.is_deterministic()
False
sage: D = C.determinisation().minimization()
sage: D.is_equivalent(Automaton([(0, 0, 0), (0, 0, 1),
....:     (1, 7, 0), (1, 0, 1), (2, 6, 0), (2, 0, 1),
....:     (3, 5, 0), (3, 0, 1), (4, 0, 0), (4, 2, 1),
....:     (5, 0, 0), (5, 3, 1), (6, 4, 0), (6, 0, 1),
....:     (7, 4, 0), (7, 3, 1)],
....:    initial_states=[1],
....:    final_states=[1, 2, 3]))
True
```

Disjoint union of transducers:

```
sage: T1 = Transducer([(0, 0, 0, 1)],
....:                 initial_states=[0],
....:                 final_states=[0])
sage: T2 = Transducer([(0, 0, 0, 2)],
....:                 initial_states=[0],
....:                 final_states=[0])
sage: T1([0])
[1]
sage: T2([0])
```

```
[2]
sage: T = T1.disjoint_union(T2)
sage: T([0])
Traceback (most recent call last):
...
ValueError: Found more than one accepting path.
sage: T.process([0])
[(True, (1, 0), [2]), (True, (0, 0), [1])]
```

Handling of the input alphabet (see trac ticket #18989):

```
sage: A = Automaton([(0, 0, 0)])
sage: B = Automaton([(0, 0, 1)], input_alphabet=[1, 2])
sage: C = Automaton([(0, 0, 2)], determine_alphabets=False)
sage: D = Automaton([(0, 0, [[0, 0]])], input_alphabet=[[0, 0]])
sage: A.input_alphabet
[0]
sage: B.input_alphabet
[1, 2]
sage: C.input_alphabet is None
True
sage: D.input_alphabet
[[0, 0]]
sage: (A + B).input_alphabet
[0, 1, 2]
sage: (A + C).input_alphabet is None
True
sage: (A + D).input_alphabet is None
True
```

See also:

`Automaton.intersection()`, `Transducer.intersection()`, `determine_alphabets()`.

empty_copy(*memo=None*, *new_class=None*)

Returns an empty deep copy of the finite state machine, i.e., `input_alphabet`, `output_alphabet`, `on_duplicate_transition` are preserved, but states and transitions are not.

INPUT:

- `memo` – a dictionary storing already processed elements.
- `new_class` – a class for the copy. By default (`None`), the class of `self` is used.

OUTPUT:

A new finite state machine.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_raise_error
sage: F = FiniteStateMachine([('A', 'A', 0, 2), ('A', 'A', 1, 3)],
....:                        input_alphabet=[0, 1],
....:                        output_alphabet=[2, 3],
....:                        on_duplicate_transition=duplicate_transition_raise_error)
sage: FE = F.empty_copy(); FE
Empty finite state machine
sage: FE.input_alphabet
[0, 1]
sage: FE.output_alphabet
[2, 3]
```

```
sage: FE.on_duplicate_transition == duplicate_transition_raise_error
True
```

TESTS:

```
sage: T = Transducer()
sage: type(T.empty_copy())
<class 'sage.combinat.finite_state_machine.Transducer'>
sage: type(T.empty_copy(new_class=Automaton))
<class 'sage.combinat.finite_state_machine.Automaton'>
```

epsilon_successors(*state*)

Returns the dictionary with states reachable from `state` without reading anything from an input tape as keys. The values are lists of outputs.

INPUT:

- `state` – the state whose epsilon successors should be determined.

OUTPUT:

A dictionary mapping states to a list of output words.

The states in the output are the epsilon successors of `state`. Each word of the list of output words is a word written when taking a path from `state` to the corresponding state.

EXAMPLES:

```
sage: T = Transducer([(0, 1, None, 'a'), (1, 2, None, 'b')])
sage: T.epsilon_successors(0)
{1: [['a']], 2: [['a', 'b']]}
sage: T.epsilon_successors(1)
{2: [['b']]}
sage: T.epsilon_successors(2)
{}
```

If there is a cycle with only epsilon transitions, then this cycle is only processed once and there is no infinite loop:

```
sage: S = Transducer([(0, 1, None, 'a'), (1, 0, None, 'b')])
sage: S.epsilon_successors(0)
{0: [['a', 'b']], 1: [['a']]}
sage: S.epsilon_successors(1)
{0: [['b']], 1: [['b', 'a']]}
```

equivalence_classes()

Returns a list of equivalence classes of states.

INPUT:

Nothing.

OUTPUT:

A list of equivalence classes of states.

Two states a and b are equivalent if and only if there is a bijection φ between paths starting at a and paths starting at b with the following properties: Let p_a be a path from a to a' and p_b a path from b to b' such that $\varphi(p_a) = p_b$, then

- $p_a.word_{in} = p_b.word_{in}$,
- $p_a.word_{out} = p_b.word_{out}$,

- a' and b' have the same output label, and
- a' and b' are both final or both non-final and have the same final output word.

The function `equivalence_classes()` returns a list of the equivalence classes to this equivalence relation.

This is one step of Moore's minimization algorithm.

See also:

`minimization()`

EXAMPLES:

```
sage: fsm = FiniteStateMachine([("A", "B", 0, 1), ("A", "B", 1, 0),
....:                           ("B", "C", 0, 0), ("B", "C", 1, 1),
....:                           ("C", "D", 0, 1), ("C", "D", 1, 0),
....:                           ("D", "A", 0, 0), ("D", "A", 1, 1)])
sage: sorted(fsm.equivalence_classes())
[['A', 'C'], ['B', 'D']]
sage: fsm.state("A").is_final = True
sage: sorted(fsm.equivalence_classes())
[['A'], ['B'], ['C'], ['D']]
sage: fsm.state("C").is_final = True
sage: sorted(fsm.equivalence_classes())
[['A', 'C'], ['B', 'D']]
sage: fsm.state("A").final_word_out = 1
sage: sorted(fsm.equivalence_classes())
[['A'], ['B'], ['C'], ['D']]
sage: fsm.state("C").final_word_out = 1
sage: sorted(fsm.equivalence_classes())
[['A', 'C'], ['B', 'D']]
```

final_components()

Returns the final components of a finite state machine as finite state machines.

INPUT:

Nothing.

OUTPUT:

A list of finite state machines, each representing a final component of `self`.

A final component of a transducer `T` is a strongly connected component `C` such that there are no transitions of `T` leaving `C`.

The final components are the only parts of a transducer which influence the main terms of the asympotic behaviour of the sum of output labels of a transducer, see [HKP2015] and [HKW2015].

EXAMPLES:

```
sage: T = Transducer([['A', 'B', 0, 0], ['B', 'C', 0, 1],
....:                 ['C', 'B', 0, 1], ['A', 'D', 1, 0],
....:                 ['D', 'D', 0, 0], ['D', 'B', 1, 0],
....:                 ['A', 'E', 2, 0], ['E', 'E', 0, 0]])
sage: FC = T.final_components()
sage: sorted(FC[0].transitions())
[Transition from 'B' to 'C': 0|1,
 Transition from 'C' to 'B': 0|1]
sage: FC[1].transitions()
[Transition from 'E' to 'E': 0|0]
```

Another example (cycle of length 2):

```
sage: T = Automaton([[0, 1, 0], [1, 0, 0]])
sage: len(T.final_components()) == 1
True
sage: T.final_components()[0].transitions()
[Transition from 0 to 1: 0|-,
 Transition from 1 to 0: 0|-]
```

final_states()

Returns a list of all final states.

INPUT:

Nothing.

OUTPUT:

A list of all final states.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A', is_final=True)
sage: B = FSMState('B', is_initial=True)
sage: C = FSMState('C', is_final=True)
sage: F = FiniteStateMachine([(A, B), (A, C)])
sage: F.final_states()
['A', 'C']
```

format_letter(*x*, *combine_all=False*)

Return a `LatexExpr` built out of the argument `x`.

INPUT:

- `x` – a Sage object
- `combine_all` – boolean (Default: `False`) If `combine_all` is `True` and the input is a tuple, then it does not return a tuple and instead returns a string with all the elements separated by a single space.

OUTPUT:

A `LatexExpr` built from `x`

EXAMPLES:

```
sage: latex(Integer(3))  # indirect doctest
3
sage: latex(1==0)
\mathrm{False}
sage: print latex([x,2])
\left[x, 2\right]
```

Check that trac ticket #11775 is fixed:

```
sage: latex((x,2), combine_all=True)
x 2
```

format_letter_negative(*letter*)

Format negative numbers as overlined numbers, everything else by standard LaTeX formatting.

INPUT:

letter – anything.

OUTPUT:

Overlined absolute value if letter is a negative integer, latex(letter) otherwise.

EXAMPLES:

```
sage: A = Automaton([(0, 0, -1)])
sage: map(A.format_letter_negative, [-1, 0, 1, 'a', None])
 ['\\overline{1}', 0, 1, \text{\texttt{a}}, \mbox{\rm None}]
sage: A.latex_options(format_letter=A.format_letter_negative)
sage: print(latex(A))
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state] (v0) at (3.000000, 0.000000) {$0$};
\path[->] (v0) edge[loop above] node {$\overline{1}$} ();
\end{tikzpicture}
```

format_transition_label(*word*)

Default formatting of words in transition labels for LaTeX output.

INPUT:

word – list of letters

OUTPUT:

String representation of word suitable to be typeset in mathematical mode.

- For a non-empty word: Concatenation of the letters, piped through self.format_letter and separated by blanks.
- For an empty word: sage.combinat.finite_state_machine.EmptyWordLaTeX.

There is also a variant format_transition_label_reversed() writing the words in reversed order.

EXAMPLES:

1. Example of a non-empty word:

```
sage: T = Transducer()
sage: print T.default_format_transition_label(
....:     ['a', 'alpha', 'a_1', '0', 0, (0, 1)])
\text{\texttt{a}} \text{\texttt{alpha}}
\text{\texttt{a{\char`\_}1}} 0 0 \left(0, 1\right)
```

2. In the example above, 'a' and 'alpha' should perhaps be symbols:

```
sage: var('a alpha a_1')
(a, alpha, a_1)
sage: print T.default_format_transition_label([a, alpha, a_1])
a \alpha a_{1}
```

3. Example of an empty word:

```
sage: print T.default_format_transition_label([])
\varepsilon
```

We can change this by setting `sage.combinat.finite_state_machine.EmptyWordLaTeX`:

```
sage: sage.combinat.finite_state_machine.EmptyWordLaTeX = ''
sage: T.default_format_transition_label([])
''
```

Finally, we restore the default value:

```
sage: sage.combinat.finite_state_machine.EmptyWordLaTeX = r'\varepsilon'
```

4. This method is the default value for `FiniteStateMachine.format_transition_label`. That can be changed to be any other function:

```
sage: A = Automaton([(0, 1, 0)])
sage: def custom_format_transition_label(word):
....:     return "t"
sage: A.latex_options(format_transition_label=custom_format_transition_label)
sage: print latex(A)
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state] (v0) at (3.000000, 0.000000) {$0$};
\node[state] (v1) at (-3.000000, 0.000000) {$1$};
\path[->] (v0) edge node[rotate=360.00, anchor=south] {$t$} (v1);
\end{tikzpicture}
```

TEST:

Check that trac ticket #16357 is fixed:

```
sage: T = Transducer()
sage: T.default_format_transition_label([])
'\\varepsilon'
sage: T.default_format_transition_label(iter([]))
'\\varepsilon'
```

format_transition_label_reversed(*word*)

Format words in transition labels in reversed order.

INPUT:

`word` – list of letters.

OUTPUT:

String representation of `word` suitable to be typeset in mathematical mode, letters are written in reversed order.

This is the reversed version of `default_format_transition_label()`.

In digit expansions, digits are frequently processed from the least significant to the most significant position, but it is customary to write the least significant digit at the right-most position. Therefore, the labels have to be reversed.

EXAMPLE:

```
sage: T = Transducer([(0, 0, 0, [1, 2, 3])])
sage: T.format_transition_label_reversed([1, 2, 3])
'3 2 1'
sage: T.latex_options(format_transition_label=T.format_transition_label_reversed)
```

```
sage: print latex(T)
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state] (v0) at (3.000000, 0.000000) {$0$};
\path[->] (v0) edge[loop above] node {$0\mid 3 2 1$} ();
\end{tikzpicture}
```

TEST:

Check that trac ticket #16357 is fixed:

```
sage: T = Transducer()
sage: T.format_transition_label_reversed([])
'\\varepsilon'
```

graph(*edge_labels='words_in_out'*)

Returns the graph of the finite state machine with labeled vertices and labeled edges.

INPUT:

- **edge_label: (default: 'words_in_out') can be**
  - 'words_in_out' (labels will be strings 'i|o')
  - a function with which takes as input a transition and outputs (returns) the label

OUTPUT:

A directed graph.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A')
sage: T = Transducer()
sage: T.graph()
Looped multi-digraph on 0 vertices
sage: T.add_state(A)
'A'
sage: T.graph()
Looped multi-digraph on 1 vertex
sage: T.add_transition(('A', 'A', 0, 1))
Transition from 'A' to 'A': 0|1
sage: T.graph()
Looped multi-digraph on 1 vertex
```

See also:

DiGraph

has_final_state(*state*)

Returns whether state is one of the final states of the finite state machine.

INPUT:

- state can be a FSMState or a label.

OUTPUT:

True or False.

EXAMPLES:

```
sage: FiniteStateMachine(final_states=['A']).has_final_state('A')
True
```

has_final_states()
Returns whether the finite state machine has a final state.

INPUT:

Nothing.

OUTPUT:

True or False.

EXAMPLES:

```
sage: FiniteStateMachine().has_final_states()
False
```

has_initial_state(*state*)
Returns whether `state` is one of the initial states of the finite state machine.

INPUT:

- `state` can be a `FSMState` or a label.

OUTPUT:

True or False.

EXAMPLES:

```
sage: F = FiniteStateMachine([('A', 'A')], initial_states=['A'])
sage: F.has_initial_state('A')
True
```

has_initial_states()
Returns whether the finite state machine has an initial state.

INPUT:

Nothing.

OUTPUT:

True or False.

EXAMPLES:

```
sage: FiniteStateMachine().has_initial_states()
False
```

has_state(*state*)
Returns whether `state` is one of the states of the finite state machine.

INPUT:

- `state` can be a `FSMState` or a label of a state.

OUTPUT:

True or False.

EXAMPLES:

```
sage: FiniteStateMachine().has_state('A')
False
```

has_transition(*transition*)
Returns whether `transition` is one of the transitions of the finite state machine.

INPUT:

- `transition` has to be a `FSMTransition`.

OUTPUT:

True or False.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: t = FSMTransition('A', 'A', 0, 1)
sage: FiniteStateMachine().has_transition(t)
False
sage: FiniteStateMachine().has_transition(('A', 'A', 0, 1))
Traceback (most recent call last):
...
TypeError: Transition is not an instance of FSMTransition.
```

induced_sub_finite_state_machine(*states*)

Returns a sub-finite-state-machine of the finite state machine induced by the given states.

INPUT:

- `states` – a list (or an iterator) of states (either labels or instances of `FSMState`) of the sub-finite-state-machine.

OUTPUT:

A new finite state machine. It consists (of deep copies) of the given states and (deep copies) of all transitions of `self` between these states.

EXAMPLE:

```
sage: FSM = FiniteStateMachine([(0, 1, 0), (0, 2, 0),
....:                           (1, 2, 0), (2, 0, 0)])
sage: sub_FSM = FSM.induced_sub_finite_state_machine([0, 1])
sage: sub_FSM.states()
[0, 1]
sage: sub_FSM.transitions()
[Transition from 0 to 1: 0|-]
sage: FSM.induced_sub_finite_state_machine([3])
Traceback (most recent call last):
...
ValueError: 3 is not a state of this finite state machine.
```

TESTS:

Make sure that the links between transitions and states are still intact:

```
sage: sub_FSM.transitions()[0].from_state is sub_FSM.state(0)
True
```

initial_states()

Returns a list of all initial states.

INPUT:

Nothing.

OUTPUT:

A list of all initial states.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A', is_initial=True)
sage: B = FSMState('B')
sage: F = FiniteStateMachine([(A, B, 1, 0)])
sage: F.initial_states()
['A']
```

input_alphabet

A list of letters representing the input alphabet of the finite state machine.

It can be set by the parameter `input_alphabet` when initializing a finite state machine, see `FiniteStateMachine`.

It can also be set by the method `determine_alphabets()`.

See also:

`FiniteStateMachine`, `determine_alphabets()`, `output_alphabet`.

input_projection()

Returns an automaton where the output of each transition of self is deleted.

INPUT:

Nothing

OUTPUT:

An automaton.

EXAMPLES:

```
sage: F = FiniteStateMachine([('A', 'B', 0, 1), ('A', 'A', 1, 1),
....:                         ('B', 'B', 1, 0)])
sage: G = F.input_projection()
sage: G.transitions()
[Transition from 'A' to 'B': 0|-,
 Transition from 'A' to 'A': 1|-,
 Transition from 'B' to 'B': 1|-]
```

intersection(*other*)

TESTS:

```
sage: FiniteStateMachine().intersection(FiniteStateMachine())
Traceback (most recent call last):
...
NotImplementedError
```

is_Markov_chain(*is_zero=None*)

Checks whether `self` is a Markov chain where the transition probabilities are modeled as input labels.

INPUT:

- `is_zero` – by default (`is_zero=None`), checking for zero is simply done by `is_zero()`. This parameter can be used to provide a more sophisticated check for zero, e.g. in the case of symbolic probabilities, see the examples below.

OUTPUT:

`True` or `False`.

on_duplicate_transition must be duplicate_transition_add_input(), the sum of the input weights of the transitions leaving a state must add up to 1 and the sum of initial probabilities must add up to 1 (or all be None).

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_add_input
sage: F = Transducer([[0, 0, 1/4, 0], [0, 1, 3/4, 1],
....:                 [1, 0, 1/2, 0], [1, 1, 1/2, 1]],
....:                on_duplicate_transition=duplicate_transition_add_input)
sage: F.is_Markov_chain()
True
```

on_duplicate_transition must be duplicate_transition_add_input():

```
sage: F = Transducer([[0, 0, 1/4, 0], [0, 1, 3/4, 1],
....:                 [1, 0, 1/2, 0], [1, 1, 1/2, 1]])
sage: F.is_Markov_chain()
False
```

Sum of input labels of the transitions leaving states must be 1:

```
sage: F = Transducer([[0, 0, 1/4, 0], [0, 1, 3/4, 1],
....:                 [1, 0, 1/2, 0]],
....:                on_duplicate_transition=duplicate_transition_add_input)
sage: F.is_Markov_chain()
False
```

The initial probabilities of all states must be None or they must sum up to 1. The initial probabilities of all states have to be set in the latter case:

```
sage: F = Transducer([[0, 0, 1/4, 0], [0, 1, 3/4, 1],
....:                 [1, 0, 1, 0]],
....:                on_duplicate_transition=duplicate_transition_add_input)
sage: F.is_Markov_chain()
True
sage: F.state(0).initial_probability = 1/4
sage: F.is_Markov_chain()
False
sage: F.state(1).initial_probability = 7
sage: F.is_Markov_chain()
False
sage: F.state(1).initial_probability = 3/4
sage: F.is_Markov_chain()
True
```

If the probabilities are variables in the symbolic ring, assume() will do the trick:

```
sage: var('p q')
(p, q)
sage: F = Transducer([(0, 0, p, 1), (0, 0, q, 0)],
....:                on_duplicate_transition=duplicate_transition_add_input)
sage: assume(p + q == 1)
sage: (p + q - 1).is_zero()
True
sage: F.is_Markov_chain()
True
sage: forget()
sage: del(p, q)
```

If the probabilities are variables in some polynomial ring, the parameter is_zero can be used:

```
sage: R.<p, q> = PolynomialRing(QQ)
sage: def is_zero_polynomial(polynomial):
....:     return polynomial in (p + q - 1)*R
sage: F = Transducer([(0, 0, p, 1), (0, 0, q, 0)],
....:                on_duplicate_transition=duplicate_transition_add_input)
sage: F.state(0).initial_probability = p + q
sage: F.is_Markov_chain()
False
sage: F.is_Markov_chain(is_zero_polynomial)
True
```

is_complete()

Returns whether the finite state machine is complete.

INPUT:

Nothing.

OUTPUT:

`True` or `False`.

A finite state machine is considered to be complete if each transition has an input label of length one and for each pair (q, a) where q is a state and a is an element of the input alphabet, there is exactly one transition from q with input label a.

EXAMPLES:

```
sage: fsm = FiniteStateMachine([(0, 0, 0, 0),
....:                           (0, 1, 1, 1),
....:                           (1, 1, 0, 0)],
....:                          determine_alphabets=False)
sage: fsm.is_complete()
Traceback (most recent call last):
...
ValueError: No input alphabet is given. Try calling determine_alphabets().
sage: fsm.input_alphabet = [0, 1]
sage: fsm.is_complete()
False
sage: fsm.add_transition((1, 1, 1, 1))
Transition from 1 to 1: 1|1
sage: fsm.is_complete()
True
sage: fsm.add_transition((0, 0, 1, 0))
Transition from 0 to 0: 1|0
sage: fsm.is_complete()
False
```

is_connected()

TESTS:

```
sage: FiniteStateMachine().is_connected()
Traceback (most recent call last):
...
NotImplementedError
```

is_deterministic()

Return whether the finite finite state machine is deterministic.

INPUT:

Nothing.

OUTPUT:

`True` or `False`.

A finite state machine is considered to be deterministic if each transition has input label of length one and for each pair (q, a) where q is a state and a is an element of the input alphabet, there is at most one transition from q with input label a. Furthermore, the finite state may not have more than one initial state.

EXAMPLES:

```
sage: fsm = FiniteStateMachine()
sage: fsm.add_transition(('A', 'B', 0, []))
Transition from 'A' to 'B': 0|-
sage: fsm.is_deterministic()
True
sage: fsm.add_transition(('A', 'C', 0, []))
Transition from 'A' to 'C': 0|-
sage: fsm.is_deterministic()
False
sage: fsm.add_transition(('A', 'B', [0,1], []))
Transition from 'A' to 'B': 0,1|-
sage: fsm.is_deterministic()
False
```

Check that trac ticket #18556 is fixed:

```
sage: Automaton().is_deterministic()
True
sage: Automaton(initial_states=[0]).is_deterministic()
True
sage: Automaton(initial_states=[0, 1]).is_deterministic()
False
```

is_monochromatic()

Checks whether the colors of all states are equal.

INPUT:

Nothing.

OUTPUT:

`True` or `False`.

EXAMPLES:

```
sage: G = transducers.GrayCode()
sage: [s.color for s in G.iter_states()]
[None, None, None]
sage: G.is_monochromatic()
True
sage: G.state(1).color = 'blue'
sage: G.is_monochromatic()
False
```

iter_final_states()

Returns an iterator of the final states.

INPUT:

Nothing.

OUTPUT:

An iterator over all initial states.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A', is_final=True)
sage: B = FSMState('B', is_initial=True)
sage: C = FSMState('C', is_final=True)
sage: F = FiniteStateMachine([(A, B), (A, C)])
sage: [s.label() for s in F.iter_final_states()]
['A', 'C']
```

iter_initial_states()

Returns an iterator of the initial states.

INPUT:

Nothing.

OUTPUT:

An iterator over all initial states.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A', is_initial=True)
sage: B = FSMState('B')
sage: F = FiniteStateMachine([(A, B, 1, 0)])
sage: [s.label() for s in F.iter_initial_states()]
['A']
```

iter_process(*input_tape=None*, *initial_state=None*, *process_iterator_class=None*, *iterator_type=None*, *automatic_output_type=False*, ***kwargs*)

This function returns an iterator for processing the input. See `process()` (which runs this iterator until the end) for more information.

INPUT:

- `iterator_type` – If `None` (default), then an instance of `FSMProcessIterator` is returned. If this is `'simple'` only an iterator over one output is returned (an exception is raised if this is not the case, i.e., if the process has branched).

See `process()` for a description of the other parameters.

OUTPUT:

An iterator.

EXAMPLES:

We can use `iter_process()` to deal with infinite words:

```
sage: inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:     initial_states=['A'], final_states=['A'])
sage: words.FibonacciWord()
word: 0100101001001010010100100101001001010010...
sage: it = inverter.iter_process(
....:     words.FibonacciWord(), iterator_type='simple')
sage: Words([0,1])(it)
word: 1011010110110101101011011010110110101101...
```

This can also be done by:

```
sage: inverter.iter_process(words.FibonacciWord(),
....:                       iterator_type='simple',
....:                       automatic_output_type=True)
word: 10110101101101011010110110101101101011101...
```

or even simpler by:

```
sage: inverter(words.FibonacciWord())
word: 10110101101101011010110110101101101011101...
```

To see what is going on, we use `iter_process()` without arguments:

```
sage: from itertools import islice
sage: it = inverter.iter_process(words.FibonacciWord())
sage: for current in islice(it, 4):
....:     print current
process (1 branch)
+ at state 'A'
+-- tape at 1, [[1]]
process (1 branch)
+ at state 'A'
+-- tape at 2, [[1, 0]]
process (1 branch)
+ at state 'A'
+-- tape at 3, [[1, 0, 1]]
process (1 branch)
+ at state 'A'
+-- tape at 4, [[1, 0, 1, 1]]
```

The following show the difference between using the `'simple'`-option and not using it. With this option, we have

```
sage: it = inverter.iter_process(input_tape=[0, 1, 1],
....:                            iterator_type='simple')
sage: for i, o in enumerate(it):
....:     print 'step %s: output %s' % (i, o)
step 0: output 1
step 1: output 0
step 2: output 0
```

So `iter_process()` is a generator expression which gives a new output letter in each step (and not more). In many cases this is sufficient.

Doing the same without the `'simple'`-option does not give the output directly; it has to be extracted first. On the other hand, additional information is presented:

```
sage: it = inverter.iter_process(input_tape=[0, 1, 1])
sage: for current in it:
....:     print current
process (1 branch)
+ at state 'A'
+-- tape at 1, [[1]]
process (1 branch)
+ at state 'A'
+-- tape at 2, [[1, 0]]
process (1 branch)
+ at state 'A'
+-- tape at 3, [[1, 0, 0]]
process (0 branches)
```

```
sage: it.result()
[Branch(accept=True, state='A', output=[1, 0, 0])]
```

One can see the growing of the output (the list of lists at the end of each entry).

Even if the transducer has transitions with empty or multiletter output, the simple iterator returns one new output letter in each step:

```
sage: T = Transducer([(0, 0, 0, []),
....:                 (0, 0, 1, [1]),
....:                 (0, 0, 2, [2, 2])],
....:                initial_states=[0])
sage: it = T.iter_process(input_tape=[0, 1, 2, 0, 1, 2],
....:                     iterator_type='simple')
sage: for i, o in enumerate(it):
....:     print 'step %s: output %s' % (i, o)
step 0: output 1
step 1: output 2
step 2: output 2
step 3: output 1
step 4: output 2
step 5: output 2
```

See also:

FiniteStateMachine.process(), Automaton.process(), Transducer.process(), __call__(), FSMProcessIterator.

iter_states()

Returns an iterator of the states.

INPUT:

Nothing.

OUTPUT:

An iterator of the states of the finite state machine.

EXAMPLES:

```
sage: FSM = Automaton([('1', '2', 1), ('2', '2', 0)])
sage: [s.label() for s in FSM.iter_states()]
['1', '2']
```

iter_transitions(*from_state=None*)

Returns an iterator of all transitions.

INPUT:

- from_state – (default: None) If from_state is given, then a list of transitions starting there is given.

OUTPUT:

An iterator of all transitions.

EXAMPLES:

```
sage: FSM = Automaton([('1', '2', 1), ('2', '2', 0)])
sage: [(t.from_state.label(), t.to_state.label())
....:     for t in FSM.iter_transitions('1')]
[('1', '2')]
```

```
sage: [(t.from_state.label(), t.to_state.label())
....:     for t in FSM.iter_transitions('2')]
[('2', '2')]
sage: [(t.from_state.label(), t.to_state.label())
....:     for t in FSM.iter_transitions()]
[('1', '2'), ('2', '2')]
```

kleene_star()

Compute the Kleene closure of this finite state machine.

OUTPUT:

A FiniteStateMachine of the same type as this finite state machine.

Assume that this finite state machine is an automaton recognizing the language $\mathcal{L}$. Then the Kleene star recognizes the language $\mathcal{L}^* = \{w_1 \ldots w_n \mid n \geq 0, w_j \in \mathcal{L} \text{ for all } j\}$.

Assume that this finite state machine is a transducer realizing a function f on some alphabet $\mathcal{L}$. Then the Kleene star realizes a function g on $\mathcal{L}^*$ with $g(w_1 \ldots w_n) = f(w_1) \ldots f(w_n)$.

EXAMPLES:

Kleene star of an automaton:

```
sage: A = automata.Word([0, 1])
sage: B = A.kleene_star()
sage: B.transitions()
[Transition from 0 to 1: 0|-,
 Transition from 2 to 0: -|-,
 Transition from 1 to 2: 1|-]
sage: from sage.combinat.finite_state_machine import (
....:     is_Automaton, is_Transducer)
sage: is_Automaton(B)
True
sage: [w for w in ([], [0, 1], [0, 1, 0], [0, 1, 0, 1], [0, 1, 1, 1])
....:  if B(w)]
[[],
 [0, 1],
 [0, 1, 0, 1]]
```

Kleene star of a transducer:

```
sage: T = Transducer([(0, 1, 0, 1), (0, 1, 1, 0)],
....:                initial_states=[0],
....:                final_states=[1])
sage: S = T.kleene_star()
sage: S.transitions()
[Transition from 0 to 1: 0|1,
 Transition from 0 to 1: 1|0,
 Transition from 1 to 0: -|-]
sage: is_Transducer(S)
True
sage: for w in ([], [0], [1], [0, 0], [0, 1]):
....:     print w, S.process(w)
[]     (True, 0, [])
[0]    [(True, 0, [1]), (True, 1, [1])]
[1]    [(True, 0, [0]), (True, 1, [0])]
[0, 0] [(True, 0, [1, 1]), (True, 1, [1, 1])]
[0, 1] [(True, 0, [1, 0]), (True, 1, [1, 0])]
```

Final output words are taken into account:

```
sage: T = Transducer([(0, 1, 0, 1)],
....:                initial_states=[0],
....:                final_states=[1])
sage: T.state(1).final_word_out = 2
sage: S = T.kleene_star()
sage: S.process([0, 0])
[(True, 0, [1, 2, 1, 2]), (True, 1, [1, 2, 1, 2])]
```

Final output words may lead to undesirable situations if initial states and final states coincide:

```
sage: T = Transducer(initial_states=[0], final_states=[0])
sage: T.state(0).final_word_out = 1
sage: T([])
[1]
sage: S = T.kleene_star()
sage: S([])
Traceback (most recent call last):
...
RuntimeError: State 0 is in an epsilon cycle (no input), but
output is written.
```

language (*max_length=None*, ***kwargs*)

Return all words that can be written by this transducer.

INPUT:

- `max_length` – an integer or `None` (default). Only output words which come from inputs of length at most `max_length` will be considered. If `None`, then this iterates over all possible words without length restrictions.
- `kwargs` – will be passed on to to the `process iterator`. See `process()` for a description.

OUTPUT:

An iterator.

EXAMPLES:

```
sage: NAF = Transducer([('I', 0, 0, None), ('I', 1, 1, None),
....:                   (0, 0, 0, 0), (0, 1, 1, 0),
....:                   (1, 0, 0, 1), (1, 2, 1, -1),
....:                   (2, 1, 0, 0), (2, 2, 1, 0)],
....:                  initial_states=['I'], final_states=[0],
....:                  input_alphabet=[0, 1])
sage: sorted(NAF.language(4),
....:        key=lambda o: (ZZ(o, base=2), len(o)))
[[], [0], [0, 0], [0, 0, 0],
 [1], [1, 0], [1, 0, 0],
 [0, 1], [0, 1, 0],
 [-1, 0, 1],
 [0, 0, 1],
 [1, 0, 1]]

sage: iterator = NAF.language()
sage: next(iterator)
[]
sage: next(iterator)
[0]
sage: next(iterator)
[1]
sage: next(iterator)
```

```
[0, 0]
sage: next(iterator)
[0, 1]
```

See also:

`Automaton.language()`, `process()`.

TESTS:

```
sage: T = Transducer([(0, 1, 0, 'a'), (1, 2, 1, 'b')],
....:                initial_states=[0], final_states=[0, 1, 2])
sage: T.determine_alphabets()
sage: list(T.language(2))
[[], ['a'], ['a', 'b']]
sage: list(T.language(3))
[[], ['a'], ['a', 'b']]
sage: from sage.combinat.finite_state_machine import  _FSMProcessIteratorAll_
sage: it = T.iter_process(
....:     process_iterator_class=_FSMProcessIteratorAll_,
....:     max_length=3,
....:     process_all_prefixes_of_input=True)
sage: for current in it:
....:     print current
....:     print "finished:", [branch.output for branch in it._finished_]
process (1 branch)
+ at state 1
+-- tape at 1, [['a']]
finished: [[]]
process (1 branch)
+ at state 2
+-- tape at 2, [['a', 'b']]
finished: [[], ['a']]
process (0 branches)
finished: [[], ['a'], ['a', 'b']]
```

latex_options(*coordinates=None*, *format_state_label=None*, *format_letter=None*, *format_transition_label=None*, *loop_where=None*, *initial_where=None*, *accepting_style=None*, *accepting_distance=None*, *accepting_where=None*, *accepting_show_empty=None*)

Set options for LaTeX output via `latex()` and therefore `view()`.

INPUT:

- `coordinates` – a dictionary or a function mapping labels of states to pairs interpreted as coordinates. If no coordinates are given, states a placed equidistantly on a circle of radius 3. See also `set_coordinates()`.
- `format_state_label` – a function mapping labels of states to a string suitable for typesetting in LaTeX's mathematics mode. If not given, `latex()` is used.
- `format_letter` – a function mapping letters of the input and output alphabets to a string suitable for typesetting in LaTeX's mathematics mode. If not given, `default_format_transition_label()` uses `latex()`.
- `format_transition_label` – a function mapping words over the input and output alphabets to a string suitable for typesetting in LaTeX's mathematics mode. If not given, `default_format_transition_label()` is used.
- `loop_where` – a dictionary or a function mapping labels of initial states to one of `'above'`,

'left', 'below', 'right'. If not given, 'above' is used.

- initial_where – a dictionary or a function mapping labels of initial states to one of 'above', 'left', 'below', 'right'. If not given, TikZ' default (currently 'left') is used.
- accepting_style – one of 'accepting by double' and 'accepting by arrow'. If not given, 'accepting by double' is used unless there are non-empty final output words.
- accepting_distance – a string giving a LaTeX length used for the length of the arrow leading from a final state. If not given, TikZ' default (currently '3ex') is used unless there are non-empty final output words, in which case '7ex' is used.
- accepting_where – a dictionary or a function mapping labels of final states to one of 'above', 'left', 'below', 'right'. If not given, TikZ' default (currently 'right') is used. If the final state has a final output word, it is also possible to give an angle in degrees.
- accepting_show_empty – if True the arrow of an empty final output word is labeled as well. Note that this implicitly implies accepting_style='accepting by arrow'. If not given, the default False is used.

OUTPUT:

Nothing.

As TikZ (cf. the Wikipedia article PGF/TikZ) is used to typeset the graphics, the syntax is oriented on TikZ' syntax.

This is a convenience function collecting all options for LaTeX output. All of its functionality can also be achieved by directly setting the attributes

- coordinates, format_label, loop_where, initial_where, and accepting_where of FSMState (here, format_label is a callable without arguments, everything else is a specific value);
- format_label of FSMTransition (format_label is a callable without arguments);
- format_state_label, format_letter, format_transition_label, accepting_style, accepting_distance, and accepting_show_empty of FiniteStateMachine.

This function, however, also (somewhat) checks its input and serves to collect documentation on all these options.

The function can be called several times, only those arguments which are not None are taken into account. By the same means, it can be combined with directly setting some attributes as outlined above.

EXAMPLES:

See also the section on *LaTeX output* in the introductory examples of this module.

```
sage: T = Transducer(initial_states=[4],
....:     final_states=[0, 3])
sage: for j in srange(4):
....:     T.add_transition(4, j, 0, [0, j])
....:     T.add_transition(j, 4, 0, [0, -j])
....:     T.add_transition(j, j, 0, 0)
Transition from 4 to 0: 0|0,0
Transition from 0 to 4: 0|0,0
Transition from 0 to 0: 0|0
Transition from 4 to 1: 0|0,1
Transition from 1 to 4: 0|0,-1
Transition from 1 to 1: 0|0
Transition from 4 to 2: 0|0,2
Transition from 2 to 4: 0|0,-2
```

```
Transition from 2 to 2: 0|0
Transition from 4 to 3: 0|0,3
Transition from 3 to 4: 0|0,-3
Transition from 3 to 3: 0|0
sage: T.add_transition(4, 4, 0, 0)
Transition from 4 to 4: 0|0
sage: T.state(3).final_word_out = [0, 0]
sage: T.latex_options(
....:     coordinates={4: (0, 0),
....:                  0: (-6, 3),
....:                  1: (-2, 3),
....:                  2: (2, 3),
....:                  3: (6, 3)},
....:     format_state_label=lambda x: r'\mathbf{%s}' % x,
....:     format_letter=lambda x: r'w_{%s}' % x,
....:     format_transition_label=lambda x:
....:         r"{\scriptstyle %s}" % T.default_format_transition_label(x),
....:     loop_where={4: 'below', 0: 'left', 1: 'above',
....:                 2: 'right', 3:'below'},
....:     initial_where=lambda x: 'above',
....:     accepting_style='accepting by double',
....:     accepting_distance='10ex',
....:     accepting_where={0: 'left', 3: 45}
....:     )
sage: T.state(4).format_label=lambda: r'\mathcal{I}'
sage: latex(T)
\begin{tikzpicture}[auto, initial text=, >=latex]
\node[state, initial, initial where=above] (v0) at (0.000000, 0.000000) {$\mathcal{I}$};
\node[state, accepting, accepting where=left] (v1) at (-6.000000, 3.000000) {$\mathbf{0}$};
\node[state, accepting, accepting where=45] (v2) at (6.000000, 3.000000) {$\mathbf{3}$};
\path[->] (v2.45.00) edge node[rotate=45.00, anchor=south] {$$ \mid {\scriptstyle w_{0} w_{0
\node[state] (v3) at (-2.000000, 3.000000) {$\mathbf{1}$};
\node[state] (v4) at (2.000000, 3.000000) {$\mathbf{2}$};
\path[->] (v1) edge[loop left] node[rotate=90, anchor=south] {${\scriptstyle w_{0}}\mid {\sc
\path[->] (v1.-21.57) edge node[rotate=-26.57, anchor=south] {${\scriptstyle w_{0}}\mid {\sc
\path[->] (v3) edge[loop above] node {${\scriptstyle w_{0}}\mid {\scriptstyle w_{0}}$} ();
\path[->] (v3.-51.31) edge node[rotate=-56.31, anchor=south] {${\scriptstyle w_{0}}\mid {\sc
\path[->] (v4) edge[loop right] node[rotate=90, anchor=north] {${\scriptstyle w_{0}}\mid {\s
\path[->] (v4.-118.69) edge node[rotate=56.31, anchor=north] {${\scriptstyle w_{0}}\mid {\sc
\path[->] (v2) edge[loop below] node {${\scriptstyle w_{0}}\mid {\scriptstyle w_{0}}$} ();
\path[->] (v2.-148.43) edge node[rotate=26.57, anchor=north] {${\scriptstyle w_{0}}\mid {\sc
\path[->] (v0.158.43) edge node[rotate=333.43, anchor=north] {${\scriptstyle w_{0}}\mid {\sc
\path[->] (v0.128.69) edge node[rotate=303.69, anchor=north] {${\scriptstyle w_{0}}\mid {\sc
\path[->] (v0.61.31) edge node[rotate=56.31, anchor=south] {${\scriptstyle w_{0}}\mid {\scri
\path[->] (v0.31.57) edge node[rotate=26.57, anchor=south] {${\scriptstyle w_{0}}\mid {\scri
\path[->] (v0) edge[loop below] node {${\scriptstyle w_{0}}\mid {\scriptstyle w_{0}}$} ();
\end{tikzpicture}
sage: view(T) # not tested
```

To actually see this, use the live documentation in the Sage notebook and execute the cells.

By changing some of the options, we get the following output:

```
sage: T.latex_options(
....:     format_transition_label=T.default_format_transition_label,
....:     accepting_style='accepting by arrow',
....:     accepting_show_empty=True
....:     )
sage: latex(T)
```

```
\begin{tikzpicture}[auto, initial text=, >=latex, accepting text=, accepting/.style=acceptin
\node[state, initial, initial where=above] (v0) at (0.000000, 0.000000) {$\mathcal{I}$};
\node[state] (v1) at (-6.000000, 3.000000) {$\mathbf{0}$};
\path[->] (v1.180.00) edge node[rotate=360.00, anchor=south] {$$ \mid \varepsilon$} ++(180.0
\node[state] (v2) at (6.000000, 3.000000) {$\mathbf{3}$};
\path[->] (v2.45.00) edge node[rotate=45.00, anchor=south] {$$ \mid w_{0} w_{0}$} ++(45.00:1
\node[state] (v3) at (-2.000000, 3.000000) {$\mathbf{1}$};
\node[state] (v4) at (2.000000, 3.000000) {$\mathbf{2}$};
\path[->] (v1) edge[loop left] node[rotate=90, anchor=south] {$w_{0}\mid w_{0}$} ();
\path[->] (v1.-21.57) edge node[rotate=-26.57, anchor=south] {$w_{0}\mid w_{0} w_{0}$} (v0.1
\path[->] (v3) edge[loop above] node {$w_{0}\mid w_{0}$} ();
\path[->] (v3.-51.31) edge node[rotate=-56.31, anchor=south] {$w_{0}\mid w_{0} w_{-1}$} (v0.
\path[->] (v4) edge[loop right] node[rotate=90, anchor=north] {$w_{0}\mid w_{0}$} ();
\path[->] (v4.-118.69) edge node[rotate=56.31, anchor=north] {$w_{0}\mid w_{0} w_{-2}$} (v0.
\path[->] (v2) edge[loop below] node {$w_{0}\mid w_{0}$} ();
\path[->] (v2.-148.43) edge node[rotate=26.57, anchor=north] {$w_{0}\mid w_{0} w_{-3}$} (v0.
\path[->] (v0.158.43) edge node[rotate=333.43, anchor=north] {$w_{0}\mid w_{0} w_{0}$} (v1.3
\path[->] (v0.128.69) edge node[rotate=303.69, anchor=north] {$w_{0}\mid w_{0} w_{1}$} (v3.2
\path[->] (v0.61.31) edge node[rotate=56.31, anchor=south] {$w_{0}\mid w_{0} w_{2}$} (v4.231
\path[->] (v0.31.57) edge node[rotate=26.57, anchor=south] {$w_{0}\mid w_{0} w_{3}$} (v2.201
\path[->] (v0) edge[loop below] node {$w_{0}\mid w_{0}$} ();
\end{tikzpicture}
sage: view(T) # not tested
```

TESTS:

```
sage: T.latex_options(format_state_label='Nothing')
Traceback (most recent call last):
...
TypeError: format_state_label must be callable.
sage: T.latex_options(format_letter='')
Traceback (most recent call last):
...
TypeError: format_letter must be callable.
sage: T.latex_options(format_transition_label='')
Traceback (most recent call last):
...
TypeError: format_transition_label must be callable.
sage: T.latex_options(loop_where=37)
Traceback (most recent call last):
...
TypeError: loop_where must be a callable or a
dictionary.
sage: T.latex_options(loop_where=lambda x: 'top')
Traceback (most recent call last):
...
ValueError: loop_where for 4 must be in ['below',
'right', 'above', 'left'].
sage: T.latex_options(initial_where=90)
Traceback (most recent call last):
...
TypeError: initial_where must be a callable or a
dictionary.
sage: T.latex_options(initial_where=lambda x: 'top')
Traceback (most recent call last):
...
ValueError: initial_where for 4 must be in ['below',
'right', 'above', 'left'].
```

```
sage: T.latex_options(accepting_style='fancy')
Traceback (most recent call last):
...
ValueError: accepting_style must be in ['accepting by
double', 'accepting by arrow'].
sage: T.latex_options(accepting_where=90)
Traceback (most recent call last):
...
TypeError: accepting_where must be a callable or a
dictionary.
sage: T.latex_options(accepting_where=lambda x: 'top')
Traceback (most recent call last):
...
ValueError: accepting_where for 0 must be in ['below',
'right', 'above', 'left'].
sage: T.latex_options(accepting_where={0: 'above', 3: 'top'})
Traceback (most recent call last):
...
ValueError: accepting_where for 3 must be a real number or
be in ['below', 'right', 'above', 'left'].
```

markov_chain_simplification()

Consider `self` as Markov chain with probabilities as input labels and simplify it.

INPUT:

Nothing.

OUTPUT:

Simplified version of `self`.

EXAMPLE:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_add_input
sage: T = Transducer([[1, 2, 1/4, 0], [1, -2, 1/4, 0], [1, -2, 1/2, 0],
....:                 [2, 2, 1/4, 1], [2, -2, 1/4, 1], [-2, -2, 1/4, 1],
....:                 [-2, 2, 1/4, 1], [2, 3, 1/2, 2], [-2, 3, 1/2, 2]],
....:                initial_states=[1],
....:                final_states=[3],
....:                on_duplicate_transition=duplicate_transition_add_input)
sage: T1 = T.markov_chain_simplification()
sage: sorted(T1.transitions())
[Transition from ((1,),) to ((2, -2),): 1|0,
 Transition from ((2, -2),) to ((2, -2),): 1/2|1,
 Transition from ((2, -2),) to ((3,),): 1/2|2]
```

merged_transitions()

Merges transitions which have the same `from_state`, `to_state` and `word_out` while adding their `word_in`.

INPUT:

Nothing.

OUTPUT:

A finite state machine with merged transitions. If no mergers occur, return `self`.

EXAMPLE:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_add_input
sage: T = Transducer([[1, 2, 1/4, 1], [1, -2, 1/4, 1], [1, -2, 1/2, 1],
....:                 [2, 2, 1/4, 1], [2, -2, 1/4, 1], [-2, -2, 1/4, 1],
....:                 [-2, 2, 1/4, 1], [2, 3, 1/2, 1], [-2, 3, 1/2, 1]],
....:                on_duplicate_transition=duplicate_transition_add_input)
sage: T1 = T.merged_transitions()
sage: T1 is T
False
sage: sorted(T1.transitions())
[Transition from -2 to -2: 1/4|1,
 Transition from -2 to 2: 1/4|1,
 Transition from -2 to 3: 1/2|1,
 Transition from 1 to 2: 1/4|1,
 Transition from 1 to -2: 3/4|1,
 Transition from 2 to -2: 1/4|1,
 Transition from 2 to 2: 1/4|1,
 Transition from 2 to 3: 1/2|1]
```

Applying the function again does not change the result:

```
sage: T2 = T1.merged_transitions()
sage: T2 is T1
True
```

moments_waiting_time(*test=<type 'bool'>*, *is_zero=None*, *expectation_only=False*)

If this finite state machine acts as a Markov chain, return the expectation and variance of the number of steps until first writing `True`.

INPUT:

- `test` – (default: `bool`) a callable deciding whether an output label is to be considered `True`. By default, the standard conversion to boolean is used.
- `is_zero` – (default: `None`) a callable deciding whether an expression for a probability is zero. By default, checking for zero is simply done by `is_zero()`. This parameter can be used to provide a more sophisticated check for zero, e.g. in the case of symbolic probabilities, see the examples below. This parameter is passed on to `is_Markov_chain()`. This parameter only affects the input of the Markov chain.
- `expectation_only` – (default: `False`) if set, the variance is not computed (in order to save time). By default, the variance is computed.

OUTPUT:

A dictionary (if `expectation_only=False`) consisting of

- `expectation`,
- `variance`.

Otherwise, just the expectation is returned (no dictionary for `expectation_only=True`).

Expectation and variance of the number of steps until first writing `True` (as determined by the parameter `test`).

ALGORITHM:

Relies on a (classical and easy) probabilistic argument, cf. [FGT1992], Eqns. (6) and (7).

For the variance, see [FHP2015], Section 2.

EXAMPLES:

1. The simplest example is to wait for the first 1 in a 0-1-string where both digits appear with probability $1/2$. In fact, the waiting time equals k if and only if the string starts with $0^{k-1}1$. This event occurs with probability 2^{-k}. Therefore, the expected waiting time and the variance are $\sum_{k\geq 1} k2^{-k} = 2$ and $\sum_{k\geq 1}(k-2)^2 2^{-k} = 2$:

```
sage: var('k')
k
sage: sum(k * 2^(-k), k, 1, infinity)
2
sage: sum((k-2)^2 * 2^(-k), k, 1, infinity)
2
```

We now compute the same expectation and variance by using a Markov chain:

```
sage: from sage.combinat.finite_state_machine import (
....:     duplicate_transition_add_input)
sage: T = Transducer(
....:     [(0, 0, 1/2, 0), (0, 0, 1/2, 1)],
....:     on_duplicate_transition=\
....:         duplicate_transition_add_input,
....:     initial_states=[0],
....:     final_states=[0])
sage: T.moments_waiting_time()
{'expectation': 2, 'variance': 2}
sage: T.moments_waiting_time(expectation_only=True)
2
```

In the following, we replace the output `0` by `-1` and demonstrate the use of the parameter `test`:

```
sage: T.delete_transition((0, 0, 1/2, 0))
sage: T.add_transition((0, 0, 1/2, -1))
Transition from 0 to 0: 1/2|-1
sage: T.moments_waiting_time(test=lambda x: x<0)
{'expectation': 2, 'variance': 2}
```

2. Make sure that the transducer is actually a Markov chain. Although this is checked by the code, unexpected behaviour may still occur if the transducer looks like a Markov chain. In the following example, we 'forget' to assign probabilities, but due to a coincidence, all 'probabilities' add up to one. Nevertheless, 0 is never written, so the expectation is 1.

```
sage: T = Transducer([(0, 0, 0, 0), (0, 0, 1, 1)],
....:                on_duplicate_transition=\
....:                    duplicate_transition_add_input,
....:                initial_states=[0],
....:                final_states=[0])
sage: T.moments_waiting_time()
{'expectation': 1, 'variance': 0}
```

3. If `True` is never written, the moments are `+Infinity`:

```
sage: T = Transducer([(0, 0, 1, 0)],
....:                on_duplicate_transition=\
```

```
....:                     duplicate_transition_add_input,
....:                 initial_states=[0],
....:                 final_states=[0])
sage: T.moments_waiting_time()
{'expectation': +Infinity, 'variance': +Infinity}
```

4. Let h and r be positive integers. We consider random strings of letters $1, \ldots, r$ where the letter j occurs with probability p_j. Let B be the random variable giving the first position of a block of h consecutive identical letters. Then

$$\mathbb{E}(B) = \frac{1}{\sum_{i=1}^{r} \frac{1}{p_i^{-1} + \cdots + p_i^{-h}}},$$

$$\mathbb{V}(B) = \frac{\sum_{i=1}^{r} \left(\frac{p_i + p_i^h}{1 - p_i^h} - 2h \frac{p_i^h (1 - p_i)}{(1 - p_i^h)^2} \right)}{\left(\sum_{i=1}^{r} \frac{1}{p_i^{-1} + \cdots + p_i^{-h}} \right)^2}$$

cf. [S1986], p. 62, or [FHP2015], Theorem 1. We now verify this with a transducer approach.

```
sage: def test(h, r):
....:     R = PolynomialRing(
....:             QQ,
....:             names=['p_%d' % j for j in range(r)])
....:     p = R.gens()
....:     def is_zero(polynomial):
....:         return polynomial in (sum(p) - 1) * R
....:     theory_expectation = 1/(sum(1/sum(p[j]^(-i)
....:                     for i in range(1, h+1))
....:                     for j in range(r)))
....:     theory_variance = sum(
....:         (p[i] + p[i]^h)/(1 - p[i]^h)
....:         - 2*h*p[i]^h * (1 - p[i])/(1 - p[i]^h)^2
....:         for i in range(r)
....:         ) * theory_expectation^2
....:     alphabet = range(r)
....:     counters = [
....:         transducers.CountSubblockOccurrences([j]*h,
....:                     alphabet)
....:         for j in alphabet]
....:     all_counter = counters[0].cartesian_product(
....:         counters[1:])
....:     adder = transducers.add(input_alphabet=[0, 1],
....:         number_of_operands=r)
....:     probabilities = Transducer(
....:        [(0, 0, p[j], j) for j in alphabet],
....:        initial_states=[0],
....:        final_states=[0],
....:        on_duplicate_transition=\
....:            duplicate_transition_add_input)
....:     chain = adder(all_counter(probabilities))
....:     result = chain.moments_waiting_time(
....:        is_zero=is_zero)
....:     return is_zero((result['expectation'] -
....:                 theory_expectation).numerator()) \
```

```
....:             and \
....:             is_zero((result['variance'] -
....:                  theory_variance).numerator())
sage: test(2, 2)
True
sage: test(2, 3)
True
sage: test(3, 3)
True
```

5. Consider the alphabet $\{0, \ldots, r-1\}$, some $1 \leq j \leq r$ and some $h \geq 1$. For some probabilities $p_0, \ldots, p_{r-1}$, we consider infinite words where the letters occur independently with the given probabilities. The random variable B_j is the first position n such that there exist j of the r letters having an h-run. The expectation of B_j is given in [FHP2015], Theorem 2. Here, we verify this result by using transducers:

```
sage: def test(h, r, j):
....:     R = PolynomialRing(
....:             QQ,
....:             names=['p_%d' % i for i in range(r)])
....:     p = R.gens()
....:     def is_zero(polynomial):
....:         return polynomial in (sum(p) - 1) * R
....:     alphabet = range(r)
....:     counters = [
....:         transducers.Wait([0, 1])(
....:             transducers.CountSubblockOccurrences(
....:                 [i]*h,
....:                 alphabet))
....:         for i in alphabet]
....:     all_counter = counters[0].cartesian_product(
....:         counters[1:])
....:     adder = transducers.add(input_alphabet=[0, 1],
....:         number_of_operands=r)
....:     threshold = transducers.map(
....:         f=lambda x: x >= j,
....:         input_alphabet=srange(r+1))
....:     probabilities = Transducer(
....:         [(0, 0, p[i], i) for i in alphabet],
....:         initial_states=[0],
....:         final_states=[0],
....:         on_duplicate_transition=\
....:             duplicate_transition_add_input)
....:     chain = threshold(adder(all_counter(
....:         probabilities)))
....:     result = chain.moments_waiting_time(
....:         is_zero=is_zero,
....:         expectation_only=True)
....:
....:     R_v = PolynomialRing(
....:             QQ,
....:             names=['p_%d' % i for i in range(r)])
....:     v = R_v.gens()
....:     S = 1/(1 - sum(v[i]/(1+v[i])
....:                    for i in range(r)))
....:     alpha = [(p[i] - p[i]^h)/(1 - p[i])
....:              for i in range(r)]
```

```
....:     gamma = [p[i]/(1 - p[i]) for i in range(r)]
....:     alphabet_set = set(alphabet)
....:     expectation = 0
....:     for q in range(j):
....:         for M in Subsets(alphabet_set, q):
....:             summand = S
....:             for i in M:
....:                 summand = summand.subs(
....:                     {v[i]: gamma[i]}) -\
....:                     summand.subs({v[i]: alpha[i]})
....:             for i in alphabet_set - set(M):
....:                 summand = summand.subs(
....:                     {v[i]: alpha[i]})
....:             expectation += summand
....:     return is_zero((result - expectation).\
....:             numerator())
sage: test(2, 3, 2)
True
```

REFERENCES:

TESTS:

Only Markov chains are acceptable:

```
sage: T = transducers.Identity([0, 1, 2])
sage: T.moments_waiting_time()
Traceback (most recent call last):
...
ValueError: Only Markov chains can compute
moments_waiting_time.
```

There must be a unique initial state:

```
sage: T = Transducer([(0, 1, 1, 1), (1, 0, 1, 0)],
....:                on_duplicate_transition=\
....:                    duplicate_transition_add_input)
sage: T.moments_waiting_time()
Traceback (most recent call last):
...
ValueError: Unique initial state is required.
```

Using 0 as initial state in this example, a 1 is written in the first step with probability 1, so the waiting time is always 1:

```
sage: T.state(0).is_initial = True
sage: T.moments_waiting_time()
{'expectation': 1, 'variance': 0}
```

Using both 0 and 1 as initial states again yields an error message:

```
sage: T.state(1).is_initial = True
sage: T.moments_waiting_time()
Traceback (most recent call last):
...
ValueError: Unique initial state is required.
```

Detection of infinite waiting time for symbolic probabilities:

```
sage: R.<p, q> = PolynomialRing(QQ)
sage: T = Transducer([(0, 0, p, 0), (0, 0, q, 0)],
....:                initial_states=[0],
....:                on_duplicate_transition=\
....:                    duplicate_transition_add_input)
sage: T.moments_waiting_time(
....:     is_zero=lambda e: e in (p + q - 1)*R)
{'expectation': +Infinity, 'variance': +Infinity}
```

number_of_words (*variable=n*, *base_ring=Algebraic Field*)

Return the number of successful input words of given length.

INPUT:

- `variable` – a symbol denoting the length of the words, by default n.
- `base_ring` – Ring (default: `QQbar`) in which to compute the eigenvalues.

OUTPUT:

A symbolic expression.

EXAMPLES:

```
sage: NAFpm = Automaton([(0, 0, 0), (0, 1, 1),
....:                    (0, 1, -1), (1, 0, 0)],
....:                   initial_states=[0],
....:                   final_states=[0, 1])
sage: N = NAFpm.number_of_words(); N
4/3*2^n - 1/3*(-1)^n
sage: all(len(list(NAFpm.language(s)))
....:     - len(list(NAFpm.language(s-1))) == N.subs(n=s)
....:     for s in srange(1, 6))
True
```

An example with non-rational eigenvalues. By default, eigenvalues are elements of the `field of algebraic numbers`.

```
sage: NAFp = Automaton([(0, 0, 0), (0, 1, 1),  (1, 0, 0)],
....:                  initial_states=[0],
....:                  final_states=[0, 1])
sage: N = NAFp.number_of_words(); N
1.170820393249937?*1.618033988749895?^n
- 0.1708203932499369?*(-0.618033988749895?)^n
sage: all(len(list(NAFp.language(s)))
....:     - len(list(NAFp.language(s-1))) == N.subs(n=s)
....:     for s in srange(1, 6))
True
```

We specify a suitable `base_ring` to obtain a radical expression. To do so, we first compute the characteristic polynomial and then construct a number field generated by its roots.

```
sage: M = NAFp.adjacency_matrix(entry=lambda t: 1)
sage: M.characteristic_polynomial()
x^2 - x - 1
sage: R.<phi> = NumberField(x^2-x-1, embedding=1.6)
sage: N = NAFp.number_of_words(base_ring=R); N
1/10*(1/2*sqrt(5) + 1/2)^n*(3*sqrt(5) + 5)
- 1/10*(-1/2*sqrt(5) + 1/2)^n*(3*sqrt(5) - 5)
sage: all(len(list(NAFp.language(s)))
....:     - len(list(NAFp.language(s-1))) == N.subs(n=s)
```

```
....:     for s in srange(1, 6))
True
```

In this special case, we might also use the constant `golden_ratio`:

```
sage: R.<phi> = NumberField(x^2-x-1, embedding=golden_ratio)
sage: N = NAFp.number_of_words(base_ring=R); N
1/5*(3*golden_ratio + 1)*golden_ratio^n
- 1/5*(3*golden_ratio - 4)*(-golden_ratio + 1)^n
sage: all(len(list(NAFp.language(s)))
....:     - len(list(NAFp.language(s-1))) == N.subs(n=s)
....:     for s in srange(1, 6))
True
```

The adjacency matrix of the following example is a Jordan matrix of size 3 to the eigenvalue 4:

```
sage: J3 = Automaton([(0, 1, -1), (1, 2, -1)],
....:     initial_states=[0],
....:     final_states=[0, 1, 2])
sage: for i in range(3):
....:     for j in range(4):
....:         new_transition = J3.add_transition(i, i, j)
sage: J3.adjacency_matrix(entry=lambda t: 1)
[4 1 0]
[0 4 1]
[0 0 4]
sage: N = J3.number_of_words(); N
1/2*4^(n - 2)*(n - 1)*n + 4^(n - 1)*n + 4^n
sage: all(len(list(J3.language(s)))
....:     - len(list(J3.language(s-1))) == N.subs(n=s)
....:     for s in range(1, 6))
True
```

Here is an automaton without cycles, so with eigenvalue 0.

```
sage: A = Automaton([(j, j+1, 0) for j in range(3)],
....:               initial_states=[0],
....:               final_states=range(3))
sage: A.number_of_words()
1/2*0^(n - 2)*(n - 1)*n + 0^(n - 1)*n + 0^n
```

TESTS:

```
sage: A = Automaton([(0, 0, 0), (0, 1, 0)],
....:               initial_states=[0])
sage: A.number_of_words()
Traceback (most recent call last):
...
NotImplementedError: Finite State Machine must be deterministic.
```

on_duplicate_transition (*old_transition*, *new_transition*)

Which function to call when a duplicate transition is inserted.

It can be set by the parameter `on_duplicate_transition` when initializing a finite state machine, see `FiniteStateMachine`.

See also:

`FiniteStateMachine`, `is_Markov_chain()`, `markov_chain_simplification()`.

output_alphabet
A list of letters representing the output alphabet of the finite state machine.

It can be set by the parameter `output_alphabet` when initializing a finite state machine, see `FiniteStateMachine`.

It can also be set by the method `determine_alphabets()`.

See also:

`FiniteStateMachine`, `determine_alphabets()`, `input_alphabet`.

output_projection()
Returns a automaton where the input of each transition of self is deleted and the new input is the original output.

INPUT:

Nothing

OUTPUT:

An automaton.

EXAMPLES:

```
sage: F = FiniteStateMachine([('A', 'B', 0, 1), ('A', 'A', 1, 1),
....:                         ('B', 'B', 1, 0)])
sage: G = F.output_projection()
sage: G.transitions()
[Transition from 'A' to 'B': 1|-,
 Transition from 'A' to 'A': 1|-,
 Transition from 'B' to 'B': 0|-]
```

Final output words are also considered correctly:

```
sage: H = Transducer([('A', 'B', 0, 1), ('A', 'A', 1, 1),
....:                 ('B', 'B', 1, 0), ('A', ('final', 0), 0, 0)],
....:                final_states=['A', 'B'])
sage: H.state('B').final_word_out = 2
sage: J = H.output_projection()
sage: J.states()
['A', 'B', ('final', 0), ('final', 1)]
sage: J.transitions()
[Transition from 'A' to 'B': 1|-,
 Transition from 'A' to 'A': 1|-,
 Transition from 'A' to ('final', 0): 0|-,
 Transition from 'B' to 'B': 0|-,
 Transition from 'B' to ('final', 1): 2|-]
sage: J.final_states()
['A', ('final', 1)]
```

plot()
Plots a graph of the finite state machine with labeled vertices and labeled edges.

INPUT:

Nothing.

OUTPUT:

A plot of the graph of the finite state machine.

TESTS:

```
sage: FiniteStateMachine([('A', 'A', 0)]).plot()
Graphics object consisting of 3 graphics primitives
```

predecessors(*state*, *valid_input=None*)

Lists all predecessors of a state.

INPUT:

- `state` – the state from which the predecessors should be listed.
- `valid_input` – If `valid_input` is a list, then we only consider transitions whose input labels are contained in `valid_input`. `state` has to be a `FSMState` (not a label of a state). If input labels of length larger than 1 are used, then `valid_input` has to be a list of lists.

OUTPUT:

A list of states.

EXAMPLES:

```
sage: A = Transducer([('I', 'A', 'a', 'b'), ('I', 'B', 'b', 'c'),
....:                 ('I', 'C', 'c', 'a'), ('A', 'F', 'b', 'a'),
....:                 ('B', 'F', ['c', 'b'], 'b'), ('C', 'F', 'a', 'c')],
....:                initial_states=['I'], final_states=['F'])
sage: A.predecessors(A.state('A'))
['A', 'I']
sage: A.predecessors(A.state('F'), valid_input=['b', 'a'])
['F', 'C', 'A', 'I']
sage: A.predecessors(A.state('F'), valid_input=[['c', 'b'], 'a'])
['F', 'C', 'B']
```

prepone_output()

For all paths, shift the output of the path from one transition to the earliest possible preceeding transition of the path.

INPUT:

Nothing.

OUTPUT:

Nothing.

Apply the following to each state s (except initial states) of the finite state machine as often as possible:

If the letter a is a prefix of the output label of all transitions from s (including the final output of s), then remove it from all these labels and append it to all output labels of all transitions leading to s.

We assume that the states have no output labels, but final outputs are allowed.

EXAMPLES:

```
sage: A = Transducer([('A', 'B', 1, 1),
....:                 ('B', 'B', 0, 0),
....:                 ('B', 'C', 1, 0)],
....:                initial_states=['A'],
....:                final_states=['C'])
sage: A.prepone_output()
sage: A.transitions()
[Transition from 'A' to 'B': 1|1,0,
 Transition from 'B' to 'B': 0|0,
 Transition from 'B' to 'C': 1|-]
```

```
sage: B = Transducer([('A', 'B', 0, 1),
....:                 ('B', 'C', 1, [1, 1]),
....:                 ('B', 'C', 0, 1)],
....:                initial_states=['A'],
....:                final_states=['C'])
sage: B.prepone_output()
sage: B.transitions()
[Transition from 'A' to 'B': 0|1,1,
 Transition from 'B' to 'C': 1|1,
 Transition from 'B' to 'C': 0|-]
```

If initial states are not labeled as such, unexpected results may be obtained:

```
sage: C = Transducer([(0,1,0,0)])
sage: C.prepone_output()
verbose 0 (...: finite_state_machine.py, prepone_output)
All transitions leaving state 0 have an output label with
prefix 0.  However, there is no inbound transition and it
is not an initial state. This routine (possibly called by
simplification) therefore erased this prefix from all
outbound transitions.
sage: C.transitions()
[Transition from 0 to 1: 0|-]
```

Also the final output of final states can be changed:

```
sage: T = Transducer([('A', 'B', 0, 1),
....:                 ('B', 'C', 1, [1, 1]),
....:                 ('B', 'C', 0, 1)],
....:                initial_states=['A'],
....:                final_states=['B'])
sage: T.state('B').final_word_out = [1]
sage: T.prepone_output()
sage: T.transitions()
[Transition from 'A' to 'B': 0|1,1,
 Transition from 'B' to 'C': 1|1,
 Transition from 'B' to 'C': 0|-]
sage: T.state('B').final_word_out
[]

sage: S = Transducer([('A', 'B', 0, 1),
....:                 ('B', 'C', 1, [1, 1]),
....:                 ('B', 'C', 0, 1)],
....:                initial_states=['A'],
....:                final_states=['B'])
sage: S.state('B').final_word_out = [0]
sage: S.prepone_output()
sage: S.transitions()
[Transition from 'A' to 'B': 0|1,
 Transition from 'B' to 'C': 1|1,1,
 Transition from 'B' to 'C': 0|1]
sage: S.state('B').final_word_out
[0]
```

Output labels do not have to be hashable:

```
sage: C = Transducer([(0, 1, 0, []),
....:                 (1, 0, 0, [vector([0, 0]), 0]),
....:                 (1, 1, 1, [vector([0, 0]), 1]),
```

```
....:                  (0, 0, 1, 0)],
....:                 determine_alphabets=False,
....:                 initial_states=[0])
sage: C.prepone_output()
sage: sorted(C.transitions())
[Transition from 0 to 1: 0|(0, 0),
 Transition from 0 to 0: 1|0,
 Transition from 1 to 0: 0|0,
 Transition from 1 to 1: 1|1,(0, 0)]
```

process(**args*, ***kwargs*)

Returns whether the finite state machine accepts the input, the state where the computation stops and which output is generated.

INPUT:

- `input_tape` – the input tape can be a list or an iterable with entries from the input alphabet. If we are working with a multi-tape machine (see parameter `use_multitape_input` and notes below), then the tape is a list or tuple of tracks, each of which can be a list or an iterable with entries from the input alphabet.
- `initial_state` or `initial_states` – the initial state(s) in which the machine starts. Either specify a single one with `initial_state` or a list of them with `initial_states`. If both are given, `initial_state` will be appended to `initial_states`. If neither is specified, the initial states of the finite state machine are taken.
- `list_of_outputs` – (default: `None`) a boolean or `None`. If `True`, then the outputs are given in list form (even if we have no or only one single output). If `False`, then the result is never a list (an exception is raised if the result cannot be returned). If `list_of_outputs=None`, the method determines automatically what to do (e.g. if a non-deterministic machine returns more than one path, then the output is returned in list form).
- `only_accepted` – (default: `False`) a boolean. If set, then the first argument in the output is guaranteed to be `True` (if the output is a list, then the first argument of each element will be `True`).
- `always_include_output` – if set (not by default), always include the output. This is inconsequential for a `FiniteStateMachine`, but can be used in derived classes where the output is suppressed by default, cf. `Automaton.process()`.
- `format_output` – a function that translates the written output (which is in form of a list) to something more readable. By default (`None`) identity is used here.
- `check_epsilon_transitions` – (default: `True`) a boolean. If `False`, then epsilon transitions are not taken into consideration during process.
- `write_final_word_out` – (default: `True`) a boolean specifying whether the final output words should be written or not.
- `use_multitape_input` – (default: `False`) a boolean. If `True`, then the multi-tape mode of the process iterator is activated. See also the notes below for multi-tape machines.
- `process_all_prefixes_of_input` – (default: `False`) a boolean. If `True`, then each prefix of the input word is processed (instead of processing the whole input word at once). Consequently, there is an output generated for each of these prefixes.
- `process_iterator_class` – (default: `None`) a class inherited from `FSMProcessIterator`. If `None`, then `FSMProcessIterator` is taken. An instance of this class is created and is used during the processing.

- `automatic_output_type` – (default: `False`) a boolean. If set and the input has a parent, then the output will have the same parent. If the input does not have a parent, then the output will be of the same type as the input.

OUTPUT:

A triple (or a list of triples, cf. parameter `list_of_outputs`), where

- the first entry is `True` if the input string is accepted,
- the second gives the reached state after processing the input tape (This is a state with label `None` if the input could not be processed, i.e., if at one point no transition to go on could be found.), and
- the third gives a list of the output labels written during processing (in the case the finite state machine runs as transducer).

Note that in the case the finite state machine is not deterministic, all possible paths are taken into account.

This function uses an iterator which, in its simplest form, goes from one state to another in each step. To decide which way to go, it uses the input words of the outgoing transitions and compares them to the input tape. More precisely, in each step, the iterator takes an outgoing transition of the current state, whose input label equals the input letter of the tape. The output label of the transition, if present, is written on the output tape.

If the choice of the outgoing transition is not unique (i.e., we have a non-deterministic finite state machine), all possibilites are followed. This is done by splitting the process into several branches, one for each of the possible outgoing transitions.

The process (iteration) stops if all branches are finished, i.e., for no branch, there is any transition whose input word coincides with the processed input tape. This can simply happen when the entire tape was read.

Also see `__call__()` for a version of `process()` with shortened output.

Internally this function creates and works with an instance of `FSMProcessIterator`. This iterator can also be obtained with `iter_process()`.

If working with multi-tape finite state machines, all input words of transitions are words of k-tuples of letters. Moreover, the input tape has to consist of k tracks, i.e., be a list or tuple of k iterators, one for each track.

Warning: Working with multi-tape finite state machines is still experimental and can lead to wrong outputs.

EXAMPLES:

```
sage: binary_inverter = FiniteStateMachine({'A': [('A', 0, 1), ('A', 1, 0)]},
....:                                      initial_states=['A'], final_states=['A'])
sage: binary_inverter.process([0, 1, 0, 0, 1, 1])
(True, 'A', [1, 0, 1, 1, 0, 0])
```

Alternatively, we can invoke this function by:

```
sage: binary_inverter([0, 1, 0, 0, 1, 1])
(True, 'A', [1, 0, 1, 1, 0, 0])
```

Below we construct a finite state machine which tests if an input is a non-adjacent form, i.e., no two neighboring letters are both nonzero (see also the example on *non-adjacent forms* in the documentation of the module *Finite State Machines, Automata, Transducers*):

```
sage: NAF = FiniteStateMachine(
....:     {'_': [('_', 0), (1, 1)], 1: [('_', 0)]},
....:     initial_states=['_'], final_states=['_', 1])
```

```
sage: [NAF.process(w)[0] for w in [[0], [0, 1], [1, 1], [0, 1, 0, 1],
....:                              [0, 1, 1, 1, 0], [1, 0, 0, 1, 1]]]
[True, True, False, True, False, False]
```

Working only with the first component (i.e., returning whether accepted or not) usually corresponds to using the more specialized class `Automaton`.

Non-deterministic finite state machines can be handeled as well.

```
sage: T = Transducer([(0, 1, 0, 0), (0, 2, 0, 0)],
....:     initial_states=[0])
sage: T.process([0])
[(False, 1, [0]), (False, 2, [0])]
```

Here is another non-deterministic finite state machine. Note that we use `format_output` (see `FSMProcessIterator`) to convert the written outputs (all characters) to strings.

```
sage: T = Transducer([(0, 1, [0, 0], 'a'), (0, 2, [0, 0, 1], 'b'),
....:                 (0, 1, 1, 'c'), (1, 0, [], 'd'),
....:                 (1, 1, 1, 'e')],
....:                initial_states=[0], final_states=[0, 1])
sage: T.process([0], format_output=lambda o: ''.join(o))
(False, None, None)
sage: T.process([0, 0], format_output=lambda o: ''.join(o))
[(True, 0, 'ad'), (True, 1, 'a')]
sage: T.process([1], format_output=lambda o: ''.join(o))
[(True, 0, 'cd'), (True, 1, 'c')]
sage: T.process([1, 1], format_output=lambda o: ''.join(o))
[(True, 0, 'cdcd'), (True, 0, 'ced'),
 (True, 1, 'cdc'), (True, 1, 'ce')]
sage: T.process([0, 0, 1], format_output=lambda o: ''.join(o))
[(True, 0, 'adcd'), (True, 0, 'aed'),
 (True, 1, 'adc'), (True, 1, 'ae'), (False, 2, 'b')]
sage: T.process([0, 0, 1], format_output=lambda o: ''.join(o),
....:           only_accepted=True)
[(True, 0, 'adcd'), (True, 0, 'aed'),
 (True, 1, 'adc'), (True, 1, 'ae')]
```

A simple example of a multi-tape finite state machine is the following: It writes the length of the first tape many letters `a` and then the length of the second tape many letters `b`:

```
sage: M = FiniteStateMachine([(0, 0, (1, None), 'a'),
....:                         (0, 1, [], []),
....:                         (1, 1, (None, 1), 'b')],
....:                        initial_states=[0],
....:                        final_states=[1])
sage: M.process(([1, 1], [1]), use_multitape_input=True)
(True, 1, ['a', 'a', 'b'])
```

See also:

`Automaton.process()`, `Transducer.process()`, `iter_process()`, `__call__()`, `FSMProcessIterator`.

TESTS:

```
sage: T = Transducer([(0, 1, [0, 0], 0), (0, 2, [0, 0, 1], 0),
....:                 (0, 1, 1, 2), (1, 0, [], 1), (1, 1, 1, 3)],
....:     initial_states=[0], final_states=[0, 1])
sage: T.process([0])
```

```
(False, None, None)
sage: T.process([0, 0])
[(True, 0, [0, 1]), (True, 1, [0])]
sage: T.process([1])
[(True, 0, [2, 1]), (True, 1, [2])]
sage: T.process([1, 1])
[(True, 0, [2, 1, 2, 1]), (True, 0, [2, 3, 1]),
 (True, 1, [2, 1, 2]), (True, 1, [2, 3])]

sage: F = FiniteStateMachine([(0, 0, 0, 0)],
....:                        initial_states=[0])
sage: F.process([0], only_accepted=True)
[]
sage: F.process([0], only_accepted=True, list_of_outputs=False)
Traceback (most recent call last):
...
ValueError: No accepting output was found but according to the
given options, an accepting output should be returned. Change
only_accepted and/or list_of_outputs options.
sage: F.process([0], only_accepted=True, list_of_outputs=True)
[]
sage: F.process([0], only_accepted=False)
(False, 0, [0])
sage: F.process([0], only_accepted=False, list_of_outputs=False)
(False, 0, [0])
sage: F.process([0], only_accepted=False, list_of_outputs=True)
[(False, 0, [0])]
sage: F.process([1], only_accepted=True)
[]
sage: F.process([1], only_accepted=True, list_of_outputs=False)
Traceback (most recent call last):
...
ValueError: No accepting output was found but according to the
given options, an accepting output should be returned. Change
only_accepted and/or list_of_outputs options.
sage: F.process([1], only_accepted=True, list_of_outputs=True)
[]
sage: F.process([1], only_accepted=False)
(False, None, None)
sage: F.process([1], only_accepted=False, list_of_outputs=False)
(False, None, None)
sage: F.process([1], only_accepted=False, list_of_outputs=True)
[]

sage: F = FiniteStateMachine([(0, 1, 1, 'a'), (0, 2, 2, 'b')],
....:                        initial_states=[0],
....:                        final_states=[1])
sage: A = Automaton([(0, 1, 1), (0, 2, 2)],
....:               initial_states=[0],
....:               final_states=[1])
sage: T = Transducer([(0, 1, 1, 'a'), (0, 2, 2, 'b')],
....:                initial_states=[0],
....:                final_states=[1])
sage: F.process([1])
(True, 1, ['a'])
sage: A.process([1])
(True, 1)
sage: T.process([1])
```

```
(True, 1, ['a'])
sage: F.process([2])
(False, 2, ['b'])
sage: A.process([2])
(False, 2)
sage: T.process([2])
(False, 2, ['b'])
sage: F.process([3])
(False, None, None)
sage: A.process([3])
(False, None)
sage: T.process([3])
(False, None, None)
```

product_FiniteStateMachine(*other*, *function*, *new_input_alphabet=None*, *only_accessible_components=True*, *final_function=None*, *new_class=None*)

Returns a new finite state machine whose states are d-tuples of states of the original finite state machines.

INPUT:

- `other` – a finite state machine (for $d = 2$) or a list (or iterable) of $d - 1$ finite state machines.
- `function` has to accept d transitions from A_j to B_j for $j \in \{1, \ldots, d\}$ and returns a pair `(word_in, word_out)` which is the label of the transition $A = (A_1, \ldots, A_d)$ to $B = (B_1, \ldots, B_d)$. If there is no transition from A to B, then `function` should raise a `LookupError`.
- `new_input_alphabet` (optional) – the new input alphabet as a list.
- `only_accessible_components` – If `True` (default), then the result is piped through `accessible_components()`. If no `new_input_alphabet` is given, it is determined by `determine_alphabets()`.
- `final_function` – A function mapping d final states of the original finite state machines to the final output of the corresponding state in the new finite state machine. By default, the final output is the empty word if both final outputs of the constituent states are empty; otherwise, a `ValueError` is raised.
- `new_class` – Class of the new finite state machine. By default (`None`), the class of `self` is used.

OUTPUT:

A finite state machine whose states are d-tuples of states of the original finite state machines. A state is initial or final if all constituent states are initial or final, respectively.

The labels of the transitions are defined by `function`.

The final output of a final state is determined by calling `final_function` on the constituent states.

The color of a new state is the tuple of colors of the constituent states of `self` and `other`. However, if all constituent states have color `None`, then the state has color `None`, too.

EXAMPLES:

```
sage: F = Automaton([('A', 'B', 1), ('A', 'A', 0), ('B', 'A', 2)],
....:               initial_states=['A'], final_states=['B'],
....:               determine_alphabets=True)
sage: G = Automaton([(1, 1, 1)], initial_states=[1], final_states=[1])
sage: def addition(transition1, transition2):
....:     return (transition1.word_in[0] + transition2.word_in[0],
....:             None)
sage: H = F.product_FiniteStateMachine(G, addition, [0, 1, 2, 3], only_accessible_components
```

```
sage: H.transitions()
[Transition from ('A', 1) to ('B', 1): 2|-,
 Transition from ('A', 1) to ('A', 1): 1|-,
 Transition from ('B', 1) to ('A', 1): 3|-]
sage: [s.color for s in H.iter_states()]
[None, None]
sage: H1 = F.product_FiniteStateMachine(G, addition, [0, 1, 2, 3], only_accessible_component
sage: H1.states()[0].label()[0] is F.states()[0]
True
sage: H1.states()[0].label()[1] is G.states()[0]
True

sage: F = Automaton([(0,1,1/4), (0,0,3/4), (1,1,3/4), (1,0,1/4)],
....:               initial_states=[0] )
sage: G = Automaton([(0,0,1), (1,1,3/4), (1,0,1/4)],
....:               initial_states=[0] )
sage: H = F.product_FiniteStateMachine(
....:         G, lambda t1,t2: (t1.word_in[0]*t2.word_in[0], None))
sage: H.states()
[(0, 0), (1, 0)]

sage: F = Automaton([(0,1,1/4), (0,0,3/4), (1,1,3/4), (1,0,1/4)],
....:               initial_states=[0] )
sage: G = Automaton([(0,0,1), (1,1,3/4), (1,0,1/4)],
....:               initial_states=[0] )
sage: H = F.product_FiniteStateMachine(G,
....:                                  lambda t1,t2: (t1.word_in[0]*t2.word_in[0], None),
....:                                  only_accessible_components=False)
sage: H.states()
[(0, 0), (1, 0), (0, 1), (1, 1)]
```

Also final output words are considered according to the function `final_function`:

```
sage: F = Transducer([(0, 1, 0, 1), (1, 1, 1, 1), (1, 1, 0, 1)],
....:                final_states=[1])
sage: F.state(1).final_word_out = 1
sage: G = Transducer([(0, 0, 0, 1), (0, 0, 1, 0)], final_states=[0])
sage: G.state(0).final_word_out = 1
sage: def minus(t1, t2):
....:     return (t1.word_in[0] - t2.word_in[0],
....:             t1.word_out[0] - t2.word_out[0])
sage: H = F.product_FiniteStateMachine(G, minus)
Traceback (most recent call last):
...
ValueError: A final function must be given.
sage: def plus(s1, s2):
....:     return s1.final_word_out[0] + s2.final_word_out[0]
sage: H = F.product_FiniteStateMachine(G, minus,
....:                                  final_function=plus)
sage: H.final_states()
[(1, 0)]
sage: H.final_states()[0].final_word_out
[2]
```

Products of more than two finite state machines are also possible:

```
sage: def plus(s1, s2, s3):
....:     if s1.word_in == s2.word_in == s3.word_in:
....:         return (s1.word_in,
```

```
....:                   sum(s.word_out[0] for s in (s1, s2, s3)))
....:     else:
....:         raise LookupError
sage: T0 = transducers.CountSubblockOccurrences([0, 0], [0, 1, 2])
sage: T1 = transducers.CountSubblockOccurrences([1, 1], [0, 1, 2])
sage: T2 = transducers.CountSubblockOccurrences([2, 2], [0, 1, 2])
sage: T = T0.product_FiniteStateMachine([T1, T2], plus)
sage: T.transitions()
[Transition from ((), (), ()) to ((0,), (), ()): 0|0,
 Transition from ((), (), ()) to ((), (1,), ()): 1|0,
 Transition from ((), (), ()) to ((), (), (2,)): 2|0,
 Transition from ((0,), (), ()) to ((0,), (), ()): 0|1,
 Transition from ((0,), (), ()) to ((), (1,), ()): 1|0,
 Transition from ((0,), (), ()) to ((), (), (2,)): 2|0,
 Transition from ((), (1,), ()) to ((0,), (), ()): 0|0,
 Transition from ((), (1,), ()) to ((), (1,), ()): 1|1,
 Transition from ((), (1,), ()) to ((), (), (2,)): 2|0,
 Transition from ((), (), (2,)) to ((0,), (), ()): 0|0,
 Transition from ((), (), (2,)) to ((), (1,), ()): 1|0,
 Transition from ((), (), (2,)) to ((), (), (2,)): 2|1]
sage: T([0, 0, 1, 1, 2, 2, 0, 1, 2, 2])
[0, 1, 0, 1, 0, 1, 0, 0, 0, 1]
```

`other` can also be an iterable:

```
sage: T == T0.product_FiniteStateMachine(iter([T1, T2]), plus)
True
```

TESTS:

Check that colors are correctly dealt with. In particular, the new colors have to be hashable such that `Automaton.determinisation()` does not fail:

```
sage: A = Automaton([[0, 0, 0]], initial_states=[0])
sage: B = A.product_FiniteStateMachine(A,
....:                                  lambda t1, t2: (0, None))
sage: B.states()[0].color is None
True
sage: B.determinisation()
Automaton with 1 state
```

Check handling of the parameter `other`:

```
sage: A.product_FiniteStateMachine(None, plus)
Traceback (most recent call last):
...
ValueError: other must be a finite state machine or a list
of finite state machines.
sage: A.product_FiniteStateMachine([None], plus)
Traceback (most recent call last):
...
ValueError: other must be a finite state machine or a list
of finite state machines.
```

Test whether `new_class` works:

```
sage: T = Transducer()
sage: type(T.product_FiniteStateMachine(T, None))
<class 'sage.combinat.finite_state_machine.Transducer'>
sage: type(T.product_FiniteStateMachine(T, None,
```

```
....:       new_class=Automaton))
<class 'sage.combinat.finite_state_machine.Automaton'>
```

Check that isolated vertices are kept (trac ticket #16762):

```
sage: F = Transducer(initial_states=[0])
sage: F.add_state(1)
1
sage: G = Transducer(initial_states=['A'])
sage: F.product_FiniteStateMachine(G, None).states()
[(0, 'A')]
sage: F.product_FiniteStateMachine(
....:     G, None, only_accessible_components=False).states()
[(0, 'A'), (1, 'A')]
```

projection(*what='input'*)

Returns an Automaton which transition labels are the projection of the transition labels of the input.

INPUT:

- `what` – (default: `input`) either `input` or `output`.

OUTPUT:

An automaton.

EXAMPLES:

```
sage: F = FiniteStateMachine([('A', 'B', 0, 1), ('A', 'A', 1, 1),
....:                         ('B', 'B', 1, 0)])
sage: G = F.projection(what='output')
sage: G.transitions()
[Transition from 'A' to 'B': 1|-,
 Transition from 'A' to 'A': 1|-,
 Transition from 'B' to 'B': 0|-]
```

quotient(*classes*)

Constructs the quotient with respect to the equivalence classes.

INPUT:

- `classes` is a list of equivalence classes of states.

OUTPUT:

A finite state machine.

The labels of the new states are tuples of states of the `self`, corresponding to `classes`.

Assume that c is a class, and a and b are states in c. Then there is a bijection φ between the transitions from a and the transitions from b with the following properties: if $\varphi(t_a) = t_b$, then

- $t_a.word_{in} = t_b.word_{in}$,
- $t_a.word_{out} = t_b.word_{out}$, and
- t_a and t_b lead to some equivalent states a' and b'.

Non-initial states may be merged with initial states, the resulting state is an initial state.

All states in a class must have the same `is_final`, `final_word_out` and `word_out` values.

EXAMPLES:

```
sage: fsm = FiniteStateMachine([("A", "B", 0, 1), ("A", "B", 1, 0),
....:                           ("B", "C", 0, 0), ("B", "C", 1, 1),
....:                           ("C", "D", 0, 1), ("C", "D", 1, 0),
....:                           ("D", "A", 0, 0), ("D", "A", 1, 1)])
sage: fsmq = fsm.quotient([[fsm.state("A"), fsm.state("C")],
....:                      [fsm.state("B"), fsm.state("D")]])
sage: fsmq.transitions()
[Transition from ('A', 'C')
              to ('B', 'D'): 0|1,
 Transition from ('A', 'C')
              to ('B', 'D'): 1|0,
 Transition from ('B', 'D')
              to ('A', 'C'): 0|0,
 Transition from ('B', 'D')
              to ('A', 'C'): 1|1]
sage: fsmq.relabeled().transitions()
[Transition from 0 to 1: 0|1,
 Transition from 0 to 1: 1|0,
 Transition from 1 to 0: 0|0,
 Transition from 1 to 0: 1|1]
sage: fsmq1 = fsm.quotient(fsm.equivalence_classes())
sage: fsmq1 == fsmq
True
sage: fsm.quotient([[fsm.state("A"), fsm.state("B"), fsm.state("C"), fsm.state("D")]])
Traceback (most recent call last):
    ...
AssertionError: Transitions of state 'A' and 'B' are incompatible.
```

TESTS:

```
sage: fsm = FiniteStateMachine([("A", "B", 0, 1), ("A", "B", 1, 0),
....:                           ("B", "C", 0, 0), ("B", "C", 1, 1),
....:                           ("C", "D", 0, 1), ("C", "D", 1, 0),
....:                           ("D", "A", 0, 0), ("D", "A", 1, 1)],
....:                          final_states=["A", "C"])
sage: fsm.state("A").final_word_out = 1
sage: fsm.state("C").final_word_out = 2
sage: fsmq = fsm.quotient([[fsm.state("A"), fsm.state("C")],
....:                      [fsm.state("B"), fsm.state("D")]])
Traceback (most recent call last):
    ...
AssertionError: Class ['A', 'C'] mixes
final states with different final output words.
```

relabeled(*memo=None*, *labels=None*)

Returns a deep copy of the finite state machine, but the states are relabeled.

INPUT:

- `memo` – (default: `None`) a dictionary storing already processed elements.
- `labels` – (default: `None`) a dictionary or callable mapping old labels to new labels. If `None`, then the new labels are integers starting with 0.

OUTPUT:

A new finite state machine.

EXAMPLES:

```
sage: FSM1 = FiniteStateMachine([('A', 'B'), ('B', 'C'), ('C', 'A')])
sage: FSM1.states()
['A', 'B', 'C']
sage: FSM2 = FSM1.relabeled()
sage: FSM2.states()
[0, 1, 2]
sage: FSM3 = FSM1.relabeled(labels={'A': 'a', 'B': 'b', 'C': 'c'})
sage: FSM3.states()
['a', 'b', 'c']
sage: FSM4 = FSM2.relabeled(labels=lambda x: 2*x)
sage: FSM4.states()
[0, 2, 4]
```

TESTS:

```
sage: FSM2.relabeled(labels=1)
Traceback (most recent call last):
...
TypeError: labels must be None, a callable or a dictionary.
```

remove_epsilon_transitions()

TESTS:

```
sage: FiniteStateMachine().remove_epsilon_transitions()
Traceback (most recent call last):
...
NotImplementedError
```

set_coordinates(*coordinates*, *default=True*)

Set coordinates of the states for the LaTeX representation by a dictionary or a function mapping labels to coordinates.

INPUT:

- `coordinates` – a dictionary or a function mapping labels of states to pairs interpreted as coordinates.
- `default` – If `True`, then states not given by `coordinates` get a default position on a circle of radius 3.

OUTPUT:

Nothing.

EXAMPLES:

```
sage: F = Automaton([[0, 1, 1], [1, 2, 2], [2, 0, 0]])
sage: F.set_coordinates({0: (0, 0), 1: (2, 0), 2: (1, 1)})
sage: F.state(0).coordinates
(0, 0)
```

We can also use a function to determine the coordinates:

```
sage: F = Automaton([[0, 1, 1], [1, 2, 2], [2, 0, 0]])
sage: F.set_coordinates(lambda l: (l, 3/(l+1)))
sage: F.state(2).coordinates
(2, 1)
```

split_transitions()

Returns a new transducer, where all transitions in self with input labels consisting of more than one letter are replaced by a path of the corresponding length.

INPUT:

Nothing.

OUTPUT:

A new transducer.

EXAMPLES:

```
sage: A = Transducer([('A', 'B', [1, 2, 3], 0)],
....:                initial_states=['A'], final_states=['B'])
sage: A.split_transitions().states()
[('A', ()), ('B', ()),
 ('A', (1,)), ('A', (1, 2))]
```

state(*state*)

Returns the state of the finite state machine.

INPUT:

- `state` – If `state` is not an instance of `FSMState`, then it is assumed that it is the label of a state.

OUTPUT:

Returns the state of the finite state machine corresponding to `state`.

If no state is found, then a `LookupError` is thrown.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMState
sage: A = FSMState('A')
sage: FSM = FiniteStateMachine([(A, 'B'), ('C', A)])
sage: FSM.state('A') == A
True
sage: FSM.state('xyz')
Traceback (most recent call last):
...
LookupError: No state with label xyz found.
```

states()

Returns the states of the finite state machine.

INPUT:

Nothing.

OUTPUT:

The states of the finite state machine as list.

EXAMPLES:

```
sage: FSM = Automaton([('1', '2', 1), ('2', '2', 0)])
sage: FSM.states()
['1', '2']
```

transition(*transition*)

Returns the transition of the finite state machine.

INPUT:

- `transition` – If `transition` is not an instance of `FSMTransition`, then it is assumed that it is a tuple `(from_state, to_state, word_in, word_out)`.

OUTPUT:

Returns the transition of the finite state machine corresponding to `transition`.

If no transition is found, then a `LookupError` is thrown.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: t = FSMTransition('A', 'B', 0)
sage: F = FiniteStateMachine([t])
sage: F.transition(('A', 'B', 0))
Transition from 'A' to 'B': 0|-
sage: id(t) == id(F.transition(('A', 'B', 0)))
True
```

transitions (*from_state=None*)

Returns a list of all transitions.

INPUT:

- `from_state` – (default: `None`) If `from_state` is given, then a list of transitions starting there is given.

OUTPUT:

A list of all transitions.

EXAMPLES:

```
sage: FSM = Automaton([('1', '2', 1), ('2', '2', 0)])
sage: FSM.transitions()
[Transition from '1' to '2': 1|-,
 Transition from '2' to '2': 0|-]
```

transposition (*reverse_output_labels=True*)

Returns a new finite state machine, where all transitions of the input finite state machine are reversed.

INPUT:

- `reverse_output_labels` – a boolean (default: `True`): whether to reverse output labels.

OUTPUT:

A new finite state machine.

EXAMPLES:

```
sage: aut = Automaton([('A', 'A', 0), ('A', 'A', 1), ('A', 'B', 0)],
....:                 initial_states=['A'], final_states=['B'])
sage: aut.transposition().transitions('B')
[Transition from 'B' to 'A': 0|-]

sage: aut = Automaton([('1', '1', 1), ('1', '2', 0), ('2', '2', 0)],
....:                 initial_states=['1'], final_states=['1', '2'])
sage: aut.transposition().initial_states()
['1', '2']

sage: A = Automaton([(0, 1, [1, 0])],
....:     initial_states=[0],
....:     final_states=[1])
sage: A([1, 0])
True
```

```
sage: A.transposition()([0, 1])
True

sage: T = Transducer([(0, 1, [1, 0], [1, 0])],
....:     initial_states=[0],
....:     final_states=[1])
sage: T([1, 0])
[1, 0]
sage: T.transposition()([0, 1])
[0, 1]
sage: T.transposition(reverse_output_labels=False)([0, 1])
[1, 0]
```

TESTS:

If a final state of `self` has a non-empty final output word, transposition is not implemented:

```
sage: T = Transducer([('1', '1', 1, 0), ('1', '2', 0, 1),
....:                 ('2', '2', 0, 2)],
....:                 initial_states=['1'],
....:                 final_states=['1', '2'])
sage: T.state('1').final_word_out = [2, 5]
sage: T.transposition()
Traceback (most recent call last):
...
NotImplementedError: Transposition for transducers with
final output words is not implemented.
```

with_final_word_out (*letters*, *allow_non_final=True*)

Constructs a new finite state machine with final output words for all states by implicitly reading trailing letters until a final state is reached.

INPUT:

- `letters` – either an element of the input alphabet or a list of such elements. This is repeated cyclically when needed.
- `allow_non_final` – a boolean (default: `True`) which indicates whether we allow that some states may be non-final in the resulting finite state machine. I.e., if `False` then each state has to have a path to a final state with input label matching `letters`.

OUTPUT:

A finite state machine.

The inplace version of this function is `construct_final_word_out()`.

Suppose for the moment a single element `letter` as input for `letters`. This is equivalent to `letters = [letter]`. We will discuss the general case below.

Let `word_in` be a word over the input alphabet and assume that the original finite state machine transforms `word_in` to `word_out` reaching a possibly non-final state `s`. Let further k be the minimum number of letters `letter` such that there is a path from `s` to some final state `f` whose input label consists of k copies of `letter` and whose output label is `path_word_out`. Then the state `s` of the resulting finite state machine is a final state with final output `path_word_out + f.final_word_out`. Therefore, the new finite state machine transforms `word_in` to `word_out + path_word_out + f.final_word_out`.

This is e.g. useful for finite state machines operating on digit expansions: there, it is sometimes required to read a sufficient number of trailing zeros (at the most significant positions) in order to reach a final state and to flush all carries. In this case, this method constructs an essentially equivalent finite state machine in

the sense that it not longer requires adding sufficiently many trailing zeros. However, it is the responsibility of the user to make sure that if adding trailing zeros to the input anyway, the output is equivalent.

If `letters` consists of more than one letter, then it is assumed that (not necessarily complete) cycles of `letters` are appended as trailing input.

See also:

example on Gray code

EXAMPLES:

1. A simple transducer transforming 00 blocks to 01 blocks:

```
sage: T = Transducer([(0, 1, 0, 0), (1, 0, 0, 1)],
....:                initial_states=[0],
....:                final_states=[0])
sage: T.process([0, 0, 0])
(False, 1, [0, 1, 0])
sage: T.process([0, 0, 0, 0])
(True, 0, [0, 1, 0, 1])
sage: F = T.with_final_word_out(0)
sage: for f in F.iter_final_states():
....:     print f, f.final_word_out
0 []
1 [1]
sage: F.process([0, 0, 0])
(True, 1, [0, 1, 0, 1])
sage: F.process([0, 0, 0, 0])
(True, 0, [0, 1, 0, 1])
```

2. A more realistic example: Addition of 1 in binary. We construct a transition function transforming the input to its binary expansion:

```
sage: def binary_transition(carry, input):
....:     value = carry + input
....:     if value.mod(2) == 0:
....:         return (value/2, 0)
....:     else:
....:         return ((value-1)/2, 1)
```

Now, we only have to start with a carry of 1 to get the required transducer:

```
sage: T = Transducer(binary_transition,
....:                input_alphabet=[0, 1],
....:                initial_states=[1],
....:                final_states=[0])
```

We test this for the binary expansion of 7:

```
sage: T.process([1, 1, 1])
(False, 1, [0, 0, 0])
```

The final carry 1 has not be flushed yet, we have to add a trailing zero:

```
sage: T.process([1, 1, 1, 0])
(True, 0, [0, 0, 0, 1])
```

We check that with this trailing zero, the transducer performs as advertised:

```
sage: all(ZZ(T(k.bits()+[0]), base=2) == k + 1
....:     for k in srange(16))
True
```

However, most of the time, we produce superfluous trailing zeros:

```
sage: T(11.bits()+[0])
[0, 0, 1, 1, 0]
```

We now use this method:

```
sage: F = T.with_final_word_out(0)
sage: for f in F.iter_final_states():
....:     print f, f.final_word_out
1 [1]
0 []
```

The same tests as above, but we do not have to pad with trailing zeros anymore:

```
sage: F.process([1, 1, 1])
(True, 1, [0, 0, 0, 1])
sage: all(ZZ(F(k.bits()), base=2) == k + 1
....:     for k in srange(16))
True
```

No more trailing zero in the output:

```
sage: F(11.bits())
[0, 0, 1, 1]
sage: all(F(k.bits())[-1] == 1
....:     for k in srange(16))
True
```

3.Here is an example, where we allow trailing repeated 10:

```
sage: T = Transducer([(0, 1, 0, 'a'),
....:                 (1, 2, 1, 'b'),
....:                 (2, 0, 0, 'c')],
....:                initial_states=[0],
....:                final_states=[0])
sage: F = T.with_final_word_out([1, 0])
sage: for f in F.iter_final_states():
....:     print f, ''.join(f.final_word_out)
0
1 bc
```

Trying this with trailing repeated 01 does not produce a `final_word_out` for state `1`, but for state `2`:

```
sage: F = T.with_final_word_out([0, 1])
sage: for f in F.iter_final_states():
....:     print f, ''.join(f.final_word_out)
0
2 c
```

4.Here another example with a more-letter trailing input:

```
sage: T = Transducer([(0, 1, 0, 'a'),
....:                 (1, 2, 0, 'b'), (1, 2, 1, 'b'),
....:                 (2, 3, 0, 'c'), (2, 0, 1, 'e'),
....:                 (3, 1, 0, 'd'), (3, 1, 1, 'd')],
....:                initial_states=[0],
....:                final_states=[0],
....:                with_final_word_out=[0, 0, 1, 1])
sage: for f in T.iter_final_states():
....:     print f, ''.join(f.final_word_out)
0
1 bcdbcdbe
2 cdbe
3 dbe
```

TESTS:

1.Reading copies of `letter` may result in a cycle. In this simple example, we have no final state at all:

```
sage: T = Transducer([(0, 1, 0, 0), (1, 0, 0, 0)],
....:                initial_states=[0])
sage: T.with_final_word_out(0)
Traceback (most recent call last):
...
ValueError: The finite state machine contains
a cycle starting at state 0 with input label 0
and no final state.
```

2.A unique transition with input word `letter` is required:

```
sage: T = Transducer([(0, 1, 0, 0), (0, 2, 0, 0)])
sage: T.with_final_word_out(0)
Traceback (most recent call last):
...
ValueError: No unique transition leaving state 0
with input label 0.
```

It is not a problem if there is no transition starting at state `1` with input word `letter`:

```
sage: T = Transducer([(0, 1, 0, 0)])
sage: F = T.with_final_word_out(0)
sage: for f in F.iter_final_states():
....:     print f, f.final_word_out
```

Anyhow, you can override this by:

```
sage: T = Transducer([(0, 1, 0, 0)])
sage: T.with_final_word_out(0, allow_non_final=False)
Traceback (most recent call last):
...
ValueError: No unique transition leaving state 1
with input label 0.
```

3.All transitions must have input labels of length 1:

```
sage: T = Transducer([(0, 0, [], 0)])
sage: T.with_final_word_out(0)
Traceback (most recent call last):
...
NotImplementedError: All transitions must have input
labels of length 1. Consider calling split_transitions().
sage: T = Transducer([(0, 0, [0, 1], 0)])
sage: T.with_final_word_out(0)
Traceback (most recent call last):
...
NotImplementedError: All transitions must have input
labels of length 1. Consider calling split_transitions().
```

4.An empty list as input is not allowed:

```
sage: T = Transducer([(0, 0, [], 0)])
sage: T.with_final_word_out([])
Traceback (most recent call last):
...
ValueError: letters is not allowed to be an empty list.
```

class sage.combinat.finite_state_machine.**Transducer**(*data=None*, *initial_states=None*, *final_states=None*, *input_alphabet=None*, *output_alphabet=None*, *determine_alphabets=None*, *with_final_word_out=None*, *store_states_dict=True*, *on_duplicate_transition=None*)

Bases: sage.combinat.finite_state_machine.FiniteStateMachine

This creates a transducer, which is a finite state machine, whose transitions have input and output labels.

An transducer has additional features like creating a simplified transducer.

See class FiniteStateMachine for more information.

EXAMPLES:

We can create a transducer performing the addition of 1 (for numbers given in binary and read from right to left) in the following way:

```
sage: T = Transducer([('C', 'C', 1, 0), ('C', 'N', 0, 1),
....:                 ('N', 'N', 0, 0), ('N', 'N', 1, 1)],
....:                initial_states=['C'], final_states=['N'])
sage: T
```

```
Transducer with 2 states
sage: T([0])
[1]
sage: T([1,1,0])
[0, 0, 1]
sage: ZZ(T(15.digits(base=2)+[0]), base=2)
16
```

Note that we have padded the binary input sequence by a 0 so that the transducer can reach its final state.

TESTS:

```
sage: Transducer()
Empty transducer
```

cartesian_product(*other*, *only_accessible_components=True*)

Return a new transducer which can simultaneously process an input with the machines `self` and `other` where the output labels are d-tuples of the original output labels.

INPUT:

- `other` - a finite state machine (if $d = 2$) or a list (or other iterable) of $d - 1$ finite state machines
- `only_accessible_components` – If `True` (default), then the result is piped through `accessible_components()`. If no `new_input_alphabet` is given, it is determined by `determine_alphabets()`.

OUTPUT:

A transducer which can simultaneously process an input with `self` and the machine(s) in `other`.

The set of states of the new transducer is the Cartesian product of the set of states of `self` and `other`.

Let (A_j, B_j, a_j, b_j) for $j \in \{1, \ldots, d\}$ be transitions in the machines `self` and in `other`. Then there is a transition $((A_1, \ldots, A_d), (B_1, \ldots, B_d), a, (b_1, \ldots, b_d))$ in the new transducer if $a_1 = \cdots = a_d =: a$.

EXAMPLES:

```
sage: transducer1 = Transducer([('A', 'A', 0, 0),
....:                           ('A', 'A', 1, 1)],
....:                          initial_states=['A'],
....:                          final_states=['A'],
....:                          determine_alphabets=True)
sage: transducer2 = Transducer([(0, 1, 0, ['b', 'c']),
....:                           (0, 0, 1, 'b'),
....:                           (1, 1, 0, 'a')],
....:                          initial_states=[0],
....:                          final_states=[1],
....:                          determine_alphabets=True)
sage: result = transducer1.cartesian_product(transducer2)
sage: result
Transducer with 2 states
sage: result.transitions()
[Transition from ('A', 0) to ('A', 1): 0|(0, 'b'),(None, 'c'),
 Transition from ('A', 0) to ('A', 0): 1|(1, 'b'),
 Transition from ('A', 1) to ('A', 1): 0|(0, 'a')]
sage: result([1, 0, 0])
[(1, 'b'), (0, 'b'), (None, 'c'),  (0, 'a')]
sage: (transducer1([1, 0, 0]), transducer2([1, 0, 0]))
([1, 0, 0], ['b', 'b', 'c', 'a'])
```

Also final output words are correctly processed:

```
sage: transducer1.state('A').final_word_out = 2
sage: result = transducer1.cartesian_product(transducer2)
sage: result.final_states()[0].final_word_out
[(2, None)]
```

The following transducer counts the number of 11 blocks minus the number of 10 blocks over the alphabet `[0, 1]`.

```
sage: count_11 = transducers.CountSubblockOccurrences(
....:     [1, 1],
....:     input_alphabet=[0, 1])
sage: count_10 = transducers.CountSubblockOccurrences(
....:     [1, 0],
....:     input_alphabet=[0, 1])
sage: count_11x10 = count_11.cartesian_product(count_10)
sage: difference = transducers.sub([0, 1])(count_11x10)
sage: T = difference.simplification().relabeled()
sage: T.initial_states()
[1]
sage: sorted(T.transitions())
[Transition from 0 to 1: 0|-1,
 Transition from 0 to 0: 1|1,
 Transition from 1 to 1: 0|0,
 Transition from 1 to 0: 1|0]
sage: input =  [0, 1, 1,  0, 1,  0, 0, 0, 1, 1, 1,  0]
sage: output = [0, 0, 1, -1, 0, -1, 0, 0, 0, 1, 1, -1]
sage: T(input) == output
True
```

If `other` is an automaton, then `cartesian_product()` returns `self` where the input is restricted to the input accepted by `other`.

For example, if the transducer transforms the standard binary expansion into the non-adjacent form and the automaton recognizes the binary expansion without adjacent ones, then the Cartesian product of these two is a transducer which does not change the input (except for changing `a` to `(a, None)` and ignoring a leading 0).

```
sage: NAF = Transducer([(0, 1, 0, None),
....:                   (0, 2, 1, None),
....:                   (1, 1, 0, 0),
....:                   (1, 2, 1, 0),
....:                   (2, 1, 0, 1),
....:                   (2, 3, 1, -1),
....:                   (3, 2, 0, 0),
....:                   (3, 3, 1, 0)],
....:                  initial_states=[0],
....:                  final_states=[1],
....:                  determine_alphabets=True)
sage: aut11 = Automaton([(0, 0, 0), (0, 1, 1), (1, 0, 0)],
....:                   initial_states=[0],
....:                   final_states=[0, 1],
....:                   determine_alphabets=True)
sage: res = NAF.cartesian_product(aut11)
sage: res([1, 0, 0, 1, 0, 1, 0])
[(1, None), (0, None), (0, None), (1, None), (0, None), (1, None)]
```

This is obvious because if the standard binary expansion does not have adjacent ones, then it is the same as the non-adjacent form.

Be aware that cartesian_product() is not commutative.

```
sage: aut11.cartesian_product(NAF)
Traceback (most recent call last):
...
TypeError: Only an automaton can be intersected with an automaton.
```

The Cartesian product of more than two finite state machines can also be computed:

```
sage: T0 = transducers.CountSubblockOccurrences([0, 0], [0, 1, 2])
sage: T1 = transducers.CountSubblockOccurrences([1, 1], [0, 1, 2])
sage: T2 = transducers.CountSubblockOccurrences([2, 2], [0, 1, 2])
sage: T = T0.cartesian_product([T1, T2])
sage: T.transitions()
[Transition from ((), (), ()) to ((0,), (), ()): 0|(0, 0, 0),
 Transition from ((), (), ()) to ((), (1,), ()): 1|(0, 0, 0),
 Transition from ((), (), ()) to ((), (), (2,)): 2|(0, 0, 0),
 Transition from ((0,), (), ()) to ((0,), (), ()): 0|(1, 0, 0),
 Transition from ((0,), (), ()) to ((), (1,), ()): 1|(0, 0, 0),
 Transition from ((0,), (), ()) to ((), (), (2,)): 2|(0, 0, 0),
 Transition from ((), (1,), ()) to ((0,), (), ()): 0|(0, 0, 0),
 Transition from ((), (1,), ()) to ((), (1,), ()): 1|(0, 1, 0),
 Transition from ((), (1,), ()) to ((), (), (2,)): 2|(0, 0, 0),
 Transition from ((), (), (2,)) to ((0,), (), ()): 0|(0, 0, 0),
 Transition from ((), (), (2,)) to ((), (1,), ()): 1|(0, 0, 0),
 Transition from ((), (), (2,)) to ((), (), (2,)): 2|(0, 0, 1)]
sage: T([0, 0, 1, 1, 2, 2, 0, 1, 2, 2])
[(0, 0, 0),
 (1, 0, 0),
 (0, 0, 0),
 (0, 1, 0),
 (0, 0, 0),
 (0, 0, 1),
 (0, 0, 0),
 (0, 0, 0),
 (0, 0, 0),
 (0, 0, 1)]
```

intersection(*other*, *only_accessible_components=True*)

Returns a new transducer which accepts an input if it is accepted by both given finite state machines producing the same output.

INPUT:

- other – a transducer
- only_accessible_components – If True (default), then the result is piped through accessible_components(). If no new_input_alphabet is given, it is determined by determine_alphabets().

OUTPUT:

A new transducer which computes the intersection (see below) of the languages of self and other.

The set of states of the transducer is the Cartesian product of the set of states of both given transducer. There is a transition $((A, B), (C, D), a, b)$ in the new transducer if there are transitions (A, C, a, b) and (B, D, a, b) in the old transducers.

EXAMPLES:

```
sage: transducer1 = Transducer([('1', '2', 1, 0),
....:                           ('2', '2', 1, 0),
```

```
....:                              ('2', '2', 0, 1)],
....:                             initial_states=['1'],
....:                             final_states=['2'])
sage: transducer2 = Transducer([('A', 'A', 1, 0),
....:                              ('A', 'B', 0, 0),
....:                              ('B', 'B', 0, 1),
....:                              ('B', 'A', 1, 1)],
....:                             initial_states=['A'],
....:                             final_states=['B'])
sage: res = transducer1.intersection(transducer2)
sage: res.transitions()
[Transition from ('1', 'A') to ('2', 'A'): 1|0,
 Transition from ('2', 'A') to ('2', 'A'): 1|0]
```

In general, transducers are not closed under intersection. But for transducer which do not have epsilon-transitions, the intersection is well defined (cf. [BaWo2012]). However, in the next example the intersection of the two transducers is not well defined. The intersection of the languages consists of $(a^n, b^n c^n)$. This set is not recognizable by a *finite* transducer.

```
sage: t1 = Transducer([(0, 0, 'a', 'b'),
....:                  (0, 1, None, 'c'),
....:                  (1, 1, None, 'c')],
....:                 initial_states=[0],
....:                 final_states=[0, 1])
sage: t2 = Transducer([('A', 'A', None, 'b'),
....:                  ('A', 'B', 'a', 'c'),
....:                  ('B', 'B', 'a', 'c')],
....:                 initial_states=['A'],
....:                 final_states=['A', 'B'])
sage: t2.intersection(t1)
Traceback (most recent call last):
...
ValueError: An epsilon-transition (with empty input or output)
was found.
```

TESTS:

```
sage: transducer1 = Transducer([('1', '2', 1, 0)],
....:                          initial_states=['1'],
....:                          final_states=['2'])
sage: transducer2 = Transducer([('A', 'B', 1, 0)],
....:                          initial_states=['A'],
....:                          final_states=['B'])
sage: res = transducer1.intersection(transducer2)
sage: res.final_states()
[('2', 'B')]
sage: transducer1.state('2').final_word_out = 1
sage: transducer2.state('B').final_word_out = 2
sage: res = transducer1.intersection(transducer2)
sage: res.final_states()
[]
```

REFERENCES:

process(**args*, ***kwargs*)

Return whether the transducer accepts the input, the state where the computation stops and which output is generated.

INPUT:

- input_tape – the input tape can be a list or an iterable with entries from the input alphabet. If we are working with a multi-tape machine (see parameter use_multitape_input and notes below), then the tape is a list or tuple of tracks, each of which can be a list or an iterable with entries from the input alphabet.
- initial_state or initial_states – the initial state(s) in which the machine starts. Either specify a single one with initial_state or a list of them with initial_states. If both are given, initial_state will be appended to initial_states. If neither is specified, the initial states of the finite state machine are taken.
- list_of_outputs – (default: None) a boolean or None. If True, then the outputs are given in list form (even if we have no or only one single output). If False, then the result is never a list (an exception is raised if the result cannot be returned). If list_of_outputs=None the method determines automatically what to do (e.g. if a non-deterministic machine returns more than one path, then the output is returned in list form).
- only_accepted – (default: False) a boolean. If set, then the first argument in the output is guaranteed to be True (if the output is a list, then the first argument of each element will be True).
- full_output – (default: True) a boolean. If set, then the full output is given, otherwise only the generated output (the third entry below only). If the input is not accepted, a ValueError is raised.
- always_include_output – if set (not by default), always include the output. This is inconsequential for a Transducer, but can be used in other classes derived from FiniteStateMachine where the output is suppressed by default, cf. Automaton.process().
- format_output – a function that translates the written output (which is in form of a list) to something more readable. By default (None) identity is used here.
- check_epsilon_transitions – (default: True) a boolean. If False, then epsilon transitions are not taken into consideration during process.
- write_final_word_out – (default: True) a boolean specifying whether the final output words should be written or not.
- use_multitape_input – (default: False) a boolean. If True, then the multi-tape mode of the process iterator is activated. See also the notes below for multi-tape machines.
- process_all_prefixes_of_input – (default: False) a boolean. If True, then each prefix of the input word is processed (instead of processing the whole input word at once). Consequently, there is an output generated for each of these prefixes.
- process_iterator_class – (default: None) a class inherited from FSMProcessIterator. If None, then FSMProcessIterator is taken. An instance of this class is created and is used during the processing.
- automatic_output_type – (default: False) a boolean If set and the input has a parent, then the output will have the same parent. If the input does not have a parent, then the output will be of the same type as the input.

OUTPUT:

The full output is a triple (or a list of triples, cf. parameter list_of_outputs), where

- the first entry is True if the input string is accepted,
- the second gives the reached state after processing the input tape (This is a state with label None if the input could not be processed, i.e., if at one point no transition to go on could be found.), and
- the third gives a list of the output labels written during processing.

If full_output is False, then only the third entry is returned.

Note that in the case the transducer is not deterministic, all possible paths are taken into account.

This function uses an iterator which, in its simplest form, goes from one state to another in each step. To decide which way to go, it uses the input words of the outgoing transitions and compares them to the input tape. More precisely, in each step, the iterator takes an outgoing transition of the current state, whose input label equals the input letter of the tape. The output label of the transition, if present, is written on the output tape.

If the choice of the outgoing transition is not unique (i.e., we have a non-deterministic finite state machine), all possibilites are followed. This is done by splitting the process into several branches, one for each of the possible outgoing transitions.

The process (iteration) stops if all branches are finished, i.e., for no branch, there is any transition whose input word coincides with the processed input tape. This can simply happen when the entire tape was read.

Also see `__call__()` for a version of `process()` with shortened output.

Internally this function creates and works with an instance of `FSMProcessIterator`. This iterator can also be obtained with `iter_process()`.

If working with multi-tape finite state machines, all input words of transitions are words of k-tuples of letters. Moreover, the input tape has to consist of k tracks, i.e., be a list or tuple of k iterators, one for each track.

Warning: Working with multi-tape finite state machines is still experimental and can lead to wrong outputs.

EXAMPLES:

```
sage: binary_inverter = Transducer({'A': [('A', 0, 1), ('A', 1, 0)]},
....:                              initial_states=['A'], final_states=['A'])
sage: binary_inverter.process([0, 1, 0, 0, 1, 1])
(True, 'A', [1, 0, 1, 1, 0, 0])
```

If we are only interested in the output, we can also use:

```
sage: binary_inverter([0, 1, 0, 0, 1, 1])
[1, 0, 1, 1, 0, 0]
```

This can also be used with words as input:

```
sage: W = Words([0, 1]); W
Finite and infinite words over {0, 1}
sage: w = W([0, 1, 0, 0, 1, 1]); w
word: 010011
sage: binary_inverter(w)
word: 101100
```

In this case it is automatically determined that the output is a word. The call above is equivalent to:

```
sage: binary_inverter.process(w,
....:                         full_output=False,
....:                         list_of_outputs=False,
....:                         automatic_output_type=True)
word: 101100
```

The following transducer transforms 0^n1 to 1^n2:

```
sage: T = Transducer([(0, 0, 0, 1), (0, 1, 1, 2)])
sage: T.state(0).is_initial = True
sage: T.state(1).is_final = True
```

We can see the different possibilites of the output by:

```
sage: [T.process(w) for w in [[1], [0, 1], [0, 0, 1], [0, 1, 1],
....:                         [0], [0, 0], [2, 0], [0, 1, 2]]]
[(True, 1, [2]), (True, 1, [1, 2]),
 (True, 1, [1, 1, 2]), (False, None, None),
 (False, 0, [1]), (False, 0, [1, 1]),
 (False, None, None), (False, None, None)]
```

If we just want a condensed output, we use:

```
sage: [T.process(w, full_output=False)
....:      for w in [[1], [0, 1], [0, 0, 1]]]
[[2], [1, 2], [1, 1, 2]]
sage: T.process([0], full_output=False)
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
sage: T.process([0, 1, 2], full_output=False)
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
```

It is equivalent to:

```
sage: [T(w) for w in [[1], [0, 1], [0, 0, 1]]]
[[2], [1, 2], [1, 1, 2]]
sage: T([0])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
sage: T([0, 1, 2])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
```

A cycle with empty input and empty output is correctly processed:

```
sage: T = Transducer([(0, 1, None, None), (1, 0, None, None)],
....:                initial_states=[0], final_states=[1])
sage: T.process([])
[(False, 0, []), (True, 1, [])]
sage: _ = T.add_transition(-1, 0, 0, 'r')
sage: T.state(-1).is_initial = True
sage: T.state(0).is_initial = False
sage: T.process([0])
[(False, 0, ['r']), (True, 1, ['r'])]
```

If there is a cycle with empty input but non-empty output, the possible outputs would be an infinite set:

```
sage: T = Transducer([(0, 1, None, 'z'), (1, 0, None, None)],
....:                initial_states=[0], final_states=[1])
sage: T.process([])
Traceback (most recent call last):
...
RuntimeError: State 0 is in an epsilon cycle (no input),
but output is written.
```

But if this cycle with empty input and non-empty output is not reached, the correct output is produced:

```
sage: _ = T.add_transition(-1, 0, 0, 'r')
sage: T.state(-1).is_initial = True
sage: T.state(0).is_initial = False
sage: T.process([])
(False, -1, [])
sage: T.process([0])
Traceback (most recent call last):
...
RuntimeError: State 0 is in an epsilon cycle (no input),
but output is written.
```

If we set `check_epsilon_transitions=False`, then no transitions with empty input are considered anymore. Thus cycles with empty input are no problem anymore:

```
sage: T.process([0], check_epsilon_transitions=False)
(False, 0, ['r'])
```

A simple example of a multi-tape transducer is the following: It writes the length of the first tape many letters `a` and then the length of the second tape many letters `b`:

```
sage: M = Transducer([(0, 0, (1, None), 'a'),
....:                 (0, 1, [], []),
....:                 (1, 1, (None, 1), 'b')],
....:                initial_states=[0],
....:                final_states=[1])
sage: M.process(([1, 1], [1]), use_multitape_input=True)
(True, 1, ['a', 'a', 'b'])
```

See also:

`FiniteStateMachine.process()`, `Automaton.process()`, `iter_process()`, `__call__()`, `FSMProcessIterator`.

TESTS:

```
sage: T = Transducer([(0, 1, 1, 'a'), (1, 0, 1, 'b')],
....:                initial_states=[0, 1], final_states=[1])
sage: T.process([1, 1])
[(False, 0, ['a', 'b']), (True, 1, ['b', 'a'])]
sage: T.process([1, 1], T.state(0))
(False, 0, ['a', 'b'])
sage: T.state(1).final_word_out = 'c'
sage: T.process([1, 1], T.state(1))
(True, 1, ['b', 'a', 'c'])
sage: T.process([1, 1], T.state(1), write_final_word_out=False)
(True, 1, ['b', 'a'])
```

The parameter `input_tape` is required:

```
sage: T.process()
Traceback (most recent call last):
...
TypeError: No input tape given.
```

simplification()

Returns a simplified transducer.

INPUT:

Nothing.

OUTPUT:

A new transducer.

This function simplifies a transducer by Moore's algorithm, first moving common output labels of transitions leaving a state to output labels of transitions entering the state (cf. `prepone_output()`).

The resulting transducer implements the same function as the original transducer.

EXAMPLES:

```
sage: fsm = Transducer([("A", "B", 0, 1), ("A", "B", 1, 0),
....:                   ("B", "C", 0, 0), ("B", "C", 1, 1),
....:                   ("C", "D", 0, 1), ("C", "D", 1, 0),
....:                   ("D", "A", 0, 0), ("D", "A", 1, 1)])
sage: fsms = fsm.simplification()
sage: fsms
Transducer with 2 states
sage: fsms.transitions()
[Transition from ('A', 'C')
              to ('B', 'D'): 0|1,
 Transition from ('A', 'C')
              to ('B', 'D'): 1|0,
 Transition from ('B', 'D')
              to ('A', 'C'): 0|0,
 Transition from ('B', 'D')
              to ('A', 'C'): 1|1]
sage: fsms.relabeled().transitions()
[Transition from 0 to 1: 0|1,
 Transition from 0 to 1: 1|0,
 Transition from 1 to 0: 0|0,
 Transition from 1 to 0: 1|1]

sage: fsm = Transducer([("A", "A", 0, 0),
....:                   ("A", "B", 1, 1),
....:                   ("A", "C", 1, -1),
....:                   ("B", "A", 2, 0),
....:                   ("C", "A", 2, 0)])
sage: fsm_simplified = fsm.simplification()
sage: fsm_simplified
Transducer with 2 states
sage: fsm_simplified.transitions()
[Transition from ('A',) to ('A',): 0|0,
 Transition from ('A',) to ('B', 'C'): 1|1,0,
 Transition from ('A',) to ('B', 'C'): 1|-1,0,
 Transition from ('B', 'C') to ('A',): 2|-]

sage: from sage.combinat.finite_state_machine import duplicate_transition_add_input
sage: T = Transducer([('A', 'A', 1/2, 0),
....:                 ('A', 'B', 1/4, 1),
....:                 ('A', 'C', 1/4, 1),
....:                 ('B', 'A', 1, 0),
....:                 ('C', 'A', 1, 0)],
....:                initial_states=[0],
....:                final_states=['A', 'B', 'C'],
....:                on_duplicate_transition=duplicate_transition_add_input)
sage: sorted(T.simplification().transitions())
[Transition from ('A',) to ('A',): 1/2|0,
 Transition from ('A',) to ('B', 'C'): 1/2|1,
 Transition from ('B', 'C') to ('A',): 1|0]
```

Illustrating the use of colors in order to avoid identification of states:

```
sage: T = Transducer( [[0,0,0,0], [0,1,1,1],
....:                  [1,0,0,0], [1,1,1,1]],
....:                 initial_states=[0],
....:                 final_states=[0,1])
sage: sorted(T.simplification().transitions())
[Transition from (0, 1) to (0, 1): 0|0,
 Transition from (0, 1) to (0, 1): 1|1]
sage: T.state(0).color = 0
sage: T.state(0).color = 1
sage: sorted(T.simplification().transitions())
[Transition from (0,) to (0,): 0|0,
 Transition from (0,) to (1,): 1|1,
 Transition from (1,) to (0,): 0|0,
 Transition from (1,) to (1,): 1|1]
```

sage.combinat.finite_state_machine.**duplicate_transition_add_input**(*old_transition*, *new_transition*)

Alternative function for handling duplicate transitions in finite state machines. This implementation adds the input label of the new transition to the input label of the old transition. This is intended for the case where a Markov chain is modelled by a finite state machine using the input labels as transition probabilities.

See the documentation of the `on_duplicate_transition` parameter of `FiniteStateMachine`.

INPUT:

- `old_transition` – A transition in a finite state machine.
- `new_transition` – A transition, identical to `old_transition`, which is to be inserted into the finite state machine.

OUTPUT:

A transition whose input weight is the sum of the input weights of `old_transition` and `new_transition`.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_add_input
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: duplicate_transition_add_input(FSMTransition('a', 'a', 1/2),
....:                                FSMTransition('a', 'a', 1/2))
Transition from 'a' to 'a': 1|-
```

Input labels must be lists of length 1:

```
sage: duplicate_transition_add_input(FSMTransition('a', 'a', [1, 1]),
....:                                FSMTransition('a', 'a', [1, 1]))
Traceback (most recent call last):
...
TypeError: Trying to use duplicate_transition_add_input on
"Transition from 'a' to 'a': 1,1|-" and
"Transition from 'a' to 'a': 1,1|-",
but input words are assumed to be lists of length 1
```

sage.combinat.finite_state_machine.**duplicate_transition_ignore**(*old_transition*, *new_transition*)

Default function for handling duplicate transitions in finite state machines. This implementation ignores the occurrence.

See the documentation of the `on_duplicate_transition` parameter of `FiniteStateMachine`.

INPUT:

- `old_transition` – A transition in a finite state machine.
- `new_transition` – A transition, identical to `old_transition`, which is to be inserted into the finite state machine.

OUTPUT:

The same transition, unchanged.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_ignore
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: duplicate_transition_ignore(FSMTransition(0, 0, 1),
....:                             FSMTransition(0, 0, 1))
Transition from 0 to 0: 1|-
```

sage.combinat.finite_state_machine.**duplicate_transition_raise_error**(*old_transition*, *new_transition*)

Alternative function for handling duplicate transitions in finite state machines. This implementation raises a `ValueError`.

See the documentation of the `on_duplicate_transition` parameter of `FiniteStateMachine`.

INPUT:

- `old_transition` – A transition in a finite state machine.
- `new_transition` – A transition, identical to `old_transition`, which is to be inserted into the finite state machine.

OUTPUT:

Nothing. A `ValueError` is raised.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import duplicate_transition_raise_error
sage: from sage.combinat.finite_state_machine import FSMTransition
sage: duplicate_transition_raise_error(FSMTransition(0, 0, 1),
....:                                  FSMTransition(0, 0, 1))
Traceback (most recent call last):
...
ValueError: Attempting to re-insert transition Transition from 0 to 0: 1|-
```

sage.combinat.finite_state_machine.**equal**(*iterator*)

Checks whether all elements of `iterator` are equal.

INPUT:

- `iterator` – an iterator of the elements to check

OUTPUT:

`True` or `False`.

This implements http://stackoverflow.com/a/3844832/1052778.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import equal
sage: equal([0, 0, 0])
True
sage: equal([0, 1, 0])
```

```
False
sage: equal([])
True
sage: equal(iter([None, None]))
True
```

We can test other properties of the elements than the elements themselves. In the following example, we check whether all tuples have the same lengths:

```
sage: equal(len(x) for x in [(1, 2), (2, 3), (3, 1)])
True
sage: equal(len(x) for x in [(1, 2), (1, 2, 3), (3, 1)])
False
```

sage.combinat.finite_state_machine.**full_group_by**(*l*, *key*=<*function* <*lambda*>>)

Group iterable `l` by values of `key`.

INPUT:

- iterable `l`
- key function `key`

OUTPUT:

A list of pairs `(k, elements)` such that `key(e)=k` for all `e` in `elements`.

This is similar to `itertools.groupby` except that lists are returned instead of iterables and no prior sorting is required.

We do not require

- that the keys are sortable (in contrast to the approach via `sorted` and `itertools.groupby`) and
- that the keys are hashable (in contrast to the implementation proposed in http://stackoverflow.com/a/15250161).

However, it is required

- that distinct keys have distinct `str`-representations.

The implementation is inspired by http://stackoverflow.com/a/15250161, but non-hashable keys are allowed.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import full_group_by
sage: t = [2/x, 1/x, 2/x]
sage: r = full_group_by([0, 1, 2], key=lambda i:t[i])
sage: sorted(r, key=lambda p:p[1])
[(2/x, [0, 2]), (1/x, [1])]
sage: from itertools import groupby
sage: for k, elements in groupby(sorted([0, 1, 2],
....:                            key=lambda i:t[i]),
....:                            key=lambda i:t[i]):
....:     print k, list(elements)
2/x [0]
1/x [1]
2/x [2]
```

Note that the behavior is different from `itertools.groupby` because neither $1/x < 2/x$ nor $2/x < 1/x$ does hold.

Here, the result `r` has been sorted in order to guarantee a consistent order for the doctest suite.

sage.combinat.finite_state_machine.**is_Automaton**(*FSM*)
Tests whether or not FSM inherits from Automaton.

TESTS:

```
sage: from sage.combinat.finite_state_machine import is_FiniteStateMachine, is_Automaton
sage: is_Automaton(FiniteStateMachine())
False
sage: is_Automaton(Automaton())
True
sage: is_FiniteStateMachine(Automaton())
True
```

sage.combinat.finite_state_machine.**is_FSMProcessIterator**(*PI*)
Tests whether or not PI inherits from FSMProcessIterator.

TESTS:

```
sage: from sage.combinat.finite_state_machine import is_FSMProcessIterator, FSMProcessIterator
sage: is_FSMProcessIterator(FSMProcessIterator(FiniteStateMachine([[0, 0, 0, 0]], initial_states
True
```

sage.combinat.finite_state_machine.**is_FSMState**(*S*)
Tests whether or not S inherits from FSMState.

TESTS:

```
sage: from sage.combinat.finite_state_machine import is_FSMState, FSMState
sage: is_FSMState(FSMState('A'))
True
```

sage.combinat.finite_state_machine.**is_FSMTransition**(*T*)
Tests whether or not T inherits from FSMTransition.

TESTS:

```
sage: from sage.combinat.finite_state_machine import is_FSMTransition, FSMTransition
sage: is_FSMTransition(FSMTransition('A', 'B'))
True
```

sage.combinat.finite_state_machine.**is_FiniteStateMachine**(*FSM*)
Tests whether or not FSM inherits from FiniteStateMachine.

TESTS:

```
sage: from sage.combinat.finite_state_machine import is_FiniteStateMachine
sage: is_FiniteStateMachine(FiniteStateMachine())
True
sage: is_FiniteStateMachine(Automaton())
True
sage: is_FiniteStateMachine(Transducer())
True
```

sage.combinat.finite_state_machine.**is_Transducer**(*FSM*)
Tests whether or not FSM inherits from Transducer.

TESTS:

```
sage: from sage.combinat.finite_state_machine import is_FiniteStateMachine, is_Transducer
sage: is_Transducer(FiniteStateMachine())
False
sage: is_Transducer(Transducer())
True
```

```
sage: is_FiniteStateMachine(Transducer())
True
```

sage.combinat.finite_state_machine.**setup_latex_preamble**()

This function adds the package `tikz` with support for automata to the preamble of Latex so that the finite state machines can be drawn nicely.

INPUT:

Nothing.

OUTPUT:

Nothing.

See the section on *LaTeX output* in the introductory examples of this module.

TESTS:

```
sage: from sage.combinat.finite_state_machine import setup_latex_preamble
sage: setup_latex_preamble()
sage: ("\usepackage{tikz}" in latex.extra_preamble()) == latex.has_file("tikz.sty")
True
```

sage.combinat.finite_state_machine.**startswith**(*list*, *prefix*)

Determine whether list starts with the given prefix.

INPUT:

- `list` – list
- `prefix` – list representing the prefix

OUTPUT:

`True` or `False`.

Similar to `str.startswith()`.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import startswith
sage: startswith([1, 2, 3], [1, 2])
True
sage: startswith([1], [1, 2])
False
sage: startswith([1, 3, 2], [1, 2])
False
```

sage.combinat.finite_state_machine.**tupleofwords_to_wordoftuples**(*tupleofwords*)

Transposes a tuple of words over the alphabet to a word of tuples.

INPUT:

- `tupleofwords` – a tuple of a list of letters.

OUTPUT:

A list of tuples.

Missing letters in the words are padded with the letter `None` (from the empty word).

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import (
....:     tupleofwords_to_wordoftuples)
sage: tupleofwords_to_wordoftuples(
....:     ([1, 2], [3, 4, 5, 6], [7]))
[(1, 3, 7), (2, 4, None), (None, 5, None), (None, 6, None)]
```

sage.combinat.finite_state_machine.**wordoftuples_to_tupleofwords**(*wordoftuples*)

Transposes a word of tuples to a tuple of words over the alphabet.

INPUT:

- `wordoftuples` – a list of tuples of letters.

OUTPUT:

A tuple of lists.

Letters `None` (empty word) are removed from each word in the output.

EXAMPLES:

```
sage: from sage.combinat.finite_state_machine import (
....:     wordoftuples_to_tupleofwords)
sage: wordoftuples_to_tupleofwords(
....:     [(1, 2), (1, None), (1, None), (1, 2), (None, 2)])
([1, 1, 1, 1], [2, 2, 2])
```

5.1.97 Common Automata and Transducers (Finite State Machines Generators)

Automata and Transducers in Sage can be built through the `automata` and `transducers` objects, respectively. It contains generators for common finite state machines. For example,

```
sage: I = transducers.Identity([0, 1, 2])
```

generates an identity transducer on the alphabet $\{0, 1, 2\}$.

To construct automata and transducers manually, you can use the classes `Automaton` and `Transducer`, respectively. See *Finite State Machines, Automata, Transducers* for more details and a lot of *examples*.

Automata

`AnyLetter()`	Return an automaton recognizing any letter.
`AnyWord()`	Return an automaton recognizing any word.
`EmptyWord()`	Return an automaton recognizing the empty word.
`Word()`	Return an automaton recognizing the given word.
`ContainsWord()`	Return an automaton recognizing words containing the given word.

Transducers

Identity()	Returns a transducer realizing the identity map.
abs()	Returns a transducer realizing absolute value.
map()	Returns a transducer realizing a function.
operator()	Returns a transducer realizing a binary operation.
all()	Returns a transducer realizing logical and.
any()	Returns a transducer realizing logical or.
add()	Returns a transducer realizing addition.
sub()	Returns a transducer realizing subtraction.
CountSubblockOccurrences()	Returns a transducer counting the occurrences of a subblock.
Wait()	Returns a transducer writing False until first (or k-th) true input is read.
weight()	Returns a transducer realizing the Hamming weight.
GrayCode()	Returns a transducer realizing binary Gray code.
Recursion()	Returns a transducer defined by recursions.

AUTHORS:

- Clemens Heuberger (2014-04-07): initial version
- Sara Kropf (2014-04-10): some changes in TransducerGenerator
- Daniel Krenn (2014-04-15): improved common docstring during review
- Clemens Heuberger, Daniel Krenn, Sara Kropf (2014-04-16–2014-05-02): A couple of improvements. Details see trac ticket #16141, trac ticket #16142, trac ticket #16143, trac ticket #16186.
- Sara Kropf (2014-04-29): weight transducer
- Clemens Heuberger, Daniel Krenn (2014-07-18): transducers Wait, all, any
- Clemens Heuberger (2014-08-10): transducer Recursion
- Clemens Heuberger (2015-07-31): automaton word
- Daniel Krenn (2015-09-14): cleanup trac ticket #18227

ACKNOWLEDGEMENT:

- Clemens Heuberger, Daniel Krenn and Sara Kropf are supported by the Austrian Science Fund (FWF): P 24644-N26.

Functions and methods

class sage.combinat.finite_state_machine_generators.**AutomatonGenerators**
Bases: object

A collection of constructors for several common automata.

A list of all automata in this database is available via tab completion. Type "automata." and then hit tab to see which automata are available.

The automata currently in this class include:

- AnyLetter()
- AnyWord()
- EmptyWord()
- Word()
- ContainsWord()

AnyLetter(*input_alphabet*)

Return an automaton recognizing any letter of the given input alphabet.

INPUT:

- `input_alphabet` – a list, the input alphabet

OUTPUT:

An `Automaton`.

EXAMPLES:

```
sage: A = automata.AnyLetter([0, 1])
sage: A([])
False
sage: A([0])
True
sage: A([1])
True
sage: A([0, 0])
False
```

See also:

`AnyWord()`

AnyWord(*input_alphabet*)

Return an automaton recognizing any word of the given input alphabet.

INPUT:

- `input_alphabet` – a list, the input alphabet

OUTPUT:

An `Automaton`.

EXAMPLES:

```
sage: A = automata.AnyWord([0, 1])
sage: A([0])
True
sage: A([1])
True
sage: A([0, 1])
True
sage: A([0, 2])
False
```

This is equivalent to taking the `kleene_star()` of `AnyLetter()` and minimizing the result. This method immediately gives a minimized version:

```
sage: B = automata.AnyLetter([0, 1]).kleene_star().minimization().relabeled()
sage: B == A
True
```

See also:

`AnyLetter()`, `Word()`.

ContainsWord(*word*, *input_alphabet*)

Return an automaton recognizing the words containing the given word as a factor.

INPUT:

- word – a list (or other iterable) of letters, the word we are looking for.
- input_alphabet – a list or other iterable, the input alphabet.

OUTPUT:

An Automaton.

EXAMPLES:

```
sage: A = automata.ContainsWord([0, 1, 0, 1, 1],
....:                            input_alphabet=[0, 1])
sage: A([1, 0, 1, 0, 1, 0, 1, 1, 0, 0])
True
sage: A([1, 0, 1, 0, 1, 0, 1, 0])
False
```

This is equivalent to taking the concatenation of AnyWord(), Word() and AnyWord() and minimizing the result. This method immediately gives a minimized version:

```
sage: B = (automata.AnyWord([0, 1]) *
....:      automata.Word([0, 1, 0, 1, 1], [0, 1]) *
....:      automata.AnyWord([0, 1])).minimization()
sage: B.is_equivalent(A)
True
```

See also:

CountSubblockOccurrences(), AnyWord(), Word().

EmptyWord(*input_alphabet=None*)

Return an automaton recognizing the empty word.

INPUT:

- input_alphabet – (default: None) an iterable or None.

OUTPUT:

An Automaton.

EXAMPLES:

```
sage: A = automata.EmptyWord()
sage: A([])
True
sage: A([0])
False
```

See also:

AnyLetter(), AnyWord().

Word(*word*, *input_alphabet=None*)

Return an automaton recognizing the given word.

INPUT:

- word – an iterable.
- input_alphabet – a list or None. If None, then the letters occurring in the word are used.

OUTPUT:

An Automaton.

EXAMPLES:

```
sage: A = automata.Word([0])
sage: A.transitions()
[Transition from 0 to 1: 0|-]
sage: [A(w) for w in ([], [0], [1])]
[False, True, False]
sage: A = automata.Word([0, 1, 0])
sage: A.transitions()
[Transition from 0 to 1: 0|-,
Transition from 1 to 2: 1|-,
Transition from 2 to 3: 0|-]
sage: [A(w) for w in ([], [0], [0, 1], [0, 1, 1], [0, 1, 0])]
[False, False, False, False, True]
```

If the input alphabet is not given, it is derived from the given word.

```
sage: A.input_alphabet
[0, 1]
sage: A = automata.Word([0, 1, 0], input_alphabet=[0, 1, 2])
sage: A.input_alphabet
[0, 1, 2]
```

See also:

AnyWord(), ContainsWord().

TESTS:

```
sage: from sage.rings.integer import is_Integer
sage: all(is_Integer(s.label()) for s in A.states())
True
```

class sage.combinat.finite_state_machine_generators.**TransducerGenerators**

Bases: object

A collection of constructors for several common transducers.

A list of all transducers in this database is available via tab completion. Type "transducers." and then hit tab to see which transducers are available.

The transducers currently in this class include:

- Identity()
- abs()
- operator()
- all()
- any()
- add()
- sub()
- CountSubblockOccurrences()
- Wait()
- GrayCode()

CountSubblockOccurrences(*block*, *input_alphabet*)

Returns a transducer counting the number of (possibly overlapping) occurrences of a block in the input.

INPUT:

- `block` – a list (or other iterable) of letters.
- `input_alphabet` – a list or other iterable.

OUTPUT:

A transducer counting (in unary) the number of occurrences of the given block in the input. Overlapping occurrences are counted several times.

Denoting the block by $b_0 \dots b_{k-1}$, the input word by $i_0 \dots i_L$ and the output word by $o_0 \dots o_L$, we have $o_j = 1$ if and only if $i_{j-k+1} \dots i_j = b_0 \dots b_{k-1}$. Otherwise, $o_j = 0$.

EXAMPLES:

1. Counting the number of `10` blocks over the alphabet `[0, 1]`:

```
sage: T = transducers.CountSubblockOccurrences(
....:     [1, 0],
....:     [0, 1])
sage: sorted(T.transitions())
[Transition from () to (): 0|0,
 Transition from () to (1,): 1|0,
 Transition from (1,) to (): 0|1,
 Transition from (1,) to (1,): 1|0]
sage: T.input_alphabet
[0, 1]
sage: T.output_alphabet
[0, 1]
sage: T.initial_states()
[()]
sage: T.final_states()
[(), (1,)]
```

Check some sequence:

```
sage: T([0, 1, 0, 1, 1, 0])
[0, 0, 1, 0, 0, 1]
```

2. Counting the number of `11` blocks over the alphabet `[0, 1]`:

```
sage: T = transducers.CountSubblockOccurrences(
....:     [1, 1],
....:     [0, 1])
sage: sorted(T.transitions())
[Transition from () to (): 0|0,
 Transition from () to (1,): 1|0,
 Transition from (1,) to (): 0|0,
 Transition from (1,) to (1,): 1|1]
```

Check some sequence:

```
sage: T([0, 1, 0, 1, 1, 0])
[0, 0, 0, 0, 1, 0]
```

3. Counting the number of `1010` blocks over the alphabet `[0, 1, 2]`:

```
sage: T = transducers.CountSubblockOccurrences(
....:     [1, 0, 1, 0],
....:     [0, 1, 2])
sage: sorted(T.transitions())
[Transition from () to (): 0|0,
 Transition from () to (1,): 1|0,
 Transition from () to (): 2|0,
 Transition from (1,) to (1, 0): 0|0,
 Transition from (1,) to (1,): 1|0,
 Transition from (1,) to (): 2|0,
 Transition from (1, 0) to (): 0|0,
 Transition from (1, 0) to (1, 0, 1): 1|0,
 Transition from (1, 0) to (): 2|0,
 Transition from (1, 0, 1) to (1, 0): 0|1,
 Transition from (1, 0, 1) to (1,): 1|0,
 Transition from (1, 0, 1) to (): 2|0]
sage: input =  [0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 2]
sage: output = [0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0]
sage: T(input) == output
True
```

See also:

`ContainsWord()`

GrayCode()

Returns a transducer converting the standard binary expansion to Gray code.

INPUT:

Nothing.

OUTPUT:

A transducer.

Cf. the Wikipedia article Gray_code for a description of the Gray code.

EXAMPLE:

```
sage: G = transducers.GrayCode()
sage: G
Transducer with 3 states
sage: for v in srange(0, 10):
....:     print v, G(v.digits(base=2))
0 []
1 [1]
2 [1, 1]
3 [0, 1]
4 [0, 1, 1]
5 [1, 1, 1]
6 [1, 0, 1]
7 [0, 0, 1]
8 [0, 0, 1, 1]
9 [1, 0, 1, 1]
```

In the example *Gray Code* in the documentation of the *Finite State Machines, Automata, Transducers* module, the Gray code transducer is derived from the algorithm converting the binary expansion to the Gray code. The result is the same as the one given here.

Identity(*input_alphabet*)

Returns the identity transducer realizing the identity map.

INPUT:

- `input_alphabet` – a list or other iterable.

OUTPUT:

A transducer mapping each word over `input_alphabet` to itself.

EXAMPLES:

```
sage: T = transducers.Identity([0, 1])
sage: sorted(T.transitions())
[Transition from 0 to 0: 0|0,
 Transition from 0 to 0: 1|1]
sage: T.initial_states()
[0]
sage: T.final_states()
[0]
sage: T.input_alphabet
[0, 1]
sage: T.output_alphabet
[0, 1]
sage: T([0, 1, 0, 1, 1])
[0, 1, 0, 1, 1]
```

Recursion(*recursions, base, function=None, var=None, input_alphabet=None, word_function=None, is_zero=None, output_rings=[Integer Ring, Rational Field]*)

Return a transducer realizing the given recursion when reading the digit expansion with base `base`.

INPUT:

- `recursions` – list or iterable of equations. Each equation has either the form
  - `f(base^K * n + r) == f(base^k * n + s) + t` for some integers `0 <= k < K`, `r` and some `t`—valid for all `n` such that the arguments on both sides are non-negative—

  or the form
  - `f(r) == t` for some integer `r` and some `t`.

  Alternatively, an equation may be replaced by a `transducers.RecursionRule` with the attributes `K`, `r`, `k`, `s`, `t` as above or a tuple `(r, t)`. Note that `t` *must* be a list in this case.
- `base` – base of the digit expansion.
- `function` – symbolic function `f` occuring in the recursions.
- `var` – symbolic variable.
- `input_alphabet` – (default: `None`) a list of digits to be used as the input alphabet. If `None` and the base is an integer, `input_alphabet` is chosen to be `srange(base.abs())`.
- `word_function` – (default: `None`) a symbolic function. If not `None`, `word_function(arg1, ..., argn)` in a symbolic recurrence relation is interpreted as a transition with output `[arg1, ..., argn]`. This could not be entered in a symbolic recurrence relation because lists do not coerce into the `SymbolicRing`.
- `is_zero` – (default: `None`) a callable. The recursion relations are only well-posed if there is no cycle with non-zero output and input consisting of zeros. This parameter is used to determine whether the output of such a cycle is non-zero. By default, the output must evaluate to `False` as a boolean.

- `output_rings` – (default: `[ZZ, QQ]`) a list of rings. The output labels are converted into the first ring of the list in which they are contained. If they are not contained in any ring, they remain in whatever ring they are after parsing the recursions, typically the symbolic ring.

OUTPUT:

A transducer `T`.

The transducer is constructed such that `T(expansion) == f(n)` if `expansion` is the digit expansion of `n` to the base `base` with the given input alphabet as set of digits. Here, the + on the right hand side of the recurrence relation is interpreted as the concatenation of words.

The formal equations and initial conditions in the recursion have to be selected such that `f` is uniquely defined.

EXAMPLES:

- The following example computes the Hamming weight of the ternary expansion of integers.

```
sage: function('f')
f
sage: var('n')
n
sage: T = transducers.Recursion([
....:     f(3*n + 1) == f(n) + 1,
....:     f(3*n + 2) == f(n) + 1,
....:     f(3*n) == f(n),
....:     f(0) == 0],
....:     3, f, n)
sage: T.transitions()
[Transition from (0, 0) to (0, 0): 0|-,
 Transition from (0, 0) to (0, 0): 1|1,
 Transition from (0, 0) to (0, 0): 2|1]
```

To illustrate what this transducer does, we consider the example of $n = 601$:

```
sage: ternary_expansion = 601.digits(base=3)
sage: ternary_expansion
[1, 2, 0, 1, 1, 2]
sage: weight_sequence = T(ternary_expansion)
sage: weight_sequence
[1, 1, 1, 1, 1]
sage: sum(weight_sequence)
5
```

Note that the digit zero does not show up in the output because the equation `f(3*n) == f(n)` means that no output is added to `f(n)`.

- The following example computes the Hamming weight of the non-adjacent form, cf. the Wikipedia article Non-adjacent_form.

```
sage: function('f')
f
sage: var('n')
n
sage: T = transducers.Recursion([
....:     f(4*n + 1) == f(n) + 1,
....:     f(4*n - 1) == f(n) + 1,
```

```
....:     f(2*n) == f(n),
....:     f(0) == 0],
....:     2, f, n)
sage: T.transitions()
[Transition from (0, 0) to (0, 0): 0|-,
 Transition from (0, 0) to (1, 1): 1|-,
 Transition from (1, 1) to (0, 0): 0|1,
 Transition from (1, 1) to (1, 0): 1|1,
 Transition from (1, 0) to (1, 1): 0|-,
 Transition from (1, 0) to (1, 0): 1|-]
sage: [(s.label(), s.final_word_out)
....:  for s in T.iter_final_states()]
[((0, 0), []),
 ((1, 1), [1]),
 ((1, 0), [1])]
```

As we are interested in the weight only, we also output 1 for numbers congruent to 3 mod 4. The actual expansion is computed in the next example.

Consider the example of $29 = (100\bar{1}01)_2$ (as usual, the digit -1 is denoted by $\bar{1}$ and digits are written from the most significant digit at the left to the least significant digit at the right; for the transducer, we have to give the digits in the reverse order):

```
sage: NAF = [1, 0, -1, 0, 0, 1]
sage: ZZ(NAF, base=2)
29
sage: binary_expansion = 29.digits(base=2)
sage: binary_expansion
[1, 0, 1, 1, 1]
sage: T(binary_expansion)
[1, 1, 1]
sage: sum(T(binary_expansion))
3
```

Indeed, the given non-adjacent form has three non-zero digits.

•The following example computes the non-adjacent form from the binary expansion, cf. the Wikipedia article Non-adjacent_form. In contrast to the previous example, we actually compute the expansion, not only the weight.

We have to write the output 0 when converting an even number. This cannot be encoded directly by an equation in the symbolic ring, because `f(2*n) == f(n) + 0` would be equivalent to `f(2*n) == f(n)` and an empty output would be written. Therefore, we wrap the output in the symbolic function `w` and use the parameter `word_function` to announce this.

Similarly, we use `w(-1, 0)` to write an output word of length 2 in one iteration. Finally, we write `f(0) == w()` to write an empty word upon completion.

Moreover, there is a cycle with output `[0]` which—from the point of view of this method—is a contradicting recursion. We override this by the parameter `is_zero`.

```
sage: var('n')
n
sage: function('f w')
(f, w)
sage: T = transducers.Recursion([
....:     f(2*n) == f(n) + w(0),
```

```
....:       f(4*n + 1) == f(n) + w(1, 0),
....:       f(4*n - 1) == f(n) + w(-1, 0),
....:       f(0) == w()],
....:       2, f, n,
....:       word_function=w,
....:       is_zero=lambda x: sum(x).is_zero())
sage: T.transitions()
[Transition from (0, 0) to (0, 0): 0|0,
 Transition from (0, 0) to (1, 1): 1|-,
 Transition from (1, 1) to (0, 0): 0|1,0,
 Transition from (1, 1) to (1, 0): 1|-1,0,
 Transition from (1, 0) to (1, 1): 0|-,
 Transition from (1, 0) to (1, 0): 1|0]
sage: for s in T.iter_states():
....:     print s, s.final_word_out
(0, 0) []
(1, 1) [1, 0]
(1, 0) [1, 0]
```

We again consider the example of $n = 29$:

```
sage: T(29.digits(base=2))
[1, 0, -1, 0, 0, 1, 0]
```

The same transducer can also be entered bypassing the symbolic equations:

```
sage: R = transducers.RecursionRule
sage: TR = transducers.Recursion([
....:        R(K=1, r=0, k=0, s=0, t=[0]),
....:        R(K=2, r=1, k=0, s=0, t=[1, 0]),
....:        R(K=2, r=-1, k=0, s=0, t=[-1, 0]),
....:        (0, [])],
....:        2,
....:        is_zero=lambda x: sum(x).is_zero())
sage: TR == T
True
```

•Here is an artificial example where some of the s are negative:

```
sage: function('f')
f
sage: var('n')
n
sage: T = transducers.Recursion([
....:     f(2*n + 1) == f(n-1) + 1,
....:     f(2*n) == f(n),
....:     f(1) == 1,
....:     f(0) == 0], 2, f, n)
sage: T.transitions()
[Transition from (0, 0) to (0, 0): 0|-,
 Transition from (0, 0) to (1, 1): 1|-,
 Transition from (1, 1) to (-1, 1): 0|1,
 Transition from (1, 1) to (0, 0): 1|1,
 Transition from (-1, 1) to (-1, 2): 0|-,
```

```
 Transition from (-1, 1) to (1, 2): 1|-,
 Transition from (-1, 2) to (-1, 1): 0|1,
 Transition from (-1, 2) to (0, 0): 1|1,
 Transition from (1, 2) to (-1, 2): 0|1,
 Transition from (1, 2) to (1, 2): 1|1]
sage: [(s.label(), s.final_word_out)
....:  for s in T.iter_final_states()]
[((0, 0), []),
 ((1, 1), [1]),
 ((-1, 1), [0]),
 ((-1, 2), [0]),
 ((1, 2), [1])]
```

•Abelian complexity of the paperfolding sequence (cf. [HKP2015], Example 2.8):

```
sage: T = transducers.Recursion([
....:     f(4*n) == f(2*n),
....:     f(4*n+2) == f(2*n+1)+1,
....:     f(16*n+1) == f(8*n+1),
....:     f(16*n+5) == f(4*n+1)+2,
....:     f(16*n+11) == f(4*n+3)+2,
....:     f(16*n+15) == f(2*n+2)+1,
....:     f(1) == 2, f(0) == 0]
....:     + [f(16*n+jj) == f(2*n+1)+2 for jj in [3,7,9,13]],
....:     2, f, n)
sage: T.transitions()
[Transition from (0, 0) to (0, 1): 0|-,
 Transition from (0, 0) to (1, 1): 1|-,
 Transition from (0, 1) to (0, 1): 0|-,
 Transition from (0, 1) to (1, 1): 1|1,
 Transition from (1, 1) to (1, 2): 0|-,
 Transition from (1, 1) to (3, 2): 1|-,
 Transition from (1, 2) to (1, 3): 0|-,
 Transition from (1, 2) to (5, 3): 1|-,
 Transition from (3, 2) to (3, 3): 0|-,
 Transition from (3, 2) to (7, 3): 1|-,
 Transition from (1, 3) to (1, 3): 0|-,
 Transition from (1, 3) to (1, 1): 1|2,
 Transition from (5, 3) to (1, 2): 0|2,
 Transition from (5, 3) to (1, 1): 1|2,
 Transition from (3, 3) to (1, 1): 0|2,
 Transition from (3, 3) to (3, 2): 1|2,
 Transition from (7, 3) to (1, 1): 0|2,
 Transition from (7, 3) to (2, 1): 1|1,
 Transition from (2, 1) to (1, 1): 0|1,
 Transition from (2, 1) to (2, 1): 1|-]
sage: for s in T.iter_states():
....:     print s, s.final_word_out
(0, 0) []
(0, 1) []
(1, 1) [2]
(1, 2) [2]
(3, 2) [2, 2]
(1, 3) [2]
(5, 3) [2, 2]
(3, 3) [2, 2]
(7, 3) [2, 2]
```

```
(2, 1) [1, 2]
sage: list(sum(T(n.bits())) for n in srange(1, 21))
[2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6, 5, 4, 5, 4, 3, 4, 5, 6, 5]
```

•We now demonstrate the use of the `output_rings` parameter. If no `output_rings` are specified, the output labels are converted into `ZZ`:

```
sage: function('f')
f
sage: var('n')
n
sage: T = transducers.Recursion([
....:     f(2*n + 1) == f(n) + 1,
....:     f(2*n) == f(n),
....:     f(0) == 2],
....:     2, f, n)
sage: for t in T.transitions():
....:     print [x.parent() for x in t.word_out]
[]
[Integer Ring]
sage: [x.parent() for x in T.states()[0].final_word_out]
[Integer Ring]
```

In contrast, if `output_rings` is set to the empty list, the results are not converted:

```
sage: T = transducers.Recursion([
....:     f(2*n + 1) == f(n) + 1,
....:     f(2*n) == f(n),
....:     f(0) == 2],
....:     2, f, n, output_rings=[])
sage: for t in T.transitions():
....:     print [x.parent() for x in t.word_out]
[]
[Symbolic Ring]
sage: [x.parent() for x in T.states()[0].final_word_out]
[Symbolic Ring]
```

Finally, we use a somewhat questionable conversion:

```
sage: T = transducers.Recursion([
....:     f(2*n + 1) == f(n) + 1,
....:     f(2*n) == f(n),
....:     f(0) == 0],
....:     2, f, n, output_rings=[GF(5)])
sage: for t in T.transitions():
....:     print [x.parent() for x in t.word_out]
[]
[Finite Field of size 5]
```

Todo

Extend the method to

•non-integral bases,

•higher dimensions.

ALGORITHM:

See [HKP2015], Section 6. However, there are also recursion transitions for states of level $< \kappa$ if the recursion rules allow such a transition. Furthermore, the intermediate step of a non-deterministic transducer is left out by implicitly using recursion transitions. The well-posedness is checked in a truncated version of the recursion digraph.

TESTS:

The following tests fail due to missing or superfluous recursions or initial conditions.

```
sage: var('n')
n
sage: function('f')
f
sage: transducers.Recursion([f(2*n) == f(n)],
....:     2, f, n)
Traceback (most recent call last):
...
ValueError: Missing recursions for input congruent to
[1] modulo 2.
```

```
sage: transducers.Recursion([f(2*n + 1) == f(n),
....:                        f(4*n) == f(2*n) + 1,
....:                        f(2*n) == f(n) +1],
....:                       2, f, n)
Traceback (most recent call last):
...
ValueError: Conflicting rules congruent to 0 modulo 4.
```

```
sage: transducers.Recursion([f(2*n + 1) == f(n) + 1,
....:                        f(2*n) == f(n),
....:                        f(0) == 0,
....:                        f(42) == 42], 2, f, n)
Traceback (most recent call last):
...
ValueError: Superfluous initial values for [42].
```

```
sage: transducers.Recursion([f(2*n + 1) == f(n) + 1,
....:                        f(2*n) == f(n - 2) + 4,
....:                        f(0) == 0], 2, f, n)
Traceback (most recent call last):
...
ValueError: Missing initial values for [2].
```

Here is an example of a transducer with a conflicting rule (it cannot hold for $n = 0$):

```
sage: T = transducers.Recursion([
....:     f(2*n + 1) == f(n - 1),
....:     f(2*n) == f(n) + 1,
```

```
....:     f(1) == 1,
....:     f(0) == 0], 2, f, n)
Traceback (most recent call last):
...
ValueError: Conflicting recursion for [0].
```

class RecursionRule

Bases: tuple

K

Alias for field number 0

k

Alias for field number 2

r

Alias for field number 1

s

Alias for field number 3

t

Alias for field number 4

TransducerGenerators.**Wait**(*input_alphabet*, *threshold=1*)

Writes False until reading the threshold-th occurrence of a true input letter; then writes True.

INPUT:

- input_alphabet – a list or other iterable.
- threshold – a positive integer specifying how many occurrences of True inputs are waited for.

OUTPUT:

A transducer writing False until the threshold-th true (Python's standard conversion to boolean is used to convert the actual input to boolean) input is read. Subsequently, the transducer writes True.

EXAMPLES:

```
sage: T = transducers.Wait([0, 1])
sage: T([0, 0, 1, 0, 1, 0])
[False, False, True, True, True, True]
sage: T2 = transducers.Wait([0, 1], threshold=2)
sage: T2([0, 0, 1, 0, 1, 0])
[False, False, False, False, True, True]
```

TransducerGenerators.**abs**(*input_alphabet*)

Returns a transducer which realizes the letter-wise absolute value of an input word over the given input alphabet.

INPUT:

- input_alphabet – a list or other iterable.

OUTPUT:

A transducer mapping $i_0 \dots i_k$ to $|i_0| \dots |i_k|$.

EXAMPLE:

The following transducer realizes letter-wise absolute value:

```
sage: T = transducers.abs([-1, 0, 1])
sage: T.transitions()
[Transition from 0 to 0: -1|1,
 Transition from 0 to 0: 0|0,
 Transition from 0 to 0: 1|1]
sage: T.initial_states()
[0]
sage: T.final_states()
[0]
sage: T([-1, -1, 0, 1])
[1, 1, 0, 1]
```

TransducerGenerators.**add**(*input_alphabet*, *number_of_operands=2*)

Returns a transducer which realizes addition on pairs over the given input alphabet.

INPUT:

- `input_alphabet` – a list or other iterable.
- `number_of_operands` – (default: 2) it specifies the number of input arguments the operator takes.

OUTPUT:

A transducer mapping an input word $(i_{01}, \ldots, i_{0d}) \ldots (i_{k1}, \ldots, i_{kd})$ to the word $(i_{01} + \cdots + i_{0d}) \ldots (i_{k1} + \cdots + i_{kd})$.

The input alphabet of the generated transducer is the Cartesian product of `number_of_operands` copies of `input_alphabet`.

EXAMPLE:

The following transducer realizes letter-wise addition:

```
sage: T = transducers.add([0, 1])
sage: T.transitions()
[Transition from 0 to 0: (0, 0)|0,
 Transition from 0 to 0: (0, 1)|1,
 Transition from 0 to 0: (1, 0)|1,
 Transition from 0 to 0: (1, 1)|2]
sage: T.input_alphabet
[(0, 0), (0, 1), (1, 0), (1, 1)]
sage: T.initial_states()
[0]
sage: T.final_states()
[0]
sage: T([(0, 0), (0, 1), (1, 0), (1, 1)])
[0, 1, 1, 2]
```

More than two operands can also be handled:

```
sage: T3 = transducers.add([0, 1], number_of_operands=3)
sage: T3.input_alphabet
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1),
 (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
sage: T3([(0, 0, 0), (0, 1, 0), (0, 1, 1), (1, 1, 1)])
[0, 1, 2, 3]
```

TransducerGenerators.**all**(*input_alphabet*, *number_of_operands=2*)

Returns a transducer which realizes logical `and` over the given input alphabet.

INPUT:

- input_alphabet – a list or other iterable.
- number_of_operands – (default: 2) specifies the number of input arguments for the and operation.

OUTPUT:

A transducer mapping an input word $(i_{01}, \ldots, i_{0d}) \ldots (i_{k1}, \ldots, i_{kd})$ to the word $(i_{01} \wedge \cdots \wedge i_{0d}) \ldots (i_{k1} \wedge \cdots \wedge i_{kd})$.

The input alphabet of the generated transducer is the Cartesian product of number_of_operands copies of input_alphabet.

EXAMPLE:

The following transducer realizes letter-wise logical and:

```
sage: T = transducers.all([False, True])
sage: T.transitions()
[Transition from 0 to 0: (False, False)|False,
 Transition from 0 to 0: (False, True)|False,
 Transition from 0 to 0: (True, False)|False,
 Transition from 0 to 0: (True, True)|True]
sage: T.input_alphabet
[(False, False), (False, True), (True, False), (True, True)]
sage: T.initial_states()
[0]
sage: T.final_states()
[0]
sage: T([(False, False), (False, True), (True, False), (True, True)])
[False, False, False, True]
```

More than two operands and other input alphabets (with conversion to boolean) are also possible:

```
sage: T3 = transducers.all([0, 1], number_of_operands=3)
sage: T3([(0, 0, 0), (1, 0, 0), (1, 1, 1)])
[False, False, True]
```

TransducerGenerators.**any**(*input_alphabet*, *number_of_operands=2*)

Returns a transducer which realizes logical or over the given input alphabet.

INPUT:

- input_alphabet – a list or other iterable.
- number_of_operands – (default: 2) specifies the number of input arguments for the or operation.

OUTPUT:

A transducer mapping an input word $(i_{01}, \ldots, i_{0d}) \ldots (i_{k1}, \ldots, i_{kd})$ to the word $(i_{01} \vee \cdots \vee i_{0d}) \ldots (i_{k1} \vee \cdots \vee i_{kd})$.

The input alphabet of the generated transducer is the Cartesian product of number_of_operands copies of input_alphabet.

EXAMPLE:

The following transducer realizes letter-wise logical or:

```
sage: T = transducers.any([False, True])
sage: T.transitions()
[Transition from 0 to 0: (False, False)|False,
 Transition from 0 to 0: (False, True)|True,
```

```
 Transition from 0 to 0: (True, False)|True,
 Transition from 0 to 0: (True, True)|True]
sage: T.input_alphabet
[(False, False), (False, True), (True, False), (True, True)]
sage: T.initial_states()
[0]
sage: T.final_states()
[0]
sage: T([(False, False), (False, True), (True, False), (True, True)])
[False, True, True, True]
```

More than two operands and other input alphabets (with conversion to boolean) are also possible:

```
sage: T3 = transducers.any([0, 1], number_of_operands=3)
sage: T3([(0, 0, 0), (1, 0, 0), (1, 1, 1)])
[False, True, True]
```

TransducerGenerators.**map**(*f*, *input_alphabet*)

Return a transducer which realizes a function on the alphabet.

INPUT:

- `f` – function to realize.
- `input_alphabet` – a list or other iterable.

OUTPUT:

A transducer mapping an input letter x to $f(x)$.

EXAMPLE:

The following binary transducer realizes component-wise absolute value (this transducer is also available as `abs()`):

```
sage: T = transducers.map(abs, [-1, 0, 1])
sage: T.transitions()
[Transition from 0 to 0: -1|1,
 Transition from 0 to 0: 0|0,
 Transition from 0 to 0: 1|1]
sage: T.input_alphabet
[-1, 0, 1]
sage: T.initial_states()
[0]
sage: T.final_states()
[0]
sage: T([-1, 1, 0, 1])
[1, 1, 0, 1]
```

See also:

`Automaton.with_output()`.

TransducerGenerators.**operator**(*operator*, *input_alphabet*, *number_of_operands=2*)

Returns a transducer which realizes an operation on tuples over the given input alphabet.

INPUT:

- `operator` – operator to realize. It is a function which takes `number_of_operands` input arguments (each out of `input_alphabet`).
- `input_alphabet` – a list or other iterable.

- number_of_operands – (default: 2) it specifies the number of input arguments the operator takes.

OUTPUT:

A transducer mapping an input letter $(i_1, \ldots, i_n)$ to operator$(i_1, \ldots, i_n)$. Here, n equals number_of_operands.

The input alphabet of the generated transducer is the Cartesian product of number_of_operands copies of input_alphabet.

EXAMPLE:

The following binary transducer realizes component-wise addition (this transducer is also available as add()):

```
sage: import operator
sage: T = transducers.operator(operator.add, [0, 1])
sage: T.transitions()
[Transition from 0 to 0: (0, 0)|0,
 Transition from 0 to 0: (0, 1)|1,
 Transition from 0 to 0: (1, 0)|1,
 Transition from 0 to 0: (1, 1)|2]
sage: T.input_alphabet
[(0, 0), (0, 1), (1, 0), (1, 1)]
sage: T.initial_states()
[0]
sage: T.final_states()
[0]
sage: T([(0, 0), (0, 1), (1, 0), (1, 1)])
[0, 1, 1, 2]
```

Note that for a unary operator the input letters of the new transducer are tuples of length 1:

```
sage: T = transducers.operator(abs,
....:                          [-1, 0, 1],
....:                          number_of_operands=1)
sage: T([-1, 1, 0])
Traceback (most recent call last):
...
ValueError: Invalid input sequence.
sage: T([(-1,), (1,), (0,)])
[1, 1, 0]
```

Compare this with the transducer generated by map():

```
sage: T = transducers.map(abs,
....:                     [-1, 0, 1])
sage: T([-1, 1, 0])
[1, 1, 0]
```

In fact, this transducer is also available as abs():

```
sage: T = transducers.abs([-1, 0, 1])
sage: T([-1, 1, 0])
[1, 1, 0]
```

TransducerGenerators.**sub**(*input_alphabet*)

Returns a transducer which realizes subtraction on pairs over the given input alphabet.

INPUT:

- input_alphabet – a list or other iterable.

OUTPUT:

A transducer mapping an input word $(i_0, i'_0) \dots (i_k, i'_k)$ to the word $(i_0 - i'_0) \dots (i_k - i'_k)$.

The input alphabet of the generated transducer is the Cartesian product of two copies of `input_alphabet`.

EXAMPLE:

The following transducer realizes letter-wise subtraction:

```
sage: T = transducers.sub([0, 1])
sage: T.transitions()
[Transition from 0 to 0: (0, 0)|0,
 Transition from 0 to 0: (0, 1)|-1,
 Transition from 0 to 0: (1, 0)|1,
 Transition from 0 to 0: (1, 1)|0]
sage: T.input_alphabet
[(0, 0), (0, 1), (1, 0), (1, 1)]
sage: T.initial_states()
[0]
sage: T.final_states()
[0]
sage: T([(0, 0), (0, 1), (1, 0), (1, 1)])
[0, -1, 1, 0]
```

`TransducerGenerators.`**`weight`**(*input_alphabet*, *zero=0*)

Returns a transducer which realizes the Hamming weight of the input over the given input alphabet.

INPUT:

- `input_alphabet` – a list or other iterable.
- `zero` – the zero symbol in the alphabet used

OUTPUT:

A transducer mapping $i_0 \dots i_k$ to $(i_0 \neq 0) \dots (i_k \neq 0)$.

The Hamming weight is defined as the number of non-zero digits in the input sequence over the alphabet `input_alphabet` (see Wikipedia article Hamming_weight). The output sequence of the transducer is a unary encoding of the Hamming weight. Thus the sum of the output sequence is the Hamming weight of the input.

EXAMPLES:

```
sage: W = transducers.weight([-1, 0, 2])
sage: W.transitions()
[Transition from 0 to 0: -1|1,
 Transition from 0 to 0: 0|0,
 Transition from 0 to 0: 2|1]
sage: unary_weight = W([-1, 0, 0, 2, -1])
sage: unary_weight
[1, 0, 0, 1, 1]
sage: weight = add(unary_weight)
sage: weight
3
```

Also the joint Hamming weight can be computed:

```
sage: v1 = vector([-1, 0])
sage: v0 = vector([0, 0])
sage: W = transducers.weight([v1, v0])
sage: unary_weight = W([v1, v0, v1, v0])
```

```
sage: add(unary_weight)
2
```

For the input alphabet `[-1, 0, 1]` the weight transducer is the same as the absolute value transducer `abs()`:

```
sage: W = transducers.weight([-1, 0, 1])
sage: A = transducers.abs([-1, 0, 1])
sage: W == A
True
```

For other input alphabets, we can specify the zero symbol:

```
sage: W = transducers.weight(['a', 'b'], zero='a')
sage: add(W(['a', 'b', 'b']))
2
```

5.1.98 Free modules

class sage.combinat.free_module.**CartesianProductWithFlattening**(*flatten*)

Bases: object

A class for Cartesian product constructor, with partial flattening

class sage.combinat.free_module.**CombinatorialFreeModule**(*R*, *basis_keys*, *element_class=None*, *category=None*, *prefix='B'*, ***kwds*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.modules.module.Module, sage.structure.indexed_generators.IndexedGenerators

Class for free modules with a named basis

INPUT:

- `R` - base ring
- `basis_keys` - list, tuple, family, set, etc. defining the indexing set for the basis of this module
- `element_class` - the class of which elements of this module should be instances (optional, default None, in which case the elements are instances of `CombinatorialFreeModuleElement`)
- `category` - the category in which this module lies (optional, default None, in which case use the "category of modules with basis" over the base ring `R`); this should be a subcategory of `ModulesWithBasis`

For the options controlling the printing of elements, see `IndexedGenerators`.

Note: These print options may also be accessed and modified using the `print_options()` method, after the module has been defined.

EXAMPLES:

We construct a free module whose basis is indexed by the letters a, b, c:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: F
Free module generated by {'a', 'b', 'c'} over Rational Field
```

Its basis is a family, indexed by a, b, c:

```
sage: e = F.basis()
sage: e
Finite family {'a': B['a'], 'c': B['c'], 'b': B['b']}

sage: [x for x in e]
[B['a'], B['b'], B['c']]
sage: [k for k in e.keys()]
['a', 'b', 'c']
```

Let us construct some elements, and compute with them:

```
sage: e['a']
B['a']
sage: 2*e['a']
2*B['a']
sage: e['a'] + 3*e['b']
B['a'] + 3*B['b']
```

Some uses of `sage.categories.commutative_additive_semigroups.CommutativeAdditiveSemigroups` and `sum()`:

```
sage: F = CombinatorialFreeModule(QQ, [1,2,3,4])
sage: F.summation(F.monomial(1), F.monomial(3))
B[1] + B[3]

sage: F = CombinatorialFreeModule(QQ, [1,2,3,4])
sage: F.sum(F.monomial(i) for i in [1,2,3])
B[1] + B[2] + B[3]
```

Note that free modules with a given basis and parameters are unique:

```
sage: F1 = CombinatorialFreeModule(QQ, (1,2,3,4))
sage: F1 is F
True
```

The identity of the constructed free module depends on the order of the basis and on the other parameters, like the prefix. Note that `CombinatorialFreeModule` is a `UniqueRepresentation`. Hence, two combinatorial free modules evaluate equal if and only if they are identical:

```
sage: F1 = CombinatorialFreeModule(QQ, (1,2,3,4))
sage: F1 is F
True
sage: F1 = CombinatorialFreeModule(QQ, [4,3,2,1])
sage: F1 == F
False
sage: F2 = CombinatorialFreeModule(QQ, [1,2,3,4], prefix='F')
sage: F2 == F
False
```

Because of this, if you create a free module with certain parameters and then modify its prefix or other print options, this affects all modules which were defined using the same parameters.

```
sage: F2.print_options(prefix='x')
sage: F2.prefix()
'x'
sage: F3 = CombinatorialFreeModule(QQ, [1,2,3,4], prefix='F')
sage: F3 is F2   # F3 was defined just like F2
True
sage: F3.prefix()
'x'
```

```
sage: F4 = CombinatorialFreeModule(QQ, [1,2,3,4], prefix='F', bracket=True)
sage: F4 == F2   # F4 was NOT defined just like F2
False
sage: F4.prefix()
'F'

sage: F2.print_options(prefix='F') #reset for following doctests
```

The constructed module is in the category of modules with basis over the base ring:

```
sage: CombinatorialFreeModule(QQ, Partitions()).category()
Category of vector spaces with basis over Rational Field
```

If furthermore the index set is finite (i.e. in the category `Sets().Finite()`), then the module is declared as being finite dimensional:

```
sage: CombinatorialFreeModule(QQ, [1,2,3,4]).category()
Category of finite dimensional vector spaces with basis over Rational Field
sage: CombinatorialFreeModule(QQ, Partitions(3),
....:                         category=Algebras(QQ).WithBasis()).category()
Category of finite dimensional algebras with basis over Rational Field
```

See `sage.categories.examples.algebras_with_basis` and `sage.categories.examples.hopf_algebra` for illustrations of the use of the `category` keyword, and see `sage.combinat.root_system.weight_space.WeightSpace` for an example of the use of `element_class`.

Customizing print and LaTeX representations of elements:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b'], prefix='x')
sage: original_print_options = F.print_options()
sage: sorted(original_print_options.items())
[('bracket', None), ('generator_cmp', <built-in function cmp>),
 ('latex_bracket', False), ('latex_prefix', None),
 ('latex_scalar_mult', None), ('prefix', 'x'),
 ('scalar_mult', '*'), ('string_quotes', True),
 ('tensor_symbol', None)]

sage: e = F.basis()
sage: e['a'] - 3 * e['b']
x['a'] - 3*x['b']

sage: F.print_options(prefix='x', scalar_mult=' ', bracket='{')
sage: e['a'] - 3 * e['b']
x{'a'} - 3 x{'b'}
sage: latex(e['a'] - 3 * e['b'])
x_{a} - 3 x_{b}

sage: F.print_options(latex_prefix='y')
sage: latex(e['a'] - 3 * e['b'])
y_{a} - 3  y_{b}

sage: F.print_options(generator_cmp = lambda x,y: -cmp(x,y))
sage: e['a'] - 3 * e['b']
-3 x{'b'} + x{'a'}
sage: F.print_options(**original_print_options) # reset print options

sage: F = CombinatorialFreeModule(QQ, [(1,2), (3,4)])
sage: e = F.basis()
```

```
sage: e[(1,2)] - 3 * e[(3,4)]
B[(1, 2)] - 3*B[(3, 4)]

sage: F.print_options(bracket=['_{', '}'])
sage: e[(1,2)] - 3 * e[(3,4)]
B_{(1, 2)} - 3*B_{(3, 4)}

sage: F.print_options(prefix='', bracket=False)
sage: e[(1,2)] - 3 * e[(3,4)]
(1, 2) - 3*(3, 4)
```

TESTS:

Before trac ticket #14054, combinatorial free modules violated the unique parent condition. That caused a problem. The tensor product construction involves maps, but maps check that their domain and the parent of a to-be-mapped element are identical (not just equal). However, the tensor product was cached by a `cached_method`, which involves comparison by equality (not identity). Hence, the last line of the following example used to fail with an assertion error:

```
sage: F = CombinatorialFreeModule(ZZ, [1,2,3], prefix="F")
sage: G = CombinatorialFreeModule(ZZ, [1,2,3,4], prefix="G")
sage: f =   F.monomial(1) + 2 * F.monomial(2)
sage: g = 2*G.monomial(3) +     G.monomial(4)
sage: tensor([f, g])
2*F[1] # G[3] + F[1] # G[4] + 4*F[2] # G[3] + 2*F[2] # G[4]
sage: F = CombinatorialFreeModule(ZZ, [1,2,3], prefix='x')
sage: G = CombinatorialFreeModule(ZZ, [1,2,3,4], prefix='y')
sage: f =   F.monomial(1) + 2 * F.monomial(2)
sage: g = 2*G.monomial(3) +     G.monomial(4)
sage: tensor([f, g])
2*x[1] # y[3] + x[1] # y[4] + 4*x[2] # y[3] + 2*x[2] # y[4]
```

CartesianProduct
alias of CombinatorialFreeModule_CartesianProduct

Element
alias of CombinatorialFreeModuleElement

Tensor
alias of CombinatorialFreeModule_Tensor

dimension()
Return the dimension of the free module (which is given by the number of elements in the basis).

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.dimension()
3
sage: F.basis().cardinality()
3
sage: F.basis().keys().cardinality()
3

sage: s = SymmetricFunctions(QQ).schur()
sage: s.dimension()
+Infinity
```

from_vector(*vector*)
Build an element of `self` from a (sparse) vector.

See also:

get_order(), CombinatorialFreeModule.Element._vector_()

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: b = QS3.from_vector(vector((2, 0, 0, 0, 0, 4))); b
2*[1, 2, 3] + 4*[3, 2, 1]
sage: a = 2*QS3([1,2,3])+4*QS3([3,2,1])
sage: a == b
True
```

gens()

Return a tuple consisting of the basis elements of self.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(ZZ, ['a', 'b', 'c'])
sage: F.gens()
(B['a'], B['b'], B['c'])
```

get_order()

Return the order of the elements in the basis.

EXAMPLES:

```
sage: QS2 = SymmetricGroupAlgebra(QQ,2)
sage: QS2.get_order()                    # note: order changed on 2009-03-13
[[2, 1], [1, 2]]
```

get_order_cmp()

Return a comparison function on the basis indices that is compatible with the current term order.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example()
sage: Acmp = A.get_order_cmp()
sage: sorted(A.basis().keys(), Acmp)
['x', 'y', 'a', 'b']
sage: A.set_order(list(reversed(A.basis().keys())))
sage: Acmp = A.get_order_cmp()
sage: sorted(A.basis().keys(), Acmp)
['b', 'a', 'y', 'x']
```

linear_combination(*iter_of_elements_coeff*, *factor_on_left=True*)

Return the linear combination $\lambda_1 v_1 + \cdots + \lambda_k v_k$ (resp. the linear combination $v_1 \lambda_1 + \cdots + v_k \lambda_k$) where iter_of_elements_coeff iterates through the sequence $((\lambda_1, v_1), ..., (\lambda_k, v_k))$.

INPUT:

- iter_of_elements_coeff – iterator of pairs (element, coeff) with element in self and coeff in self.base_ring()
- factor_on_left – (optional) if True, the coefficients are multiplied from the left if False, the coefficients are multiplied from the right

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ,[1,2])
sage: f = F.an_element(); f
2*B[1] + 2*B[2]
sage: F.linear_combination( (f,i) for i in range(5) )
20*B[1] + 20*B[2]
```

monomial()
Return the basis element indexed by i.

INPUT:

- i – an element of the index set

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.monomial('a')
B['a']
```

F.monomial is in fact (almost) a map:

```
sage: F.monomial
Term map from {'a', 'b', 'c'} to Free module generated by {'a', 'b', 'c'} over Rational Fiel
```

set_order(*order*)
Set the order of the elements of the basis.

If set_order() has not been called, then the ordering is the one used in the generation of the elements of self's associated enumerated set.

Warning: Many cached methods depend on this order, in particular for constructing subspaces and quotients. Changing the order after some computations have been cached does not invalidate the cache, and is likely to introduce inconsistencies.

EXAMPLES:

```
sage: QS2 = SymmetricGroupAlgebra(QQ,2)
sage: b = list(QS2.basis().keys())
sage: b.reverse()
sage: QS2.set_order(b)
sage: QS2.get_order()
[[2, 1], [1, 2]]
```

sum(*iter_of_elements*)
Return the sum of all elements in iter_of_elements.

Overrides method inherited from commutative additive monoid as it is much faster on dicts directly.

INPUT:

- iter_of_elements – iterator of elements of self

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ,[1,2])
sage: f = F.an_element(); f
2*B[1] + 2*B[2]
sage: F.sum( f for _ in range(5) )
10*B[1] + 10*B[2]
```

sum_of_terms(*terms*, *distinct=False*)
Construct a sum of terms of self.

INPUT:

- terms – a list (or iterable) of pairs (index, coeff)

- distinct – (default: False) whether the indices are guaranteed to be distinct

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.sum_of_terms([('a',2), ('c',3)])
2*B['a'] + 3*B['c']
```

If distinct is True, then the construction is optimized:

```
sage: F.sum_of_terms([('a',2), ('c',3)], distinct = True)
2*B['a'] + 3*B['c']
```

Warning: Use distinct=True only if you are sure that the indices are indeed distinct:

```
sage: F.sum_of_terms([('a',2), ('a',3)], distinct = True)
3*B['a']
```

Extreme case:

```
sage: F.sum_of_terms([])
0
```

term(*index*, *coeff=None*)

Construct a term in self.

INPUT:

- index – the index of a basis element
- coeff – an element of the coefficient ring (default: one)

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.term('a',3)
3*B['a']
sage: F.term('a')
B['a']
```

Design: should this do coercion on the coefficient ring?

zero()

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.zero()
0
```

class sage.combinat.free_module.**CombinatorialFreeModuleElement**(*M*, *x*)

Bases: sage.structure.element.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
```

```
sage: f == loads(dumps(f))
True
```

monomial_coefficients(*copy=True*)

Return the internal dictionary which has the combinatorial objects indexing the basis as keys and their corresponding coefficients as values.

INPUT:

- copy – (default: True) if self is internally represented by a dictionary d, then make a copy of d; if False, then this can cause undesired behavior by mutating d

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']
sage: d = f.monomial_coefficients()
sage: d['a']
1
sage: d['c']
3
```

To run through the monomials of an element, it is better to use the idiom:

```
sage: for (t,c) in f:
...       print t,c
a 1
c 3

sage: s = SymmetricFunctions(QQ).schur()
sage: a = s([2,1])+2*s([3,2])
sage: d = a.monomial_coefficients()
sage: type(d)
<type 'dict'>
sage: d[ Partition([2,1]) ]
1
sage: d[ Partition([3,2]) ]
2
```

to_vector(*new_base_ring=None*)

Returns self as a dense vector

INPUT:

- new_base_ring – a ring (default: None)

OUTPUT: a dense FreeModule() vector

Warning: This will crash/run forever if self is infinite dimensional!

See also:

- vector()
- CombinatorialFreeModule.get_order()
- CombinatorialFreeModule.from_vector()
- CombinatorialFreeModule._dense_free_module()

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] - 3*B['c']
sage: f._vector_()
(1, 0, -3)
```

One can use equivalently:

```
sage: f.to_vector()
(1, 0, -3)
sage: vector(f)
(1, 0, -3)
```

More examples:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: a = 2*QS3([1,2,3])+4*QS3([3,2,1])
sage: a._vector_()
(2, 0, 0, 0, 0, 4)
sage: a.to_vector()
(2, 0, 0, 0, 0, 4)
sage: vector(a)
(2, 0, 0, 0, 0, 4)
sage: a == QS3.from_vector(a.to_vector())
True
```

If `new_base_ring` is specified, then a vector over `new_base_ring` is returned:

```
sage: a._vector_(RDF)
(2.0, 0.0, 0.0, 0.0, 0.0, 4.0)
```

Note: trac ticket #13406: the current implementation has been optimized, at the price of breaking the encapsulation for FreeModule elements creation, with the following use case as metric, on a 2008' Macbook Pro:

```
sage: F = CombinatorialFreeModule(QQ, range(10))
sage: f = F.an_element()
sage: %timeit f._vector_()   # not tested
625 loops, best of 3: 17.5 micros per loop

Other use cases may call for different or further
optimizations.
```

---

class sage.combinat.free_module.**CombinatorialFreeModule_CartesianProduct**(*modules*, ***options*)

Bases: sage.combinat.free_module.CombinatorialFreeModule

An implementation of Cartesian products of modules with basis

EXAMPLES:

We construct two free modules, assign them short names, and construct their Cartesian product:

```
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.__custom_name = "F"
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.__custom_name = "G"
sage: H = CombinatorialFreeModule(ZZ, [4,7]); H.__custom_name = "H"
sage: S = cartesian_product([F, G])
sage: S
```

```
F (+) G
sage: S.basis()
Lazy family (Term map from Disjoint union of Family ({4, 5}, {4, 6}) to F (+) G(i))_{i in Disjoi
```

Note that the indices of the basis elements of F and G intersect non trivially. This is handled by forcing the union to be disjoint:

```
sage: list(S.basis())
[B[(0, 4)], B[(0, 5)], B[(1, 4)], B[(1, 6)]]
```

We now compute the Cartesian product of elements of free modules:

```
sage: f =   F.monomial(4) + 2 * F.monomial(5)
sage: g = 2*G.monomial(4) +     G.monomial(6)
sage: h =   H.monomial(4) +     H.monomial(7)
sage: cartesian_product([f,g])
B[(0, 4)] + 2*B[(0, 5)] + 2*B[(1, 4)] + B[(1, 6)]
sage: cartesian_product([f,g,h])
B[(0, 4)] + 2*B[(0, 5)] + 2*B[(1, 4)] + B[(1, 6)] + B[(2, 4)] + B[(2, 7)]
sage: cartesian_product([f,g,h]).parent()
F (+) G (+) H
```

TODO: choose an appropriate semantic for Cartesian products of Cartesian products (associativity?):

```
sage: S = cartesian_product([cartesian_product([F, G]), H]) # todo: not implemented
F (+) G (+) H
```

class `Element`(*M*, *x*)

Bases: `sage.combinat.free_module.CombinatorialFreeModuleElement`

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

`CombinatorialFreeModule_CartesianProduct.`**`cartesian_embedding`**(*i*)

Return the natural embedding morphism of the `i`-th Cartesian factor (summand) of `self` into `self`.

INPUT:

- `i` – an integer

EXAMPLES:

```
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.__custom_name = "F"
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.__custom_name = "G"
sage: S = cartesian_product([F, G])
sage: phi = S.cartesian_embedding(0)
sage: phi(F.monomial(4) + 2 * F.monomial(5))
B[(0, 4)] + 2*B[(0, 5)]
sage: phi(F.monomial(4) + 2 * F.monomial(6)).parent() == S
True
```

TESTS:

```
sage: phi(G.monomial(4))
Traceback (most recent call last):
...
AssertionError
```

CombinatorialFreeModule_CartesianProduct.**cartesian_factors**()

Return the factors of the Cartesian product.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.__custom_name = "F"
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.__custom_name = "G"
sage: S = cartesian_product([F, G])
sage: S.cartesian_factors()
(F, G)
```

CombinatorialFreeModule_CartesianProduct.**cartesian_projection**(*i*)

Return the natural projection onto the i-th Cartesian factor (summand) of `self`.

INPUT:

- `i` – an integer

EXAMPLE:

```
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.__custom_name = "F"
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.__custom_name = "G"
sage: S = cartesian_product([F, G])
sage: x = S.monomial((0,4)) + 2 * S.monomial((0,5)) + 3 * S.monomial((1,6))
sage: S.cartesian_projection(0)(x)
B[4] + 2*B[5]
sage: S.cartesian_projection(1)(x)
3*B[6]
sage: S.cartesian_projection(0)(x).parent() == F
True
sage: S.cartesian_projection(1)(x).parent() == G
True
```

CombinatorialFreeModule_CartesianProduct.**summand_embedding**(*i*)

Return the natural embedding morphism of the `i`-th Cartesian factor (summand) of `self` into `self`.

INPUT:

- `i` – an integer

EXAMPLES:

```
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.__custom_name = "F"
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.__custom_name = "G"
sage: S = cartesian_product([F, G])
sage: phi = S.cartesian_embedding(0)
sage: phi(F.monomial(4) + 2 * F.monomial(5))
B[(0, 4)] + 2*B[(0, 5)]
sage: phi(F.monomial(4) + 2 * F.monomial(6)).parent() == S
True
```

TESTS:

```
sage: phi(G.monomial(4))
Traceback (most recent call last):
...
AssertionError
```

CombinatorialFreeModule_CartesianProduct.**summand_projection**(*i*)

Return the natural projection onto the i-th Cartesian factor (summand) of `self`.

INPUT:

- `i` – an integer

EXAMPLE:

```
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.__custom_name = "F"
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.__custom_name = "G"
sage: S = cartesian_product([F, G])
sage: x = S.monomial((0,4)) + 2 * S.monomial((0,5)) + 3 * S.monomial((1,6))
sage: S.cartesian_projection(0)(x)
B[4] + 2*B[5]
sage: S.cartesian_projection(1)(x)
3*B[6]
sage: S.cartesian_projection(0)(x).parent() == F
True
sage: S.cartesian_projection(1)(x).parent() == G
True
```

class sage.combinat.free_module.**CombinatorialFreeModule_Tensor**(*modules*, ***options*)

Bases: sage.combinat.free_module.CombinatorialFreeModule

Tensor Product of Free Modules

EXAMPLES:

We construct two free modules, assign them short names, and construct their tensor product:

```
sage: F = CombinatorialFreeModule(ZZ, [1,2]); F.__custom_name = "F"
sage: G = CombinatorialFreeModule(ZZ, [3,4]); G.__custom_name = "G"
sage: T = tensor([F, G]); T
F # G

sage: T.category()
Category of finite dimensional tensor products of modules with basis over Integer Ring

sage: T.construction() # todo: not implemented
[tensor, ]
```

T is a free module, with same base ring as F and G:

```
sage: T.base_ring()
Integer Ring
```

The basis of T is indexed by tuples of basis indices of F and G:

```
sage: T.basis().keys()
Image of Cartesian product of {1, 2}, {3, 4} by <type 'tuple'>
sage: T.basis().keys().list()
[(1, 3), (1, 4), (2, 3), (2, 4)]
```

FIXME: Should elements of a CartesianProduct be tuples (making them hashable)?

Here are the basis elements themselves:

```
sage: T.basis().cardinality()
4
sage: list(T.basis())
[B[1] # B[3], B[1] # B[4], B[2] # B[3], B[2] # B[4]]
```

The tensor product is associative and flattens sub tensor products:

```
sage: H = CombinatorialFreeModule(ZZ, [5,6]); H.rename("H")
sage: tensor([F, tensor([G, H])])
F # G # H
sage: tensor([tensor([F, G]), H])
F # G # H
sage: tensor([F, G, H])
F # G # H
```

We now compute the tensor product of elements of free modules:

```
sage: f =   F.monomial(1) + 2 * F.monomial(2)
sage: g = 2*G.monomial(3) +     G.monomial(4)
sage: h =   H.monomial(5) +     H.monomial(6)
sage: tensor([f, g])
2*B[1] # B[3] + B[1] # B[4] + 4*B[2] # B[3] + 2*B[2] # B[4]
```

Again, the tensor product is associative on elements:

```
sage: tensor([f, tensor([g, h])]) == tensor([f, g, h])
True
sage: tensor([tensor([f, g]), h]) == tensor([f, g, h])
True
```

Note further that the tensor product spaces need not preexist:

```
sage: t = tensor([f, g, h])
sage: t.parent()
F # G # H
```

TESTS:

```
sage: tensor([tensor([F, G]), H]) == tensor([F, G, H])
True
sage: tensor([F, tensor([G, H])]) == tensor([F, G, H])
True
```

tensor_constructor(*modules*)

INPUT:

- `modules` – a tuple $(F_1, \ldots, F_n)$ of free modules whose tensor product is self

Returns the canonical multilinear morphism from $F_1 \times \cdots \times F_n$ to $F_1 \otimes \cdots \otimes F_n$

EXAMPLES:

```
sage: F = CombinatorialFreeModule(ZZ, [1,2]); F.__custom_name = "F"
sage: G = CombinatorialFreeModule(ZZ, [3,4]); G.__custom_name = "G"
sage: H = CombinatorialFreeModule(ZZ, [5,6]); H.rename("H")

sage: f =   F.monomial(1) + 2 * F.monomial(2)
sage: g = 2*G.monomial(3) +     G.monomial(4)
sage: h =   H.monomial(5) +     H.monomial(6)
sage: FG  = tensor([F, G   ])
sage: phi_fg = FG.tensor_constructor((F, G))
sage: phi_fg(f,g)
2*B[1] # B[3] + B[1] # B[4] + 4*B[2] # B[3] + 2*B[2] # B[4]

sage: FGH = tensor([F, G, H])
sage: phi_fgh = FGH.tensor_constructor((F, G, H))
sage: phi_fgh(f, g, h)
```

```
2*B[1] # B[3] # B[5] + 2*B[1] # B[3] # B[6] + B[1] # B[4] # B[5] + B[1] # B[4] # B[6] + 4*B[
```

```
sage: phi_fg_h = FGH.tensor_constructor((FG, H))
sage: phi_fg_h(phi_fg(f, g), h)
2*B[1] # B[3] # B[5] + 2*B[1] # B[3] # B[6] + B[1] # B[4] # B[5] + B[1] # B[4] # B[6] + 4*B[
```

5.1.99 Free Pre-Lie Algebras

AUTHORS:

- Florent Hivert, Frédéric Chapoton (2011)

class sage.combinat.free_prelie_algebra.**FreePreLieAlgebra**(*R*, *names=None*)
Bases: sage.combinat.free_module.CombinatorialFreeModule

The free pre-Lie algebra.

Pre-Lie algebras are non-associative algebras, where the product $*$ satisfies

$$(x * y) * z - x * (y * z) = (x * z) * y - x * (z * y).$$

We use here the convention where the associator

$$(x, y, z) := (x * y) * z - x * (y * z)$$

is symmetric in its two rightmost arguments. This is sometimes called a right pre-Lie algebra.

They have appeared in numerical analysis and deformation theory.

The free Pre-Lie algebra on a given set E has an explicit description using rooted trees, just as the free associative algebra can be described using words. The underlying vector space has a basis indexed by finite rooted trees endowed with a map from their vertices to E. In this basis, the product of two (decorated) rooted trees $S * T$ is the sum over vertices of S of the rooted tree obtained by adding one edge from the root of T to the given vertex of S. The root of these trees is taken to be the root of S. The free pre-Lie algebra can also be considered as the free algebra over the PreLie operad.

Warning: The usual binary operator $*$ can be used for the pre-Lie product. Beware that it but must be parenthesized properly, as the pre-Lie product is not associative. By default, a multiple product will be taken with left parentheses.

EXAMPLES:

```
sage: F = algebras.FreePreLie(ZZ, 'xyz')
sage: x,y,z = F.gens()
sage: (x * y) * z
B[x[y[], z[]]] + B[x[y[z[]]]]
sage: (x * y) * z - x * (y * z) == (x * z) * y - x * (z * y)
True
```

The free pre-Lie algebra is non-associative:

```
sage: x * (y * z) == (x * y) * z
False
```

The default product is with left parentheses:

```
sage: x * y * z == (x * y) * z
True
sage: x * y * z * x == ((x * y) * z) * x
True
```

The NAP product as defined in [Liv] is also implemented on the same vector space:

```
sage: N = F.nap_product
sage: N(x*y,z*z)
B[x[y[], z[z[]]]]
```

When there is only one generator, unlabelled trees are used instead:

```
sage: F1 = algebras.FreePreLie(QQ, 'w')
sage: w = F1.gen(0); w
B[[]]
sage: w * w * w * w
B[[[], [], []] + 3*B[[[], [[]]]] + B[[[[], []]]] + B[[[[[[]]]]]]
```

REFERENCES:

algebra_generators()

Return the generators of this algebra.

These are the rooted trees with just one vertex.

EXAMPLES:

```
sage: A = algebras.FreePreLie(ZZ, 'fgh'); A
Free PreLie algebra on 3 generators ['f', 'g', 'h']
 over Integer Ring
sage: list(A.algebra_generators())
[B[f[]], B[g[]], B[h[]]]

sage: A = algebras.FreePreLie(QQ, ['x1','x2'])
sage: list(A.algebra_generators())
[B[x1[]], B[x2[]]]
```

an_element()

Return an element of `self`.

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, 'xy')
sage: A.an_element()
B[x[x[], x[x[]]]] + B[x[x[x[x[]]]]]
```

degree_on_basis(*t*)

Return the degree of a rooted tree in the free Pre-Lie algebra.

This is the number of vertices.

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ,'@')
sage: RT = A.basis().keys()
sage: A.degree_on_basis(RT([RT([])]))
2
```

gen(*i*)

Return the `i`-th generator of the algebra.

INPUT:

- `i` – an integer

EXAMPLES:

```
sage: F = algebras.FreePreLie(ZZ, 'xyz')
sage: F.gen(0)
B[x[]]

sage: F.gen(4)
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2
```

gens()

Return the generators of `self` (as an algebra).

EXAMPLES:

```
sage: A = algebras.FreePreLie(ZZ, 'fgh')
sage: A.gens()
(B[f[]], B[g[]], B[h[]])
```

nap_product()

Return the NAP product.

See also:

`nap_product_on_basis()`

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, '@')
sage: RT = A.basis().keys()
sage: x = A(RT([RT([])]))
sage: A.nap_product(x, x)
B[[[], [[]]]]
```

nap_product_on_basis(x, y)

Return the NAP product of two trees.

This is the grafting of the root of y over the root of x. The root of the resulting tree is the root of x.

See also:

`nap_product()`

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, '@')
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
sage: A.nap_product_on_basis(x, x)
B[[[], [[]]]]
```

pre_Lie_product()

Return the pre-Lie product.

See also:

`pre_Lie_product_on_basis()`

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, '@')
sage: RT = A.basis().keys()
sage: x = A(RT([RT([])]))
```

```
sage: A.pre_Lie_product(x, x)
B[[[], [[]]]] + B[[[[[]]]]]
```

pre_Lie_product_on_basis(*x*, *y*)
Return the pre-Lie product of two trees.

This is the sum over all graftings of the root of y over a vertex of x. The root of the resulting trees is the root of x.

See also:

pre_Lie_product()

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, '@')
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
sage: A.product_on_basis(x, x)
B[[[], [[]]]] + B[[[[[]]]]]
```

product_on_basis(*x*, *y*)
Return the pre-Lie product of two trees.

This is the sum over all graftings of the root of y over a vertex of x. The root of the resulting trees is the root of x.

See also:

pre_Lie_product()

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, '@')
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
sage: A.product_on_basis(x, x)
B[[[], [[]]]] + B[[[[[]]]]]
```

some_elements()
Return some elements of the free pre-Lie algebra.

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ,'@')
sage: A.some_elements()
[B[[]], B[[[]]], B[[[], [[]]]] + B[[[[]]]],
 B[[[], []]] + B[[[[]]]], B[[[]]]]
```

With several generators:

```
sage: A = algebras.FreePreLie(QQ, 'xy')
sage: A.some_elements()
[B[x[]],
 B[x[x[]]],
 B[x[x[], x[x[]]]] + B[x[x[x[x[]]]]],
 B[x[x[], x[]]] + B[x[x[x[]]]],
 B[x[x[], y[]]] + B[x[x[y[]]]]]
```

variable_names()
Return the names of the variables.

EXAMPLES:

```
sage: R = algebras.FreePreLie(QQ, 'xy')
sage: R.variable_names()
{'x', 'y'}
```

5.1.100 Fully packed loops

class sage.combinat.fully_packed_loop.**FullyPackedLoop**(*parent*, *generator*)

Bases: sage.structure.element.Element

A class for fully packed loops.

A fully packed loop is a collection of non-intersecting lattice paths on a square grid such that every vertex is part of some path, and the paths are either closed internal loops or have endpoints corresponding to alternate points on the boundary [Propp2001]. They are known to be in bijection with alternating sign matrices.

See also:

AlternatingSignMatrix

To each fully packed loop, we assign a link pattern, which is the non-crossing matching attained by seeing which points on the boundary are connected by open paths in the fully packed loop.

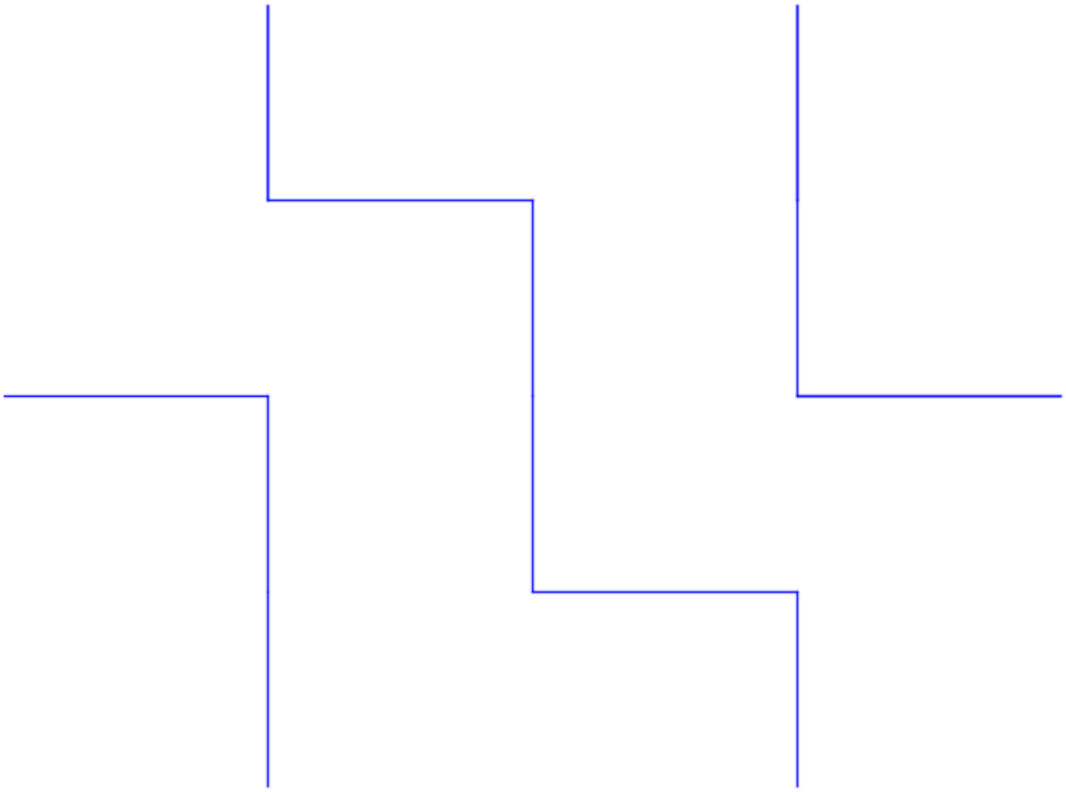
We can create a fully packed loop using the corresponding alternating sign matrix and also extract the link pattern:

```
sage: A = AlternatingSignMatrix([[0, 0, 1], [0, 1, 0], [1, 0, 0]])
sage: fpl = FullyPackedLoop(A)
sage: fpl.link_pattern()
[(1, 4), (2, 3), (5, 6)]
sage: fpl
        |         |
        |         |
        + -- +    +
             |    |
             |    |
     -- +    +    + --
        |    |
        |    |
        +    + -- +
        |         |
        |         |
sage: B = AlternatingSignMatrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
sage: fplb = FullyPackedLoop(B)
sage: fplb.link_pattern()
[(1, 6), (2, 5), (3, 4)]
sage: fplb
        |         |
        |         |
        +    + -- +
        |    |
        |    |
     -- +    +    + --
             |    |
             |    |
        + -- +    +
        |         |
        |         |
```

The class also has a plot method:

```
sage: fpl.plot()
Graphics object consisting of 15 graphics primitives
```

which gives:



Note that we can also create a fully packed loop from a six vertex model configuration:

```
sage: S = SixVertexModel(3, boundary_conditions='ice').from_alternating_sign_matrix(A)
sage: S
    ^    ^    ^
    |    |    |
--> # -> # -> # <--
    ^    ^    |
    |    |    V
--> # -> # <- # <--
    ^    |    |
    |    V    V
--> # <- # <- # <--
    |    |    |
    V    V    V
sage: fpl = FullyPackedLoop(S)
sage: fpl
    |         |
    |         |
    + -- +    +
         |    |
         |    |
```

```
 -- +    +    + --
    |    |
    |    |
    +    + -- +
    |         |
    |         |
```

Once we have a fully packed loop we can obtain the corresponding alternating sign matrix:

```
sage: fpl.to_alternating_sign_matrix()
[0 0 1]
[0 1 0]
[1 0 0]
```

Here are some more examples using bigger ASMs:

```
sage: A = AlternatingSignMatrix([[0,1,0,0],[0,0,1,0],[1,-1,0,1],[0,1,0,0]])
sage: S = SixVertexModel(4, boundary_conditions='ice').from_alternating_sign_matrix(A)
sage: fpl = FullyPackedLoop(S)
sage: fpl.link_pattern()
[(1, 2), (3, 6), (4, 5), (7, 8)]
sage: fpl
    |         |
    |         |
    + -- + -- +    + --
                   |
                   |
 -- +    + -- + -- +
    |    |
    |    |
    +    +    + -- + --
    |    |    |
    |    |    |
 -- +    +    + -- +
         |         |
         |         |

sage: m = AlternatingSignMatrix([[0,0,1,0,0,0],
....:                            [1,0,-1,0,1,0],
....:                            [0,0,0,1,0,0],
....:                            [0,1,0,0,-1,1],
....:                            [0,0,0,0,1,0],
....:                            [0,0,1,0,0,0]])
sage: fpl = FullyPackedLoop(m)
sage: fpl.link_pattern()
[(1, 12), (2, 7), (3, 4), (5, 6), (8, 9), (10, 11)]
sage: fpl
    |         |         |
    |         |         |
    + -- +    +    + -- +    + --
         |    |    |         |
         |    |    |         |
 -- + -- +    +    + -- + -- +
              |
              |
    + -- +    + -- + -- +    + --
    |    |              |    |
    |    |              |    |
 -- +    +    + -- +    +    +
```

```
          |    |    |    |    |
          |    |    |    |    |
    + -- +    + -- +    +    + --
    |                   |
    |                   |
 -- +    + -- + -- +    + -- +
          |         |         |
          |         |         |
```

```
sage: m = AlternatingSignMatrix([[0,1,0,0,0,0,0],
....:                            [1,-1,0,0,1,0,0],
....:                            [0,0,0,1,0,0,0],
....:                            [0,1,0,0,-1,1,0],
....:                            [0,0,0,0,1,0,0],
....:                            [0,0,1,0,-1,0,1],
....:                            [0,0,0,0,1,0,0]])
sage: fpl = FullyPackedLoop(m)
sage: fpl.link_pattern()
[(1, 2), (3, 4), (5, 6), (7, 8), (9, 14), (10, 11), (12, 13)]
sage: fpl
    |         |         |         |
    |         |         |         |
    + -- + -- +    + -- +    + -- +
                   |         |
                   |         |
 -- + -- + -- +    + -- + -- +    + --
               |                   |
               |                   |
    + -- +    + -- + -- +    + -- +
    |    |               |    |
    |    |               |    |
 -- +    +    + -- +    +    +    + --
          |    |    |    |    |    |
          |    |    |    |    |    |
    + -- +    + -- +    +    + -- +
    |                   |
    |                   |
 -- +    + -- + -- +    +    + -- + --
          |         |    |    |
          |         |    |    |
    + -- +    + -- +    +    + -- +
    |         |         |         |
    |         |         |         |
```

Gyration on an alternating sign matrix/fully packed loop `fpl` of the link pattern corresponding to `fpl`:

```
sage: ASMs = AlternatingSignMatrices(3).list()
sage: ncp = FullyPackedLoop(ASMs[1]).link_pattern() # fpl's gyration orbit size is 2
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(0,5):
....:         a,b=a%6+1,b%6+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(ASMs[1].gyration().to_fully_packed_loop().link_pattern()) ==\
PerfectMatching(rotated_ncp)
True

sage: fpl = FullyPackedLoop(ASMs[0])
```

```
sage: ncp = fpl.link_pattern() # fpl's gyration size is 3
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(0,5):
....:         a,b=a%6+1,b%6+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(ASMs[0].gyration().to_fully_packed_loop().link_pattern()) ==\
PerfectMatching(rotated_ncp)
True

sage: mat = AlternatingSignMatrix([[0,0,1,0,0,0,0],[1,0,-1,0,1,0,0],\
[0,0,1,0,0,0,0],[0,1,-1,0,0,1,0],[0,0,1,0,0,0,0],[0,0,0,1,0,0,0],[0,0,0,0,0,0,1]])
sage: fpl = FullyPackedLoop(mat) # n=7
sage: ncp = fpl.link_pattern()
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(0,13):
....:         a,b=a%14+1,b%14+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(mat.gyration().to_fully_packed_loop().link_pattern()) ==\
PerfectMatching(rotated_ncp)
True

sage: mat = AlternatingSignMatrix([[0,0,0,1,0,0], [0,0,1,-1,1,0],\
[0,1,0,0,-1,1], [1,0,-1,1,0,0], [0,0,1,0,0,0], [0,0,0,0,1,0]])
sage: fpl = FullyPackedLoop(mat) # n =6
sage: ncp = fpl.link_pattern()
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(0,11):
....:         a,b=a%12+1,b%12+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(mat.gyration().to_fully_packed_loop().link_pattern()) ==\
PerfectMatching(rotated_ncp)
True
```

More examples:

We can initiate a fully packed loop using an alternating sign matrix::

```
sage: A = AlternatingSignMatrix([[0, 0, 1], [0, 1, 0], [1, 0, 0]])
sage: fpl = FullyPackedLoop(A)
sage: fpl
    |         |
    |         |
    + -- +    +
         |    |
         |    |
 -- +    +    + --
    |    |
    |    |
    +    + -- +
    |         |
    |         |
sage: FullyPackedLoops(3)(A) == fpl
True
```

We can also input a matrix::

```
    sage: FullyPackedLoop([[0, 0, 1], [0, 1, 0], [1, 0, 0]])
        |         |
        |         |
        + -- +    +
             |    |
             |    |
     -- +    +    + --
        |    |
        |    |
        +    + -- +
        |         |
        |         |
    sage: FullyPackedLoop([[0, 0, 1], [0, 1, 0], [1, 0, 0]]) ==\
    ....: FullyPackedLoops(3)([[0, 0, 1], [0, 1, 0], [1, 0, 0]])
    True
```

Otherwise we initiate a fully packed loop using a six vertex model::

```
    sage: S = SixVertexModel(3, boundary_conditions='ice').from_alternating_sign_matrix(A)
    sage: fpl = FullyPackedLoop(S)
    sage: fpl
        |         |
        |         |
        + -- +    +
             |    |
             |    |
     -- +    +    + --
        |    |
        |    |
        +    + -- +
        |         |
        |         |

    sage: FullyPackedLoops(3)(S) == FullyPackedLoop(S)
    True

    sage: fpl.six_vertex_model().to_alternating_sign_matrix()
    [0 0 1]
    [0 1 0]
    [1 0 0]
```

We can also input the matrix associated to a six vertex model::

```
    sage: SixVertexModel(2)([[3,1],[5,3]])
        ^    ^
        |    |
    --> # <- # <--
        |    ^
        V    |
    --> # -> # <--
        |    |
        V    V

    sage: FullyPackedLoop([[3,1],[5,3]])
        |
        |
        +    + --
        |    |
```

```
    |    |
 -- +    +
         |
         |
```

```
sage: FullyPackedLoops(2)([[3,1],[5,3]]) == FullyPackedLoop([[3,1],[5,3]])
True
```

Note that the matrix corresponding to a six vertex model without
the ice boundary condition is not allowed::

```
sage: SixVertexModel(2)([[3,1],[5,5]])
    ^    ^
    |    |
--> # <- # <--
    |    ^
    V    V
--> # -> # -->
    |    |
    V    V
```

```
sage: FullyPackedLoop([[3,1],[5,5]])
Traceback (most recent call last):
...
ValueError: Invalid alternating sign matrix
```

```
sage: FullyPackedLoops(2)([[3,1],[5,5]])
Traceback (most recent call last):
...
ValueError: Invalid alternating sign matrix
```

Note that if anything else is used to generate the fully packed loop an error will occur::

```
sage: fpl = FullyPackedLoop(5)
Traceback (most recent call last):
...
ValueError: Invalid alternating sign matrix
```

```
sage: fpl = FullyPackedLoop((1, 2, 3))
Traceback (most recent call last):
...
ValueError: The alternating sign matrices must be square
```

```
sage: SVM = SixVertexModel(3)[0]
sage: FullyPackedLoop(SVM)
Traceback (most recent call last):
...
ValueError: Invalid alternating sign matrix
```

REFERENCES:

gyration()

Return the fully packed loop obtained by applying gyration to the alternating sign matrix in bijection with `self`.

Gyration was first defined in [Wieland00] as an action on fully-packed loops.

REFERENCES:

EXAMPLES:

```
sage: A = AlternatingSignMatrix([[1, 0, 0],[0, 1, 0],[0, 0, 1]])
sage: fpl = FullyPackedLoop(A)
sage: fpl.gyration().to_alternating_sign_matrix()
[0 0 1]
[0 1 0]
[1 0 0]
sage: asm = AlternatingSignMatrix([[0, 0, 1],[1, 0, 0],[0, 1, 0]])
sage: f = FullyPackedLoop(asm)
sage: f.gyration().to_alternating_sign_matrix()
[0 1 0]
[0 0 1]
[1 0 0]
```

link_pattern()

Return a link pattern corresponding to a fully packed loop.

Here we define a link pattern LP to be a partition of the list $[1, ..., 2k]$ into 2-element sets (such a partition is also known as a perfect matching) such that the following non-crossing condition holds: Let the numbers $1, ..., 2k$ be written on the perimeter of a circle. For every 2-element set (a, b) of the partition LP, draw an arc linking the two numbers a and b. We say that LP is non-crossing if every arc can be drawn so that no two arcs intersect.

Since every endpoint of a fully packed loop fpl is connected to a different endpoint, there is a natural surjection from the fully packed loops on an nxn grid onto the link patterns on the list $[1, \ldots, 2n]$. The pairs of connected endpoints of a fully packed loop fpl correspond to the 2-element tuples of the corresponding link pattern.

See also:

`PerfectMatching`

Note: by convention, we choose the top left vertex to be even. See [Propp2001] and [Striker2015].

EXAMPLES:

We can extract the underlying link pattern (a non-crossing partition) from a fully packed loop:

```
sage: A = AlternatingSignMatrix([[0, 1, 0], [1, -1, 1], [0, 1, 0]])
sage: fpl = FullyPackedLoop(A)
sage: fpl.link_pattern()
[(1, 2), (3, 6), (4, 5)]

sage: B = AlternatingSignMatrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
sage: fpl = FullyPackedLoop(B)
sage: fpl.link_pattern()
[(1, 6), (2, 5), (3, 4)]
```

Gyration on an alternating sign matrix/fully packed loop `fpl` corresponds to a rotation (i.e. a becomes a-1 mod 2n) of the link pattern corresponding to `fpl`:

```
sage: ASMs = AlternatingSignMatrices(3).list()
sage: ncp = FullyPackedLoop(ASMs[1]).link_pattern()
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(0,5):
....:         a,b=a%6+1,b%6+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(ASMs[1].gyration().to_fully_packed_loop().link_pattern()) ==
True
```

```
sage: fpl = FullyPackedLoop(ASMs[0])
sage: ncp = fpl.link_pattern()
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(0,5):
....:         a,b=a%6+1,b%6+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(ASMs[0].gyration().to_fully_packed_loop().link_pattern()) ==
True

sage: mat = AlternatingSignMatrix([[0,0,1,0,0,0,0],[1,0,-1,0,1,0,0],[0,0,1,0,0,0,0],
sage: fpl = FullyPackedLoop(mat) # n=7
sage: ncp = fpl.link_pattern()
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(0,13):
....:         a,b=a%14+1,b%14+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(mat.gyration().to_fully_packed_loop().link_pattern()) ==            Pe
True

sage: mat = AlternatingSignMatrix([[0,0,0,1,0,0], [0,0,1,-1,1,0], [0,1,0,0,-1,1], [1,0,-1,1,
sage: fpl = FullyPackedLoop(mat)
sage: ncp = fpl.link_pattern()
sage: rotated_ncp=[]
sage: for (a,b) in ncp:
....:     for i in range(0,11):
....:         a,b=a%12+1,b%12+1;
....:     rotated_ncp.append((a,b))
sage: PerfectMatching(mat.gyration().to_fully_packed_loop().link_pattern()) ==            Pe
True
```

TESTS:

We test previous two bugs which showed up when this method is called twice:

```
sage: A = AlternatingSignMatrices(6)
sage: B = A.random_element()
sage: C = FullyPackedLoop(B)
sage: D = C.link_pattern()
sage: E = C.link_pattern()
sage: D == E
True
```

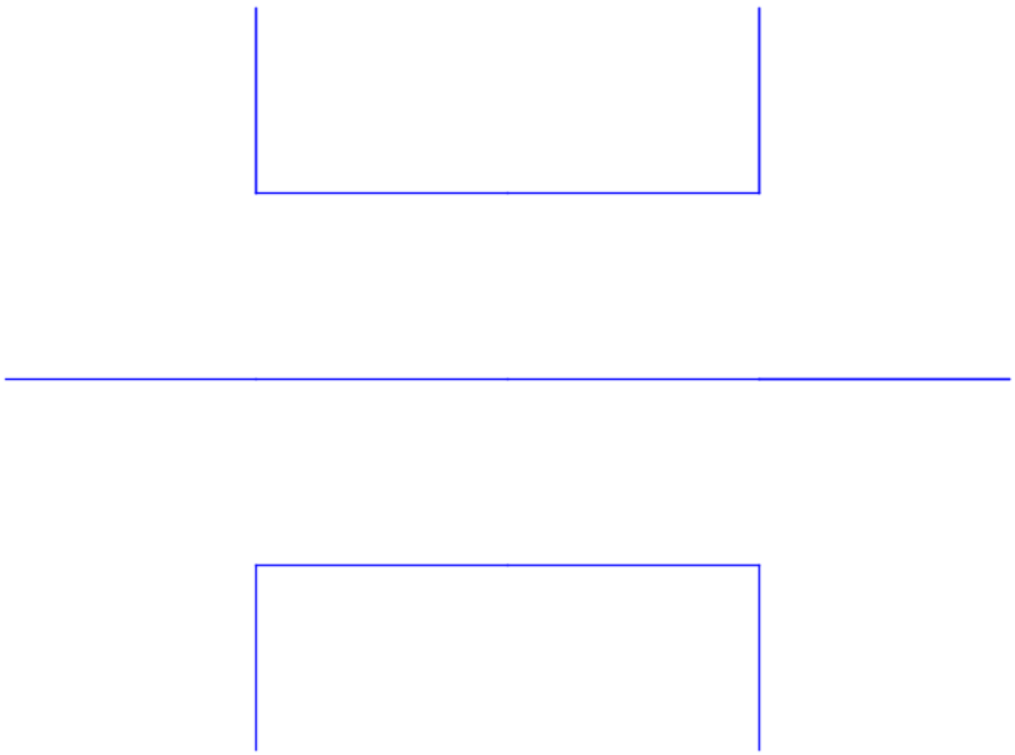
plot()

Return a graphical object of the Fully Packed Loop

EXAMPLES:

Here is the fully packed loop for

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & -1 & 1 \\ 0 & 1 & 0 \end{pmatrix}:$$



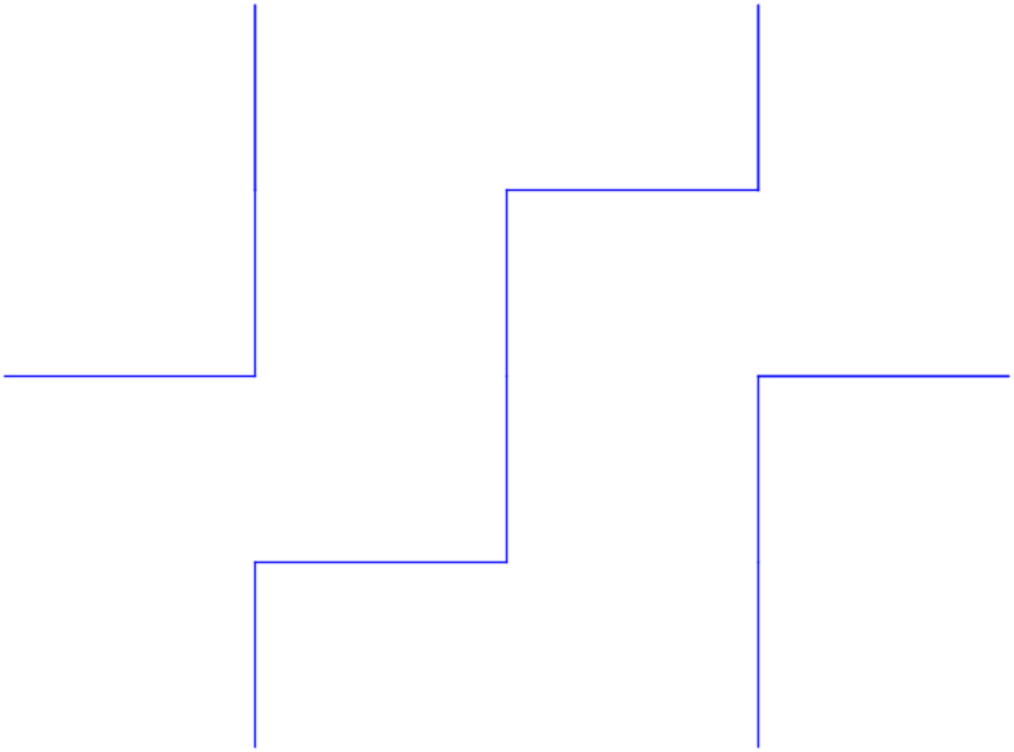
Here is how Sage represents this:

```
sage: A = AlternatingSignMatrix([[0, 1, 0], [1, -1, 1], [0, 1, 0]])
sage: fpl = FullyPackedLoop(A)
sage: print fpl.plot().description()
Line defined by 2 points:       [(-1.0, 1.0), (0.0, 1.0)]
Line defined by 2 points:       [(0.0, 0.0), (0.0, -1.0)]
Line defined by 2 points:       [(0.0, 0.0), (1.0, 0.0)]
Line defined by 2 points:       [(0.0, 2.0), (0.0, 3.0)]
Line defined by 2 points:       [(0.0, 2.0), (0.0, 3.0)]
Line defined by 2 points:       [(0.0, 2.0), (1.0, 2.0)]
Line defined by 2 points:       [(1.0, 1.0), (0.0, 1.0)]
Line defined by 2 points:       [(1.0, 1.0), (2.0, 1.0)]
Line defined by 2 points:       [(2.0, 0.0), (1.0, 0.0)]
Line defined by 2 points:       [(2.0, 0.0), (2.0, -1.0)]
Line defined by 2 points:       [(2.0, 2.0), (1.0, 2.0)]
Line defined by 2 points:       [(2.0, 2.0), (2.0, 3.0)]
Line defined by 2 points:       [(2.0, 2.0), (2.0, 3.0)]
Line defined by 2 points:       [(3.0, 1.0), (2.0, 1.0)]
Line defined by 2 points:       [(3.0, 1.0), (2.0, 1.0)]
```

Here are the other 3 by 3 Alternating Sign Matrices and their corresponding fully packed loops:

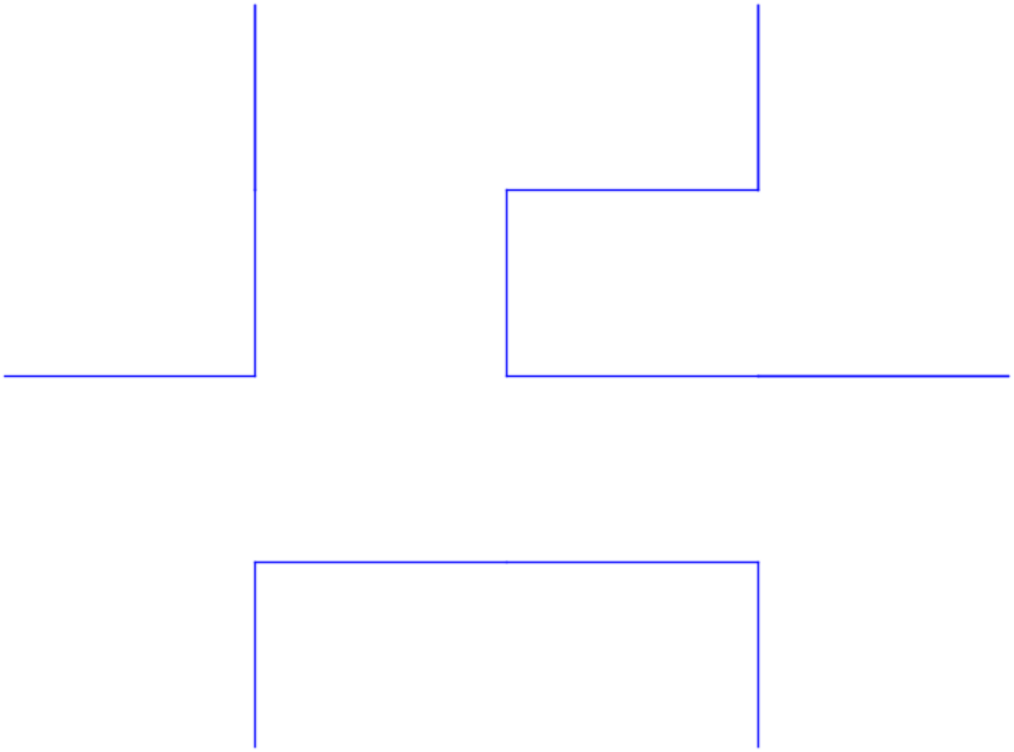
$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

gives:



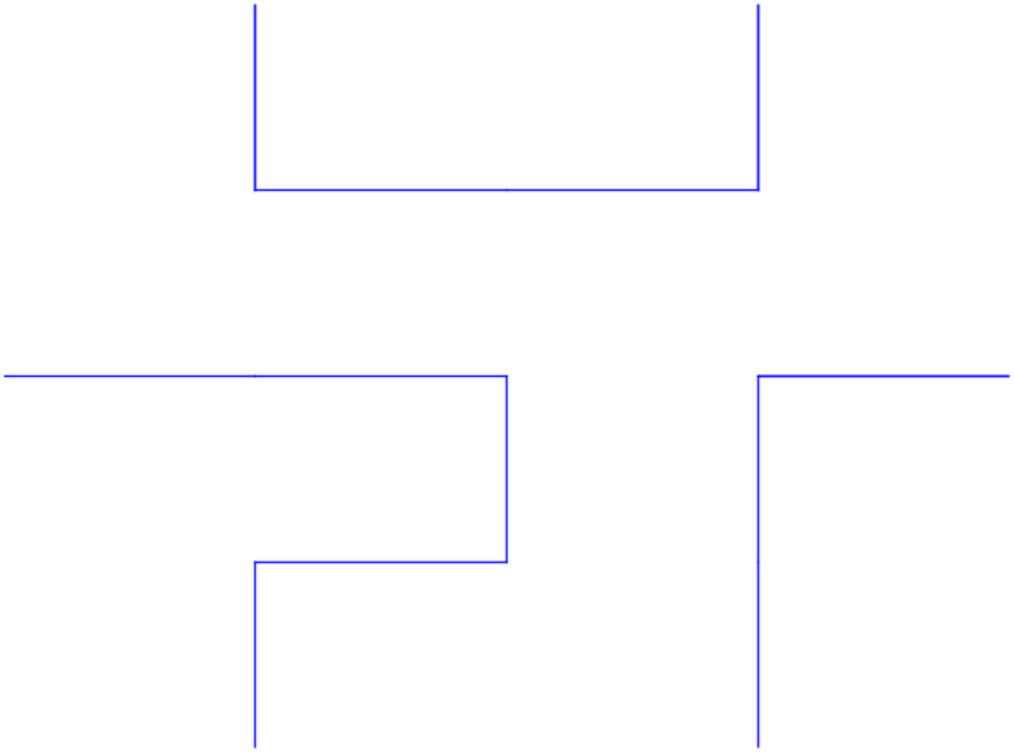
$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

gives:



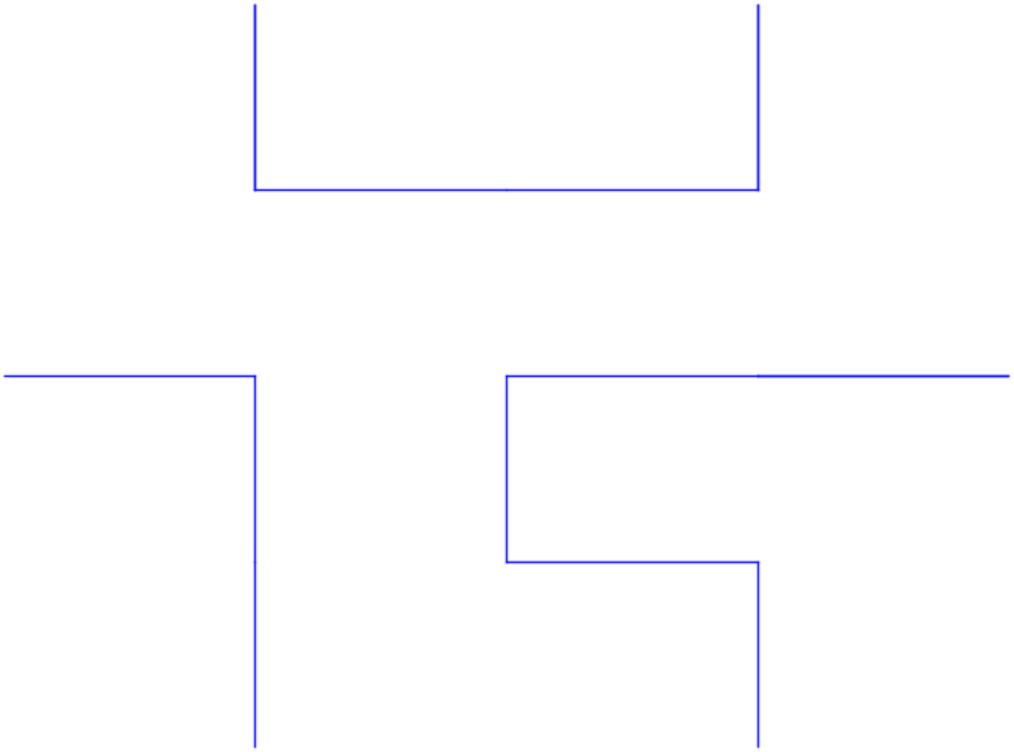
$$A = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

gives:



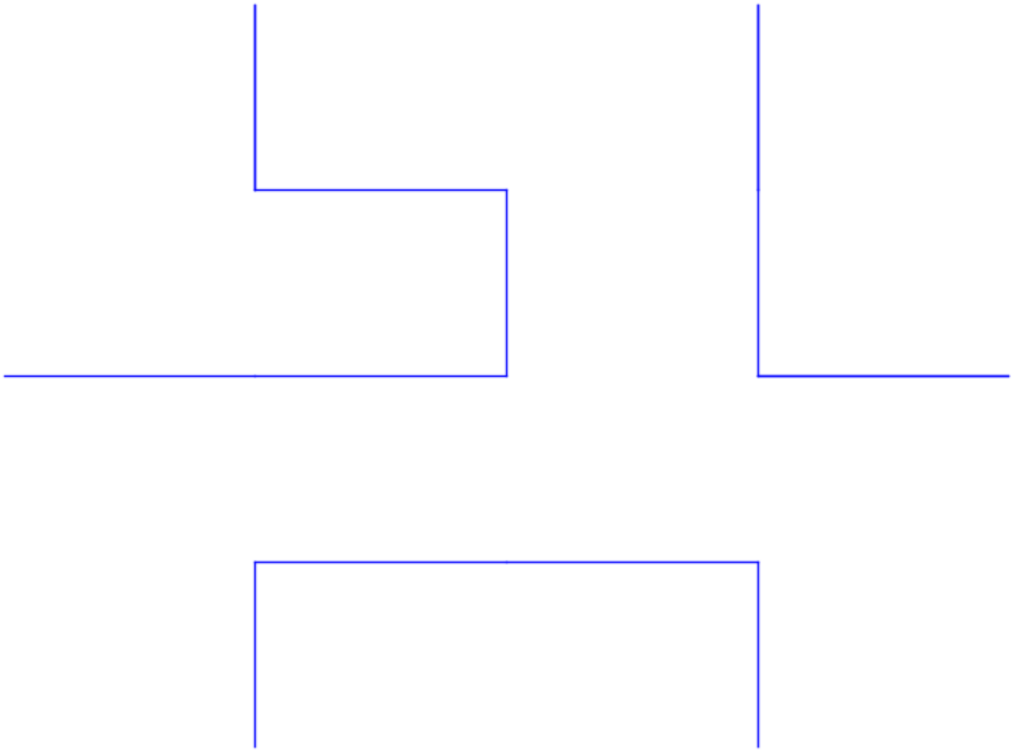
$$A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

gives:



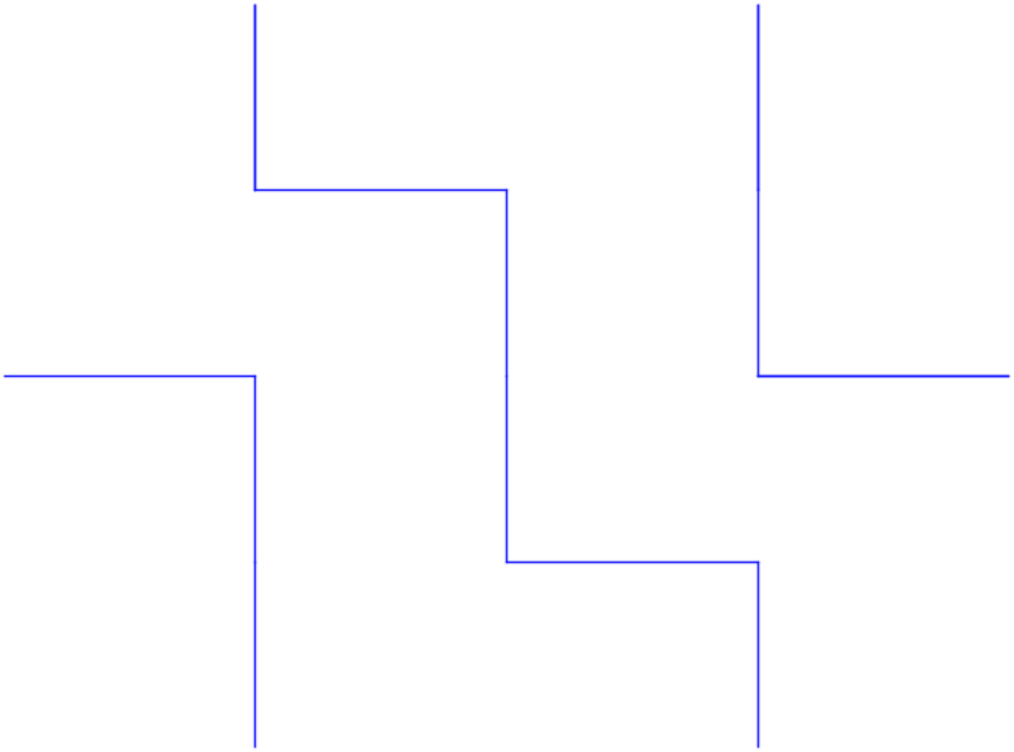
$$A = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

gives:



$$A = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

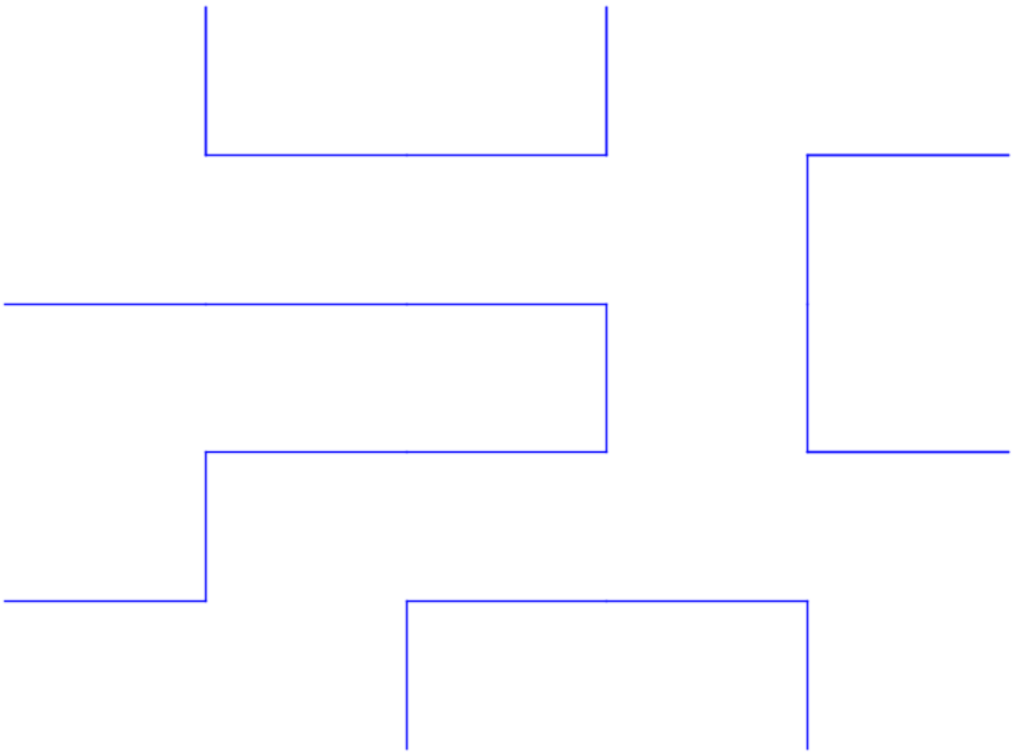
gives:



EXAMPLES:

```
sage: A = AlternatingSignMatrix([[0, 1, 0, 0], [1, -1, 0, 1], \
[0, 1, 0, 0],[0, 0, 1, 0]])
sage: fpl = FullyPackedLoop(A)
sage: print fpl.plot().description()
Line defined by 2 points:       [(-1.0, 0.0), (0.0, 0.0)]
Line defined by 2 points:       [(-1.0, 2.0), (0.0, 2.0)]
Line defined by 2 points:       [(0.0, 1.0), (0.0, 0.0)]
Line defined by 2 points:       [(0.0, 1.0), (1.0, 1.0)]
Line defined by 2 points:       [(0.0, 3.0), (0.0, 4.0)]
Line defined by 2 points:       [(0.0, 3.0), (0.0, 4.0)]
Line defined by 2 points:       [(0.0, 3.0), (1.0, 3.0)]
Line defined by 2 points:       [(1.0, 0.0), (1.0, -1.0)]
Line defined by 2 points:       [(1.0, 0.0), (2.0, 0.0)]
Line defined by 2 points:       [(1.0, 2.0), (0.0, 2.0)]
Line defined by 2 points:       [(1.0, 2.0), (2.0, 2.0)]
Line defined by 2 points:       [(2.0, 1.0), (1.0, 1.0)]
Line defined by 2 points:       [(2.0, 1.0), (2.0, 2.0)]
Line defined by 2 points:       [(2.0, 3.0), (1.0, 3.0)]
Line defined by 2 points:       [(2.0, 3.0), (2.0, 4.0)]
Line defined by 2 points:       [(2.0, 3.0), (2.0, 4.0)]
Line defined by 2 points:       [(3.0, 0.0), (2.0, 0.0)]
Line defined by 2 points:       [(3.0, 0.0), (3.0, -1.0)]
Line defined by 2 points:       [(3.0, 2.0), (3.0, 1.0)]
Line defined by 2 points:       [(3.0, 2.0), (3.0, 3.0)]
Line defined by 2 points:       [(4.0, 1.0), (3.0, 1.0)]
Line defined by 2 points:       [(4.0, 1.0), (3.0, 1.0)]
Line defined by 2 points:       [(4.0, 3.0), (3.0, 3.0)]
Line defined by 2 points:       [(4.0, 3.0), (3.0, 3.0)]
```

Here is the plot:



six_vertex_model()

Return the underlying six vertex model configuration.

EXAMPLES:

```
sage: B = AlternatingSignMatrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
sage: fpl = FullyPackedLoop(B)
sage: fpl
    |           |
    |           |
    +     + -- +
    |     |
    |     |
 -- +     +     + --
          |     |
          |     |
    + -- +     +
    |           |
    |           |
sage: fpl.six_vertex_model()
    ^     ^     ^
    |     |     |
--> # <- # <- # <--
    |     ^     ^
    V     |     |
--> # -> # <- # <--
    |     |     ^
```

```
    V    V    |
--> # -> # -> # <--
    |    |    |
    V    V    V
```

to_alternating_sign_matrix()

Returns the alternating sign matrix corresponding to this class.

See also:

AlternatingSignMatrix

EXAMPLES:

```
sage: A = AlternatingSignMatrix([[0, 1, 0], [1, -1, 1], [0, 1, 0]])
sage: S = SixVertexModel(3, boundary_conditions='ice').from_alternating_sign_matrix(A)
sage: fpl = FullyPackedLoop(S)
sage: fpl.to_alternating_sign_matrix()
[ 0  1  0]
[ 1 -1  1]
[ 0  1  0]
sage: A = AlternatingSignMatrix([[0,1,0,0],[0,0,1,0],[1,-1,0,1],[0,1,0,0]])
sage: S = SixVertexModel(4, boundary_conditions='ice').from_alternating_sign_matrix(A)
sage: fpl = FullyPackedLoop(S)
sage: fpl.to_alternating_sign_matrix()
[ 0  1  0  0]
[ 0  0  1  0]
[ 1 -1  0  1]
[ 0  1  0  0]
```

class sage.combinat.fully_packed_loop.**FullyPackedLoops**(n)

Bases: sage.structure.parent.Parent, sage.structure.unique_representation.UniqueRepresenta

Class of all fully packed loops on an $n \times n$ grid.

They are known to be in bijection with alternating sign matrices.

See also:

AlternatingSignMatrices

INPUT:

- n – the number of row (and column) or grid

EXAMPLES:

This will create an instance to manipulate the fully packed loops of size 3:

```
sage: FPLs = FullyPackedLoops(3)
sage: FPLs
Fully packed loops on a 3x3 grid
sage: FPLs.cardinality()
7
```

When using the square ice model, it is known that the number of configurations is equal to the number of alternating sign matrices:

```
sage: M = FullyPackedLoops(1)
sage: len(M)
1
sage: M = FullyPackedLoops(4)
sage: len(M)
```

```
42
sage: all(len(SixVertexModel(n, boundary_conditions='ice'))
....:     == FullyPackedLoops(n).cardinality() for n in range(1, 7))
True
```

Element
alias of FullyPackedLoop

cardinality()
Return the cardinality of self.

The number of fully packed loops on $n \times n$ grid

$$\prod_{k=0}^{n-1} \frac{(3k+1)!}{(n+k)!} = \frac{1!4!7!10!\cdots(3n-2)!}{n!(n+1)!(n+2)!(n+3)!\cdots(2n-1)!}.$$

EXAMPLES:

```
sage: [AlternatingSignMatrices(n).cardinality() for n in range(0, 11)]
[1, 1, 2, 7, 42, 429, 7436, 218348, 10850216, 911835460, 129534272700]
```

size()
Return the size of the matrices in self.

TESTS:

```
sage: FPLs = FullyPackedLoops(4)
sage: FPLs.size()
4
```

5.1.101 Gelfand-Tsetlin Patterns

AUTHORS:

- Travis Scrimshaw (2013-15-03): Initial version

REFERENCES:

class sage.combinat.gelfand_tsetlin_patterns.**GelfandTsetlinPattern**
Bases: sage.structure.list_clone.ClonableArray

A Gelfand-Tsetlin (sometimes written as Gelfand-Zetlin or Gelfand-Cetlin) pattern. They were originally defined in [GC50].

A Gelfand-Tsetlin pattern is a triangular array:

$$\begin{array}{ccccccccccc}
a_{1,1} & & a_{1,2} & & a_{1,3} & & \cdots & & a_{1,n} \\
& a_{2,2} & & a_{2,3} & & \cdots & & a_{2,n} & \\
& & a_{3,3} & & \cdots & & a_{3,n} & & \\
& & & \ddots & & & & & \\
& & & & a_{n,n} & & & &
\end{array}$$

such that $a_{i,j} \geq a_{i+1,j+1} \geq a_{i,j+1}$.

Gelfand-Tsetlin patterns are in bijection with semistandard Young tableaux by the following algorithm. Let G be a Gelfand-Tsetlin pattern with $\lambda^{(k)}$ being the $(n-k+1)$-st row (note that this is a partition). The definition of G implies

$$\lambda^{(0)} \subseteq \lambda^{(1)} \subseteq \cdots \subseteq \lambda^{(n)},$$

where $\lambda^{(0)}$ is the empty partition, and each skew shape $\lambda^{(k)}/\lambda^{(k-1)}$ is a horizontal strip. Thus define $T(G)$ by inserting k into the squares of the skew shape $\lambda^{(k)}/\lambda^{(k-1)}$, for $k = 1, \ldots, n$.

To each entry in a Gelfand-Tsetlin pattern, one may attach a decoration of a circle or a box (or both or neither). These decorations appear in the study of Weyl group multiple Dirichlet series, and are implemented here following the exposition in [BBF].

Note: We use the "right-hand" rule for determining circled and boxed entries.

Warning: The entries in Sage are 0-based and are thought of as flushed to the left in a matrix. In other words, the coordinates of entries in the Gelfand-Tsetlin patterns are thought of as the matrix:

$$\begin{bmatrix} g_{0,0} & g_{0,1} & g_{0,2} & \cdots & g_{0,n-2} & g_{n-1,n-1} \\ g_{1,0} & g_{1,1} & g_{1,2} & \cdots & g_{1,n-2} & \\ g_{2,0} & g_{2,1} & g_{2,2} & \cdots & & \\ \vdots & \vdots & \vdots & & & \\ g_{n-2,0} & g_{n-2,1} & & & & \\ g_{n-1,0} & & & & & \end{bmatrix}.$$

However, in the discussions, we will be using the **standard** numbering system.

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[3, 2, 1], [2, 1], [1]]); G
[[3, 2, 1], [2, 1], [1]]
sage: G.pp()
  3  2  1
    2  1
       1
sage: G = GelfandTsetlinPattern([[7, 7, 4, 0], [7, 7, 3], [7, 5], [5]]); G.pp()
  7  7  4  0
    7  7  3
       7  5
          5
sage: G.to_tableau().pp()
  1  1  1  1  1  2  2
  2  2  2  2  2  3  3
  3  3  3  4
```

Tokuyama_coefficient(*name='t'*)

Return the Tokuyama coefficient attached to `self`.

Following the exposition of [BBF], Tokuyama's formula asserts

$$\sum_G (t+1)^{s(G)} t^{l(G)} z_1^{d_{n+1}} z_2^{d_n - d_{n+1}} \cdots z_{n+1}^{d_1 - d_2} = s_\lambda(z_1, \ldots, z_{n+1}) \prod_{i<j} (z_j + t z_i),$$

where the sum is over all strict Gelfand-Tsetlin patterns with fixed top row $\lambda + \rho$, with λ a partition with at most $n+1$ parts and $\rho = (n, n-1, \ldots, 1, 0)$, and s_λ is a Schur function.

INPUT:

- `name` – (Default: `'t'`) An alternative name for the variable t.

EXAMPLES:

```
sage: P = GelfandTsetlinPattern([[3,2,1],[2,2],[2]])
sage: P.Tokuyama_coefficient()
0
```

```
sage: G = GelfandTsetlinPattern([[3,2,1],[3,1],[2]])
sage: G.Tokuyama_coefficient()
t^2 + t
sage: G = GelfandTsetlinPattern([[2,1,0],[1,1],[1]])
sage: G.Tokuyama_coefficient()
0
sage: G = GelfandTsetlinPattern([[5,3,2,1,0],[4,3,2,0],[4,2,1],[3,2],[3]])
sage: G.Tokuyama_coefficient()
t^8 + 3*t^7 + 3*t^6 + t^5
```

boxed_entries()

Return the position of the boxed entries of `self`.

Using the *right-hand* rule, an entry $a_{i,j}$ is boxed if $a_{i,j} = a_{i-1,j-1}$; i.e., $a_{i,j}$ has the same value as its neighbor to the northwest.

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[3,2,1],[3,1],[1]])
sage: G.boxed_entries()
((1, 0),)
```

check()

Check that this is a valid Gelfand-Tsetlin pattern.

EXAMPLES:

```
sage: G = GelfandTsetlinPatterns()
sage: G([[3,2,1],[2,1],[1]]).check()
```

circled_entries()

Return the circled entries of `self`.

Using the *right-hand* rule, an entry $a_{i,j}$ is circled if $a_{i,j} = a_{i-1,j}$; i.e., $a_{i,j}$ has the same value as its neighbor to the northeast.

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[3,2,1],[3,1],[1]])
sage: G.circled_entries()
((1, 1), (2, 0))
```

is_strict()

Return `True` if `self` is a strict Gelfand-Tsetlin pattern.

A Gelfand-Tsetlin pattern is said to be *strict* if every row is strictly decreasing.

EXAMPLES:

```
sage: GelfandTsetlinPattern([[7,3,1],[6,2],[4]]).is_strict()
True
sage: GelfandTsetlinPattern([[3,2,1],[3,1],[1]]).is_strict()
True
sage: GelfandTsetlinPattern([[6,0,0],[3,0],[2]]).is_strict()
False
```

number_of_boxes()

Return the number of boxed entries. See `boxed_entries()`.

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[3,2,1],[3,1],[1]])
sage: G.number_of_boxes()
1
```

number_of_circles()
Return the number of boxed entries. See `circled_entries()`.

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[3,2,1],[3,1],[1]])
sage: G.number_of_circles()
2
```

number_of_special_entries()
Return the number of special entries. See `special_entries()`.

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[4,2,1],[4,1],[2]])
sage: G.number_of_special_entries()
1
```

pp()
Pretty print `self`.

EXAMPLES:

```
sage: G = GelfandTsetlinPatterns()
sage: G([[3,2,1],[2,1],[1]]).pp()
  3     2     1
     2     1
        1
```

row_sums()
Return the list of row sums.

For a Gelfand-Tsetlin pattern G, the i-th row sum d_i is

$$d_i = d_i(G) = \sum_{j=i}^{n} a_{i,j}.$$

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[5,3,2,1,0],[4,3,2,0],[4,2,1],[3,2],[3]])
sage: G.row_sums()
[11, 9, 7, 5, 3]
sage: G = GelfandTsetlinPattern([[3,2,1],[3,1],[2]])
sage: G.row_sums()
[6, 4, 2]
```

special_entries()
Return the special entries.

An entry $a_{i,j}$ is special if $a_{i-1,j-1} > a_{i,j} > a_{i-1,j}$, that is to say, the entry is neither boxed nor circled and is **not** in the first row. The name was coined by [Tok88].

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[3,2,1],[3,1],[1]])
sage: G.special_entries()
()
```

```
sage: G = GelfandTsetlinPattern([[4,2,1],[4,1],[2]])
sage: G.special_entries()
((2, 0),)
```

to_tableau()

Return `self` as a semistandard Young tableau.

The conversion from a Gelfand-Tsetlin pattern to a semistandard Young tableaux is as follows. Let G be a Gelfand-Tsetlin pattern with $\lambda^{(k)}$ being the $(n-k+1)$-st row (note that this is a partition). The definition of G implies

$$\lambda^{(0)} \subseteq \lambda^{(1)} \subseteq \cdots \subseteq \lambda^{(n)},$$

where $\lambda^{(0)}$ is the empty partition, and each skew shape $\lambda^{(k)}/\lambda^{(k-1)}$ is a horizontal strip. Thus define $T(G)$ by inserting k into the squares of the skew shape $\lambda^{(k)}/\lambda^{(k-1)}$, for $k = 1, \ldots, n$.

EXAMPLES:

```
sage: G = GelfandTsetlinPatterns()
sage: elt = G([[3,2,1],[2,1],[1]])
sage: T = elt.to_tableau(); T
[[1, 2, 3], [2, 3], [3]]
sage: T.pp()
  1  2  3
  2  3
  3
sage: G(T) == elt
True
```

weight()

Return the weight of `self`.

Define the weight of G to be the content of the tableau to which G corresponds under the bijection between Gelfand-Tsetlin patterns and semistandard tableaux. More precisely,

$$\mathrm{wt}(G) = (d_n, d_{n-1} - d_n, \ldots, d_1 - d_2),$$

where the d_i are the row sums.

EXAMPLES:

```
sage: G = GelfandTsetlinPattern([[2,1,0],[1,0],[1]])
sage: G.weight()
(1, 0, 2)
sage: G = GelfandTsetlinPattern([[4,2,1],[3,1],[2]])
sage: G.weight()
(2, 2, 3)
```

class sage.combinat.gelfand_tsetlin_patterns.**GelfandTsetlinPatterns**(*n*, *k*, *strict*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Gelfand-Tsetlin patterns.

INPUT:

- `n` – The width or depth of the array, also known as the rank
- `k` – (Default: `None`) If specified, this is the maximum value that can occur in the patterns
- `top_row` – (Default: `None`) If specified, this is the fixed top row of all patterns

- `strict` – (Default: `False`) Set to `True` if all patterns are strict patterns

TESTS:

Check that the number of Gelfand-Tsetlin patterns is equal to the number of semistandard Young tableaux:

```
sage: G = GelfandTsetlinPatterns(3,3)
sage: c = 0
sage: from sage.combinat.crystals.kirillov_reshetikhin import partitions_in_box
sage: for p in partitions_in_box(3,3):
...       S = SemistandardTableaux(p, max_entry=3)
...       c += S.cardinality()
sage: c == G.cardinality()
True
```

Note that the top row in reverse of the Gelfand-Tsetlin pattern is the shape of the corresponding semistandard Young tableau under the bijection described in `GelfandTsetlinPattern.to_tableau()`:

```
sage: G = GelfandTsetlinPatterns(top_row=[2,2,1])
sage: S = SemistandardTableaux([2,2,1], max_entry=3)
sage: G.cardinality() == S.cardinality()
True
```

Element
alias of `GelfandTsetlinPattern`

random_element()
Return a uniformly random Gelfand-Tsetlin pattern.

EXAMPLES:

```
sage: g = GelfandTsetlinPatterns(4, 5)
sage: g.random_element()
[[5, 2, 2, 1], [2, 2, 1], [2, 1], [1]]
sage: g = GelfandTsetlinPatterns(4, 5, strict=True)
sage: g.random_element()
[[5, 4, 1, 0], [5, 2, 1], [2, 1], [2]]
```

class sage.combinat.gelfand_tsetlin_patterns.**GelfandTsetlinPatternsTopRow**(*top_row*, *strict*)

Bases: `sage.combinat.gelfand_tsetlin_patterns.GelfandTsetlinPatterns`

Gelfand-Tsetlin patterns with a fixed top row.

Tokuyama_formula(*name='t'*)
Return the Tokuyama formula of `self`.

Following the exposition of [BBF], Tokuyama's formula asserts

$$\sum_G (t+1)^{s(G)} t^{l(G)} z_1^{d_{n+1}} z_2^{d_n - d_{n+1}} \cdots z_{n+1}^{d_1 - d_2} = s_\lambda(z_1, \ldots, z_{n+1}) \prod_{i<j} (z_j + t z_i),$$

where the sum is over all strict Gelfand-Tsetlin patterns with fixed top row $\lambda + \rho$, with λ a partition with at most $n+1$ parts and $\rho = (n, n-1, \ldots, 1, 0)$, and s_λ is a Schur function.

INPUT:

- `name` – (Default: `'t'`) An alternative name for the variable t.

EXAMPLES:

```
sage: GT = GelfandTsetlinPatterns(top_row=[2,1,0],strict=True)
sage: GT.Tokuyama_formula()
t^3*x1^2*x2 + t^2*x1*x2^2 + t^2*x1^2*x3 + t^2*x1*x2*x3 + t*x1*x2*x3 + t*x2^2*x3 + t*x1*x3^2
```

```
sage: GT = GelfandTsetlinPatterns(top_row=[3,2,1],strict=True)
sage: GT.Tokuyama_formula()
t^3*x1^3*x2^2*x3 + t^2*x1^2*x2^3*x3 + t^2*x1^3*x2*x3^2 + t^2*x1^2*x2^2*x3^2 + t*x1^2*x2^2*x3
sage: GT = GelfandTsetlinPatterns(top_row=[1,1,1],strict=True)
sage: GT.Tokuyama_formula()
0
```

random_element()

Return a uniformly random Gelfand-Tsetlin pattern with specified top row.

EXAMPLES:

```
sage: g = GelfandTsetlinPatterns(top_row = [4, 3, 1, 1])
sage: g.random_element()
[[4, 3, 1, 1], [4, 3, 1], [4, 1], [3]]
sage: g = GelfandTsetlinPatterns(top_row=[4, 3, 2, 1], strict=True)
sage: g.random_element()
[[4, 3, 2, 1], [4, 2, 1], [4, 1], [2]]
```

top_row()

Return the top row of `self`.

EXAMPLES:

```
sage: G = GelfandTsetlinPatterns(top_row=[4,4,3,1])
sage: G.top_row()
(4, 4, 3, 1)
```

5.1.102 Paths in Directed Acyclic Graphs

sage.combinat.graph_path.**GraphPaths**(*g*, *source=None*, *target=None*)

Returns the combinatorial class of paths in the directed acyclic graph g.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
```

If source and target are not given, then the returned class contains all paths (including trivial paths containing only one vertex).

```
sage: p = GraphPaths(G); p
Paths in Multi-digraph on 5 vertices
sage: p.cardinality()
37
sage: p.random_element()
[1, 2, 3, 4, 5]
```

If the source is specified, then the returned class contains all of the paths starting at the vertex source (including the trivial path).

```
sage: p = GraphPaths(G, source=3); p
Paths in Multi-digraph on 5 vertices starting at 3
sage: p.list()
[[3], [3, 4], [3, 4, 5], [3, 4, 5]]
```

If the target is specified, then the returned class contains all of the paths ending at the vertex target (including the trivial path).

```
sage: p = GraphPaths(G, target=3); p
Paths in Multi-digraph on 5 vertices ending at 3
sage: p.cardinality()
5
sage: p.list()
[[3], [1, 3], [2, 3], [1, 2, 3], [1, 2, 3]]
```

If both the target and source are specified, then the returned class contains all of the paths from source to target.

```
sage: p = GraphPaths(G, source=1, target=3); p
Paths in Multi-digraph on 5 vertices starting at 1 and ending at 3
sage: p.cardinality()
3
sage: p.list()
[[1, 2, 3], [1, 2, 3], [1, 3]]
```

Note that G must be a directed acyclic graph.

```
sage: G = DiGraph({1:[2,2,3,5], 2:[3,4], 3:[4], 4:[2,5,7], 5:[6]}, multiedges=True)
sage: GraphPaths(G)
Traceback (most recent call last):
...
TypeError: g must be a directed acyclic graph
```

class sage.combinat.graph_path.**GraphPaths_all**(*g*)

Bases: sage.combinat.combinat.CombinatorialClass, sage.combinat.graph_path.GraphPaths_comm

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G)
sage: p.cardinality()
37
```

list()

Returns a list of the paths of self.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: len(GraphPaths(G).list())
37
```

class sage.combinat.graph_path.**GraphPaths_common**

incoming_edges(*v*)

Returns a list of v's incoming edges.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G)
sage: p.incoming_edges(2)
[(1, 2, None), (1, 2, None)]
```

incoming_paths(*v*)

Returns a list of paths that end at v.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: gp = GraphPaths(G)
sage: gp.incoming_paths(2)
[[2], [1, 2], [1, 2]]
```

outgoing_edges(*v*)

Returns a list of v's outgoing edges.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G)
sage: p.outgoing_edges(2)
[(2, 3, None), (2, 4, None)]
```

outgoing_paths(*v*)

Returns a list of the paths that start at v.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: gp = GraphPaths(G)
sage: gp.outgoing_paths(3)
[[3], [3, 4], [3, 4, 5], [3, 4, 5]]
sage: gp.outgoing_paths(2)
[[2],
 [2, 3],
 [2, 3, 4],
 [2, 3, 4, 5],
 [2, 3, 4, 5],
 [2, 4],
 [2, 4, 5],
 [2, 4, 5]]
```

paths()

Returns a list of all the paths of self.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: gp = GraphPaths(G)
sage: len(gp.paths())
37
```

paths_from_source_to_target(*source*, *target*)

Returns a list of paths from source to target.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: gp = GraphPaths(G)
sage: gp.paths_from_source_to_target(2,4)
[[2, 3, 4], [2, 4]]
```

class sage.combinat.graph_path.**GraphPaths_s**(*g*, *source*)

Bases: sage.combinat.combinat.CombinatorialClass, sage.combinat.graph_path.GraphPaths_comm

TESTS:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G, 4)
```

```
sage: p == loads(dumps(p))
True
```

list()

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G, 4)
sage: p.list()
[[4], [4, 5], [4, 5]]
```

class sage.combinat.graph_path.**GraphPaths_st**(*g*, *source*, *target*)

Bases: sage.combinat.combinat.CombinatorialClass, sage.combinat.graph_path.GraphPaths_comm

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: GraphPaths(G,1,2).cardinality()
2
sage: GraphPaths(G,1,3).cardinality()
3
sage: GraphPaths(G,1,4).cardinality()
5
sage: GraphPaths(G,1,5).cardinality()
10
sage: GraphPaths(G,2,3).cardinality()
1
sage: GraphPaths(G,2,4).cardinality()
2
sage: GraphPaths(G,2,5).cardinality()
4
sage: GraphPaths(G,3,4).cardinality()
1
sage: GraphPaths(G,3,5).cardinality()
2
sage: GraphPaths(G,4,5).cardinality()
2
```

list()

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G,1,2)
sage: p.list()
[[1, 2], [1, 2]]
```

class sage.combinat.graph_path.**GraphPaths_t**(*g*, *target*)

Bases: sage.combinat.combinat.CombinatorialClass, sage.combinat.graph_path.GraphPaths_comm

TESTS:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G, target=4)
sage: p == loads(dumps(p))
True
```

list()

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G, target=4)
sage: p.list()
[[4],
 [2, 4],
 [1, 2, 4],
 [1, 2, 4],
 [3, 4],
 [1, 3, 4],
 [2, 3, 4],
 [1, 2, 3, 4],
 [1, 2, 3, 4]]
```

5.1.103 Gray codes

REFERENCES:

Functions

sage.combinat.gray_codes.**combinations**(*n*, *t*)

Iterator through the switches of the revolving door algorithm.

The revolving door algorithm is a way to generate all combinations of a set (i.e. the subset of given cardinality) in such way that two consecutive subsets differ by one element. At each step, the iterator output a pair `(i, j)` where the item `i` has to be removed and `j` has to be added.

The ground set is always $\{0, 1, ..., n-1\}$. Note that `n` can be infinity in that algorithm.

See [Knuth-TAOCP3A].

INPUT:

- `n` – (integer or `Infinity`) – size of the ground set
- `t` – (integer) – size of the subsets

EXAMPLES:

```
sage: from sage.combinat.gray_codes import combinations
sage: b = [1, 1, 1, 0, 0]
sage: for i,j in combinations(5,3):
....:     b[i] = 0; b[j] = 1
....:     print b
[1, 0, 1, 1, 0]
[0, 1, 1, 1, 0]
[1, 1, 0, 1, 0]
[1, 0, 0, 1, 1]
[0, 1, 0, 1, 1]
[0, 0, 1, 1, 1]
[1, 0, 1, 0, 1]
[0, 1, 1, 0, 1]
[1, 1, 0, 0, 1]

sage: s = set([0,1])
sage: for i,j in combinations(4,2):
....:     s.remove(i)
....:     s.add(j)
....:     print s
```

```
set([1, 2])
set([0, 2])
set([2, 3])
set([1, 3])
set([0, 3])
```

Note that `n` can be infinity:

```
sage: c = combinations(Infinity,4)
sage: s = set([0,1,2,3])
sage: for _ in xrange(10):
....:     i,j = next(c)
....:     s.remove(i); s.add(j)
....:     print s
set([0, 1, 3, 4])
set([1, 2, 3, 4])
set([0, 2, 3, 4])
set([0, 1, 2, 4])
set([0, 1, 4, 5])
set([1, 2, 4, 5])
set([0, 2, 4, 5])
set([2, 3, 4, 5])
set([1, 3, 4, 5])
set([0, 3, 4, 5])
sage: for _ in xrange(1000):
....:     i,j = next(c)
....:     s.remove(i); s.add(j)
sage: print s
set([0, 4, 13, 14])
```

TESTS:

```
sage: def check_sets_from_iter(n,k):
....:     l = []
....:     s = set(range(k))
....:     l.append(frozenset(s))
....:     for i,j in combinations(n,k):
....:         s.remove(i)
....:         s.add(j)
....:         assert len(s) == k
....:         l.append(frozenset(s))
....:     assert len(set(l)) == binomial(n,k)
sage: check_sets_from_iter(9,5)
sage: check_sets_from_iter(8,5)
sage: check_sets_from_iter(5,6)
Traceback (most recent call last):
...
AssertionError: t(=6) must be >=0 and <=n(=5)
```

`sage.combinat.gray_codes.`**`product`**`(`m`)`

Iterator over the switch for the iteration of the product $[m_0] \times [m_1] \dots \times [m_k]$.

The iterator return at each step a pair `(p,i)` which corresponds to the modification to perform to get the next element. More precisely, one has to apply the increment `i` at the position `p`. By construction, the increment is either `+1` or `-1`.

This is algorithm H in [Knuth-TAOCP2A]: loopless reflected mixed-radix Gray generation.

INPUT:

- m – a list or tuple of positive integers that correspond to the size of the sets in the product

EXAMPLES:

```
sage: from sage.combinat.gray_codes import product
sage: l = [0,0,0]
sage: for p,i in product([3,3,3]):
....:     l[p] += i
....:     print l
[1, 0, 0]
[2, 0, 0]
[2, 1, 0]
[1, 1, 0]
[0, 1, 0]
[0, 2, 0]
[1, 2, 0]
[2, 2, 0]
[2, 2, 1]
[1, 2, 1]
[0, 2, 1]
[0, 1, 1]
[1, 1, 1]
[2, 1, 1]
[2, 0, 1]
[1, 0, 1]
[0, 0, 1]
[0, 0, 2]
[1, 0, 2]
[2, 0, 2]
[2, 1, 2]
[1, 1, 2]
[0, 1, 2]
[0, 2, 2]
[1, 2, 2]
[2, 2, 2]
sage: l = [0,0]
sage: for i,j in product([2,1]):
....:     l[i] += j
....:     print l
[1, 0]
```

TESTS:

```
sage: for t in [[2,2,2],[2,1,2],[3,2,1],[2,1,3]]:
....:     assert sum(1 for _ in product(t)) == prod(t)-1
```

5.1.104 Hall Polynomials

sage.combinat.hall_polynomial.**hall_polynomial**(*nu*, *mu*, *la*, *q=None*)

Return the (classical) Hall polynomial $P^{\nu}_{\mu,\lambda}$ (where ν, μ and λ are the inputs `nu`, `mu` and `la`).

Let ν, μ, λ be partitions. The Hall polynomial $P^{\nu}_{\mu,\lambda}(q)$ (in the indeterminate q) is defined as follows: Specialize q to a prime power, and consider the category of $\mathbf{F}_q$-vector spaces with a distinguished nilpotent endomorphism. The morphisms in this category shall be the linear maps commuting with the distinguished endomorphisms. The *type* of an object in the category will be the Jordan type of the distinguished endomorphism; this is a partition. Now, if N is any fixed object of type ν in this category, then the polynomial $P^{\nu}_{\mu,\lambda}(q)$ specialized at the prime power q counts the number of subobjects L of N having type λ such that the quotient object N/L has type μ. This determines the values of the polynomial $P^{\nu}_{\mu,\lambda}$ at infinitely many points (namely, at all prime powers),

and hence $P^{\nu}_{\mu,\lambda}$ is uniquely determined. The degree of this polynomial is at most $n(\nu) - n(\lambda) - n(\mu)$, where $n(\kappa) = \sum_i (i-1)\kappa_i$ for every partition κ. (If this is negative, then the polynomial is zero.)

These are the structure coefficients of the `(classical) Hall algebra`.

If $|\nu| \neq |\mu| + |\lambda|$, then we have $P^{\nu}_{\mu,\lambda} = 0$. More generally, if the Littlewood-Richardson coefficient $c^{\nu}_{\mu,\lambda}$ vanishes, then $P^{\nu}_{\mu,\lambda} = 0$.

The Hall polynomials satisfy the symmetry property $P^{\nu}_{\mu,\lambda} = P^{\nu}_{\lambda,\mu}$.

ALGORITHM:

If $\lambda = (1^r)$ and $|\nu| = |\mu| + |\lambda|$, then we compute $P^{\nu}_{\mu,\lambda}$ as follows (cf. Example 2.4 in [Schiffmann]):

First, write $\nu = (1^{l_1}, 2^{l_2}, \ldots, n^{l_n})$, and define a sequence $r = r_0 \geq r_1 \geq \cdots \geq r_n$ such that

$$\mu = \left(1^{l_1 - r_0 + 2r_1 - r_2}, 2^{l_2 - r_1 + 2r_2 - r_3}, \ldots, (n-1)^{l_{n-1} - r_{n-2} + 2r_{n-1} - r_n}, n^{l_n - r_{n-1} + r_n}\right).$$

Thus if $\mu = (1^{m_1}, \ldots, n^{m_n})$, we have the following system of equations:

$$\begin{aligned} m_1 &= l_1 - r + 2r_1 - r_2, \\ m_2 &= l_2 - r_1 + 2r_2 - r_3, \\ &\vdots, \\ m_{n-1} &= l_{n-1} - r_{n-2} + 2r_{n-1} - r_n, \\ m_n &= l_n - r_{n-1} + r_n \end{aligned}$$

and solving for r_i and back substituting we obtain the equations:

$$\begin{aligned} r_n &= r_{n-1} + m_n - l_n, \\ r_{n-1} &= r_{n-2} + m_{n-1} - l_{n-1} + m_n - l_n, \\ &\vdots, \\ r_1 &= r + \sum_{k=1}^{n} (m_k - l_k), \end{aligned}$$

or in general we have the recursive equation:

$$r_i = r_{i-1} + \sum_{k=i}^{n} (m_k - l_k).$$

This, combined with the condition that $r_0 = r$, determines the r_i uniquely (recursively). Next we define

$$t = (r_{n-2} - r_{n-1})(l_n - r_{n-1}) + (r_{n-3} - r_{n-2})(l_{n-1} + l_n - r_{n-2}) + \cdots + (r_0 - r_1)(l_2 + \cdots + l_n - r_1),$$

and with these notations we have

$$P^{\nu}_{\mu,(1^r)} = q^t \binom{l_n}{r_{n-1}}_q \binom{l_{n-1}}{r_{n-2} - r_{n-1}}_q \cdots \binom{l_1}{r_0 - r_1}_q.$$

To compute $P^{\nu}_{\mu,\lambda}$ in general, we compute the product $I_\mu I_\lambda$ in the Hall algebra and return the coefficient of I_ν.

EXAMPLES:

```
sage: from sage.combinat.hall_polynomial import hall_polynomial
sage: hall_polynomial([1,1],[1],[1])
q + 1
sage: hall_polynomial([2],[1],[1])
1
```

```
sage: hall_polynomial([2,1],[2],[1])
q
sage: hall_polynomial([2,2,1],[2,1],[1,1])
q^2 + q
sage: hall_polynomial([2,2,2,1],[2,2,1],[1,1])
q^4 + q^3 + q^2
sage: hall_polynomial([3,2,2,1], [3,2], [2,1])
q^6 + q^5
sage: hall_polynomial([4,2,1,1], [3,1,1], [2,1])
2*q^3 + q^2 - q - 1
sage: hall_polynomial([4,2], [2,1], [2,1], 0)
1
```

5.1.105 Enumerated set of lists of integers with constraints: base classes

- `IntegerListsBackend`: base class for the Cython back-end of an enumerated set of lists of integers with specified constraints.
- `Envelope`: a utility class for upper (lower) envelope of a function under constraints.

class `sage.combinat.integer_lists.base.`**Envelope**

Bases: `object`

The (currently approximated) upper (lower) envelope of a function under the specified constraints.

INPUT:

- `f` – a function, list, or tuple; if `f` is a list, it is considered as the function `f(i)=f[i]`, completed for larger i with `f(i)=max_part`.
- `min_part`, `max_part`, `min_slope`, `max_slope`, ... as for `IntegerListsLex` (please consult for details).
- `sign` – (+1 or -1) multiply the input values with `sign` and multiply the output with `sign`. Setting this to -1 can be used to implement a lower envelope.

The *upper envelope* $U(f)$ of f is the (pointwise) largest function which is bounded above by f and satisfies the `max_part` and `max_slope` conditions. Furthermore, for `i,i+1<min_length`, the upper envelope also satisfies the `min_slope` condition.

Upon computing $U(f)(i)$, all the previous values for $j \leq i$ are computed and cached; in particular $f(i)$ will be computed at most once for each i.

Todo

- This class is a good candidate for Cythonization, especially to get the critical path in `__call__` super fast.
- To get full envelopes, we would want both the `min_slope` and `max_slope` conditions to always be satisfied. This is only properly defined for the restriction of f to a finite interval $0, .., k$, and depends on k.
- This is the core "data structure" of `IntegerListsLex`. Improving the lookahead there essentially depends on having functions with a good complexity to compute the area below an envelope; and in particular how it evolves when increasing the length.

EXAMPLES:

```
sage: from sage.combinat.integer_lists import Envelope
```

Trivial upper and lower envelopes:

```
sage: f = Envelope([3,2,2])
sage: [f(i) for i in range(10)]
[3, 2, 2, inf, inf, inf, inf, inf, inf, inf]
sage: f = Envelope([3,2,2], sign=-1)
sage: [f(i) for i in range(10)]
[3, 2, 2, 0, 0, 0, 0, 0, 0, 0]
```

A more interesting lower envelope:

```
sage: f = Envelope([4,1,5,3,5], sign=-1, min_part=2, min_slope=-1)
sage: [f(i) for i in range(10)]
[4, 3, 5, 4, 5, 4, 3, 2, 2, 2]
```

Currently, adding `max_slope` has no effect:

```
sage: f = Envelope([4,1,5,3,5], sign=-1, min_part=2, min_slope=-1, max_slope=0)
sage: [f(i) for i in range(10)]
[4, 3, 5, 4, 5, 4, 3, 2, 2, 2]
```

unless `min_length` is large enough:

```
sage: f = Envelope([4,1,5,3,5], sign=-1, min_part=2, min_slope=-1, max_slope=0, min_length=2)
sage: [f(i) for i in range(10)]
[4, 3, 5, 4, 5, 4, 3, 2, 2, 2]

sage: f = Envelope([4,1,5,3,5], sign=-1, min_part=2, min_slope=-1, max_slope=0, min_length=4)
sage: [f(i) for i in range(10)]
[5, 5, 5, 4, 5, 4, 3, 2, 2, 2]

sage: f = Envelope([4,1,5,3,5], sign=-1, min_part=2, min_slope=-1, max_slope=0, min_length=5)
sage: [f(i) for i in range(10)]
[5, 5, 5, 5, 5, 4, 3, 2, 2, 2]
```

A non trivial upper envelope:

```
sage: f = Envelope([9,1,5,4], max_part=7, max_slope=2)
sage: [f(i) for i in range(10)]
[7, 1, 3, 4, 6, 7, 7, 7, 7, 7]
```

TESTS:

```
sage: f = Envelope(3, min_slope=1)
sage: [f(i) for i in range(10)]
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]

sage: f = Envelope(3, min_slope=1, min_length=5)
sage: [f(i) for i in range(10)]
[-1, 0, 1, 2, 3, 3, 3, 3, 3, 3]

sage: f = Envelope(3, sign=-1, min_slope=1)
sage: [f(i) for i in range(10)]
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

sage: f = Envelope(3, sign=-1, max_slope=-1, min_length=4)
sage: [f(i) for i in range(10)]
[6, 5, 4, 3, 3, 3, 3, 3, 3, 3]
```

adapt (m, j)

Return this envelope adapted to an additional local constraint.

INPUT:

- m – a nonnegative integer (starting value)
- j – a nonnegative integer (position)

This method adapts this envelope to the additional local constraint imposed by having a part m at position j. Namely, this returns a function which computes, for any $i > j$, the minimum of the ceiling function and the value restriction given by the slope conditions.

EXAMPLES:

```
sage: from sage.combinat.integer_lists import Envelope
sage: f = Envelope(3)
sage: g = f.adapt(1,1)
sage: g is f
True
sage: [g(i) for i in range(10)]
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]

sage: f = Envelope(3, max_slope=1)
sage: g = f.adapt(1,1)
sage: [g(i) for i in range(10)]
[0, 1, 2, 3, 3, 3, 3, 3, 3, 3]
```

Note that, in both cases above, the adapted envelope is only guaranteed to be valid for $i > j$! This is to leave potential room in the future for sharing similar adapted envelopes:

```
sage: g = f.adapt(0,0)
sage: [g(i) for i in range(10)]
[0, 1, 2, 3, 3, 3, 3, 3, 3, 3]

sage: g = f.adapt(2,2)
sage: [g(i) for i in range(10)]
[0, 1, 2, 3, 3, 3, 3, 3, 3, 3]

sage: g = f.adapt(3,3)
sage: [g(i) for i in range(10)]
[0, 1, 2, 3, 3, 3, 3, 3, 3, 3]
```

Now with a lower envelope:

```
sage: f = Envelope(1, sign=-1, min_slope=-1)
sage: g = f.adapt(2,2)
sage: [g(i) for i in range(10)]
[4, 3, 2, 1, 1, 1, 1, 1, 1, 1]
sage: g = f.adapt(1,3)
sage: [g(i) for i in range(10)]
[4, 3, 2, 1, 1, 1, 1, 1, 1, 1]
```

limit()

Return a bound on the limit of self.

OUTPUT: a nonnegative integer or ∞

This returns some upper bound for the accumulation points of this upper envelope. For a lower envelope, a lower bound is returned instead.

In particular this gives a bound for the value of self at i for i large enough. Special case: for a lower envelop, and when the limit is ∞, the envelope is guaranteed to tend to ∞ instead.

When s=self.limit_start() is finite, this bound is guaranteed to be valid for $i >= s$.

Sometimes it's better to have a loose bound that starts early; sometimes the converse holds. At this point which specific bound and starting point is returned is not set in stone, in order to leave room for later optimizations.

EXAMPLES:

```
sage: from sage.combinat.integer_lists import Envelope
sage: Envelope([4,1,5]).limit()
inf
sage: Envelope([4,1,5], max_part=2).limit()
2
sage: Envelope([4,1,5], max_slope=0).limit()
1
sage: Envelope(lambda x: 3, max_part=2).limit()
2
```

Lower envelopes:

```
sage: Envelope(lambda x: 3, min_part=2, sign=-1).limit()
2
sage: Envelope([4,1,5], min_slope=0, sign=-1).limit()
5
sage: Envelope([4,1,5], sign=-1).limit()
0
```

See also:

`limit_start()`

limit_start()

Return from which i on the bound returned by `limit` holds.

See also:

`limit()` for the precise specifications.

EXAMPLES:

```
sage: from sage.combinat.integer_lists import Envelope
sage: Envelope([4,1,5]).limit_start()
3
sage: Envelope([4,1,5], sign=-1).limit_start()
3

sage: Envelope([4,1,5], max_part=2).limit_start()
3

sage: Envelope(4).limit_start()
0
sage: Envelope(4, sign=-1).limit_start()
0

sage: Envelope(lambda x: 3).limit_start() == Infinity
True
sage: Envelope(lambda x: 3, max_part=2).limit_start() == Infinity
True

sage: Envelope(lambda x: 3, sign=-1, min_part=2).limit_start() == Infinity
True
```

max_part

max_slope

min_slope

sign

class sage.combinat.integer_lists.base.**IntegerListsBackend**
Bases: object

Base class for the Cython back-end of an enumerated set of lists of integers with specified constraints.

This base implements the basic operations, including checking for containment using `_contains()`, but not iteration. For iteration, subclass this class and implement an `_iter()` method.

EXAMPLES:

```
sage: from sage.combinat.integer_lists.base import IntegerListsBackend
sage: L = IntegerListsBackend(6, max_slope=-1)
sage: L._contains([3,2,1])
True
```

ceiling

floor

max_length

max_part

max_slope

max_sum

min_length

min_part

min_slope

min_sum

5.1.106 Enumerated set of lists of integers with constraints: front-end

- `IntegerLists`: class which models an enumerated set of lists of integers with certain constraints. This is a Python front-end where all user-accessible functionality should be implemented.

class sage.combinat.integer_lists.lists.**IntegerList**
Bases: sage.structure.list_clone.ClonableArray

Element class for `IntegerLists`.

check()
Check to make sure this is a valid element in its `IntegerLists` parent.

EXAMPLES:

```
sage: C = IntegerListsLex(4)
sage: C([4]).check()
True
sage: C([5]).check()
False
```

class sage.combinat.integer_lists.lists.**IntegerLists**(**args*, ***kwds*)
Bases: sage.structure.parent.Parent

Enumerated set of lists of integers with constraints.

Currently, this is simply an abstract base class which should not be used by itself. See `IntegerListsLex` for a class which can be used by end users.

`IntegerLists` is just a Python front-end, acting as a `Parent`, supporting element classes and so on. The attribute `.backend` which is an instance of `sage.combinat.integer_lists.base.IntegerListsBackend` is the Cython back-end which implements all operations such as iteration.

The front-end (i.e. this class) and the back-end are supposed to be orthogonal: there is no imposed correspondence between front-ends and back-ends.

For example, the set of partitions of 5 and the set of weakly decreasing sequences which sum to 5 might be implemented by the same back-end, but they will be presented to the user by a different front-end.

EXAMPLES:

```
sage: from sage.combinat.integer_lists import IntegerLists
sage: L = IntegerLists(5)
sage: L
Integer lists of sum 5 satisfying certain constraints

sage: IntegerListsLex(2, length=3, name="A given name")
A given name
```

Element
alias of `IntegerList`

backend_class
alias of `IntegerListsBackend`

5.1.107 Enumerated set of lists of integers with constraints, in inverse lexicographic order

- `IntegerListsLex`: the enumerated set of lists of nonnegative integers with specified constraints, in inverse lexicographic order.
- `IntegerListsBackend_invlex`: Cython back-end for `IntegerListsLex`.

HISTORY:

This generic tool was originally written by Hivert and Thiery in MuPAD-Combinat in 2000 and ported over to Sage by Mike Hansen in 2007. It was then completely rewritten in 2015 by Gillespie, Schilling, and Thiery, with the help of many, to deal with limitations and lack of robustness w.r.t. input.

class `sage.combinat.integer_lists.invlex.`**IntegerListsBackend_invlex**
Bases: `sage.combinat.integer_lists.base.IntegerListsBackend`

Cython back-end of an set of lists of integers with specified constraints enumerated in inverse lexicographic order.

check

class `sage.combinat.integer_lists.invlex.`**IntegerListsLex**(**args*, ***kwds*)
Bases: `sage.combinat.integer_lists.lists.IntegerLists`

Lists of nonnegative integers with constraints, in inverse lexicographic order.

An *integer list* is a list l of nonnegative integers, its *parts*. The slope (at position i) is the difference `l[i+1]-l[i]` between two consecutive parts.

This class allows to construct the set S of all integer lists l satisfying specified bounds on the sum, the length, the slope, and the individual parts, enumerated in *inverse* lexicographic order, that is from largest to smallest in lexicographic order. Note that, to admit such an enumeration, S is almost necessarily finite (see *IntegerListsLex_finiteness*).

The main purpose is to provide a generic iteration engine for all the enumerated sets like `Partitions`, `Compositions`, `IntegerVectors`. It can also be used to generate many other combinatorial objects like Dyck paths, Motzkin paths, etc. Mathematically speaking, this is a special case of set of integral points of a polytope (or union thereof, when the length is not fixed).

INPUT:

- `min_sum` – a nonnegative integer (default: 0): a lower bound on `sum(l)`.
- `max_sum` – a nonnegative integer or ∞ (default: ∞): an upper bound on `sum(l)`.
- `n` – a nonnegative integer (optional): if specified, this overrides `min_sum` and `max_sum`.
- `min_length` – a nonnegative integer (default: 0): a lower bound on `len(l)`.
- `max_length` – a nonnegative integer or ∞ (default: ∞): an upper bound on `len(l)`.
- `length` – an integer (optional); overrides `min_length` and `max_length` if specified;
- **`min_part` – a nonnegative integer: a lower bounds on all** parts: `min_part <= l[i]` for `0 <= i < len(l)`.
- `floor` – a list of nonnegative integers or a function: lower bounds on the individual parts $l[i]$.

  If `floor` is a list of integers, then `floor<=l[i]` for `0 <= i < min(len(l), len(floor)`. Similarly, if `floor` is a function, then `floor(i) <= l[i]` for `0 <= i < len(l)`.
- `max_part` – a nonnegative integer or ∞: an upper bound on all parts: `l[i] <= max_part` for `0 <= i < len(l)`.
- `ceiling` – upper bounds on the individual parts `l[i]`; this takes the same type of input as `floor`, except that ∞ is allowed in addition to integers, and the default value is ∞.
- `min_slope` – an integer or $-\infty$ (default: $-\infty$): an lower bound on the slope between consecutive parts: `min_slope <= l[i+1]-l[i]` for `0 <= i < len(l)-1`
- `max_slope` – an integer or $+\infty$ (defaults: $+\infty$) an upper bound on the slope between consecutive parts: `l[i+1]-l[i] <= max_slope` for `0 <= i < len(l)-1`
- `category` – a category (default: `FiniteEnumeratedSets`)
- `check` – boolean (default: `True`): whether to display the warnings raised when functions are given as input to `floor` or `ceiling` and the errors raised when there is no proper enumeration.
- `name` – a string or `None` (default: `None`) if set, this will be passed down to `Parent.rename()` to specify the name of `self`. It is recommented to use directly the rename method as this feature may become deprecated.
- `element_constructor` – a function (or callable) that creates elements of `self` from a list. See also `Parent`.
- `element_class` – a class for the elements of `self` (default: $ClonableArray$). This merely sets the attribute `self.Element`. See the examples for details.
- `global_options` – a `GlobalOptions` object that will be assigned to the attribute `_global_options`; for internal use only (subclasses, ...).

Note: When several lists satisfying the constraints differ only by trailing zeroes, only the shortest one is enumerated (and therefore counted). The others are still considered valid. See the examples below.

This feature is questionable. It is recommended not to rely on it, as it may eventually be discontinued.

EXAMPLES:

We create the enumerated set of all lists of nonnegative integers of length 3 and sum 2:

```
sage: C = IntegerListsLex(2, length=3)
sage: C
Integer lists of sum 2 satisfying certain constraints
sage: C.cardinality()
6
sage: [p for p in C]
[[2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]

sage: [2, 0, 0] in C
True
sage: [2, 0, 1] in C
False
sage: "a" in C
False
sage: ["a"] in C
False
sage: C.first()
[2, 0, 0]
```

One can specify lower and upper bounds on each part:

```
sage: list(IntegerListsLex(5, length=3, floor=[1,2,0], ceiling=[3,2,3]))
[[3, 2, 0], [2, 2, 1], [1, 2, 2]]
```

When the length is fixed as above, one can also use `IntegerVectors`:

```
sage: IntegerVectors(2,3).list()
[[2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]
```

Using the slope condition, one can generate integer partitions (but see `Partitions`):

```
sage: list(IntegerListsLex(4, max_slope=0))
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

The following is the list of all partitions of 7 with parts at least 2:

```
sage: list(IntegerListsLex(7, max_slope=0, min_part=2))
[[7], [5, 2], [4, 3], [3, 2, 2]]
```

floor and ceiling conditions

Next we list all partitions of 5 of length at most 3 which are bounded below by `[2,1,1]`:

```
sage: list(IntegerListsLex(5, max_slope=0, max_length=3, floor=[2,1,1]))
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1]]
```

Note that `[5]` is considered valid, because the floor constraints only apply to existing positions in the list. To obtain instead the partitions containing `[2,1,1]`, one needs to use `min_length` or `length`:

```
sage: list(IntegerListsLex(5, max_slope=0, length=3, floor=[2,1,1]))
[[3, 1, 1], [2, 2, 1]]
```

Here is the list of all partitions of 5 which are contained in `[3,2,2]`:

```
sage: list(IntegerListsLex(5, max_slope=0, max_length=3, ceiling=[3,2,2]))
[[3, 2], [3, 1, 1], [2, 2, 1]]
```

This is the list of all compositions of 4 (but see `Compositions`):

```
sage: list(IntegerListsLex(4, min_part=1))
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

This is the list of all integer vectors of sum 4 and length 3:

```
sage: list(IntegerListsLex(4, length=3))
[[4, 0, 0], [3, 1, 0], [3, 0, 1], [2, 2, 0], [2, 1, 1],
 [2, 0, 2], [1, 3, 0], [1, 2, 1], [1, 1, 2], [1, 0, 3],
 [0, 4, 0], [0, 3, 1], [0, 2, 2], [0, 1, 3], [0, 0, 4]]
```

For whatever it is worth, the `floor` and `min_part` constraints can be combined:

```
sage: L = IntegerListsLex(5, floor=[2,0,2], min_part=1)
sage: L.list()
[[5], [4, 1], [3, 2], [2, 3], [2, 1, 2]]
```

This is achieved by updating the floor upon constructing `L`:

```
sage: [L.floor(i) for i in range(5)]
[2, 1, 2, 1, 1]
```

Similarly, the `ceiling` and `max_part` constraints can be combined:

```
sage: L = IntegerListsLex(4, ceiling=[2,3,1], max_part=2, length=3)
sage: L.list()
[[2, 2, 0], [2, 1, 1], [1, 2, 1]]
sage: [L.ceiling(i) for i in range(5)]
[2, 2, 1, 2, 2]
```

This can be used to generate Motzkin words (see Wikipedia article Motzkin_number):

```
sage: def motzkin_words(n):
....:     return IntegerListsLex(length=n+1, min_slope=-1, max_slope=1,
....:                            ceiling=[0]+[+oo for i in range(n-1)]+[0])
sage: motzkin_words(4).list()
[[0, 1, 2, 1, 0],
 [0, 1, 1, 1, 0],
 [0, 1, 1, 0, 0],
 [0, 1, 0, 1, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 1, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 0]]
sage: [motzkin_words(n).cardinality() for n in range(8)]
[1, 1, 2, 4, 9, 21, 51, 127]
sage: oeis(_)                  # optional -- internet
0: A001006: Motzkin numbers: number of ways of drawing any number
of nonintersecting chords joining n (labeled) points on a circle.
```

or Dyck words (see also `DyckWords`), through the bijection with paths from $(0,0)$ to (n,n) with left and up steps that remain below the diagonal:

```
sage: def dyck_words(n):
....:     return IntegerListsLex(length=n, ceiling=range(n+1), min_slope=0)
sage: [dyck_words(n).cardinality() for n in range(8)]
```

```
[1, 1, 2, 5, 14, 42, 132, 429]
sage: dyck_words(3).list()
[[0, 1, 2], [0, 1, 1], [0, 0, 2], [0, 0, 1], [0, 0, 0]]
```

On finiteness and inverse lexicographic enumeration

The set of all lists of integers cannot be enumerated in inverse lexicographic order, since there is no largest list (take $[n]$ for n as large as desired):

```
sage: IntegerListsLex().first()
Traceback (most recent call last):
...
ValueError: could not prove that the specified constraints yield a finite set
```

Here is a variant which could be enumerated in lexicographic order but not in inverse lexicographic order:

```
sage: L = IntegerListsLex(length=2, ceiling=[Infinity, 0], floor=[0,1])
sage: for l in L: print l
Traceback (most recent call last):
...
ValueError: infinite upper bound for values of m
```

Even when the sum is specified, it is not necessarily possible to enumerate *all* elements in inverse lexicographic order. In the following example, the list `[1, 1, 1]` will never appear in the enumeration:

```
sage: IntegerListsLex(3).first()
Traceback (most recent call last):
...
ValueError: could not prove that the specified constraints yield a finite set
```

If one wants to proceed anyway, one can sign a waiver by setting `check=False` (again, be warned that some valid lists may never appear):

```
sage: L = IntegerListsLex(3, check=False)
sage: it = iter(L)
sage: [next(it) for i in range(6)]
[[3], [2, 1], [2, 0, 1], [2, 0, 0, 1], [2, 0, 0, 0, 1], [2, 0, 0, 0, 0, 1]]
```

In fact, being inverse lexicographically enumerable is almost equivalent to being finite. The only infinity that can occur would be from a tail of numbers $0, 1$ as in the previous example, where the 1 moves further and further to the right. If there is any list that is inverse lexicographically smaller than such a configuration, the iterator would not reach it and hence would not be considered iterable. Given that the infinite cases are very specific, at this point only the finite cases are supported (without signing the waiver).

The finiteness detection is not complete yet, so some finite cases may not be supported either, at least not without disabling the checks. Practical examples of such are welcome.

On trailing zeroes, and their caveats

As mentioned above, when several lists satisfying the constraints differ only by trailing zeroes, only the shortest one is listed:

```
sage: L = IntegerListsLex(max_length=4, max_part=1)
sage: L.list()
[[1, 1, 1, 1],
 [1, 1, 1],
```

```
 [1, 1, 0, 1],
 [1, 1],
 [1, 0, 1, 1],
 [1, 0, 1],
 [1, 0, 0, 1],
 [1],
 [0, 1, 1, 1],
 [0, 1, 1],
 [0, 1, 0, 1],
 [0, 1],
 [0, 0, 1, 1],
 [0, 0, 1],
 [0, 0, 0, 1],
 []]
```

and counted:

```
sage: L.cardinality()
16
```

Still, the others are considered as elements of L:

```
sage: L = IntegerListsLex(4,min_length=3,max_length=4)
sage: L.list()
[..., [2, 2, 0], ...]

sage: [2, 2, 0] in L       # in L.list()
True
sage: [2, 2, 0, 0] in L    # not in L.list() !
True
sage: [2, 2, 0, 0, 0] in L
False
```

Specifying functions as input for the floor or ceiling

We construct all lists of sum 4 and length 4 such that `l[i] <= i`:

```
sage: list(IntegerListsLex(4, length=4, ceiling=lambda i: i, check=False))
[[0, 1, 2, 1], [0, 1, 1, 2], [0, 1, 0, 3], [0, 0, 2, 2], [0, 0, 1, 3]]
```

Warning: When passing a function as `floor` or `ceiling`, it may become undecidable to detect improper inverse lexicographic enumeration. For example, the following example has a finite enumeration:

```
sage: L = IntegerListsLex(3, floor=lambda i: 1 if i>=2 else 0, check=False)
sage: L.list()
[[3],
 [2, 1],
 [2, 0, 1],
 [1, 2],
 [1, 1, 1],
 [1, 0, 2],
 [1, 0, 1, 1],
 [0, 3],
 [0, 2, 1],
 [0, 1, 2],
 [0, 1, 1, 1],
 [0, 0, 3],
 [0, 0, 2, 1],
 [0, 0, 1, 2],
 [0, 0, 1, 1, 1]]
```

but one cannot decide whether the following has an improper inverse lexicographic enumeration without computing the floor all the way to `Infinity`:

```
sage: L = IntegerListsLex(3, floor=lambda i: 0, check=False)
sage: it = iter(L)
sage: [next(it) for i in range(6)]
[[3], [2, 1], [2, 0, 1], [2, 0, 0, 1], [2, 0, 0, 0, 1], [2, 0, 0, 0, 0, 1]]
```

Hence a warning is raised when a function is specified as input, unless the waiver is signed by setting `check=False`:

```
sage: L = IntegerListsLex(3, floor=lambda i: 1 if i>=2 else 0)
doctest:...
A function has been given as input of the floor=[...] or ceiling=[...]
arguments of IntegerListsLex. Please see the documentation for the caveats.
If you know what you are doing, you can set check=False to skip this warning.
```

Similarly, the algorithm may need to search forever for a solution when the ceiling is ultimately zero:

```
sage: L = IntegerListsLex(2,ceiling=lambda i:0, check=False)
sage: L.first()           # not tested: will hang forever
sage: L = IntegerListsLex(2,ceiling=lambda i:0 if i<20 else 1, check=False)
sage: it = iter(L)
sage: next(it)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]
sage: next(it)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1]
sage: next(it)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1]
```

Tip: using disjoint union enumerated sets for additional flexibility

Sometimes, specifying a range for the sum or the length may be too restrictive. One would want instead to specify a list, or iterable L, of acceptable values. This is easy to achieve using a `disjoint union of enumerated sets`. Here we want to accept the values $n = 0, 2, 3$:

```
sage: C = DisjointUnionEnumeratedSets(Family([0,2,3],
....:         lambda n: IntegerListsLex(n, length=2)))
sage: C
Disjoint union of Finite family
{0: Integer lists of sum 0 satisfying certain constraints,
 2: Integer lists of sum 2 satisfying certain constraints,
 3: Integer lists of sum 3 satisfying certain constraints}
sage: C.list()
[[0, 0],
 [2, 0], [1, 1], [0, 2],
 [3, 0], [2, 1], [1, 2], [0, 3]]
```

The price to pay is that the enumeration order is now *graded lexicographic* instead of lexicographic: first choose the value according to the order specified by L, and use lexicographic order within each value. Here is we reverse L:

```
sage: DisjointUnionEnumeratedSets(Family([3,2,0],
....:     lambda n: IntegerListsLex(n, length=2))).list()
[[3, 0], [2, 1], [1, 2], [0, 3],
 [2, 0], [1, 1], [0, 2],
 [0, 0]]
```

Note that if a given value appears several times, the corresponding elements will be enumerated several times, which may, or not, be what one wants:

```
sage: DisjointUnionEnumeratedSets(Family([2,2],
....:     lambda n: IntegerListsLex(n, length=2))).list()
[[2, 0], [1, 1], [0, 2], [2, 0], [1, 1], [0, 2]]
```

Here is a variant where we specify acceptable values for the length:

```
sage: DisjointUnionEnumeratedSets(Family([0,1,3],
....:     lambda l: IntegerListsLex(2, length=l))).list()
[[2],
 [2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]
```

This technique can also be useful to obtain a proper enumeration on infinite sets by using a graded lexicographic enumeration:

```
sage: C = DisjointUnionEnumeratedSets(Family(NN,
....:         lambda n: IntegerListsLex(n, length=2)))
sage: C
Disjoint union of Lazy family (<lambda>(i))_{i in Non negative integer semiring}
sage: it = iter(C)
sage: [next(it) for i in range(10)]
[[0, 0],
 [1, 0], [0, 1],
 [2, 0], [1, 1], [0, 2],
 [3, 0], [2, 1], [1, 2], [0, 3]]
```

Specifying how to construct elements

This is the list of all monomials of degree 4 which divide the monomial $x^3y^1z^2$ (a monomial being identified with its exponent vector):

```
sage: R.<x,y,z> = QQ[]
sage: m = [3,1,2]
sage: def term(exponents):
....:     return x^exponents[0] * y^exponents[1] * z^exponents[2]
sage: list( IntegerListsLex(4, length=len(m), ceiling=m, element_constructor=term) )
[x^3*y, x^3*z, x^2*y*z, x^2*z^2, x*y*z^2]
```

Note the use of the `element_constructor` option to specify how to construct elements from a plain list.

A variant is to specify a class for the elements. With the default element constructor, this class should take as input the parent `self` and a list. Here we want the elements to be constructed in the class `Partition`:

```
sage: IntegerListsLex(3, max_slope=0, element_class=Partition, global_options=Partitions.global_
doctest:...: DeprecationWarning: the global_options argument is
 deprecated since, in general, pickling is broken;
 create your own class instead
See http://trac.sagemath.org/15525 for details.
[[3], [2, 1], [1, 1, 1]]
```

Note that the `Partition` further assumes the existence of an attribute `_global_options` in the parent, hence the use of the `global_options` parameter.

Warning: The protocol for specifying the element class and constructor is subject to changes.

ALGORITHM:

The iteration algorithm uses a depth first search through the prefix tree of the list of integers (see also *Lexicographic generation of lists of integers*). While doing so, it does some lookahead heuristics to attempt to cut dead branches.

In most practical use cases, most dead branches are cut. Then, roughly speaking, the time needed to iterate through all the elements of S is proportional to the number of elements, where the proportion factor is controlled by the length l of the longest element of S. In addition, the memory usage is also controlled by l, which is to say negligible in practice.

Still, there remains much room for efficiency improvements; see trac ticket #18055, trac ticket #18056.

Note: The generation algorithm could in principle be extended to deal with non-constant slope constraints and with negative parts.

TESTS:

This example from the combinatorics tutorial used to fail before trac ticket #17979 because the floor conditions did not satisfy the slope conditions:

```
sage: I = IntegerListsLex(16, min_length=2, max_slope=-1, floor=[5,3,3])
sage: I.list()
[[13, 3], [12, 4], [11, 5], [10, 6], [9, 7], [9, 4, 3], [8, 5, 3], [8, 4, 3, 1],
 [7, 6, 3], [7, 5, 4], [7, 5, 3, 1], [7, 4, 3, 2], [6, 5, 4, 1], [6, 5, 3, 2],
 [6, 4, 3, 2, 1]]

sage: Partitions(2, max_slope=-1, length=2).list()
[]
sage: list(IntegerListsLex(0, floor=ConstantFunction(1), min_slope=0))
```

```
[[]]
sage: list(IntegerListsLex(0, floor=ConstantFunction(1), min_slope=0, max_slope=0))
[[]]
sage: list(IntegerListsLex(0, max_length=0, floor=ConstantFunction(1), min_slope=0, max_slope=0)
[[]]
sage: list(IntegerListsLex(0, max_length=0, floor=ConstantFunction(0), min_slope=0, max_slope=0)
[[]]
sage: list(IntegerListsLex(0, min_part=1, min_slope=0))
[[]]
sage: list(IntegerListsLex(1, min_part=1, min_slope=0))
[[1]]
sage: list(IntegerListsLex(0, min_length=1, min_part=1, min_slope=0))
[]
sage: list(IntegerListsLex(0, min_length=1, min_slope=0))
[[0]]
sage: list(IntegerListsLex(3, max_length=2))
[[3], [2, 1], [1, 2], [0, 3]]
sage: partitions = {"min_part": 1, "max_slope": 0}
sage: partitions_min_2 = {"floor": ConstantFunction(2), "max_slope": 0}
sage: compositions = {"min_part": 1}
sage: integer_vectors = lambda l: {"length": l}
sage: lower_monomials = lambda c: {"length": c, "floor": lambda i: c[i]}
sage: upper_monomials = lambda c: {"length": c, "ceiling": lambda i: c[i]}
sage: constraints = { "min_part":1, "min_slope": -1, "max_slope": 0}
sage: list(IntegerListsLex(6, **partitions))
[[6],
 [5, 1],
 [4, 2],
 [4, 1, 1],
 [3, 3],
 [3, 2, 1],
 [3, 1, 1, 1],
 [2, 2, 2],
 [2, 2, 1, 1],
 [2, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1]]
sage: list(IntegerListsLex(6, **constraints))
[[6],
 [3, 3],
 [3, 2, 1],
 [2, 2, 2],
 [2, 2, 1, 1],
 [2, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1]]
sage: list(IntegerListsLex(1, **partitions_min_2))
[]
sage: list(IntegerListsLex(2, **partitions_min_2))
[[2]]
sage: list(IntegerListsLex(3, **partitions_min_2))
[[3]]
sage: list(IntegerListsLex(4, **partitions_min_2))
[[4], [2, 2]]
sage: list(IntegerListsLex(5, **partitions_min_2))
[[5], [3, 2]]
sage: list(IntegerListsLex(6, **partitions_min_2))
[[6], [4, 2], [3, 3], [2, 2, 2]]
sage: list(IntegerListsLex(7, **partitions_min_2))
[[7], [5, 2], [4, 3], [3, 2, 2]]
```

```
sage: list(IntegerListsLex(9, **partitions_min_2))
[[9], [7, 2], [6, 3], [5, 4], [5, 2, 2], [4, 3, 2], [3, 3, 3], [3, 2, 2, 2]]
sage: list(IntegerListsLex(10, **partitions_min_2))
[[10],
 [8, 2],
 [7, 3],
 [6, 4],
 [6, 2, 2],
 [5, 5],
 [5, 3, 2],
 [4, 4, 2],
 [4, 3, 3],
 [4, 2, 2, 2],
 [3, 3, 2, 2],
 [2, 2, 2, 2, 2]]
sage: list(IntegerListsLex(4, **compositions))
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
sage: list(IntegerListsLex(6, min_length=1, floor=[7]))
[]
sage: L = IntegerListsLex(10**100,length=1)
sage: L.list()
[[1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Noted on trac ticket #17898:

```
sage: list(IntegerListsLex(4, min_part=1, length=3, min_slope=1))
[]
sage: IntegerListsLex(6, ceiling=[4,2], floor=[3,3]).list()
[]
sage: IntegerListsLex(6, min_part=1, max_part=3, max_slope=-4).list()
[]
```

Noted in trac ticket #17548, which are now fixed:

```
sage: IntegerListsLex(10, min_part=2, max_slope=-1).list()
[[10], [8, 2], [7, 3], [6, 4], [5, 3, 2]]
sage: IntegerListsLex(5, min_slope=1, floor=[2,1,1], max_part=2).list()
[]
sage: IntegerListsLex(4, min_slope=0, max_slope=0).list()
[[4], [2, 2], [1, 1, 1, 1]]
sage: IntegerListsLex(6, min_slope=-1, max_slope=-1).list()
[[6], [3, 2, 1]]
sage: IntegerListsLex(6, min_length=3, max_length=2, min_part=1).list()
[]
sage: I = IntegerListsLex(3, max_length=2, min_part=1)
sage: I.list()
[[3], [2, 1], [1, 2]]
sage: [1,1,1] in I
False
sage: I=IntegerListsLex(10, ceiling=[4], max_length=1, min_part=1)
sage: I.list()
[]
sage: [4,6] in I
False
sage: I = IntegerListsLex(4, min_slope=1, min_part=1, max_part=2)
sage: I.list()
[]
sage: I = IntegerListsLex(7, min_slope=1, min_part=1, max_part=4)
sage: I.list()
```

```
[[3, 4], [1, 2, 4]]
sage: I = IntegerListsLex(4, floor=[2,1], ceiling=[2,2], max_length=2, min_slope=0)
sage: I.list()
[[2, 2]]
sage: I = IntegerListsLex(10, min_part=1, max_slope=-1)
sage: I.list()
[[10], [9, 1], [8, 2], [7, 3], [7, 2, 1], [6, 4], [6, 3, 1], [5, 4, 1],
 [5, 3, 2], [4, 3, 2, 1]]
```

TESTS from comments on trac ticket #17979

Comment 191:

```
sage: list(IntegerListsLex(1, min_length=2, min_slope=0, max_slope=0))
[]
```

Comment 240:

```
sage: L = IntegerListsLex(min_length=2, max_part=0)
sage: L.list()
[[0, 0]]
```

Tests on the element constructor feature and mutability

Internally, the iterator works on a single list that is mutated along the way. Therefore, you need to make sure that the `element_constructor` actually **copies** its input. This example shows what can go wrong:

```
sage: P = IntegerListsLex(n=3, max_slope=0, min_part=1, element_constructor=lambda x: x)
sage: list(P)
[[], [], []]
```

However, specifying `list()` as constructor solves this problem:

```
sage: P = IntegerListsLex(n=3, max_slope=0, min_part=1, element_constructor=list)
sage: list(P)
[[3], [2, 1], [1, 1, 1]]
```

Same, step by step:

```
sage: it = iter(P)
sage: a = next(it); a
[3]
sage: b = next(it); b
[2, 1]
sage: a
[3]
sage: a is b
False
```

Tests from MuPAD-Combinat:

```
sage: IntegerListsLex(7, min_length=2, max_length=6, floor=[0,0,2,0,0,1], ceiling=[3,2,3,2,1,2])
83
sage: IntegerListsLex(7, min_length=2, max_length=6, floor=[0,0,2,0,1,1], ceiling=[3,2,3,2,1,2])
53
sage: IntegerListsLex(5, min_length=2, max_length=6, floor=[0,0,2,0,0,0], ceiling=[2,2,2,2,2,2])
30
```

```
sage: IntegerListsLex(5, min_length=2, max_length=6, floor=[0,0,1,1,0,0], ceiling=[2,2,2,2,2,2])
43

sage: IntegerListsLex(0, min_length=0, max_length=7, floor=[1,1,0,0,1,0], ceiling=[4,3,2,3,2,2,1
[]

sage: IntegerListsLex(0, min_length=1, max_length=7, floor=[0,1,0,0,1,0], ceiling=[4,3,2,3,2,2,1
[0]
sage: IntegerListsLex(0, min_length=1, max_length=7, floor=[1,1,0,0,1,0], ceiling=[4,3,2,3,2,2,1
0

sage: IntegerListsLex(2, min_length=0, max_length=7, floor=[1,1,0,0,0,0], ceiling=[4,3,2,3,2,2,1
[2]
sage: IntegerListsLex(1, min_length=1, max_length=7, floor=[1,1,0,0,0,0], ceiling=[4,3,2,3,2,2,1
[1]
sage: IntegerListsLex(1, min_length=2, max_length=7, floor=[1,1,0,0,0,0], ceiling=[4,3,2,3,2,2,1
0
sage: IntegerListsLex(2, min_length=5, max_length=7, floor=[1,1,0,0,0,0], ceiling=[4,3,2,3,2,2,1
[1, 1, 0, 0, 0]
sage: IntegerListsLex(2, min_length=5, max_length=7, floor=[1,1,0,0,0,1], ceiling=[4,3,2,3,2,2,1
[1, 1, 0, 0, 0]
sage: IntegerListsLex(2, min_length=5, max_length=7, floor=[1,1,0,0,1,0], ceiling=[4,3,2,3,2,2,1
0

sage: IntegerListsLex(4, min_length=3, max_length=6, floor=[2, 1, 2, 1, 1, 1], ceiling=[3, 1, 2,
0
sage: IntegerListsLex(5, min_length=3, max_length=6, floor=[2, 1, 2, 1, 1, 1], ceiling=[3, 1, 2,
[2, 1, 2]
sage: IntegerListsLex(6, min_length=3, max_length=6, floor=[2, 1, 2, 1, 1, 1], ceiling=[3, 1, 2,
[3, 1, 2]
sage: IntegerListsLex(12, min_length=3, max_length=6, floor=[2, 1, 2, 1, 1, 1], ceiling=[3, 1, 2
[3, 1, 2, 3, 2, 1]
sage: IntegerListsLex(13, min_length=3, max_length=6, floor=[2, 1, 2, 1, 1, 1], ceiling=[3, 1, 2
[3, 1, 2, 3, 2, 2]
sage: IntegerListsLex(14, min_length=3, max_length=6, floor=[2, 1, 2, 1, 1, 1], ceiling=[3, 1, 2
0
```

This used to hang (see comment 389 and fix in `Envelope.__init__()`):

```
sage: IntegerListsLex(7, max_part=0, ceiling=lambda i:i, check=False).list()
[]
```

backend_class
alias of `IntegerListsBackend_invlex`

class `sage.combinat.integer_lists.invlex.`**IntegerListsLexIter**(*backend*)
Bases: `object`

Iterator class for IntegerListsLex.

Let `T` be the prefix tree of all lists of nonnegative integers that satisfy all constraints except possibly for `min_length` and `min_sum`; let the children of a list be sorted decreasingly according to their last part.

The iterator is based on a depth-first search exploration of a subtree of this tree, trying to cut branches that do not contain a valid list. Each call of `next` iterates through the nodes of this tree until it finds a valid list to return.

Here are the attributes describing the current state of the iterator, and their invariants:

- `backend` – the `IntegerListsBackend` object this is iterating on;
- `_current_list` – the list corresponding to the current node of the tree;

- _j – the index of the last element of _current_list: self._j == len(self._current_list) - 1;
- _current_sum – the sum of the parts of _current_list;
- _search_ranges – a list of same length as _current_list: the range for each part.

Furthermore, we assume that there is no obvious contradiction in the contraints:

- self.backend.min_length <= self.backend.max_length;
- self.backend.min_slope <= self.backend.max_slope unless self.backend.min_length <= 1.

Along this iteration, next switches between the following states:

- LOOKAHEAD: determine whether the current list could be a prefix of a valid list;
- PUSH: go deeper into the prefix tree by appending the largest possible part to the current list;
- ME: check whether the current list is valid and if yes return it
- DECREASE: decrease the last part;
- POP: pop the last part of the current list;
- STOP: the iteration is finished.

The attribute _next_state contains the next state next should enter in.

next()

Return the next element in the iteration.

EXAMPLES:

```
sage: from sage.combinat.integer_lists.invlex import IntegerListsLexIter
sage: C = IntegerListsLex(2, length=3)
sage: I = IntegerListsLexIter(C)
sage: next(I)
[2, 0, 0]
sage: next(I)
[1, 1, 0]
```

5.1.108 Counting, generating, and manipulating non-negative integer matrices

Counting, generating, and manipulating non-negative integer matrices with prescribed row sums and column sums.

AUTHORS:

- Franco Saliola

class sage.combinat.integer_matrices.**IntegerMatrices**(*row_sums*, *column_sums*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The class of non-negative integer matrices with prescribed row sums and column sums.

An *integer matrix* m with column sums $c := (c_1, ..., c_k)$ and row sums $l := (l_1, ..., l_n)$ where $c_1 + ... + c_k$ is equal to $l_1 + ... + l_n$, is a $n \times k$ matrix $m = (m_{i,j})$ such that $m_{1,j} + \cdots + m_{n,j} = c_j$, for all j and $m_{i,1} + \cdots + m_{i,k} = l_i$, for all i.

EXAMPLES:

There are 6 integer matrices with row sums $[3, 2, 2]$ and column sums $[2, 5]$:

```
sage: from sage.combinat.integer_matrices import IntegerMatrices
sage: IM = IntegerMatrices([3,2,2], [2,5]); IM
Non-negative integer matrices with row sums [3, 2, 2] and column sums [2, 5]
sage: IM.list()
[
[2 1]  [1 2]  [1 2]  [0 3]  [0 3]  [0 3]
[0 2]  [1 1]  [0 2]  [2 0]  [1 1]  [0 2]
[0 2], [0 2], [1 1], [0 2], [1 1], [2 0]
]
sage: IM.cardinality()
6
```

cardinality()

The number of integer matrices with the prescribed row sums and columns sums.

EXAMPLES:

```
sage: from sage.combinat.integer_matrices import IntegerMatrices
sage: IntegerMatrices([2,5], [3,2,2]).cardinality()
6
sage: IntegerMatrices([1,1,1,1,1], [1,1,1,1,1]).cardinality()
120
sage: IntegerMatrices([2,2,2,2], [2,2,2,2]).cardinality()
282
sage: IntegerMatrices([4], [3]).cardinality()
0
sage: len(IntegerMatrices([0,0], [0]).list())
1
```

This method computes the cardinality using symmetric functions. Below are the same examples, but computed by generating the actual matrices:

```
sage: from sage.combinat.integer_matrices import IntegerMatrices
sage: len(IntegerMatrices([2,5], [3,2,2]).list())
6
sage: len(IntegerMatrices([1,1,1,1,1], [1,1,1,1,1]).list())
120
sage: len(IntegerMatrices([2,2,2,2], [2,2,2,2]).list())
282
sage: len(IntegerMatrices([4], [3]).list())
0
sage: len(IntegerMatrices([0], [0]).list())
1
```

column_sums()

The column sums of the integer matrices in `self`.

OUTPUT:

- Composition

EXAMPLES:

```
sage: from sage.combinat.integer_matrices import IntegerMatrices
sage: IM = IntegerMatrices([3,2,2], [2,5])
sage: IM.column_sums()
[2, 5]
```

row_sums()

The row sums of the integer matrices in `self`.

OUTPUT:

- Composition

EXAMPLES:

```
sage: from sage.combinat.integer_matrices import IntegerMatrices
sage: IM = IntegerMatrices([3,2,2], [2,5])
sage: IM.row_sums()
[3, 2, 2]
```

to_composition(*x*)

The composition corresponding to the integer matrix.

This is the composition obtained by reading the entries of the matrix from left to right along each row, and reading the rows from top to bottom, ignore zeros.

INPUT:

- x – matrix

EXAMPLES:

```
sage: from sage.combinat.integer_matrices import IntegerMatrices
sage: IM = IntegerMatrices([3,2,2], [2,5]); IM
Non-negative integer matrices with row sums [3, 2, 2] and column sums [2, 5]
sage: IM.list()
[
[2 1]  [1 2]  [1 2]  [0 3]  [0 3]  [0 3]
[0 2]  [1 1]  [0 2]  [2 0]  [1 1]  [0 2]
[0 2], [0 2], [1 1], [0 2], [1 1], [2 0]
]
sage: for m in IM: print IM.to_composition(m)
[2, 1, 2, 2]
[1, 2, 1, 1, 2]
[1, 2, 2, 1, 1]
[3, 2, 2]
[3, 1, 1, 1, 1]
[3, 2, 2]
```

sage.combinat.integer_matrices.**integer_matrices_generator**(*row_sums*, *column_sums*)

Recursively generate the integer matrices with the prescribed row sums and column sums.

INPUT:

- row_sums – list or tuple
- column_sums – list or tuple

OUTPUT:

- an iterator producing a list of lists

EXAMPLES:

```
sage: from sage.combinat.integer_matrices import integer_matrices_generator
sage: iter = integer_matrices_generator([3,2,2], [2,5]); iter
<generator object integer_matrices_generator at ...>
sage: for m in iter: print m
[[2, 1], [0, 2], [0, 2]]
[[1, 2], [1, 1], [0, 2]]
[[1, 2], [0, 2], [1, 1]]
[[0, 3], [2, 0], [0, 2]]
```

```
[[0, 3], [1, 1], [1, 1]]
[[0, 3], [0, 2], [2, 0]]
```

5.1.109 (Non-negative) Integer vectors

AUTHORS:

- Mike Hansen (2007) - original module
- Nathann Cohen, David Joyner (2009-2010) - Gale-Ryser stuff
- Nathann Cohen, David Joyner (2011) - Gale-Ryser bugfix
- Travis Scrimshaw (2012-05-12) - Updated doc-strings to tell the user of that the class's name is a misnomer (that they only contains non-negative entries).
- Federico Poloni (2013) - specialized rank()

sage.combinat.integer_vector.**IntegerVectors**(*n=None*, *k=None*, ***kwargs*)

Returns the combinatorial class of (non-negative) integer vectors.

INPUT:

- n – if set to an integer, returns the combinatorial class of integer vectors whose sum is n. If set to `None` (default), no such constraint is defined.
- k – the length of the vectors. Set to `None` (default) if you do not want such a constraint.

All other arguments given to this function are forwarded to the instance of `IntegerVectors_all` `IntegerVectors_nconstraints` `IntegerVectors_nk` or `IntegerVectors_nkconstraints` that it returns.

NOTE - These integer vectors are non-negative.

EXAMPLES: If n is not specified, it returns the class of all integer vectors.

```
sage: IntegerVectors()
Integer vectors
sage: [] in IntegerVectors()
True
sage: [1,2,1] in IntegerVectors()
True
sage: [1, 0, 0] in IntegerVectors()
True
```

Entries are non-negative.

```
sage: [-1, 2] in IntegerVectors()
False
```

If n is specified, then it returns the class of all integer vectors which sum to n.

```
sage: IV3 = IntegerVectors(3); IV3
Integer vectors that sum to 3
```

Note that trailing zeros are ignored so that [3, 0] does not show up in the following list (since [3] does)

```
sage: IntegerVectors(3, max_length=2).list()
[[3], [2, 1], [1, 2], [0, 3]]
```

If n and k are both specified, then it returns the class of integer vectors that sum to n and are of length k.

```
sage: IV53 = IntegerVectors(5,3); IV53
Integer vectors of length 3 that sum to 5
sage: IV53.cardinality()
21
sage: IV53.first()
[5, 0, 0]
sage: IV53.last()
[0, 0, 5]
sage: IV53.random_element()
[4, 0, 1]
```

Further examples:

```
sage: IntegerVectors(-1, 0, min_part = 1).list()
[]
sage: IntegerVectors(-1, 2, min_part = 1).list()
[]
sage: IntegerVectors(0, 0, min_part=1).list()
[[]]
sage: IntegerVectors(3, 0, min_part=1).list()
[]
sage: IntegerVectors(0, 1, min_part=1).list()
[]
sage: IntegerVectors(2, 2, min_part=1).list()
[[1, 1]]
sage: IntegerVectors(2, 3, min_part=1).list()
[]
sage: IntegerVectors(4, 2, min_part=1).list()
[[3, 1], [2, 2], [1, 3]]

sage: IntegerVectors(0, 3, outer=[0,0,0]).list()
[[0, 0, 0]]
sage: IntegerVectors(1, 3, outer=[0,0,0]).list()
[]
sage: IntegerVectors(2, 3, outer=[0,2,0]).list()
[[0, 2, 0]]
sage: IntegerVectors(2, 3, outer=[1,2,1]).list()
[[1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1]]
sage: IntegerVectors(2, 3, outer=[1,1,1]).list()
[[1, 1, 0], [1, 0, 1], [0, 1, 1]]
sage: IntegerVectors(2, 5, outer=[1,1,1,1,1]).list()
[[1, 1, 0, 0, 0],
 [1, 0, 1, 0, 0],
 [1, 0, 0, 1, 0],
 [1, 0, 0, 0, 1],
 [0, 1, 1, 0, 0],
 [0, 1, 0, 1, 0],
 [0, 1, 0, 0, 1],
 [0, 0, 1, 1, 0],
 [0, 0, 1, 0, 1],
 [0, 0, 0, 1, 1]]

sage: iv = [ IntegerVectors(n,k) for n in range(-2, 7) for k in range(7) ]
sage: all(map(lambda x: x.cardinality() == len(x.list()), iv))
True
sage: essai = [[1,1,1], [2,5,6], [6,5,2]]
sage: iv = [ IntegerVectors(x[0], x[1], max_part = x[2]-1) for x in essai ]
sage: all(map(lambda x: x.cardinality() == len(x.list()), iv))
True
```

TESTS:

trac ticket #17927:

```
sage: IntegerVectors(None, length=3)
Traceback (most recent call last):
...
TypeError: __init__() got an unexpected keyword argument 'length'
sage: IntegerVectors(None, 4)
Traceback (most recent call last):
...
NotImplementedError: k must be None when n is None
```

class sage.combinat.integer_vector.**IntegerVectors_all**(*category=None*)

Bases: sage.combinat.combinat.CombinatorialClass

TESTS:

```
sage: C = sage.combinat.combinat.CombinatorialClass()
sage: C.category()
Category of enumerated sets
sage: C.__class__
<class 'sage.combinat.combinat.CombinatorialClass_with_category'>
sage: isinstance(C, Parent)
True
sage: C = sage.combinat.combinat.CombinatorialClass(category = FiniteEnumeratedSets())
sage: C.category()
Category of finite enumerated sets
```

cardinality()

EXAMPLES:

```
sage: IntegerVectors().cardinality()
+Infinity
```

list()

EXAMPLES:

```
sage: IntegerVectors().list()
Traceback (most recent call last):
...
NotImplementedError: infinite list
```

class sage.combinat.integer_vector.**IntegerVectors_nconstraints**(*n*, *constraints*)

Bases: sage.combinat.integer_vector.IntegerVectors_nkconstraints

TESTS:

```
sage: IV = IntegerVectors(3, max_length=2)
sage: IV == loads(dumps(IV))
True

sage: IntegerVectors(3, max_length=2).cardinality()
4
sage: IntegerVectors(3).cardinality()
+Infinity

sage: IntegerVectors(3, max_length=2).list()
[[3], [2, 1], [1, 2], [0, 3]]
```

```
sage: IntegerVectors(3).list()
Traceback (most recent call last):
...
NotImplementedError: infinite list
```

class sage.combinat.integer_vector.**IntegerVectors_nk**(*n*, *k*)

Bases: sage.combinat.combinat.CombinatorialClass

TESTS:

```
sage: IV = IntegerVectors(2,3)
sage: IV == loads(dumps(IV))
True
```

AUTHORS:

- Martin Albrecht
- Mike Hansen

list()

EXAMPLE:

```
sage: IV = IntegerVectors(2,3)
sage: IV.list()
[[2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]
sage: IntegerVectors(3, 0).list()
[]
sage: IntegerVectors(3, 1).list()
[[3]]
sage: IntegerVectors(0, 1).list()
[[0]]
sage: IntegerVectors(0, 2).list()
[[0, 0]]
sage: IntegerVectors(2, 2).list()
[[2, 0], [1, 1], [0, 2]]
```

rank(*x*)

Returns the position of a given element.

INPUT:

- `x` - a list with `sum(x) == n` and `len(x) == k`

TESTS:

```
sage: IV = IntegerVectors(4,5)
sage: range(IV.cardinality()) == [IV.rank(x) for x in IV]
True
```

class sage.combinat.integer_vector.**IntegerVectors_nkconstraints**(*n*, *k*, *constraints*, *category=None*)

Bases: sage.combinat.integer_lists.invlex.IntegerListsLex

EXAMPLES:

```
sage: IV = IntegerVectors(2,3,min_slope=0)
sage: IV == loads(dumps(IV))
True

sage: v = IntegerVectors(2,3,min_slope=0).first(); v
[0, 1, 1]
```

```
sage: type(v)
<type 'list'>
```

TESTS:

```
sage: IV.min_length
3
sage: IV.max_length
3
sage: floor = IV.floor
sage: [floor(i) for i in range(1,10)]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
sage: ceiling = IV.ceiling
sage: [ceiling(i) for i in range(1,5)]
[inf, inf, inf, inf]
sage: IV.min_slope
0
sage: IV.max_slope
inf

sage: IV = IntegerVectors(3, 10, inner=[4,1,3], min_part=2)
sage: floor = IV.floor
sage: floor(0), floor(1), floor(2)
(4, 2, 3)

sage: IV = IntegerVectors(3, 10, outer=[4,1,3], max_part=3)
sage: ceiling = IV.ceiling
sage: ceiling(0), ceiling(1), ceiling(2)
(3, 1, 3)
```

cardinality()

EXAMPLES:

```
sage: IntegerVectors(3,3, min_part=1).cardinality()
1
sage: IntegerVectors(5,3, min_part=1).cardinality()
6
sage: IntegerVectors(13, 4, min_part=2, max_part=4).cardinality()
16
```

next(*x*)

EXAMPLES:

```
sage: a = IntegerVectors(2,3,min_slope=0).first()
sage: IntegerVectors(2,3,min_slope=0).next(a)
[0, 0, 2]
```

class sage.combinat.integer_vector.**IntegerVectors_nnondescents**(*n*, *comp*)

Bases: sage.combinat.combinat.CombinatorialClass

The combinatorial class of integer vectors v graded by two parameters:

- n: the sum of the parts of v
- comp: the non descents composition of v

In other words: the length of v equals c[1]+...+c[k], and v is decreasing in the consecutive blocs of length c[1], ..., c[k]

Those are the integer vectors of sum n which are lexicographically maximal (for the natural left->right reading)

in their orbit by the young subgroup S_c_1 x x S_c_k. In particular, they form a set of orbit representative of integer vectors w.r.t. this young subgroup.

sage.combinat.integer_vector.**constant_func**(*i*)

Returns the constant function i.

EXAMPLES:

```
sage: f = sage.combinat.integer_vector.constant_func(3)
sage: f(-1)
3
sage: f('asf')
3
```

sage.combinat.integer_vector.**gale_ryser_theorem**(*p1*, *p2*, *algorithm='gale'*)

Returns the binary matrix given by the Gale-Ryser theorem.

The Gale Ryser theorem asserts that if p_1, p_2 are two partitions of n of respective lengths k_1, k_2, then there is a binary $k_1 \times k_2$ matrix M such that p_1 is the vector of row sums and p_2 is the vector of column sums of M, if and only if the conjugate of p_2 dominates p_1.

INPUT:

- `p1, p2` – list of integers representing the vectors of row/column sums
- `algorithm` – two possible string values :
  - `"ryser"` implements the construction due to Ryser [Ryser63].
  - `"gale"` (default) implements the construction due to Gale [Gale57].

OUTPUT:

- A binary matrix if it exists, `None` otherwise.

Gale's Algorithm:

(Gale [Gale57]): A matrix satisfying the constraints of its sums can be defined as the solution of the following Linear Program, which Sage knows how to solve.

$$\forall i \sum_{j=1}^{k_2} b_{i,j} = p_{1,j}$$

$$\forall i \sum_{j=1}^{k_1} b_{j,i} = p_{2,j}$$

$b_{i,j}$ is a binary variable

Ryser's Algorithm:

(Ryser [Ryser63]): The construction of an $m \times n$ matrix $A = A_{r,s}$, due to Ryser, is described as follows. The construction works if and only if have $s \preceq r^*$.

- Construct the $m \times n$ matrix B from r by defining the i-th row of B to be the vector whose first r_i entries are 1, and the remainder are 0's, $1 \leq i \leq m$. This maximal matrix B with row sum r and ones left justified has column sum r^*.
- Shift the last 1 in certain rows of B to column n in order to achieve the sum s_n. Call this B again.
  - The 1's in column n are to appear in those rows in which A has the largest row sums, giving preference to the bottom-most positions in case of ties.
  - Note: When this step automatically "fixes" other columns, one must skip ahead to the first column index with a wrong sum in the step below.

•Proceed inductively to construct columns $n-1$, ..., 2, 1. Note: when performing the induction on step k, we consider the row sums of the first k columns.

•Set $A = B$. Return A.

EXAMPLES:

Computing the matrix for $p_1 = p_2 = 2 + 2 + 1$

```
sage: from sage.combinat.integer_vector import gale_ryser_theorem
sage: p1 = [2,2,1]
sage: p2 = [2,2,1]
sage: print gale_ryser_theorem(p1, p2)      # not tested
[1 1 0]
[1 0 1]
[0 1 0]
sage: A = gale_ryser_theorem(p1, p2)
sage: rs = [sum(x) for x in A.rows()]
sage: cs = [sum(x) for x in A.columns()]
sage: p1 == rs; p2 == cs
True
True
```

Or for a non-square matrix with $p_1 = 3 + 3 + 2 + 1$ and $p_2 = 3 + 2 + 2 + 1 + 1$, using Ryser's algorithm

```
sage: from sage.combinat.integer_vector import gale_ryser_theorem
sage: p1 = [3,3,1,1]
sage: p2 = [3,3,1,1]
sage: gale_ryser_theorem(p1, p2, algorithm = "ryser")
[1 1 1 0]
[1 1 0 1]
[1 0 0 0]
[0 1 0 0]
sage: p1 = [4,2,2]
sage: p2 = [3,3,1,1]
sage: gale_ryser_theorem(p1, p2, algorithm = "ryser")
[1 1 1 1]
[1 1 0 0]
[1 1 0 0]
sage: p1 = [4,2,2,0]
sage: p2 = [3,3,1,1,0,0]
sage: gale_ryser_theorem(p1, p2, algorithm = "ryser")
[1 1 1 1 0 0]
[1 1 0 0 0 0]
[1 1 0 0 0 0]
[0 0 0 0 0 0]
sage: p1 = [3,3,2,1]
sage: p2 = [3,2,2,1,1]
sage: print gale_ryser_theorem(p1, p2, algorithm="gale")  # not tested
[1 1 1 0 0]
[1 1 0 0 1]
[1 0 1 0 0]
[0 0 0 1 0]
```

With 0 in the sequences, and with unordered inputs

```
sage: from sage.combinat.integer_vector import gale_ryser_theorem
sage: gale_ryser_theorem([3,3,0,1,1,0], [3,1,3,1,0], algorithm = "ryser")
[1 1 1 0 0]
[1 0 1 1 0]
[0 0 0 0 0]
```

```
[1 0 0 0 0]
[0 0 1 0 0]
[0 0 0 0 0]
sage: p1 = [3,1,1,1,1]; p2 = [3,2,2,0]
sage: gale_ryser_theorem(p1, p2, algorithm = "ryser")
[1 1 1 0]
[1 0 0 0]
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
```

TESTS:

This test created a random bipartite graph on $n+m$ vertices. Its adjacency matrix is binary, and it is used to create some "random-looking" sequences which correspond to an existing matrix. The `gale_ryser_theorem` is then called on these sequences, and the output checked for correctness.:

```
sage: def test_algorithm(algorithm, low = 10, high = 50):
....:     n,m = randint(low,high), randint(low,high)
....:     g = graphs.RandomBipartite(n, m, .3)
....:     s1 = sorted(g.degree([(0,i) for i in range(n)]), reverse = True)
....:     s2 = sorted(g.degree([(1,i) for i in range(m)]), reverse = True)
....:     m = gale_ryser_theorem(s1, s2, algorithm = algorithm)
....:     ss1 = sorted(map(lambda x : sum(x) , m.rows()), reverse = True)
....:     ss2 = sorted(map(lambda x : sum(x) , m.columns()), reverse = True)
....:     if ((ss1 != s1) or (ss2 != s2)):
....:         print "Algorithm %s failed with this input:" % algorithm
....:         print s1, s2

sage: for algorithm in ["gale", "ryser"]:                          # long time
....:     for i in range(50):                                      # long time
....:         test_algorithm(algorithm, 3, 10)                     # long time
```

Null matrix:

```
sage: gale_ryser_theorem([0,0,0],[0,0,0,0], algorithm="gale")
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
sage: gale_ryser_theorem([0,0,0],[0,0,0,0], algorithm="ryser")
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
```

Check that trac ticket #16638 is fixed:

```
sage: tests = [([4, 3, 3, 2, 1, 1, 1, 1, 0], [6, 5, 1, 1, 1, 1, 1]),
....:          ([4, 4, 3, 3, 1, 1, 0], [5, 5, 2, 2, 1, 1]),
....:          ([4, 4, 3, 2, 1, 1], [5, 5, 1, 1, 1, 1, 1, 0, 0]),
....:          ([3, 3, 3, 3, 2, 1, 1, 1, 0], [7, 6, 2, 1, 1, 0]),
....:          ([3, 3, 3, 1, 1, 0], [4, 4, 1, 1, 1])]
sage: for s1, s2 in tests:
....:     m = gale_ryser_theorem(s1, s2, algorithm="ryser")
....:     ss1 = sorted(map(lambda x : sum(x) , m.rows()), reverse = True)
....:     ss2 = sorted(map(lambda x : sum(x) , m.columns()), reverse = True)
....:     if ((ss1 != s1) or (ss2 != s2)):
....:         print("Error in Ryser algorithm")
....:         print(s1, s2)
```

REFERENCES:

sage.combinat.integer_vector.**is_gale_ryser**(*r*, *s*)

Tests whether the given sequences satisfy the condition of the Gale-Ryser theorem.

Given a binary matrix B of dimension $n \times m$, the vector of row sums is defined as the vector whose i^{th} component is equal to the sum of the i^{th} row in A. The vector of column sums is defined similarly.

If, given a binary matrix, these two vectors are easy to compute, the Gale-Ryser theorem lets us decide whether, given two non-negative vectors r, s, there exists a binary matrix whose row/colum sums vectors are r and s.

This functions answers accordingly.

INPUT:

- `r`, `s` – lists of non-negative integers.

ALGORITHM:

Without loss of generality, we can assume that:

- The two given sequences do not contain any 0 (which would correspond to an empty column/row)
- The two given sequences are ordered in decreasing order (reordering the sequence of row (resp. column) sums amounts to reordering the rows (resp. columns) themselves in the matrix, which does not alter the columns (resp. rows) sums.

We can then assume that r and s are partitions (see the corresponding class `Partition`)

If r^* denote the conjugate of r, the Gale-Ryser theorem asserts that a binary Matrix satisfying the constraints exists if and only if $s \preceq r^*$, where $\preceq$ denotes the domination order on partitions.

EXAMPLES:

```
sage: from sage.combinat.integer_vector import is_gale_ryser
sage: is_gale_ryser([4,2,2],[3,3,1,1])
True
sage: is_gale_ryser([4,2,1,1],[3,3,1,1])
True
sage: is_gale_ryser([3,2,1,1],[3,3,1,1])
False
```

REMARK: In the literature, what we are calling a Gale-Ryser sequence sometimes goes by the (rather generic-sounding) term ‘’realizable sequence’‘.

sage.combinat.integer_vector.**list2func**(*l*, *default=None*)

Given a list `l`, return a function that takes in a value i and return $l[i]$. If default is not None, then the function will return the default value for out of range i‘s.

EXAMPLES:

```
sage: f = sage.combinat.integer_vector.list2func([1,2,3])
sage: f(0)
1
sage: f(1)
2
sage: f(2)
3
sage: f(3)
Traceback (most recent call last):
...
IndexError: list index out of range
```

```
sage: f = sage.combinat.integer_vector.list2func([1,2,3], 0)
sage: f(2)
3
sage: f(3)
0
```

5.1.110 Weighted Integer Vectors

AUTHORS:

- Mike Hansen (2007): initial version, ported from MuPAD-Combinat
- Nicolas M. Thiery (2010-10-30): WeightedIntegerVectors(weights) + cleanup

Warning: The list(self) function in this file used the `Permutation` class improperly, returning a list of, generally speaking, invalid permutations (repeated entries, including 0).

`sage.combinat.integer_vector_weighted.`**`WeightedIntegerVectors`**(*n=None*, *weight=None*)

Returns the combinatorial class of integer vectors of `n` weighted by `weight`, that is, the nonnegative integer vectors $(v_1, \ldots, v_{length(weight)})$ satisfying $\sum_i v_i weight[i] == n$.

INPUT:

- `n` – a non negative integer (optional)
- `weight` – a tuple (or list or iterable) of positive integers

EXAMPLES:

```
sage: WeightedIntegerVectors(8, [1,1,2])
Integer vectors of 8 weighted by [1, 1, 2]
sage: WeightedIntegerVectors(8, [1,1,2]).first()
[0, 0, 4]
sage: WeightedIntegerVectors(8, [1,1,2]).last()
[8, 0, 0]
sage: WeightedIntegerVectors(8, [1,1,2]).cardinality()
25
sage: WeightedIntegerVectors(8, [1,1,2]).random_element()
[1, 1, 3]

sage: WeightedIntegerVectors([1,1,2])
Integer vectors weighted by [1, 1, 2]
sage: WeightedIntegerVectors([1,1,2]).cardinality()
+Infinity
sage: WeightedIntegerVectors([1,1,2]).first()
[0, 0, 0]
```

TESTS:

```
sage: WeightedIntegerVectors(None,None)
Traceback (most recent call last):
...
ValueError: weights should be specified
```

Todo

should the order of the arguments `n` and `weight` be exchanged to simplify the logic ?

class sage.combinat.integer_vector_weighted.**WeightedIntegerVectors_all**(*weights*)
Bases: sage.sets.disjoint_union_enumerated_sets.DisjointUnionEnumeratedSets

Set of weighted integer vectors.

EXAMPLES:

```
sage: W = WeightedIntegerVectors([3,1,1,2,1]); W
Integer vectors weighted by [3, 1, 1, 2, 1]
sage: W.cardinality()
+Infinity

sage: W12 = W.graded_component(12)
sage: W12.an_element()
[4, 0, 0, 0, 0]
sage: W12.last()
[0, 12, 0, 0, 0]
sage: W12.cardinality()
441
sage: for w in W12: print w
[4, 0, 0, 0, 0]
[3, 0, 0, 1, 1]
[3, 0, 1, 1, 0]
...
[0, 11, 1, 0, 0]
[0, 12, 0, 0, 0]
```

grading(*x*)

EXAMPLES:

```
sage: C = WeightedIntegerVectors([2,1,3])
sage: C.grading((2,1,1))
8
```

subset(*size=None*)

EXAMPLES:

```
sage: C = WeightedIntegerVectors([2,1,3])
sage: C.subset(4)
Integer vectors of 4 weighted by [2, 1, 3]
```

class sage.combinat.integer_vector_weighted.**WeightedIntegerVectors_nweight**(*n*, *weight*)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

TESTS:

```
sage: C = WeightedIntegerVectors(8, [1,1,2])
sage: C.__class__
<class 'sage.combinat.integer_vector_weighted.WeightedIntegerVectors_nweight_with_category'>
sage: TestSuite(C).run()
```

5.1.111 Integer vectors modulo the action of a permutation group

class sage.combinat.integer_vectors_mod_permgroup.**IntegerVectorsModPermutationGroup**
Bases: sage.structure.unique_representation.UniqueRepresentation

Returns an enumerated set containing integer vectors which are maximal in their orbit under the action of the permutation group G for the lexicographic order.

In Sage, a permutation group G is viewed as a subgroup of the symmetric group S_n of degree n and n is said to be the degree of G. Any integer vector v is said to be canonical if it is maximal in its orbit under the action of G. v is canonical if and only if

$$v = \max_{\text{lex order}} \{g \cdot v | g \in G\}$$

The action of G is on position. This means for example that the simple transposition $s_1 = (1, 2)$ swaps the first and the second entries of any integer vector $v = [a_1, a_2, a_3, \ldots, a_n]$

$$s_1 \cdot v = [a_2, a_1, a_3, \ldots, a_n]$$

This functions returns a parent which contains a single integer vector by orbit under the action of the permutation group G. The approach chosen here is to keep the maximal integer vector for the lexicographic order in each orbit. Such maximal vector will be called canonical integer vector under the action of the permutation group G.

INPUT:

- G - a permutation group
- sum - (default: None) - a nonnegative integer
- max_part - (default: None) - a nonnegative integer setting the maximum of entries of elements
- sgs - (default: None) - a strong generating system of the group G. If you do not provide it, it will be calculated at the creation of the parent

OUTPUT:

- If sum and max_part are None, it returns the infinite enumerated set of all integer vectors (list of integers) maximal in their orbit for the lexicographic order.
- If sum is an integer, it returns a finite enumerated set containing all integer vectors maximal in their orbit for the lexicographic order and whose entries sum to sum.

EXAMPLES:

Here is the set enumerating integer vectors modulo the action of the cyclic group of 3 elements:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3)]]))
sage: I.category()
Join of Category of infinite enumerated sets and Category of quotients of sets
sage: I.cardinality()
+Infinity
sage: I.list()
Traceback (most recent call last):
...
NotImplementedError: infinite list
sage: p = iter(I)
sage: for i in range(10): next(p)
[0, 0, 0]
[1, 0, 0]
[2, 0, 0]
[1, 1, 0]
[3, 0, 0]
[2, 1, 0]
[2, 0, 1]
[1, 1, 1]
[4, 0, 0]
[3, 1, 0]
```

The method `is_canonical()` tests if any integer vector is maximal in its orbit. This method is also used in the containment test:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: I.is_canonical([5,2,0,4])
True
sage: I.is_canonical([5,0,6,4])
False
sage: I.is_canonical([1,1,1,1])
True
sage: [2,3,1,0] in I
False
sage: [5,0,5,0] in I
True
sage: 'Bla' in I
False
sage: I.is_canonical('bla')
Traceback (most recent call last):
...
AssertionError: bla should be a list or a integer vector
```

If you give a value to the extra argument `sum`, the set returned will be a finite set containing only canonical vectors whose entries sum to `sum`.:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3)]]), sum=6)
sage: I.cardinality()
10
sage: I.list()
[[6, 0, 0], [5, 1, 0], [5, 0, 1], [4, 2, 0], [4, 1, 1],
 [4, 0, 2], [3, 3, 0], [3, 2, 1], [3, 1, 2], [2, 2, 2]]
sage: I.category()
Join of Category of finite enumerated sets and Category of subquotients of finite sets and Categ
```

To get the orbit of any integer vector v under the action of the group, use the method `orbit()`; we convert the returned set of vectors into a list in increasing lexicographic order, to get a reproducible test:

```
sage: sorted(I.orbit([6,0,0]))
[[0, 0, 6], [0, 6, 0], [6, 0, 0]]
sage: sorted(I.orbit([5,1,0]))
[[0, 5, 1], [1, 0, 5], [5, 1, 0]]
sage: I.orbit([2,2,2])
{[2, 2, 2]}
```

TESTS:

Let us check that canonical integer vectors of the symmetric group are just sorted list of integers:

```
sage: I = IntegerVectorsModPermutationGroup(SymmetricGroup(5)) # long time
sage: p = iter(I) # long time
sage: for i in range(100): # long time
...       v = list(next(p))
...       assert sorted(v, reverse=True) == v
```

We now check that there is as much of canonical vectors under the symmetric group S_n whose entries sum to d than partitions of d of at most n parts:

```
sage: I = IntegerVectorsModPermutationGroup(SymmetricGroup(5)) # long time
sage: for i in range(10): # long time
...       d1 = I.subset(i).cardinality()
...       d2 = Partitions(i, max_length=5).cardinality()
...       print d1
```

```
...       assert d1 == d2
1
1
2
3
5
7
10
13
18
23
```

We present a last corner case: trivial groups. For the trivial group `G` acting on a list of length n, all integer vectors of length n are canonical:

```
sage: G = PermutationGroup([[(6,)]]) # long time
sage: G.cardinality() # long time
1
sage: I = IntegerVectorsModPermutationGroup(G) # long time
sage: for i in range(10): # long time
...       d1 = I.subset(i).cardinality()
...       d2 = IntegerVectors(i,6).cardinality()
...       print d1
...       assert d1 == d2
1
6
21
56
126
252
462
792
1287
2002
```

class sage.combinat.integer_vectors_mod_permgroup.**IntegerVectorsModPermutationGroup_All**(*G*, *sgs=N*

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.combinat.backtrack.SearchForest

A class for integer vectors enumerated up to the action of a permutation group.

A Sage permutation group is viewed as a subgroup of the symmetric group S_n for a certain n. This group has a natural action by position on vectors of length n. This class implements a set which keeps a single vector for each orbit. We say that a vector is canonical if it is the maximum in its orbit under the action of the permutation group for the lexicographic order.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: I
Integer vectors of length 4 enumerated up to the action of Permutation Group with generators [(1
sage: I.cardinality()
+Infinity
sage: TestSuite(I).run()
sage: it = iter(I)
sage: [next(it), next(it), next(it), next(it), next(it)]
[[0, 0, 0, 0],
 [1, 0, 0, 0],
 [2, 0, 0, 0],
```

```
 [1, 1, 0, 0],
 [1, 0, 1, 0]]
sage: x = next(it); x
[3, 0, 0, 0]
sage: I.first()
[0, 0, 0, 0]
```

TESTS:

```
sage: TestSuite(I).run()
```

class Element

Bases: sage.structure.list_clone.ClonableIntArray

Element class for the set of integer vectors of given sum enumerated modulo the action of a permutation group. These vector are clonable lists of integers which must check conditions comming form the parent appearing in the method check().

TESTS:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: v = I.element_class(I, [4,3,2,1]); v
[4, 3, 2, 1]
sage: TestSuite(v).run()
sage: I.element_class(I, [4,3,2,5])
Traceback (most recent call last):
...
AssertionError
```

check()

Checks that self verify the invariants needed for living in self.parent().

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: v = I.an_element()
sage: v.check()
sage: w = I([0,4,0,0], check=False); w
[0, 4, 0, 0]
sage: w.check()
Traceback (most recent call last):
...
AssertionError
```

IntegerVectorsModPermutationGroup_All.**ambient**()

Return the ambient space from which self is a quotient.

EXAMPLES:

```
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: S.ambient()
Integer vectors
```

IntegerVectorsModPermutationGroup_All.**children**(*x*)

Returns the list of children of the element x. This method is required to build the tree structure of self which inherits from the class SearchForest.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: I.children(I([2,1,0,0], check=False))
[[2, 2, 0, 0], [2, 1, 1, 0], [2, 1, 0, 1]]
```

IntegerVectorsModPermutationGroup_All.**is_canonical**(*v*, *check=True*)

Returns `True` if the integer list `v` is maximal in its orbit under the action of the permutation group given to define `self`. Such integer vectors are said to be canonical. A vector v is canonical if and only if

$$v = \max_{\text{lex order}} \{g \cdot v | g \in G\}$$

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: I.is_canonical([4,3,2,1])
True
sage: I.is_canonical([4,0,0,1])
True
sage: I.is_canonical([4,0,3,3])
True
sage: I.is_canonical([4,0,4,4])
False
```

IntegerVectorsModPermutationGroup_All.**lift**(*elt*)

Lift the element `elt` inside the ambient space from which `self` is a quotient.

EXAMPLES:

```
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: v = S.lift(S([4,3,0,1])); v
[4, 3, 0, 1]
sage: type(v)
<type 'list'>
```

IntegerVectorsModPermutationGroup_All.**orbit**(*v*)

Returns the orbit of the integer vector `v` under the action of the permutation group defining `self`. The result is a set.

EXAMPLES:

In order to get reproducible doctests, we convert the returned sets into lists in increasing lexicographic order:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: sorted(I.orbit([2,2,0,0]))
[[0, 0, 2, 2], [0, 2, 2, 0], [2, 0, 0, 2], [2, 2, 0, 0]]
sage: sorted(I.orbit([2,1,0,0]))
[[0, 0, 2, 1], [0, 2, 1, 0], [1, 0, 0, 2], [2, 1, 0, 0]]
sage: sorted(I.orbit([2,0,1,0]))
[[0, 1, 0, 2], [0, 2, 0, 1], [1, 0, 2, 0], [2, 0, 1, 0]]
sage: sorted(I.orbit([2,0,2,0]))
[[0, 2, 0, 2], [2, 0, 2, 0]]
sage: I.orbit([1,1,1,1])
{[1, 1, 1, 1]}
```

IntegerVectorsModPermutationGroup_All.**permutation_group**()

Returns the permutation group given to define `self`.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: I.permutation_group()
Permutation Group with generators [(1,2,3,4)]
```

IntegerVectorsModPermutationGroup_All.**retract**(*elt*)

Return the canonical representative of the orbit of the integer `elt` under the action of the permutation group defining `self`.

If the element `elt` is already maximal in its orbit for the lexicographic order, `elt` is thus the good representative for its orbit.

EXAMPLES:

```
sage: [0,0,0,0] in IntegerVectors(0,4)
True
sage: [1,0,0,0] in IntegerVectors(1,4)
True
sage: [0,1,0,0] in IntegerVectors(1,4)
True
sage: [1,0,1,0] in IntegerVectors(2,4)
True
sage: [0,1,0,1] in IntegerVectors(2,4)
True
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: S.retract([0,0,0,0])
[0, 0, 0, 0]
sage: S.retract([1,0,0,0])
[1, 0, 0, 0]
sage: S.retract([0,1,0,0])
[1, 0, 0, 0]
sage: S.retract([1,0,1,0])
[1, 0, 1, 0]
sage: S.retract([0,1,0,1])
[1, 0, 1, 0]
```

IntegerVectorsModPermutationGroup_All.**roots**()

Returns the root of generation of `self`. This method is required to build the tree structure of `self` which inherits from the class `SearchForest`.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: I.roots()
[[0, 0, 0, 0]]
```

IntegerVectorsModPermutationGroup_All.**subset**(*sum=None*, *max_part=None*)

Returns the subset of `self` containing integer vectors whose entries sum to `sum`.

EXAMPLES:

```
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: S.subset(4)
Integer vectors of length 4 and of sum 4 enumerated up to
the action of Permutation Group with generators
[(1,2,3,4)]
```

class sage.combinat.integer_vectors_mod_permgroup.**IntegerVectorsModPermutationGroup_with_cons**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.combinat.backtrack.SearchForest

This class models finite enumerated sets of integer vectors with constraint enumerated up to the action of a permutation group. Integer vectors are enumerated modulo the action of the permutation group. To implement

that, we keep a single integer vector by orbit under the action of the permutation group. Elements chosen are vectors maximal in their orbit for the lexicographic order.

For more information see IntegerVectorsModPermutationGroup.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), max_part=1)
sage: I.list()
[[0, 0, 0, 0], [1, 0, 0, 0], [1, 1, 0, 0], [1, 0, 1, 0], [1, 1, 1, 0], [1, 1, 1, 1]]
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), sum=6, max_part=4)
sage: I.list()
[[4, 2, 0, 0], [4, 1, 1, 0], [4, 1, 0, 1], [4, 0, 2, 0], [4, 0, 1, 1],
 [4, 0, 0, 2], [3, 3, 0, 0], [3, 2, 1, 0], [3, 2, 0, 1], [3, 1, 2, 0],
 [3, 1, 1, 1], [3, 1, 0, 2], [3, 0, 3, 0], [3, 0, 2, 1], [3, 0, 1, 2],
 [2, 2, 2, 0], [2, 2, 1, 1], [2, 1, 2, 1]]
```

Here is the enumeration of unlabeled graphs over 5 vertices:

```
sage: G = IntegerVectorsModPermutationGroup(TransitiveGroup(10,12), max_part=1) # optional - dat
sage: G.cardinality()  # optional - database_gap
34
```

TESTS:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),4)
sage: TestSuite(I).run()
```

class Element

Bases: sage.structure.list_clone.ClonableIntArray

Element class for the set of integer vectors with constraints enumerated modulo the action of a permutation group. These vectors are clonable lists of integers which must check conditions comming form the parent as in the method check().

TESTS:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), 4)
sage: v = I.element_class(I, [3,1,0,0]); v
[3, 1, 0, 0]
sage: TestSuite(v).run()
sage: v = I.element_class(I, [3,2,0,0])
Traceback (most recent call last):
...
AssertionError: [3, 2, 0, 0] should be a integer vector of sum 4
```

check()

Checks that self meets the constraints of being an element of self.parent().

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), 4)
sage: v = I.an_element()
sage: v.check()
sage: w = I([0,4,0,0], check=False); w
[0, 4, 0, 0]
sage: w.check()
Traceback (most recent call last):
...
AssertionError
```

IntegerVectorsModPermutationGroup_with_constraints.**ambient**()

Return the ambient space from which self is a quotient.

EXAMPLES:

```
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), 6); S.ambient()
Integer vectors that sum to 6
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), 6, max_part=12)
Integer vectors that sum to 6 with constraints: max_part=12
```

Todo

Integer vectors should accept `max_part` as a single argument, and the following should change:

```
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), max_part=12); S
Integer vectors
```

IntegerVectorsModPermutationGroup_with_constraints.**an_element**()

Returns an element of `self` or raises an EmptySetError when `self` is empty.

EXAMPLES:

```
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), sum=0, max_part
[0, 0, 0, 0]
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), sum=1, max_part
[1, 0, 0, 0]
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), sum=2, max_part
[1, 1, 0, 0]
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), sum=3, max_part
[1, 1, 1, 0]
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), sum=4, max_part
[1, 1, 1, 1]
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), sum=5, max_part
Traceback (most recent call last):
...
EmptySetError
```

IntegerVectorsModPermutationGroup_with_constraints.**children**(*x*)

Returns the list of children of the element `x`. This method is required to build the tree structure of `self` which inherits from the class `SearchForest`.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: I.children(I([2,1,0,0], check=False))
[[2, 2, 0, 0], [2, 1, 1, 0], [2, 1, 0, 1]]
```

IntegerVectorsModPermutationGroup_with_constraints.**is_canonical**(*v*, *check=True*)

Returns `True` if the integer list `v` is maximal in its orbit under the action of the permutation group given to define `self`. Such integer vectors are said to be canonical. A vector v is canonical if and only if

$$v = \max_{\text{lex order}} \{g \cdot v | g \in G\}$$

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), max_part=3)
sage: I.is_canonical([3,0,0,0])
True
sage: I.is_canonical([1,0,2,0])
False
sage: I.is_canonical([2,0,1,0])
True
```

IntegerVectorsModPermutationGroup_with_constraints.**lift**(*elt*)

Lift the element `elt` inside the ambient space from which `self` is a quotient.

EXAMPLES:

```
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), max_part=1)
sage: v = S.lift([1,0,1,0]); v
[1, 0, 1, 0]
sage: v in IntegerVectors(2,4,max_part=1)
True
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), sum=6)
sage: v = S.lift(S.list()[5]); v
[4, 1, 1, 0]
sage: v in IntegerVectors(n=6)
True
```

IntegerVectorsModPermutationGroup_with_constraints.**orbit**(*v*)

Returns the orbit of the vector `v` under the action of the permutation group defining `self`. The result is a set.

INPUT:

- `v` - an element of `self` or any list of length the degree of the permutation group.

EXAMPLES:

We convert the result in a list in increasing lexicographic order, to get a reproducible doctest:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]),4)
sage: I.orbit([1,1,1,1])
{[1, 1, 1, 1]}
sage: sorted(I.orbit([3,0,0,1]))
[[0, 0, 1, 3], [0, 1, 3, 0], [1, 3, 0, 0], [3, 0, 0, 1]]
```

IntegerVectorsModPermutationGroup_with_constraints.**permutation_group**()

Returns the permutation group given to define `self`.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3)]]), 5)
sage: I.permutation_group()
Permutation Group with generators [(1,2,3)]
```

IntegerVectorsModPermutationGroup_with_constraints.**retract**(*elt*)

Return the canonical representative of the orbit of the integer `elt` under the action of the permutation group defining `self`.

If the element `elt` is already maximal in its orbits for the lexicographic order, `elt` is thus the good representative for its orbit.

EXAMPLES:

```
sage: S = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]), sum=2, max_part
sage: S.retract([1,1,0,0])
[1, 1, 0, 0]
sage: S.retract([1,0,1,0])
[1, 0, 1, 0]
sage: S.retract([1,0,0,1])
[1, 1, 0, 0]
sage: S.retract([0,1,1,0])
[1, 1, 0, 0]
sage: S.retract([0,1,0,1])
[1, 0, 1, 0]
```

```
sage: S.retract([0,0,1,1])
[1, 1, 0, 0]
```

IntegerVectorsModPermutationGroup_with_constraints.**roots**()

Returns the root of generation of `self`.This method is required to build the tree structure of `self` which inherits from the class `SearchForest`.

EXAMPLES:

```
sage: I = IntegerVectorsModPermutationGroup(PermutationGroup([[(1,2,3,4)]]))
sage: I.roots()
[[0, 0, 0, 0]]
```

5.1.112 Tamari Interval-posets

This module implements Tamari interval-posets: combinatorial objects which represent intervals of the Tamari order. They have been introduced in [PCh2013] and allow for many combinatorial operations on Tamari intervals. In particular, they are linked to `DyckWords` and `BinaryTrees`. An introduction into Tamari interval-posets is given in Chapter 7 of [Pons2013].

The Tamari lattice can be defined as a lattice structure on either of several classes of Catalan objects, especially binary trees and Dyck paths [TamBrack1962] [HuangTamari1972] [Sta-EC2]. An interval can be seen as a pair of comparable elements. The number of intervals has been given in [ChapTamari08].

REFERENCES:

AUTHORS:

- Viviane Pons 2014: initial implementation
- Frederic Chapoton 2014: review
- Darij Grinberg 2014: review
- Travis Scrimshaw 2014: review

class `sage.combinat.interval_posets.`**TamariIntervalPoset**(*parent*, *size*, *relations*, *check=True*)

Bases: `sage.structure.element.Element`

The class of Tamari interval-posets.

An interval-poset is a labelled poset of size n, with labels $1, 2, \ldots, n$, satisfying the following conditions:

•if $a < c$ (as integers) and a precedes c in the poset, then, for all b such that $a < b < c$, b precedes c,

•if $a < c$ (as integers) and c precedes a in the poset, then, for all b such that $a < b < c$, b precedes a.

We use the word "precedes" here to distinguish the poset order and the natural order on numbers. "Precedes" means "is smaller than with respect to the poset structure"; this does not imply a covering relation.

Interval-posets of size n are in bijection with intervals of the Tamari lattice of binary trees of size n. Specifically, if P is an interval-poset of size n, then the set of linear extensions of P (as permutations in S_n) is an interval in the right weak order (see `permutohedron_lequal()`), and is in fact the preimage of an interval in the Tamari lattice (of binary trees of size n) under the operation which sends a permutation to its right-to-left binary search tree (`binary_search_tree()` with the `left_to_right` variable set to `False`) without its labelling.

INPUT:

•`size` – an integer, the size of the interval-posets (number of vertices)

- `relations` – a list (or tuple) of pairs `(a,b)` (themselves lists or tuples), each representing a relation of the form 'a precedes b' in the poset.
- `check` – (default: `True`) whether to check the interval-poset condition or not.

Warning: The `relations` input can be a list or tuple, but not an iterator (nor should its entries be iterators).

NOTATION:

Here and in the following, the signs $<$ and $>$ always refer to the natural ordering on integers, whereas the word "precedes" refers to the order of the interval-poset. "Minimal" and "maximal" refer to the natural ordering on integers.

The *increasing relations* of an interval-poset P mean the pairs (a, b) of elements of P such that $a < b$ as integers and a precedes b in P. The *initial forest* of P is the poset obtained by imposing (only) the increasing relations on the ground set of P. It is a sub-interval poset of P, and is a forest with its roots on top. This forest is usually given the structure of a planar forest by ordering brother nodes by their labels; it then has the property that if its nodes are traversed in post-order (see :meth:~sage.combinat.abstract_tree.AbstractTree.post_order_traversal', and traverse the trees of the forest from left to right as well), then the labels encountered are $1, 2, \ldots, n$ in this order.

The *decreasing relations* of an interval-poset P mean the pairs (a, b) of elements of P such that $b < a$ as integers and a precedes b in P. The *final forest* of P is the poset obtained by imposing (only) the decreasing relations on the ground set of P. It is a sub-interval poset of P, and is a forest with its roots on top. This forest is usually given the structure of a planar forest by ordering brother nodes by their labels; it then has the property that if its nodes are traversed in pre-order (see `pre_order_traversal()`, and traverse the trees of the forest from left to right as well), then the labels encountered are $1, 2, \ldots, n$ in this order.

EXAMPLES:

```
sage: TamariIntervalPoset(0,[])
The Tamari interval of size 0 induced by relations []
sage: TamariIntervalPoset(3,[])
The Tamari interval of size 3 induced by relations []
sage: TamariIntervalPoset(3,[(1,2)])
The Tamari interval of size 3 induced by relations [(1, 2)]
sage: TamariIntervalPoset(3,[(1,2),(2,3)])
The Tamari interval of size 3 induced by relations [(1, 2), (2, 3)]
sage: TamariIntervalPoset(3,[(1,2),(2,3),(1,3)])
The Tamari interval of size 3 induced by relations [(1, 2), (2, 3)]
sage: TamariIntervalPoset(3,[(1,2),(3,2)])
The Tamari interval of size 3 induced by relations [(1, 2), (3, 2)]
sage: TamariIntervalPoset(3,[[1,2],[2,3]])
The Tamari interval of size 3 induced by relations [(1, 2), (2, 3)]
sage: TamariIntervalPoset(3,[[1,2],[2,3],[1,2],[1,3]])
The Tamari interval of size 3 induced by relations [(1, 2), (2, 3)]

sage: TamariIntervalPoset(3,[(3,4)])
Traceback (most recent call last):
...
ValueError: The relations do not correspond to the size of the poset.

sage: TamariIntervalPoset(2,[(2,1),(1,2)])
Traceback (most recent call last):
...
ValueError: The graph is not directed acyclic

sage: TamariIntervalPoset(3,[(1,3)])
```

```
Traceback (most recent call last):
...
ValueError: This does not satisfy the Tamari interval-poset condition.
```

It is also possible to transform a poset directly into an interval-poset:

```
sage: TIP = TamariIntervalPosets()
sage: p = Poset( ([1,2,3], [(1,2)]))
sage: TIP(p)
The Tamari interval of size 3 induced by relations [(1, 2)]
sage: TIP(Poset({1: []}))
The Tamari interval of size 1 induced by relations []
sage: TIP(Poset({}))
The Tamari interval of size 0 induced by relations []
```

binary_trees()

Return an iterator on all the binary trees in the interval represented by `self`.

EXAMPLES:

```
sage: list(TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)]).binary_trees())
[[., [[., [., .]], .]],
 [[., [., [., .]]], .],
 [., [[[., .], .], .]],
 [[., [[., .], .]], .]]
sage: set(TamariIntervalPoset(4,[]).binary_trees()) == set(BinaryTrees(4))
True
```

complement()

Return the complement of the interval-poset `self`.

If P is a Tamari interval-poset of size n, then the *complement* of P is defined as the interval-poset Q whose base set is $[n] = \{1, 2, \ldots, n\}$ (just as for P), but whose order relation has a precede b if and only if $n + 1 - a$ precedes $n + 1 - b$ in P.

In terms of the Tamari lattice, the *complement* is the symmetric of `self`. It is formed from the left-right symmeterized of the binary trees of the interval (switching left and right subtrees, see `left_right_symmetry()`). In particular, initial intervals are sent to final intervals and vice-versa.

EXAMPLES:

```
sage: TamariIntervalPoset(3, [(2, 1), (3, 1)]).complement()
The Tamari interval of size 3 induced by relations [(1, 3), (2, 3)]
sage: TamariIntervalPoset(0, []).complement()
The Tamari interval of size 0 induced by relations []
sage: ip = TamariIntervalPoset(4, [(1, 2), (2, 4), (3, 4)])
sage: ip.complement() == TamariIntervalPoset(4, [(2, 1), (3, 1), (4, 3)])
True
sage: ip.lower_binary_tree() == ip.complement().upper_binary_tree().left_right_symmetry()
True
sage: ip.upper_binary_tree() == ip.complement().lower_binary_tree().left_right_symmetry()
True
sage: ip.is_initial_interval()
True
sage: ip.complement().is_final_interval()
True
```

contains_binary_tree(*binary_tree*)

Return whether the interval represented by `self` contains the binary tree `binary_tree`.

INPUT:

- `binary_tree` – a binary tree

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: ip.contains_binary_tree(BinaryTree([[None,[None,[]]],None]))
True
sage: ip.contains_binary_tree(BinaryTree([None,[[[],None],None]]))
True
sage: ip.contains_binary_tree(BinaryTree([[],[[],None]]))
False
sage: ip.contains_binary_tree(ip.lower_binary_tree())
True
sage: ip.contains_binary_tree(ip.upper_binary_tree())
True
sage: all([ip.contains_binary_tree(bt) for bt in ip.binary_trees()])
True
```

contains_dyck_word(*dyck_word*)

Return whether the interval represented by `self` contains the Dyck word `dyck_word`.

INPUT:

- `dyck_word` – a Dyck word

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: ip.contains_dyck_word(DyckWord([1,1,1,0,0,0,1,0]))
True
sage: ip.contains_dyck_word(DyckWord([1,1,0,1,0,1,0,0]))
True
sage: ip.contains_dyck_word(DyckWord([1,0,1,1,0,1,0,0]))
False
sage: ip.contains_dyck_word(ip.lower_dyck_word())
True
sage: ip.contains_dyck_word(ip.upper_dyck_word())
True
sage: all([ip.contains_dyck_word(bt) for bt in ip.dyck_words()])
True
```

contains_interval(*other*)

Return whether the interval represented by `other` is contained in `self` as an interval of the Tamari lattice.

In terms of interval-posets, it means that all relations of `self` are relations of `other`.

INPUT:

- `other` – an interval-poset

EXAMPLES:

```
sage: ip1 = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: ip2 = TamariIntervalPoset(4,[(2,3)])
sage: ip2.contains_interval(ip1)
True
sage: ip3 = TamariIntervalPoset(4,[(2,1)])
sage: ip2.contains_interval(ip3)
False
sage: ip4 = TamariIntervalPoset(3,[(2,3)])
```

```
sage: ip2.contains_interval(ip4)
False
```

decreasing_children(*v*)

Return the children of v in the final forest of self.

INPUT:

- v – an integer representing a vertex of self (between 1 and size)

OUTPUT:

The list of all children of v in the final forest of self, in increasing order.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(3,5),(4,5)]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5), (6, 5), (5, 2),
sage: ip.decreasing_children(2)
[3, 5]
sage: ip.decreasing_children(3)
[4]
sage: ip.decreasing_children(1)
[]
```

decreasing_cover_relations()

Return the cover relations of the final forest of self (the poset formed by keeping only the relations of the form a precedes b with $a > b$).

The final forest of self is a forest with its roots being on top. It is also called the decreasing poset of self.

Warning: This method computes the cover relations of the final forest. This is not identical with the cover relations of self which happen to be decreasing!

See also:

final_forest()

EXAMPLES:

```
sage: TamariIntervalPoset(4,[(2,1),(3,2),(3,4),(4,2)]).decreasing_cover_relations()
[(4, 2), (3, 2), (2, 1)]
sage: TamariIntervalPoset(4,[(2,1),(4,3),(2,3)]).decreasing_cover_relations()
[(4, 3), (2, 1)]
sage: TamariIntervalPoset(3,[(2,1),(3,1),(3,2)]).decreasing_cover_relations()
[(3, 2), (2, 1)]
```

decreasing_parent(*v*)

Return the vertex parent of v in the final forest of self. This is the highest (as integer!) vertex $a < v$ such that v precedes a. If there is no such vertex (that is, v is a decreasing root), then None is returned.

INPUT:

- v – an integer representing a vertex of self (between 1 and size)

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(3,5),(4,5)]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5), (6, 5), (5, 2),
sage: ip.decreasing_parent(4)
3
```

```
sage: ip.decreasing_parent(3)
2
sage: ip.decreasing_parent(5)
2
sage: ip.decreasing_parent(2) is None
True
```

decreasing_roots()

Return the root vertices of the final forest of `self`, i.e., the vertices b such that there is no $a < b$ with b preceding a.

OUTPUT:

The list of all roots of the final forest of `self`, in increasing order.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(3,5),(4,5)]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5), (6, 5), (5, 2),
sage: ip.decreasing_roots()
[1, 2]
sage: ip.final_forest().decreasing_roots()
[1, 2]
```

dyck_words()

Return an iterator on all the Dyck words in the interval represented by `self`.

EXAMPLES:

```
sage: list(TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)]).dyck_words())
[[1, 1, 1, 0, 0, 1, 0, 0],
 [1, 1, 1, 0, 0, 0, 1, 0],
 [1, 1, 0, 1, 0, 1, 0, 0],
 [1, 1, 0, 1, 0, 0, 1, 0]]
sage: set(TamariIntervalPoset(4,[]).dyck_words()) == set(DyckWords(4))
True
```

final_forest()

Return the final forest of `self`, i.e., the interval-poset formed with only the decreasing relations of `self`.

EXAMPLES:

```
sage: TamariIntervalPoset(4,[(2,1),(3,2),(3,4),(4,2)]).final_forest()
The Tamari interval of size 4 induced by relations [(4, 2), (3, 2), (2, 1)]
sage: ip = TamariIntervalPoset(3,[(2,1),(3,1)])
sage: ip.final_forest() == ip
True
```

ge(*e1*, *e2*)

Return whether `e2` precedes or equals `e1` in `self`.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip.ge(2,1)
True
sage: ip.ge(3,1)
True
sage: ip.ge(3,2)
True
sage: ip.ge(4,3)
False
```

```
sage: ip.ge(1,1)
True
```

gt(*e1*, *e2*)

Return whether `e2` strictly precedes `e1` in `self`.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip.gt(2,1)
True
sage: ip.gt(3,1)
True
sage: ip.gt(3,2)
True
sage: ip.gt(4,3)
False
sage: ip.gt(1,1)
False
```

increasing_children(*v*)

Return the children of `v` in the initial forest of `self`.

INPUT:

- `v` – an integer representing a vertex of `self` (between 1 and `size`)

OUTPUT:

The list of all children of `v` in the initial forest of `self`, in decreasing order.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(3,5),(4,5)]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5), (6, 5), (5, 2),
sage: ip.increasing_children(2)
[1]
sage: ip.increasing_children(5)
[4, 3]
sage: ip.increasing_children(1)
[]
```

increasing_cover_relations()

Return the cover relations of the initial forest of `self` (the poset formed by keeping only the relations of the form a precedes b with $a < b$).

The initial forest of `self` is a forest with its roots being on top. It is also called the increasing poset of `self`.

Warning: This method computes the cover relations of the initial forest. This is not identical with the cover relations of `self` which happen to be increasing!

See also:

`initial_forest()`

EXAMPLES:

```
sage: TamariIntervalPoset(4,[(1,2),(3,2),(2,4),(3,4)]).increasing_cover_relations()
[(1, 2), (2, 4), (3, 4)]
sage: TamariIntervalPoset(3,[(1,2),(1,3),(2,3)]).increasing_cover_relations()
[(1, 2), (2, 3)]
```

increasing_parent(*v*)

Return the vertex parent of v in the initial forest of self.

This is the lowest (as integer!) vertex $b > v$ such that v precedes b. If there is no such vertex (that is, v is an increasing root), then None is returned.

INPUT:

- v – an integer representing a vertex of self (between 1 and size)

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(3,5),(4,5)]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5), (6, 5), (5, 2),
sage: ip.increasing_parent(1)
2
sage: ip.increasing_parent(3)
5
sage: ip.increasing_parent(4)
5
sage: ip.increasing_parent(5) is None
True
```

increasing_roots()

Return the root vertices of the initial forest of self, i.e., the vertices a of self such that there is no $b > a$ with a precedes b.

OUTPUT:

The list of all roots of the initial forest of self, in decreasing order.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(3,5),(4,5)]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5), (6, 5), (5, 2),
sage: ip.increasing_roots()
[6, 5, 2]
sage: ip.initial_forest().increasing_roots()
[6, 5, 2]
```

initial_forest()

Return the initial forest of self, i.e., the interval-poset formed from only the increasing relations of self.

EXAMPLES:

```
sage: TamariIntervalPoset(4,[(1,2),(3,2),(2,4),(3,4)]).initial_forest()
The Tamari interval of size 4 induced by relations [(1, 2), (2, 4), (3, 4)]
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip.initial_forest() == ip
True
```

insertion(*i*)

Return the Tamari insertion of an integer i into the interval-poset self.

If P is a Tamari interval-poset of size n and i is an integer with $1 \leq i \leq n+1$, then the Tamari insertion of i into P is defined as the Tamari interval-poset of size $n+1$ which corresponds to the interval $[C_1, C_2]$ on the Tamari lattice, where the binary trees C_1 and C_2 are defined as follows: We write the interval-poset P as $[B_1, B_2]$ for two binary trees B_1 and B_2. We label the vertices of each of these two trees with the integers $1, 2, \ldots, i-1, i+1, i+2, \ldots, n+1$ in such a way that the trees are binary search

trees (this labelling is unique). Then, we insert i into each of these trees (in the way as explained in `binary_search_insert()`). The shapes of the resulting two trees are denoted C_1 and C_2.

An alternative way to construct the insertion of i into P is by relabeling each vertex u of P satisfying $u \geq i$ (as integers) as $u + 1$, and then adding a vertex i which should precede $i - 1$ and $i + 1$.

Todo

To study this, it would be more natural to define interval-posets on arbitrary ordered sets rather than just on $\{1, 2, \ldots, n\}$.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4, [(2, 3), (4, 3)]); ip
The Tamari interval of size 4 induced by relations [(2, 3), (4, 3)]
sage: ip.insertion(1)
The Tamari interval of size 5 induced by relations [(1, 2), (3, 4), (5, 4)]
sage: ip.insertion(2)
The Tamari interval of size 5 induced by relations [(2, 3), (3, 4), (5, 4), (2, 1)]
sage: ip.insertion(3)
The Tamari interval of size 5 induced by relations [(2, 4), (3, 4), (5, 4), (3, 2)]
sage: ip.insertion(4)
The Tamari interval of size 5 induced by relations [(2, 3), (4, 5), (5, 3), (4, 3)]
sage: ip.insertion(5)
The Tamari interval of size 5 induced by relations [(2, 3), (5, 4), (4, 3)]

sage: ip = TamariIntervalPoset(0, [])
sage: ip.insertion(1)
The Tamari interval of size 1 induced by relations []

sage: ip = TamariIntervalPoset(1, [])
sage: ip.insertion(1)
The Tamari interval of size 2 induced by relations [(1, 2)]
sage: ip.insertion(2)
The Tamari interval of size 2 induced by relations [(2, 1)]
```

TESTS:

Verifying that the two ways of computing insertion are equivalent:

```
sage: def insert_alternative(T, i):
....:     # Just another way to compute the insertion of i into T.
....:     from sage.combinat.binary_tree import LabelledBinaryTree
....:     B1 = T.lower_binary_tree().canonical_labelling()
....:     B2 = T.upper_binary_tree().canonical_labelling()
....:     # We should relabel the trees to "make space" for a label i,
....:     # but we don't, because it doesn't make a difference: The
....:     # binary search insertion will go precisely the same, because
....:     # an integer equal to the label of the root gets sent onto
....:     # the left branch.
....:     C1 = B1.binary_search_insert(i)
....:     C2 = B2.binary_search_insert(i)
....:     return TamariIntervalPosets.from_binary_trees(C1, C2)
sage: def test_equivalence(n):
....:     for T in TamariIntervalPosets(n):
....:         for i in range(1, n + 2):
....:             if not (insert_alternative(T, i) == T.insertion(i)):
....:                 print T, i
....:                 return False
....:     return True
```

```
sage: test_equivalence(3)
True
```

intersection(*other*)

Return the interval-poset formed by combining the relations from both `self` and `other`. It corresponds to the intersection of the two corresponding intervals of the Tamari lattice.

INPUT:

- `other` – an interval-poset of the same size as `self`

EXAMPLES:

```
sage: ip1 = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip2 = TamariIntervalPoset(4,[(4,3)])
sage: ip1.intersection(ip2)
The Tamari interval of size 4 induced by relations [(1, 2), (2, 3), (4, 3)]
sage: ip3 = TamariIntervalPoset(4,[(2,1)])
sage: ip1.intersection(ip3)
Traceback (most recent call last):
...
ValueError: This intersection is empty, it does not correspond to an interval-poset.
sage: ip4 = TamariIntervalPoset(3,[(2,3)])
sage: ip2.intersection(ip4)
Traceback (most recent call last):
...
ValueError: Intersections are only possible on interval-posets of the same size.
```

interval_cardinality()

Return the cardinality of the interval, i.e., the number of elements (binary trees or Dyck words) in the interval represented by `self`.

Not to be confused with `size()` which is the number of vertices.

EXAMPLES:

```
sage: TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)]).interval_cardinality()
4
sage: TamariIntervalPoset(4,[]).interval_cardinality()
14
sage: TamariIntervalPoset(4,[(1,2),(2,3),(3,4)]).interval_cardinality()
1
```

is_final_interval()

Return if `self` corresponds to a final interval of the Tamari lattice, i.e. if its upper end is the largest element of the lattice. It consists of checking that `self` does not contain any increasing relations.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4, [(4, 3), (3, 1), (2, 1)])
sage: ip.is_final_interval()
True
sage: ip.upper_dyck_word()
[1, 1, 1, 1, 0, 0, 0, 0]
sage: ip = TamariIntervalPoset(4, [(4, 3), (3, 1), (2, 1), (2, 3)])
sage: ip.is_final_interval()
False
sage: ip.upper_dyck_word()
[1, 1, 0, 1, 1, 0, 0, 0]
sage: all([ dw.tamari_interval(DyckWord([1, 1, 1, 0, 0, 0])).is_final_interval() for dw in D
True
```

is_initial_interval()

Return if `self` corresponds to an initial interval of the Tamari lattice, i.e. if its lower end is the smallest element of the lattice. It consists of checking that `self` does not contain any decreasing relations.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4, [(1, 2), (2, 4), (3, 4)])
sage: ip.is_initial_interval()
True
sage: ip.lower_dyck_word()
[1, 0, 1, 0, 1, 0, 1, 0]
sage: ip = TamariIntervalPoset(4, [(1, 2), (2, 4), (3, 4), (3, 2)])
sage: ip.is_initial_interval()
False
sage: ip.lower_dyck_word()
[1, 0, 1, 1, 0, 0, 1, 0]
sage: all([ DyckWord([1,0,1,0,1,0]).tamari_interval(dw).is_initial_interval() for dw in Dyck
True
```

is_linear_extension(*perm*)

Return whether the permutation `perm` is a linear extension of `self`.

INPUT:

- `perm` – a permutation of the size of `self`

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: ip.is_linear_extension([1,4,2,3])
True
sage: ip.is_linear_extension(Permutation([1,4,2,3]))
True
sage: ip.is_linear_extension(Permutation([1,4,3,2]))
False
```

is_new()

Return `True` if `self` is a new Tamari interval.

Here 'new' means that the interval is not contained in any facet of the associahedron.

They have been considered in section 9 of [ChapTamari08].

EXAMPLES:

```
sage: TIP4 = TamariIntervalPosets(4)
sage: len([u for u in TIP4 if u.is_new()])
12

sage: TIP3 = TamariIntervalPosets(3)
sage: len([u for u in TIP3 if u.is_new()])
3
```

latex_options()

Return the latex options for use in the `_latex_` function as a dictionary. The default values are set using the global options.

- `tikz_scale` – (default: 1) scale for use with the tikz package
- `line_width` – (default: 1) value representing the line width (additionally scaled by `tikz_scale`)

- color_decreasing – (default: 'red') the color for decreasing relations
- color_increasing – (default: 'blue') the color for increasing relations
- hspace – (default: 1) the difference between horizontal coordinates of adjacent vertices
- vspace – (default: 1) the difference between vertical coordinates of adjacent vertices

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: ip.latex_options()['color_decreasing']
'red'
sage: ip.latex_options()['hspace']
1
```

le(*e1*, *e2*)

Return whether e1 precedes or equals e2 in self.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip.le(1,2)
True
sage: ip.le(1,3)
True
sage: ip.le(2,3)
True
sage: ip.le(3,4)
False
sage: ip.le(1,1)
True
```

linear_extensions()

Return an iterator on the permutations which are linear extensions of self.

They form an interval of the right weak order (also called the right permutohedron order – see permutohedron_lequal() for a definition).

EXAMPLES:

```
sage: ip = TamariIntervalPoset(3,[(1,2),(3,2)])
sage: list(ip.linear_extensions())
[[3, 1, 2], [1, 3, 2]]
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: list(ip.linear_extensions())
[[4, 1, 2, 3], [1, 4, 2, 3], [1, 2, 4, 3]]
```

lower_binary_tree()

Return the lowest binary tree in the interval of the Tamari lattice represented by self.

This is a binary tree. It is the shape of the unique binary search tree whose left-branch ordered forest (i.e., the result of applying to_ordered_tree_left_branch() and cutting off the root) is the final forest of self.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(4,5)]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (4, 5), (6, 5), (5, 2), (4, 3),
sage: ip.lower_binary_tree()
[[., .], [[., [., .]], [., .]]]
sage: TamariIntervalPosets.final_forest(ip.lower_binary_tree()) == ip.final_forest()
True
```

```
sage: ip == TamariIntervalPosets.from_binary_trees(ip.lower_binary_tree(),ip.upper_binary_tr
True
```

lower_contained_intervals()

If `self` represents the interval $[t_1, t_2]$ of the Tamari lattice, return an iterator on all intervals $[t_1, t]$ with $t \leq t_2$ for the Tamari lattice.

In terms of interval-posets, it corresponds to adding all possible relations of the form n precedes m with $n < m$.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: list(ip.lower_contained_intervals())
[The Tamari interval of size 4 induced by relations [(2, 4), (3, 4), (3, 1), (2, 1)],
 The Tamari interval of size 4 induced by relations [(1, 4), (2, 4), (3, 4), (3, 1), (2, 1)]
 The Tamari interval of size 4 induced by relations [(2, 3), (3, 4), (3, 1), (2, 1)],
 The Tamari interval of size 4 induced by relations [(1, 4), (2, 3), (3, 4), (3, 1), (2, 1)]
sage: ip = TamariIntervalPoset(4,[])
sage: len(list(ip.lower_contained_intervals()))
14
```

lower_contains_interval(*other*)

Return whether the interval represented by `other` is contained in `self` as an interval of the Tamari lattice and if they share the same lower bound.

As interval-posets, it means that `other` contains the relations of `self` plus some extra increasing relations.

INPUT:

- `other` – an interval-poset

EXAMPLES:

```
sage: ip1 = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)]);
sage: ip2 = TamariIntervalPoset(4,[(4,3)])
sage: ip2.lower_contains_interval(ip1)
True
sage: ip2.contains_interval(ip1) and ip2.lower_binary_tree() == ip1.lower_binary_tree()
True
sage: ip3 = TamariIntervalPoset(4,[(4,3),(2,1)])
sage: ip2.contains_interval(ip3)
True
sage: ip2.lower_binary_tree() == ip3.lower_binary_tree()
False
sage: ip2.lower_contains_interval(ip3)
False
```

lower_dyck_word()

Return the lowest Dyck word in the interval of the Tamari lattice represented by `self`.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(4,5)]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (4, 5), (6, 5), (5, 2), (4, 3),
sage: ip.lower_dyck_word()
[1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0]
sage: TamariIntervalPosets.final_forest(ip.lower_dyck_word()) == ip.final_forest()
True
```

```
sage: ip == TamariIntervalPosets.from_dyck_words(ip.lower_dyck_word(),ip.upper_dyck_word())
True
```

lt(*e1*, *e2*)

Return whether `e1` strictly precedes `e2` in `self`.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip.lt(1,2)
True
sage: ip.lt(1,3)
True
sage: ip.lt(2,3)
True
sage: ip.lt(3,4)
False
sage: ip.lt(1,1)
False
```

max_linear_extension()

Return the maximal permutation for the right weak order which is a linear extension of `self`.

This is also the maximal permutation in the sylvester class of `self.upper_binary_tree()` and is a 132-avoiding permutation.

The right weak order is also known as the right permutohedron order. See `permutohedron_lequal()` for its definition.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: ip.max_linear_extension()
[4, 1, 2, 3]
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(4,5)]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (4, 5), (6, 5), (5, 2), (4, 3),
sage: ip.max_linear_extension()
[6, 4, 5, 3, 1, 2]
sage: ip = TamariIntervalPoset(0,[]); ip
The Tamari interval of size 0 induced by relations []
sage: ip.max_linear_extension()
[]
sage: ip = TamariIntervalPoset(5, [(1, 4), (2, 4), (3, 4), (5, 4)]); ip
The Tamari interval of size 5 induced by relations [(1, 4), (2, 4), (3, 4), (5, 4)]
sage: ip.max_linear_extension()
[5, 3, 2, 1, 4]
```

maximal_chain_binary_trees()

Return an iterator on the binary trees forming a longest chain of `self` (regarding `self` as an interval of the Tamari lattice).

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: list(ip.maximal_chain_binary_trees())
[[[., [[., .], .]], .], [., [[[., .], .], .]], [., [[., [., .]], .]]]
sage: ip = TamariIntervalPoset(4,[])
sage: list(ip.maximal_chain_binary_trees())
[[[[[., .], .], .], .],
 [[[., [., .]], .], .],
 [[., [[., .], .]], .],
```

```
 [., [[[., .], .], .]],
 [., [[., [., .]], .]],
 [., [., [[., .], .]]],
 [., [., [., [., .]]]]]
```

maximal_chain_dyck_words()

Return an iterator on the Dyck words forming a longest chain of `self` (regarding `self` as an interval of the Tamari lattice).

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: list(ip.maximal_chain_dyck_words())
[[1, 1, 0, 1, 0, 0, 1, 0], [1, 1, 0, 1, 0, 1, 0, 0], [1, 1, 1, 0, 0, 1, 0, 0]]
sage: ip = TamariIntervalPoset(4,[])
sage: list(ip.maximal_chain_dyck_words())
[[1, 0, 1, 0, 1, 0, 1, 0],
 [1, 1, 0, 0, 1, 0, 1, 0],
 [1, 1, 0, 1, 0, 0, 1, 0],
 [1, 1, 0, 1, 0, 1, 0, 0],
 [1, 1, 1, 0, 0, 1, 0, 0],
 [1, 1, 1, 0, 1, 0, 0, 0],
 [1, 1, 1, 1, 0, 0, 0, 0]]
```

maximal_chain_tamari_intervals()

Return an iterator on the upper contained intervals of one longest chain of the Tamari interval represented by `self`.

If `self` represents the interval $[T_1, T_2]$ of the Tamari lattice, this returns intervals $[T', T_2]$ with T' following one longest chain between T_1 and T_2.

To obtain a longest chain, we use the Tamari inversions of `self`. The elements of the chain are obtained by adding one by one the relations (b, a) from each Tamari inversion (a, b) to `self`, where the Tamari inversions are taken in lexicographic order.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: list(ip.maximal_chain_tamari_intervals())
[The Tamari interval of size 4 induced by relations [(2, 4), (3, 4), (3, 1), (2, 1)],
 The Tamari interval of size 4 induced by relations [(2, 4), (3, 4), (4, 1), (3, 1), (2, 1)]
 The Tamari interval of size 4 induced by relations [(2, 4), (3, 4), (4, 1), (3, 2), (2, 1)]
sage: ip = TamariIntervalPoset(4,[])
sage: list(ip.maximal_chain_tamari_intervals())
[The Tamari interval of size 4 induced by relations [],
 The Tamari interval of size 4 induced by relations [(2, 1)],
 The Tamari interval of size 4 induced by relations [(3, 1), (2, 1)],
 The Tamari interval of size 4 induced by relations [(4, 1), (3, 1), (2, 1)],
 The Tamari interval of size 4 induced by relations [(4, 1), (3, 2), (2, 1)],
 The Tamari interval of size 4 induced by relations [(4, 2), (3, 2), (2, 1)],
 The Tamari interval of size 4 induced by relations [(4, 3), (3, 2), (2, 1)]]
```

min_linear_extension()

Return the minimal permutation for the right weak order which is a linear extension of `self`.

This is also the minimal permutation in the sylvester class of `self.lower_binary_tree()` and is a 312-avoiding permutation.

The right weak order is also known as the right permutohedron order. See `permutohedron_lequal()` for its definition.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: ip.min_linear_extension()
[1, 2, 4, 3]
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(4,5)])
sage: ip.min_linear_extension()
[1, 4, 3, 6, 5, 2]
sage: ip = TamariIntervalPoset(0,[])
sage: ip.min_linear_extension()
[]
sage: ip = TamariIntervalPoset(5, [(1, 4), (2, 4), (3, 4), (5, 4)]); ip
The Tamari interval of size 5 induced by relations [(1, 4), (2, 4), (3, 4), (5, 4)]
sage: ip.min_linear_extension()
[1, 2, 3, 5, 4]
```

number_of_tamari_inversions()

Return the number of Tamari inversions of `self`. This is also the length the longest chain of the Tamari interval represented by `self`.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: ip.number_of_tamari_inversions()
2
sage: ip = TamariIntervalPoset(4,[])
sage: ip.number_of_tamari_inversions()
6
sage: ip = TamariIntervalPoset(3,[])
sage: ip.number_of_tamari_inversions()
3
```

plot(***kwds*)

Return a picture.

The picture represents the Hasse diagram, where the covers are colored in blue if they are increasing and in red if they are decreasing.

This uses the same coordinates as the latex view.

EXAMPLES:

```
sage: ti = TamariIntervalPosets(4)[2]
sage: ti.plot()
Graphics object consisting of 6 graphics primitives
```

poset()

Return `self` as a labelled poset.

An interval-poset is indeed constructed from a labelled poset which is stored internally. This method allows to access the poset and all the associated methods.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(3,2),(2,4),(3,4)])
sage: pos = ip.poset(); pos
Finite poset containing 4 elements
sage: pos.maximal_chains()
[[3, 2, 4], [1, 2, 4]]
sage: pos.maximal_elements()
[4]
```

```
sage: pos.is_lattice()
False
```

set_latex_options(*D*)

Set the latex options for use in the _latex_ function. The default values are set in the __init__ function.

- tikz_scale – (default: 1) scale for use with the tikz package
- line_width – (default: 1*‘‘tikz_scale‘‘) value representing the line width
- color_decreasing – (default: red) the color for decreasing relations
- color_increasing – (default: blue) the color for increasing relations
- hspace – (default: 1) the difference between horizontal coordinates of adjacent vertices
- vspace – (default: 1) the difference between vertical coordinates of adjacent vertices

INPUT:

- D – a dictionary with a list of latex parameters to change

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: ip.latex_options()["color_decreasing"]
'red'
sage: ip.set_latex_options({"color_decreasing":'green'})
sage: ip.latex_options()["color_decreasing"]
'green'
sage: ip.set_latex_options({"color_increasing":'black'})
sage: ip.latex_options()["color_increasing"]
'black'
```

To change the default options for all interval-posets, use the parent's global latex options:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: ip2 = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip.latex_options()["color_decreasing"]
'red'
sage: ip2.latex_options()["color_decreasing"]
'red'
sage: TamariIntervalPosets.global_options(latex_color_decreasing='green')
sage: ip.latex_options()["color_decreasing"]
'green'
sage: ip2.latex_options()["color_decreasing"]
'green'
```

Next we set a local latex option and show the global does not override it:

```
sage: ip.set_latex_options({"color_decreasing": 'black'})
sage: ip.latex_options()["color_decreasing"]
'black'
sage: TamariIntervalPosets.global_options(latex_color_decreasing='blue')
sage: ip.latex_options()["color_decreasing"]
'black'
sage: ip2.latex_options()["color_decreasing"]
'blue'
sage: TamariIntervalPosets.global_options.reset()
```

size()
Return the size (number of vertices) of the interval-poset.

EXAMPLES:
```
sage: TamariIntervalPoset(3,[(2,1),(3,1)]).size()
3
```

sub_poset(*start*, *end*)
Return the renormalized sub-poset of `self` consisting solely of integers from `start` (inclusive) to `end` (not inclusive).

"Renormalized" means that these integers are relabelled $1, 2, \ldots, k$ in the obvious way (i.e., by subtracting `start - 1`).

INPUT:

- `start` – an integer, the starting vertex (inclusive)
- `end` – an integer, the ending vertex (not inclusive)

EXAMPLES:
```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(3,5),(4,5)]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (3, 5), (4, 5), (6, 5), (5, 2),
sage: ip.sub_poset(1,3)
The Tamari interval of size 2 induced by relations [(1, 2)]
sage: ip.sub_poset(1,4)
The Tamari interval of size 3 induced by relations [(1, 2), (3, 2)]
sage: ip.sub_poset(1,5)
The Tamari interval of size 4 induced by relations [(1, 2), (4, 3), (3, 2)]
sage: ip.sub_poset(1,7) == ip
True
sage: ip.sub_poset(1,1)
The Tamari interval of size 0 induced by relations []
```

tamari_inversions()
Return the Tamari inversions of `self`. A Tamari inversion is a pair of vertices (a, b) with $a < b$ such that:

- the decreasing parent of b is strictly smaller than a (or does not exist), and
- the increasing parent of a is strictly bigger than b (or does not exist).

"Smaller" and "bigger" refer to the numerical values of the elements, not to the poset order.

This method returns the list of all Tamari inversions in lexicographic order.

The number of Tamari inversions is the length of the longest chain of the Tamari interval represented by `self`.

Indeed, when an interval consists of just one binary tree, it has no inversion. One can also prove that if a Tamari interval $I' = [T'_1, T'_2]$ is a proper subset of a Tamari interval $I = [T_1, T_2]$, then the inversion number of I' is strictly lower than the inversion number of I. And finally, by adding the relation (b, a) to the interval-poset where (a, b) is the first inversion of I in lexicographic order, one reduces the inversion number by exactly 1.

See also:

`tamari_inversions_iter()`.

EXAMPLES:
```
sage: ip = TamariIntervalPoset(3,[])
sage: ip.tamari_inversions()
[(1, 2), (1, 3), (2, 3)]
```

```
sage: ip = TamariIntervalPoset(3,[(2,1)])
sage: ip.tamari_inversions()
[(1, 3), (2, 3)]
sage: ip = TamariIntervalPoset(3,[(1,2)])
sage: ip.tamari_inversions()
[(2, 3)]
sage: ip = TamariIntervalPoset(3,[(1,2),(3,2)])
sage: ip.tamari_inversions()
[]
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: ip.tamari_inversions()
[(1, 4), (2, 3)]
sage: ip = TamariIntervalPoset(4,[])
sage: ip.tamari_inversions()
[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
sage: all([len(TamariIntervalPosets.from_binary_trees(bt,bt).tamari_inversions())==0 for bt
True
sage: all([len(TamariIntervalPosets.from_binary_trees(bt,bt).tamari_inversions())==0 for bt
True
```

tamari_inversions_iter()

Iterate over the Tamari inversions of `self`, in lexicographic order.

See `tamari_inversions()` for the definition of the terms involved.

EXAMPLES:

```
sage: T = TamariIntervalPoset(5, [[1,2],[3,4],[3,2],[5,2],[4,2]])
sage: list(T.tamari_inversions_iter())
[(4, 5)]

sage: T = TamariIntervalPoset(8, [(2, 7), (3, 7), (4, 7), (5, 7), (6, 7), (8, 7), (6, 4), (5
sage: list(T.tamari_inversions_iter())
[(1, 2), (1, 7), (5, 6)]

sage: T = TamariIntervalPoset(1, [])
sage: list(T.tamari_inversions_iter())
[]

sage: T = TamariIntervalPoset(0, [])
sage: list(T.tamari_inversions_iter())
[]
```

upper_binary_tree()

Return the highest binary tree in the interval of the Tamari lattice represented by `self`.

This is a binary tree. It is the shape of the unique binary search tree whose right-branch ordered forest (i.e., the result of applying `to_ordered_tree_right_branch()` and cutting off the root) is the initial forest of `self`.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(4,5)]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (4, 5), (6, 5), (5, 2), (4, 3),
sage: ip.upper_binary_tree()
[[., .], [., [[., .], [., .]]]]
sage: TamariIntervalPosets.initial_forest(ip.upper_binary_tree()) == ip.initial_forest()
True
sage: ip == TamariIntervalPosets.from_binary_trees(ip.lower_binary_tree(),ip.upper_binary_tr
True
```

upper_contains_interval(*other*)

Return whether the interval represented by `other` is contained in `self` as an interval of the Tamari lattice and if they share the same upper bound.

As interval-posets, it means that `other` contains the relations of `self` plus some extra decreasing relations.

INPUT:

- `other` – an interval-poset

EXAMPLES:

```
sage: ip1 = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: ip2 = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: ip2.upper_contains_interval(ip1)
True
sage: ip2.contains_interval(ip1) and ip2.upper_binary_tree() == ip1.upper_binary_tree()
True
sage: ip3 = TamariIntervalPoset(4,[(1,2),(2,3),(3,4)])
sage: ip2.upper_contains_interval(ip3)
False
sage: ip2.contains_interval(ip3)
True
sage: ip2.upper_binary_tree() == ip3.upper_binary_tree()
False
```

upper_dyck_word()

Return the highest Dyck word in the interval of the Tamari lattice represented by `self`.

EXAMPLES:

```
sage: ip = TamariIntervalPoset(6,[(3,2),(4,3),(5,2),(6,5),(1,2),(4,5)]); ip
The Tamari interval of size 6 induced by relations [(1, 2), (4, 5), (6, 5), (5, 2), (4, 3),
sage: ip.upper_dyck_word()
[1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0]
sage: TamariIntervalPosets.initial_forest(ip.upper_dyck_word()) == ip.initial_forest()
True
sage: ip == TamariIntervalPosets.from_dyck_words(ip.lower_dyck_word(),ip.upper_dyck_word())
True
```

class sage.combinat.interval_posets.**TamariIntervalPosets**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Factory for interval-posets.

INPUT:

- `size` – (optional) an integer

OUTPUT:

- the set of all interval-posets (of the given `size` if specified)

EXAMPLES:

```
sage: TamariIntervalPosets()
Interval-posets

sage: TamariIntervalPosets(2)
Interval-posets of size 2
```

Note: This is a factory class whose constructor returns instances of subclasses.

static check_poset (*poset*)

Check if the given poset `poset` is a interval-poset, that is, if it satisfies the following properties:

- Its labels are exactly $1, \ldots, n$ where n is its size.
- If $a < c$ (as numbers) and a precedes c, then b precedes c for all b such that $a < b < c$.
- If $a < c$ (as numbers) and c precedes a, then b precedes a for all b such that $a < b < c$.

INPUT:

- `poset` – a finite labeled poset

EXAMPLES:

```
sage: p = Poset(([1,2,3],[(1,2),(3,2)]))
sage: TamariIntervalPosets.check_poset(p)
True
sage: p = Poset(([2,3],[(3,2)]))
sage: TamariIntervalPosets.check_poset(p)
False
sage: p = Poset(([1,2,3],[(3,1)]))
sage: TamariIntervalPosets.check_poset(p)
False
sage: p = Poset(([1,2,3],[(1,3)]))
sage: TamariIntervalPosets.check_poset(p)
False
```

static final_forest (*element*)

Return the final forest of a binary tree, an interval-poset or a Dyck word.

A final forest is an interval-poset corresponding to a final interval of the Tamari lattice, i.e., containing only decreasing relations.

It can be constructed from a binary tree by its binary search tree labeling with the rule: b precedes a in the final forest iff b is in the right subtree of a in the binary search tree.

INPUT:

- `element` – a binary tree, a Dyck word or an interval-poset

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: TamariIntervalPosets.final_forest(ip)
The Tamari interval of size 4 induced by relations [(1, 2), (2, 3)]
```

From binary trees:

```
sage: bt = BinaryTree(); bt
.
sage: TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 0 induced by relations []
sage: bt = BinaryTree([]); bt
[., .]
sage: TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 1 induced by relations []
sage: bt = BinaryTree([[],None]); bt
[[., .], .]
sage: TamariIntervalPosets.final_forest(bt)
```

```
The Tamari interval of size 2 induced by relations []
sage: bt = BinaryTree([None,[]]); bt
[., [., .]]
sage: TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 2 induced by relations [(2, 1)]
sage: bt = BinaryTree([[],[]]); bt
[[., .], [., .]]
sage: TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 3 induced by relations [(3, 2)]
sage: bt = BinaryTree([[None,[[],None]],[]]); bt
[[., [[., .], .]], [., .]]
sage: TamariIntervalPosets.final_forest(bt)
The Tamari interval of size 5 induced by relations [(5, 4), (3, 1), (2, 1)]
```

From Dyck words:

```
sage: dw = DyckWord([1,0])
sage: TamariIntervalPosets.final_forest(dw)
The Tamari interval of size 1 induced by relations []
sage: dw = DyckWord([1,1,0,1,0,0,1,1,0,0])
sage: TamariIntervalPosets.final_forest(dw)
The Tamari interval of size 5 induced by relations [(5, 4), (3, 1), (2, 1)]
```

static from_binary_trees (*tree1*, *tree2*)

Return the interval-poset corresponding to the interval [`tree1`, `tree2`] of the Tamari lattice. Raise an exception if `tree1` is not $\leq$ `tree2` in the Tamari lattice.

INPUT:

- `tree1` – a binary tree
- `tree2` – a binary tree greater or equal than `tree1` for the Tamari lattice

EXAMPLES:

```
sage: tree1 = BinaryTree([[],None])
sage: tree2 = BinaryTree([None,[]])
sage: TamariIntervalPosets.from_binary_trees(tree1,tree2)
The Tamari interval of size 2 induced by relations []
sage: TamariIntervalPosets.from_binary_trees(tree1,tree1)
The Tamari interval of size 2 induced by relations [(1, 2)]
sage: TamariIntervalPosets.from_binary_trees(tree2,tree2)
The Tamari interval of size 2 induced by relations [(2, 1)]

sage: tree1 = BinaryTree([[],[[None,[]],[]]])
sage: tree2 = BinaryTree([None,[None,[None,[[],[]]]]])
sage: TamariIntervalPosets.from_binary_trees(tree1,tree2)
The Tamari interval of size 6 induced by relations [(4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]

sage: tree3 = BinaryTree([None,[None,[[],[None,[]]]]])
sage: TamariIntervalPosets.from_binary_trees(tree1,tree3)
Traceback (most recent call last):
...
ValueError: The two binary trees are not comparable on the Tamari lattice.
sage: TamariIntervalPosets.from_binary_trees(tree1,BinaryTree())
Traceback (most recent call last):
...
ValueError: The two binary trees are not comparable on the Tamari lattice.
```

static from_dyck_words (*dw1*, *dw2*)

Return the interval-poset corresponding to the interval [dw1, dw2] of the Tamari lattice. Raise an exception if the two Dyck words dw1 and dw2 do not satisfy dw1 $\leq$ dw2 in the Tamari lattice.

INPUT:

- dw1 – a Dyck word
- dw2 – a Dyck word greater or equal than dw1 for the Tamari lattice

EXAMPLES:

```
sage: dw1 = DyckWord([1,0,1,0])
sage: dw2 = DyckWord([1,1,0,0])
sage: TamariIntervalPosets.from_dyck_words(dw1,dw2)
The Tamari interval of size 2 induced by relations []
sage: TamariIntervalPosets.from_dyck_words(dw1,dw1)
The Tamari interval of size 2 induced by relations [(1, 2)]
sage: TamariIntervalPosets.from_dyck_words(dw2,dw2)
The Tamari interval of size 2 induced by relations [(2, 1)]

sage: dw1 = DyckWord([1,0,1,1,1,0,0,1,1,0,0,0])
sage: dw2 = DyckWord([1,1,1,1,0,1,1,0,0,0,0,0])
sage: TamariIntervalPosets.from_dyck_words(dw1,dw2)
The Tamari interval of size 6 induced by relations [(4, 5), (6, 5), (5, 2), (4, 3), (3, 2)]

sage: dw3 = DyckWord([1,1,1,0,1,1,1,0,0,0,0,0])
sage: TamariIntervalPosets.from_dyck_words(dw1,dw3)
Traceback (most recent call last):
...
ValueError: The two Dyck words are not comparable on the Tamari lattice.
sage: TamariIntervalPosets.from_dyck_words(dw1,DyckWord([1,0]))
Traceback (most recent call last):
...
ValueError: The two Dyck words are not comparable on the Tamari lattice.
```

global_options(**get_value*, ***set_value*)

Set and display the global options for Tamari interval-posets. If no parameters are set, then the function returns a copy of the options dictionary.

The options to Tamari interval-posets can be accessed as the method TamariIntervalPosets.global_options of TamariIntervalPosets and related parent classes.

OPTIONS:

- latex_color_decreasing – (default: red) the default color of decreasing relations when latexed
- latex_color_increasing – (default: blue) the default color of increasing relations when latexed
- latex_hspace – (default: 1) the default difference between horizontal coordinates of vertices when latexed
- latex_line_width_scalar – (default: 0.5) the default value for the line width as amultiple of the tikz scale when latexed
- latex_tikz_scale – (default: 1) the default value for the tikz scale when latexed
- latex_vspace – (default: 1) the default difference between vertical coordinates of vertices when latexed

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(2,4),(3,4),(2,1),(3,1)])
sage: ip.latex_options()["color_decreasing"]
'red'
sage: TamariIntervalPosets.global_options(latex_color_decreasing='green')
sage: ip.latex_options()["color_decreasing"]
'green'
sage: TamariIntervalPosets.global_options.reset()
sage: ip.latex_options()["color_decreasing"]
'red'
```

See `GlobalOptions` for more features of these options.

static initial_forest(*element*)

Return the inital forest of a binary tree, an interval-poset or a Dyck word.

An initial forest is an interval-poset corresponding to an initial interval of the Tamari lattice, i.e., containing only increasing relations.

It can be constructed from a binary tree by its binary search tree labeling with the rule: a precedes b in the initial forest iff a is in the left subtree of b in the binary search tree.

INPUT:

- `element` – a binary tree, a Dyck word or an interval-poset

EXAMPLES:

```
sage: ip = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: TamariIntervalPosets.initial_forest(ip)
The Tamari interval of size 4 induced by relations [(1, 2), (2, 3)]
```

with binary trees:

```
sage: bt = BinaryTree(); bt
.
sage: TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 0 induced by relations []
sage: bt = BinaryTree([]); bt
[., .]
sage: TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 1 induced by relations []
sage: bt = BinaryTree([[],None]); bt
[[., .], .]
sage: TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 2 induced by relations [(1, 2)]
sage: bt = BinaryTree([None,[]]); bt
[., [., .]]
sage: TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 2 induced by relations []
sage: bt = BinaryTree([[],[]]); bt
[[., .], [., .]]
sage: TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 3 induced by relations [(1, 2)]
sage: bt = BinaryTree([[None,[[],None]],[]]); bt
[[., [[., .], .]], [., .]]
sage: TamariIntervalPosets.initial_forest(bt)
The Tamari interval of size 5 induced by relations [(1, 4), (2, 3), (3, 4)]
```

from Dyck words:

```
sage: dw = DyckWord([1,0])
sage: TamariIntervalPosets.initial_forest(dw)
The Tamari interval of size 1 induced by relations []
sage: dw = DyckWord([1,1,0,1,0,0,1,1,0,0])
sage: TamariIntervalPosets.initial_forest(dw)
The Tamari interval of size 5 induced by relations [(1, 4), (2, 3), (3, 4)]
```

le(*el1*, *el2*)

Poset stucture on the set of interval-posets through interval containment.

Return whether the interval represented by `el1` is contained in the interval represented by `el2`.

INPUT:

- `el1` – an interval-poset
- `el2` – an interval-poset

EXAMPLES:

```
sage: ip1 = TamariIntervalPoset(4,[(1,2),(2,3),(4,3)])
sage: ip2 = TamariIntervalPoset(4,[(1,2),(2,3)])
sage: TamariIntervalPosets().le(ip1,ip2)
True
sage: TamariIntervalPosets().le(ip2,ip1)
False
```

class sage.combinat.interval_posets.**TamariIntervalPosets_all**

Bases: sage.sets.disjoint_union_enumerated_sets.DisjointUnionEnumeratedSets, sage.combinat.interval_posets.TamariIntervalPosets

The enumerated set of all Tamari interval-posets.

Element

alias of TamariIntervalPoset

class sage.combinat.interval_posets.**TamariIntervalPosets_size**(*size*)

Bases: sage.combinat.interval_posets.TamariIntervalPosets

The enumerated set of interval-posets of a given size.

TESTS:

```
sage: from sage.combinat.interval_posets import TamariIntervalPosets_size
sage: for i in xrange(6): TestSuite(TamariIntervalPosets_size(i)).run()
```

cardinality()

The cardinality of `self`. That is, the number of interval-posets of size n.

The formula was given in [ChapTamari08]:

$$\frac{2(4n+1)!}{(n+1)!(3n+2)!} = \frac{2}{n(n+1)}\binom{4n+1}{n-1}.$$

EXAMPLES:

```
sage: [TamariIntervalPosets(i).cardinality() for i in range(6)]
[1, 1, 3, 13, 68, 399]
```

element_class()

TESTS:

```
sage: S = TamariIntervalPosets(3)
sage: S.element_class
<class 'sage.combinat.interval_posets.TamariIntervalPosets_all_with_category.element_class'>
sage: S.first().__class__ == TamariIntervalPosets().first().__class__
True
```

5.1.113 Strong and weak tableaux

There are two types of k-tableaux: strong k-tableaux and weak k-tableaux. Standard weak k-tableaux correspond to saturated chains in the weak order, whereas standard strong k-tableaux correspond to saturated chains in the strong Bruhat order. For semistandard tableaux, the notion of weak and strong horizontal strip is necessary. More information can be found in [LLMS2006] .

See also:

`sage.combinat.k_tableau.StrongTableau()`, `sage.combinat.k_tableau.WeakTableau()`

REFERENCES:

Authors:

- Anne Schilling and Mike Zabrocki (2013): initial version
- Avi Dalal and Nate Gallup (2013): implementation of k-charge

class `sage.combinat.k_tableau.`**`StrongTableau`**(*parent*, *T*)
Bases: `sage.structure.list_clone.ClonableList`

A (standard) strong k-tableau is a (saturated) chain in Bruhat order.

Combinatorially, it is a sequence of embedded $k+1$-cores (subject to some conditions) together with a set of markings.

A strong cover in terms of cores corresponds to certain translated ribbons. A marking corresponds to the choice of one of the translated ribbons, which is indicated by marking the head (southeast most cell in French notation) of the chosen ribbon. For more information, see [LLMS2006] and [LLMSSZ2013].

In Sage, a strong k-tableau is created by specifying k, a standard strong tableau together with its markings, and a weight μ. Here the standard tableau is represented by a sequence of $k+1$-cores

$$\lambda^{(0)} \subseteq \lambda^{(1)} \subseteq \cdots \subseteq \lambda^{(m)}$$

where each of the $\lambda^{(i)}$ is a $k+1$-core. The standard tableau is a filling of the diagram for the core $\lambda^{(m)}/\lambda^{(0)}$ where a strong cover is represented by letters $\pm i$ in the skew shape $\lambda^{(i)}/\lambda^{(i-1)}$. Each skew $(k+1)$-core $\lambda^{(i)}/\lambda^{(i-1)}$ is a ribbon or multiple copies of the same ribbon which are separated by $k+1$ diagonals. Precisely one of the copies of the ribbons will be marked in the largest diagonal of the connected component (the 'head' of the ribbon). The marked cells are indicated by negative signs.

The strong tableau is stored as a standard strong marked tableau (referred to as the standard part of the strong tableau) and a vector representing the weight.

EXAMPLES:

```
sage: StrongTableau( [[-1, -2, -3], [3]], 2, [3] )
[[-1, -1, -1], [1]]
sage: StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3, [2,2,3,1])
[[-1, -1, -2, -3], [-2, 3, -3, 4], [2, 3], [-3, -4]]
```

Alternatively, the strong k-tableau can also be entered directly in semistandard format and then the standard tableau and the weight are computed and stored:

```
sage: T = StrongTableau([[-1,-1,-1],[1]], 2); T
[[-1, -1, -1], [1]]
sage: T.to_standard_list()
[[-1, -2, -3], [3]]
sage: T.weight()
(3,)
sage: T = StrongTableau([[-1, -1, -2, -3], [-2, 3, -3, 4], [2, 3], [-3, -4]], 3); T
[[-1, -1, -2, -3], [-2, 3, -3, 4], [2, 3], [-3, -4]]
sage: T.to_standard_list()
[[-1, -2, -4, -7], [-3, 6, -6, 8], [4, 7], [-5, -8]]
sage: T.weight()
(2, 2, 3, 1)
```

cell_of_highest_head(*v*)

Return the cell of the highest head of label v in the standard part of self.

Return the cell where the head of the ribbon in the highest row is located in the underlying standard tableau. If there is no cell with entry v then the cell returned is $(0, r)$ where r is the length of the first row.

This cell is calculated by iterating through the diagonals of the tableau.

INPUT:

- v – an integer indicating the label in the standard tableau

OUTPUT:

- a pair of integers indicating the coordinates of the head of the highest ribbon with label v

EXAMPLES:

```
sage: T = StrongTableau([[-1,2,-3],[-2,3],[3]], 1)
sage: [T.cell_of_highest_head(v) for v in range(1,5)]
[(0, 0), (1, 0), (2, 0), (0, 3)]
sage: T = StrongTableau([[None,None,-3,4],[3,-4]],2)
sage: [T.cell_of_highest_head(v) for v in range(1,5)]
[(1, 0), (1, 1), (0, 4), (0, 4)]
```

TESTS:

```
sage: StrongTableau([],2).cell_of_highest_head(1)
(0, 0)
```

cell_of_marked_head(*v*)

Return location of marked head labeled by v in the standard part of self.

Return the coordinates of the v-th marked cell in the strong standard tableau self. If there is no mark, then the value returned is $(0, r)$ where r is the length of the first row.

INPUT:

- v – an integer representing the label in the standard tableau

OUTPUT:

- a pair of the coordinates of the marked cell with entry v

EXAMPLES:

```
sage: T = StrongTableau([[-1, -3, 4, -5], [-2], [-4]], 3)
sage: [ T.cell_of_marked_head(i) for i in range(1,7)]
[(0, 0), (1, 0), (0, 1), (2, 0), (0, 3), (0, 4)]
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3
```

```
sage: [ T.cell_of_marked_head(i) for i in range(1,7)]
[(2, 0), (0, 2), (2, 1), (0, 3), (4, 0), (0, 4)]
```

TESTS:

```
sage: StrongTableau([],4).cell_of_marked_head(4)
(0, 0)
```

cells_head_dictionary()

Return a dictionary with the locations of the heads of all markings.

Return a dictionary of values and lists of cells where the heads with the values are located.

OUTPUT:

- a dictionary with keys the entries in the tableau and values are the coordinates of the heads with those entries

EXAMPLES:

```
sage: T = StrongTableau([[-1,-2,-4,7],[-3,6,-6,8],[4,-7],[-5,-8]], 3)
sage: T.cells_head_dictionary()
{1: [(0, 0)],
 2: [(0, 1)],
 3: [(1, 0)],
 4: [(2, 0), (0, 2)],
 5: [(3, 0)],
 6: [(1, 2)],
 7: [(2, 1), (0, 3)],
 8: [(3, 1), (1, 3)]}
sage: T = StrongTableau([[None, 4, -4, -6, -7, 8, 8, -8], [None, -5, 8, 8, 8], [-3, 6]],3)
sage: T.cells_head_dictionary()
{1: [(2, 0)],
 2: [(0, 2)],
 3: [(1, 1)],
 4: [(2, 1), (0, 3)],
 5: [(0, 4)],
 6: [(1, 4), (0, 7)]}
 sage: StrongTableau([[None, None], [None, -1]], 4).cells_head_dictionary()
 {1: [(1, 1)]}
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).cells_head_dictionary()
{}
sage: StrongTableau([],4).cells_head_dictionary()
{}
```

cells_of_heads(*v*)

Return a list of cells of the heads with label `v` in the standard part of `self`.

A list of cells which are heads of the ribbons with label `v` in the standard part of the tableau `self`. If there is no cell labelled by `v` then return the empty list.

INPUT:

- `v` – an integer label

OUTPUT:

- a list of pairs of integers of the coordinates of the heads of the ribbons with label `v`

EXAMPLES:

```
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3
sage: T.cells_of_heads(1)
[(2, 0)]
sage: T.cells_of_heads(2)
[(3, 0), (0, 2)]
sage: T.cells_of_heads(3)
[(2, 1)]
sage: T.cells_of_heads(4)
[(3, 1), (0, 3)]
sage: T.cells_of_heads(5)
[(4, 0)]
sage: T.cells_of_heads(6)
[]
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).cells_of_heads(1)
[]
sage: StrongTableau([],4).cells_of_heads(1)
[]
```

cells_of_marked_ribbon(*v*)

Return a list of all cells the marked ribbon labeled by v in the standard part of self.

Return the list of coordinates of the cells which are in the marked ribbon with label v in the standard part of the tableau. Note that the result is independent of the weight of the tableau.

The cells are listed from largest content (where the mark is located) to the smallest. Hence, the first entry in this list will be the marked cell.

INPUT:

- v – the entry of the standard tableau

OUTPUT:

- a list of pairs representing the coordinates of the cells of the marked ribbon

EXAMPLES:

```
sage: T = StrongTableau([[-1, -1, -2, -2, 3], [2, -3], [-3]],3)
sage: T.to_standard_list()
[[-1, -2, -3, -4, 6], [4, -6], [-5]]
sage: T.cells_of_marked_ribbon(1)
[(0, 0)]
sage: T.cells_of_marked_ribbon(4)
[(0, 3)]
sage: T = StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3)
sage: T.cells_of_marked_ribbon(6)
[(1, 2), (1, 1)]
sage: T.cells_of_marked_ribbon(9)
[]
sage: T = StrongTableau([[None, None, -1, -1, 3], [1, -3], [-3]],3)
sage: T.to_standard_list()
[[None, None, -1, -2, 4], [2, -4], [-3]]
sage: T.cells_of_marked_ribbon(1)
[(0, 2)]
```

TESTS:

```
sage: StrongTableau([],3).cells_of_marked_ribbon(1)
[]
```

check()

Check that `self` is a valid strong k-tableau.

This function verifies that the outer and inner shape of the parent class is equal to the outer and inner shape of the tableau, that the tableau portion of `self` is a valid standard tableau, that the marks are placed correctly and that the size and weight agree.

EXAMPLES:

```
sage: T = StrongTableau([[-1, -1, -2], [2]], 2)
sage: T.check()
sage: T = StrongTableau([[None, None, 2, -4, -4], [-1, 4], [-2]], 3)
sage: T.check()
```

TESTS:

```
sage: ST = StrongTableaux(2, [3,1], [1,1,1,1])
sage: ST([[-1,-2,3],[-3]])
Traceback (most recent call last):
...
ValueError: The size of the tableau [[-1, -2, 3], [-3]] and weight (1, 1, 1, 1) do not match
sage: ST([[-1,-3],[-2],[3]])
Traceback (most recent call last):
...
ValueError: The outer shape of the parent does not agree with the outer shape of the tableau

sage: StrongTableau([[-1, -2, 2], [1]], 2)
Traceback (most recent call last):
...
ValueError: The marks in [[-1, -2, 2], [1]] are not correctly placed.

sage: StrongTableau([[-1, -2, 3], [3]], 2)
Traceback (most recent call last):
...
ValueError: The marks in [[-1, -2, 3], [3]] are not correctly placed.

sage: StrongTableau([[-1,-2,-4,7],[-3,6,-6,8],[4,-7],[-5,-8]], 3, [2,2,3,1])
Traceback (most recent call last):
...
ValueError: The weight=(2, 2, 3, 1) and the markings on the standard tableau=[[-1, -2, -4, 7
```

content_of_highest_head(v)

Return the diagonal of the highest head of the cells labeled `v` in the standard part of `self`.

Return the content of the cell of the head in the highest row of all ribbons labeled by `v` of the underlying standard tableau. If there is no cell with entry `v` then the value returned is the length of the first row.

INPUT:

- `v` – an integer representing the label in the standard tableau

OUTPUT:

- an integer representing the content of the head of the highest ribbon with label `v`

EXAMPLES:

```
sage: [StrongTableau([[-1,2,-3],[-2,3],[3]], 1).content_of_highest_head(v) for v in range(1,
[0, -1, -2, 3]
```

TESTS:

```
sage: StrongTableau([], 4).content_of_highest_head(1)
0
sage: StrongTableau([[-1,-1]], 4).content_of_highest_head(3)
2
```

content_of_marked_head(*v*)

Return the diagonal of the marked label `v` in the standard part of `self`.

Return the content (the $j - i$ coordinate of the cell) of the `v`-th marked cell in the strong standard tableau `self`. If there is no mark, then the value returned is the size of first row.

INPUT:

- `v` – an integer representing the label in the standard tableau

OUTPUT:

- an integer representing the residue of the location of the mark

EXAMPLES:

```
sage: [ StrongTableau([[-1, -3, 4, -5], [-2], [-4]], 3).content_of_marked_head(i) for i in r
[0, -1, 1, -2, 3, 4]
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3
sage: [ T.content_of_marked_head(i) for i in range(1,7)]
[-2, 2, -1, 3, -4, 4]
```

TESTS:

```
sage: StrongTableau([],4).content_of_marked_head(4)
0
```

contents_of_heads(*v*)

A list of contents of the cells which are heads of the ribbons with label `v`.

If there is no cell labelled by `v` then return the empty list.

INPUT:

- `v` – an integer label

OUTPUT:

- a list of integers of the content of the heads of the ribbons with label `v`

EXAMPLES:

```
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3
sage: T.contents_of_heads(1)
[-2]
sage: T.contents_of_heads(2)
[-3, 2]
sage: T.contents_of_heads(3)
[-1]
sage: T.contents_of_heads(4)
[-2, 3]
sage: T.contents_of_heads(5)
[-4]
```

```
sage: T.contents_of_heads(6)
[]
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).contents_of_heads(1)
[]
sage: StrongTableau([],4).contents_of_heads(1)
[]
```

entries_by_content(*diag*)

Return the entries on the diagonal of `self`.

Return the entries in the tableau that are in the cells (i, j) with $j - i$ equal to `diag` (that is, with content equal to `diag`).

INPUT:

- `diag` – an integer indicating the diagonal

OUTPUT:

- a list (perhaps empty) of labels on the diagonal `diag`

EXAMPLES:

```
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3
sage: T.entries_by_content(0)
[]
sage: T.entries_by_content(1)
[]
sage: T.entries_by_content(2)
[-1]
sage: T.entries_by_content(-2)
[-1, 2]
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).entries_by_content(1)
[]
sage: StrongTableau([],4).entries_by_content(1)
[]
```

entries_by_content_standard(*diag*)

Return the entries on the diagonal of the standard part of `self`.

Return the entries in the tableau that are in the cells (i, j) with $j - i$ equal to `diag` (that is, with content equal to `diag`) in the standard tableau.

INPUT:

- `diag` – an integer indicating the diagonal

OUTPUT:

- a list (perhaps empty) of labels on the diagonal `diag`

EXAMPLES:

```
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3
sage: T.entries_by_content_standard(0)
[]
sage: T.entries_by_content_standard(1)
```

```
[]
sage: T.entries_by_content_standard(2)
[-2]
sage: T.entries_by_content_standard(-2)
[-1, 4]
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).entries_by_content_standard(1)
[]
sage: StrongTableau([],4).entries_by_content_standard(1)
[]
```

follows_tableau()

Return a list of strong marked tableaux with length one longer than `self`.

Return list of all strong tableaux obtained from `self` by extending to a core which follows the shape of `self` in the strong order.

OUTPUT:

- a list of strong tableaux which follow `self` in strong order

EXAMPLES:

```
sage: T = StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3, [2,2,3,1])
sage: T.follows_tableau()
[[[-1, -1, -2, -3, 5, 5, -5], [-2, 3, -3, 4], [2, 3], [-3, -4]],
 [[-1, -1, -2, -3, 5], [-2, 3, -3, 4], [2, 3, 5], [-3, -4], [-5]],
 [[-1, -1, -2, -3, 5], [-2, 3, -3, 4], [2, 3, -5], [-3, -4], [5]],
 [[-1, -1, -2, -3, -5], [-2, 3, -3, 4], [2, 3, 5], [-3, -4], [5]],
 [[-1, -1, -2, -3], [-2, 3, -3, 4], [2, 3], [-3, -4], [-5], [5], [5]]]
sage: StrongTableau([[-1,-2],[-3,-4]],3).follows_tableau()
[[[-1, -2, 5, 5, -5], [-3, -4]], [[-1, -2, 5], [-3, -4], [-5]],
 [[-1, -2, -5], [-3, -4], [5]], [[-1, -2], [-3, -4], [-5], [5], [5]]]
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).follows_tableau()
[[[None, None, -1], [None]], [[None, None], [None, -1]], [[None, None], [None], [-1]]]
sage: StrongTableau([],4).follows_tableau()
[[[-1]]]
```

height_of_ribbon(*v*)

The number of rows occupied by one of the ribbons with label `v`.

The number of rows occupied by the marked ribbon with label `v` (and by consequence the number of rows occupied by any ribbon with the same label) in the standard part of `self`.

INPUT:

- `v` – the label of the standard marked tableau

OUTPUT:

- a non-negative integer representing the number of rows occupied by the ribbon which is marked

EXAMPLES:

```
sage: T = StrongTableau([[-1, -1, -2, -2, 3], [2, -3], [-3]],3)
sage: T.to_standard_list()
[[-1, -2, -3, -4, 6], [4, -6], [-5]]
sage: T.height_of_ribbon(1)
```

```
1
sage: T.height_of_ribbon(4)
1
sage: T = StrongTableau([[None,None,1,-2],[None,-3,4,-5],[-1,3],[-4,5]], 3)
sage: T.height_of_ribbon(3)
2
sage: T.height_of_ribbon(6)
0
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).height_of_ribbon(1)
0
sage: StrongTableau([],4).height_of_ribbon(1)
0
```

inner_shape()

Return the inner shape of `self`.

If `self` is a strong skew tableau, then this method returns the inner shape (the shape of the cells labelled with `None`). If `self` is not skew, then the inner shape is empty.

OUTPUT:

- a $(k+1)$-core

EXAMPLES:

```
sage: StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3], [
[2, 2]
sage: StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3, [2,2,3,1]).inner_shape()
[]
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).inner_shape()
[2, 1]
sage: StrongTableau([],4).inner_shape()
[]
```

intermediate_shapes()

Return the intermediate shapes of `self`.

A (skew) tableau with letters $1, 2, \ldots, \ell$ can be viewed as a sequence of shapes, where the i-th shape is given by the shape of the subtableau on letters $1, 2, \ldots, i$.

The output is the list of these shapes. The marked cells are ignored so to recover the strong tableau one would need the intermediate shapes and the `content_of_marked_head()` for each pair of adjacent shapes in the list.

OUTPUT:

- a list of lists of integers representing $k+1$-cores

EXAMPLES:

```
sage: T = StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3, [2,2,3,1])
sage: T.intermediate_shapes()
[[], [2], [3, 1, 1], [4, 3, 2, 1], [4, 4, 2, 2]]
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3
sage: T.intermediate_shapes()
[[2, 2], [3, 2, 1, 1], [4, 2, 2, 2], [4, 2, 2, 2, 1, 1, 1, 1]]
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).intermediate_shapes()
[[2, 1]]
sage: StrongTableau([],4).intermediate_shapes()
[[]]
```

is_column_strict_with_weight(*mu*)

Test if `self` is a column strict tableau with respect to the weight `mu`.

INPUT:

- `mu` – a vector of weights

OUTPUT:

- a boolean, `True` means the underlying column strict strong marked tableau is valid

EXAMPLES:

```
sage: StrongTableau([[-1, -2, -3], [3]], 2).is_column_strict_with_weight([3])
True
sage: StrongTableau([[-1, -2, 3], [-3]], 2).is_column_strict_with_weight([3])
False
```

TESTS:

```
sage: StrongTableau([[None, None, None], [None]], 2).is_column_strict_with_weight([])
True
sage: StrongTableau([], 4).is_column_strict_with_weight([])
True
```

left_action(*tij*)

Action of transposition `tij` on `self` by adding marked ribbons.

Computes the left action of the transposition `tij` on the tableau. If `tij` acting on the element of the affine grassmannian raises the length by 1, then this function will add a cell to the standard tableau.

INPUT:

- `tij` – a transposition represented as a pair (i, j).

OUTPUT:

- `self` after it has been modified by the action of the transposition `tij`

EXAMPLES:

```
sage: StrongTableau( [[None, -1, -2, -3], [3], [-4]], 3, weight=[1,1,1,1] ).left_action([0,1
[[None, -1, -2, -3, 5], [3, -5], [-4]]
sage: StrongTableau( [[None, -1, -2, -3], [3], [-4]], 3, weight=[1,1,1,1] ).left_action([4,5
[[None, -1, -2, -3, -5], [3, 5], [-4]]
sage: T = StrongTableau( [[None, -1, -2, -3], [3], [-4]], 3, weight=[1,1,1,1] )
sage: T.left_action([-3,-2])
[[None, -1, -2, -3], [3], [-4], [-5]]
sage: T = StrongTableau( [[None, -1, -2, -3], [3], [-4]], 3, weight=[3,1] )
sage: T.left_action([-3,-2])
[[None, -1, -1, -1], [1], [-2], [-3]]
sage: T
[[None, -1, -1, -1], [1], [-2]]
sage: T.check()
sage: T.weight()
(3, 1)
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).left_action([-2,-1])
[[None, None], [None], [-1]]
sage: StrongTableau([],4).left_action([0,1])
[[-1]]
```

number_of_connected_components(*v*)

Number of connected components of ribbons with label `v` in the standard part.

The number of connected components is calculated by finding the number of cells with label `v` in the standard part of the tableau and dividing by the number of cells in the ribbon.

INPUT:

- `v` – the label of the standard marked tableau

OUTPUT:

- a non-negative integer representing the number of connected components

EXAMPLES:

```
sage: T = StrongTableau([[-1, -1, -2, -2, 3], [2, -3], [-3]],3)
sage: T.to_standard_list()
[[-1, -2, -3, -4, 6], [4, -6], [-5]]
sage: T.number_of_connected_components(1)
1
sage: T.number_of_connected_components(4)
2
sage: T = StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3)
sage: T.number_of_connected_components(6)
1
sage: T.number_of_connected_components(9)
0
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).number_of_connected_components(1)
0
sage: StrongTableau([],4).number_of_connected_components(1)
0
```

outer_shape()

Return the outer shape of `self`.

This method returns the outer shape of `self` as viewed as a `Core`. The outer shape of a strong tableau is always a $(k+1)$-core.

OUTPUT:

- a $(k+1)$-core

EXAMPLES:

```
sage: StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3], [
[4, 2, 2, 2, 1, 1, 1, 1]
sage: StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3, [2,2,3,1]).outer_shape()
[4, 4, 2, 2]
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).outer_shape()
[2, 1]
```

```
sage: StrongTableau([],4).outer_shape()
[]
```

pp()

Print the strong tableau `self` in pretty print format.

EXAMPLES:

```
sage: T = StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3, [2,2,3,1])
sage: T.pp()
-1 -1 -2 -3
-2  3 -3  4
 2  3
-3 -4
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3
sage: T.pp()
  .  . -1 -2
  .  .
 -1 -2
  1  2
 -3
  3
  3
  3
sage: Tableaux.global_options(convention="French")
sage: T.pp()
  3
  3
  3
 -3
  1  2
 -1 -2
  .  .
  .  . -1 -2
sage: Tableaux.global_options(convention="English")
```

restrict(*r*)

Restrict the standard part of the tableau to the labels $1, 2, \ldots, r$.

Return the tableau consisting of the labels of the standard part of `self` restricted to the labels of 1 through `r`. The result is another `StrongTableau` object.

INPUT:

- `r` – an integer

OUTPUT:

- A strong tableau

EXAMPLES:

```
sage: T = StrongTableau([[None, None, -4, 5, -5], [None, None], [-1, -3], [-2], [2], [2], [3
sage: T.restrict(3)
[[None, None], [None, None], [-1, -3], [-2], [2], [2], [3]]
sage: TT = T.restrict(0)
sage: TT
[[None, None], [None, None]]
sage: TT == StrongTableau( [[None, None], [None, None]], 4 )
True
```

```
sage: T.restrict(5) == T
True
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).restrict(1)
[[None, None], [None]]
sage: StrongTableau([],4).restrict(1)
[]
```

ribbons_above_marked(*v*)

Number of ribbons of label v higher than the marked ribbon in the standard part.

Return the number of copies of the ribbon with label v in the standard part of self which are in a higher row than the marked ribbon. Note that the result is independent of the weight of the tableau.

INPUT:

- v – the entry of the standard tableau

OUTPUT:

- an integer representing the number of copies of the ribbon above the marked ribbon

EXAMPLES:

```
sage: T = StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3)
sage: T.ribbons_above_marked(4)
1
sage: T.ribbons_above_marked(6)
0
sage: T.ribbons_above_marked(9)
0
sage: StrongTableau([[-1,-2,-3,-4],[2,3,4],[3,4],[4]], 1).ribbons_above_marked(4)
3
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).ribbons_above_marked(1)
0
sage: StrongTableau([],4).ribbons_above_marked(1)
0
```

set_weight(*mu*)

Sets a new weight mu for self.

This method first tests if the underlying standard tableau is column-strict with respect to the weight mu. If it is, then it changes the weight and returns the tableau; otherwise it raises an error.

INPUT:

- mu – a list of non-negative integers representing the new weight

EXAMPLES:

```
sage: StrongTableau( [[-1, -2, -3], [3]], 2 ).set_weight( [3] )
[[-1, -1, -1], [1]]
sage: StrongTableau( [[-1, -2, -3], [3]], 2 ).set_weight( [0,3] )
[[-2, -2, -2], [2]]
sage: StrongTableau( [[-1, -2, 3], [-3]], 2 ).set_weight( [2, 0, 1] )
[[-1, -1, 3], [-3]]
sage: StrongTableau( [[-1, -2, 3], [-3]], 2 ).set_weight( [3] )
Traceback (most recent call last):
```

```
...
ValueError: [[-1, -2, 3], [-3]] is not a semistandard strong tableau with respect to the par
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).set_weight([])
[[None, None], [None]]
sage: StrongTableau([],4).set_weight([])
[]
```

shape()

Return the shape of `self`.

If `self` is a skew tableau then return a pair of $k+1$-cores consisting of the outer and the inner shape. If `self` is strong tableau with no inner shape then return a $k+1$-core.

INPUT:

- `form` - optional argument to indicate 'inner', 'outer' or 'skew' (default : 'outer')

OUTPUT:

- a $k+1$-core or a pair of $k+1$-cores if form is not 'inner' or 'outer'

EXAMPLES:

```
sage: T = StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3
sage: T.shape()
([4, 2, 2, 2, 1, 1, 1, 1], [2, 2])
sage: StrongTableau([[-1, -2, 3], [-3]], 2).shape()
[3, 1]
sage: type(StrongTableau([[-1, -2, 3], [-3]], 2).shape())
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
```

TESTS:

```
sage: StrongTableau([[None, None, None], [None]], 2).shape()
([3, 1], [3, 1])
sage: StrongTableau([],4).shape()
[]
```

size()

Return the size of the strong tableau.

The size of the strong tableau is the sum of the entries in the `weight()`. It will also be equal to the length of the outer shape (as a $k+1$-core) minus the length of the inner shape.

See also:

`sage.combinat.core.Core.length()`

OUTPUT:

- a non-negative integer

EXAMPLES:

```
sage: StrongTableau([[-1, -2, -3, 4], [-4], [-5]], 3).size()
5
sage: StrongTableau([[None, None, -1, 2], [-2], [-3]], 3).size()
3
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).size()
0
sage: StrongTableau([],4).size()
0
```

spin()

Return the spin statistic of the tableau `self`.

The spin is an integer statistic on a strong marked tableau. It is the sum of $(h-1)r$ plus the number of connected components above the marked one where h is the height of the marked ribbon and r is the number of connected components.

See also:

`height_of_ribbon()`, `number_of_connected_components()`, `ribbons_above_marked()`

The k-Schur functions with a parameter t can be defined as

$$s_\lambda^{(k)}[X;t] = \sum_T t^{spin(T)} m_{weight(T)}[X]$$

where the sum is over all column strict marked strong k-tableaux of shape λ and partition content.

OUTPUT:

- an integer value representing the spin.

EXAMPLES:

```
sage: StrongTableau([[-1,-2,5,6],[-3,-4,-7,8],[-5,-6],[7,-8]], 3, [2,2,3,1]).spin()
1
sage: StrongTableau([[-1,-2,-4,-7],[-3,6,-6,8],[4,7],[-5,-8]], 3, [2,2,3,1]).spin()
2
sage: StrongTableau([[None,None,-1,-3],[-2,3,-3,4],[2,3],[-3,-4]], 3).spin()
2
sage: ks3 = SymmetricFunctions(QQ['t'].fraction_field()).kschur(3)
sage: t = ks3.realization_of().t
sage: m = ks3.ambient().realization_of().m()
sage: myks221 = sum(sum(t**T.spin() for T in StrongTableaux(3,[3,2,1],weight=mu))*m(mu) for
sage: myks221 == m(ks3[2,2,1])
True
sage: h = ks3.ambient().realization_of().h()
sage: Core([4,4,2,2],4).to_bounded_partition()
[2, 2, 2, 2]
sage: ks3[2,2,2,2].lift().scalar(h[3,3,2]) == sum( t**T.spin() for T in StrongTableaux(3, [4
True
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).spin()
0
sage: StrongTableau([],4).spin()
0
```

spin_of_ribbon(v)

Return the spin of the ribbon with label `v` in the standard part of `self`.

The spin of a ribbon is an integer statistic. It is the sum of $(h-1)r$ plus the number of connected components above the marked one where h is the height of the marked ribbon and r is the number of connected components.

See also:

height_of_ribbon(), number_of_connected_components(), ribbons_above_marked()

INPUT:

- v – a label of the standard part of the tableau

OUTPUT:

- an integer value representing the spin of the ribbon with label v.

EXAMPLES:

```
sage: T = StrongTableau([[-1,-2,5,6],[-3,-4,-7,8],[-5,-6],[7,-8]], 3)
sage: [T.spin_of_ribbon(v) for v in range(1,9)]
[0, 0, 0, 0, 0, 0, 1, 0]
sage: T = StrongTableau([[None,None,-1,-3],[-2,3,-3,4],[2,3],[-3,-4]], 3)
sage: [T.spin_of_ribbon(v) for v in range(1,7)]
[0, 1, 0, 0, 1, 0]
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).spin_of_ribbon(1)
0
sage: StrongTableau([],4).spin_of_ribbon(1)
0
```

to_list()

Return the marked column strict (possibly skew) tableau as a list of lists.

OUTPUT:

- a list of lists of integers or None

EXAMPLES:

```
sage: StrongTableau([[-1, -2, -3, 4], [-4], [-5]], 3).set_weight([2,1,1,1]).to_list()
[[-1, -1, -2, 3], [-3], [-4]]
sage: StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3], [
[[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3], [3]]
sage: StrongTableau([[-1, -2, -3, 4], [-4], [-5]], 3, [3,1,1]).to_list()
[[-1, -1, -1, 2], [-2], [-3]]
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).to_list()
[[None, None], [None]]
sage: StrongTableau([],4).to_list()
[]
```

to_standard_list()

Return the underlying standard strong tableau as a list of lists.

Internally, for a strong tableau the standard strong tableau and its weight is stored separately. This method returns the underlying standard part.

OUTPUT:

- a list of lists of integers or None

EXAMPLES:

```
sage: StrongTableau([[-1, -2, -3, 4], [-4], [-5]], 3, [3,1,1]).to_standard_list()
[[-1, -2, -3, 4], [-4], [-5]]
sage: StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3], [
[[None, None, -2, -4], [None, None], [-1, -3], [2, 4], [-5], [5], [5], [5]]
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).to_standard_list()
[[None, None], [None]]
sage: StrongTableau([],4).to_standard_list()
[]
```

to_standard_tableau()

Return the underlying standard strong tableau as a `StrongTableau` object.

Internally, for a strong tableau the standard strong tableau and its weight is stored separately. This method returns the underlying standard part as a `StrongTableau`.

OUTPUT:

- a strong tableau with standard weight

EXAMPLES:

```
sage: T = StrongTableau([[-1, -2, -3, 4], [-4], [-5]], 3, [3,1,1])
sage: T.to_standard_tableau()
[[-1, -2, -3, 4], [-4], [-5]]
sage: T.to_standard_tableau() == T.to_standard_list()
False
sage: StrongTableau([[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3], [
[[None, None, -2, -4], [None, None], [-1, -3], [2, 4], [-5], [5], [5], [5]]
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).to_standard_tableau()
[[None, None], [None]]
sage: StrongTableau([],4).to_standard_tableau()
[]
```

to_transposition_sequence()

Return a list of transpositions corresponding to `self`.

Given a strong column strict tableau `self` returns the list of transpositions which when applied to the left of an empty tableau gives the corresponding strong standard tableau.

OUTPUT:

- a list of pairs of values `[i,j]` representing the transpositions t_{ij}

EXAMPLES:

```
sage: T = StrongTableau([[-1, -1, -1], [1]],2)
sage: T.to_transposition_sequence()
[[2, 3], [1, 2], [0, 1]]
sage: T = StrongTableau([[-1, -1, 2], [-2]],2)
sage: T.to_transposition_sequence()
[[-1, 0], [1, 2], [0, 1]]
sage: T = StrongTableau([[None, -1, 2, -3], [-2, 3]],2)
sage: T.to_transposition_sequence()
[[3, 4], [-1, 0], [1, 2]]
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).to_transposition_sequence()
[]
sage: StrongTableau([],4).to_transposition_sequence()
[]
```

to_unmarked_list()

Return the tableau as a list of lists with markings removed.

Return the list of lists of the rows of the tableau where the markings have been removed.

OUTPUT:

- a list of lists of integers or `None`

EXAMPLES:

```
sage: T = StrongTableau( [[-1, -2, -3, 4], [-4], [-5]], 3, [3,1,1])
sage: T.to_unmarked_list()
[[1, 1, 1, 2], [2], [3]]
sage: TT = T.set_weight([2,1,1,1])
sage: TT.to_unmarked_list()
[[1, 1, 2, 3], [3], [4]]
sage: StrongTableau( [[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3],
[[None, None, 1, 2], [None, None], [1, 2], [1, 2], [3], [3], [3], [3]]
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).to_unmarked_list()
[[None, None], [None]]
sage: StrongTableau([],4).to_unmarked_list()
[]
```

to_unmarked_standard_list()

Return the standard part of the tableau as a list of lists with markings removed.

Return the list of lists of the rows of the tableau where the markings have been removed.

OUTPUT:

- a list of lists of integers or `None`

EXAMPLES:

```
sage: StrongTableau( [[-1, -2, -3, 4], [-4], [-5]], 3, [3,1,1]).to_unmarked_standard_list()
[[1, 2, 3, 4], [4], [5]]
sage: StrongTableau( [[None, None, -1, -2], [None, None], [-1, -2], [1, 2], [-3], [3], [3],
[[None, None, 2, 4], [None, None], [1, 3], [2, 4], [5], [5], [5], [5]]
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).to_unmarked_standard_list()
[[None, None], [None]]
sage: StrongTableau([],4).to_unmarked_standard_list()
[]
```

weight()

Return the weight of the tableau.

The weight is a list of non-negative integers indicating the number of 1s, number of 2s, number of 3s, etc.

OUTPUT:

- a list of non-negative integers

EXAMPLES:

```
sage: T = StrongTableau([[-1, -2, -3, 4], [-4], [-5]], 3); T.weight()
(1, 1, 1, 1, 1)
sage: T.set_weight([3,1,1]).weight()
(3, 1, 1)
sage: StrongTableau([[-1,-1,-2,-3],[-2,3,-3,4],[2,3],[-3,-4]], 3).weight()
(2, 2, 3, 1)
```

TESTS:

```
sage: StrongTableau([[None, None], [None]], 4).weight()
()
sage: StrongTableau([],4).weight()
()
```

class sage.combinat.k_tableau.**StrongTableaux**(*k*, *shape*, *weight*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

TESTS:

```
sage: strongT = StrongTableaux(2, [3,1], weight=[2,1])
sage: TestSuite(strongT).run()

sage: strongT = StrongTableaux(0, [2,2], weight=[2,2])
Traceback (most recent call last):
...
ValueError: The input k has to be a positive integer
```

Element

alias of StrongTableau

classmethod add_marking(*unmarkedT*, *marking*, *k*, *weight*)

Add markings to a partially marked strong tableau.

Given an partially marked standard tableau and a list of cells where the marks should be placed along with a weight, return the semi-standard marked strong tableau. The marking should complete the marking so that the result is a strong standard marked tableau.

INPUT:

- unmarkedT - a list of lists which is a partially marked strong k-tableau
- marking - a list of pairs of coordinates where cells are to be marked
- k - a positive integer
- weight - a tuple of the weight of the output tableau

OUTPUT:

- a StrongTableau object

EXAMPLES:

```
sage: StrongTableaux.add_marking([[None,1,2],[2]], [(0,1), (1,0)], 2, [1,1])
[[None, -1, 2], [-2]]
sage: StrongTableaux.add_marking([[None,1,2],[2]], [(0,1), (1,0)], 2, [2])
Traceback (most recent call last):
...
ValueError: The weight=(2,) and the markings on the standard tableau=[[None, -1, 2], [-2]] c
sage: StrongTableaux.add_marking([[None,1,2],[2]], [(0,1), (0,2)], 2, [2])
[[None, -1, -1], [1]]
```

TESTS:

```
sage: StrongTableaux.add_marking([[None,None,None],[None]], [], 2, [])
[[None, None, None], [None]]
sage: StrongTableaux.add_marking([], [], 2, [])
[]
```

an_element()

Return the first generated element of the class of StrongTableaux.

EXAMPLES:

```
sage: ST = StrongTableaux(3, [3], weight=[3])
sage: ST.an_element()
[[-1, -1, -1]]
```

classmethod cells_head_dictionary(*T*)

Return a dictionary with the locations of the heads of all markings.

Return a dictionary of values and lists of cells where the heads with the values are located in a strong standard unmarked tableau T.

INPUT:

- T – a strong standard unmarked tableau as a list of lists

OUTPUT:

- a dictionary with keys the entries in the tableau and values are the coordinates of the heads with those entries

EXAMPLES:

```
sage: StrongTableaux.cells_head_dictionary([[1,2,4,7],[3,6,6,8],[4,7],[5,8]])
{1: [(0, 0)],
 2: [(0, 1)],
 3: [(1, 0)],
 4: [(2, 0), (0, 2)],
 5: [(3, 0)],
 6: [(1, 2)],
 7: [(2, 1), (0, 3)],
 8: [(3, 1), (1, 3)]}
sage: StrongTableaux.cells_head_dictionary([[None, 2, 2, 4, 5, 6, 6, 6], [None, 3, 6, 6, 6],
{1: [(2, 0)],
 2: [(0, 2)],
 3: [(1, 1)],
 4: [(2, 1), (0, 3)],
 5: [(0, 4)],
 6: [(1, 4), (0, 7)]}
```

TESTS:

```
sage: StrongTableaux.cells_head_dictionary([[None, None, None],[None]])
{}
sage: StrongTableaux.cells_head_dictionary([])
{}
```

classmethod follows_tableau_unsigned_standard(*Tlist*, *k*)

Return a list of strong tableaux one longer in length than Tlist.

Return list of all standard strong tableaux obtained from `Tlist` by extending to a core which follows the shape of `Tlist` in the strong order. It does not put the markings on the last entry that it adds but it does keep the markings on all entries smaller. The objects returned are not `StrongTableau` objects (and cannot be) because the last entry will not properly marked.

INPUT:

- `Tlist` – a filling of a $k+1$-core as a list of lists
- `k` - an integer

OUTPUT:

- a list of strong tableaux which follow `Tlist` in strong order

EXAMPLES:

```
sage: StrongTableaux.follows_tableau_unsigned_standard([[-1, -1, -2, -3], [-2, 3, -3, 4], [2
[[[-1, -1, -2, -3, 5, 5, 5], [-2, 3, -3, 4], [2, 3], [-3, -4]],
 [[-1, -1, -2, -3, 5], [-2, 3, -3, 4], [2, 3, 5], [-3, -4], [5]],
 [[-1, -1, -2, -3], [-2, 3, -3, 4], [2, 3], [-3, -4], [5], [5], [5]]]
sage: StrongTableaux.follows_tableau_unsigned_standard([[None,-1],[-2,-3]],3)
[[[None, -1, 4, 4, 4], [-2, -3]], [[None, -1, 4], [-2, -3], [4]],
 [[None, -1], [-2, -3], [4], [4], [4]]]
```

TESTS:

```
sage: StrongTableaux.follows_tableau_unsigned_standard([[None, None, None], [None]], 2)
[[[None, None, None, 1], [None, 1]], [[None, None, None], [None], [1]]]
sage: StrongTableaux.follows_tableau_unsigned_standard([], 4)
[[[1]]]
```

global_options(**get_value*, ***set_value*)

Sets the global options for elements of the tableau, skew_tableau, and tableau tuple classes. The defaults are for tableau to be displayed as a list, latexed as a Young diagram using the English convention.

OPTIONS:

- `ascii_art` – (default: `repr`) Controls the ascii art output for tableaux
  - `compact` – minimal length ascii art
  - `repr` – display using the diagram string representation
  - `table` – display as a table
- `convention` – (default: `English`) Sets the convention used for displaying tableaux and partitions
  - `English` – use the English convention
  - `French` – use the French convention
- `display` – (default: `list`) Controls the way in which tableaux are printed
  - `array` – alias for `diagram`
  - `compact` – minimal length string representation
  - `diagram` – display as Young diagram (similar to `pp()`
  - `ferrers_diagram` – alias for `diagram`
  - `list` – print tableaux as lists
  - `young_diagram` – alias for `diagram`
- `latex` – (default: `diagram`) Controls the way in which tableaux are latexed

– `array` – alias for `diagram`

– `diagram` – as a Young diagram

– `ferrers_diagram` – alias for `diagram`

– `list` – as a list

– `young_diagram` – alias for `diagram`

- `notation` – alternative name for `convention`

Note: Changing the `convention` for tableaux also changes the `convention` for partitions.

If no parameters are set, then the function returns a copy of the options dictionary.

EXAMPLES:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T
[[1, 2, 3], [4, 5]]
sage: Tableaux.global_options(display="array")
sage: T
  1  2  3
  4  5
sage: Tableaux.global_options(convention="french")
sage: T
  4  5
  1  2  3
```

Changing the `convention` for tableaux also changes the `convention` for partitions and vice versa:

```
sage: P = Partition([3,3,1])
sage: print P.ferrers_diagram()
*
***
***
sage: Partitions.global_options(convention="english")
sage: print P.ferrers_diagram()
***
***
*
sage: T
  1  2  3
  4  5
```

The ASCII art can also be changed:

```
sage: t = Tableau([[1,2,3],[4,5]])
sage: ascii_art(t)
  1  2  3
  4  5
sage: Tableaux.global_options(ascii_art="table")
sage: ascii_art(t)
+---+---+
| 4 | 5 |
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
sage: Tableaux.global_options(ascii_art="compact")
sage: ascii_art(t)
|4|5|
```

```
|1|2|3|
sage: Tableaux.global_options.reset()
```

See `GlobalOptions` for more features of these options.

inner_shape()

Return the inner shape of the class of strong tableaux.

OUTPUT:

- a $k+1$-core

EXAMPLES:

```
sage: StrongTableaux( 2, [3,1] ).inner_shape()
[]
sage: type(StrongTableaux( 2, [3,1] ).inner_shape())
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
sage: StrongTableaux( 4, [[2,1], [1]] ).inner_shape()
[1]
```

classmethod marked_CST_to_transposition_sequence(T, k)

Return a list of transpositions corresponding to `T`.

Given a strong column strict tableau `T` returns the list of transpositions which when applied to the left of an empty tableau gives the corresponding strong standard tableau.

INPUT:

- `T` – a non-empty column strict tableau as a list of lists
- `k` – a positive integer

OUTPUT:

- a list of pairs of values `[i,j]` representing the transpositions t_{ij}

EXAMPLES:

```
sage: CST_to_trans = StrongTableaux.marked_CST_to_transposition_sequence
sage: CST_to_trans([[-1, -1, -1], [1]], 2)
[[2, 3], [1, 2], [0, 1]]
sage: CST_to_trans([], 2)
[]
sage: CST_to_trans([[-2, -2, -2], [2]], 2)
[[2, 3], [1, 2], [0, 1]]
sage: CST_to_trans([[-1, -2, -2, -2, -2], [-2, 2], [2]], 3)
[[4, 5], [3, 4], [2, 3], [1, 2], [-1, 0], [0, 1]]
sage: CST_to_trans([[-1, -2, -5, 5, -5, 5, -5], [-3, -4, 5, 5], [5]],3)
[[5, 7], [3, 5], [2, 3], [0, 1], [-1, 0], [1, 2], [0, 1]]
sage: CST_to_trans([[-1, -2, -3, 4, -7], [-4, -6], [-5, 6]],3)
[[4, 5], [-1, 1], [-2, -1], [-1, 0], [2, 3], [1, 2], [0, 1]]
```

TESTS:

```
sage: StrongTableaux.marked_CST_to_transposition_sequence([[None, None, None], [None]], 2)
[]
sage: StrongTableaux.marked_CST_to_transposition_sequence([], 4)
[]
```

classmethod marked_given_unmarked_and_weight_iterator(*unmarkedT*, *k*, *weight*)

An iterator generating strong marked tableaux from an unmarked strong tableau.

Iterator which lists all marked tableaux of weight `weight` such that the standard unmarked part of the tableau is equal to `unmarkedT`.

INPUT:

- `unmarkedT` - a list of lists representing a strong unmarked tableau
- `k` - a positive integer
- `weight` - a list of non-negative integers indicating the weight

OUTPUT:

- an iterator that returns `StrongTableau` objects

EXAMPLES:

```
sage: ST = StrongTableaux.marked_given_unmarked_and_weight_iterator([[1,2,3],[3]], 2, [3])
sage: list(ST)
[[[-1, -1, -1], [1]]]
sage: ST = StrongTableaux.marked_given_unmarked_and_weight_iterator([[1,2,3],[3]], 2, [0,3])
sage: list(ST)
[[[-2, -2, -2], [2]]]
sage: ST = StrongTableaux.marked_given_unmarked_and_weight_iterator([[1,2,3],[3]], 2, [1,2])
sage: list(ST)
[[[-1, -2, -2], [2]]]
sage: ST = StrongTableaux.marked_given_unmarked_and_weight_iterator([[1,2,3],[3]], 2, [2,1])
sage: list(ST)
[[[-1, -1, 2], [-2]], [[-1, -1, -2], [2]]]
sage: ST = StrongTableaux.marked_given_unmarked_and_weight_iterator([[None, None, 1, 2, 4],
sage: list(ST)
[]
sage: ST = StrongTableaux.marked_given_unmarked_and_weight_iterator([[None, None, 1, 2, 4],
sage: list(ST)
[[[None, None, -1, -1, 2], [1, -2], [-2]],
 [[None, None, -1, -1, -2], [1, 2], [-2]]]
```

TESTS:

```
sage: list(StrongTableaux.marked_given_unmarked_and_weight_iterator([[None, None, None],[Nor
[[[None, None, None], [None]]]
sage: list(StrongTableaux.marked_given_unmarked_and_weight_iterator([], 4, weight=[]))
[[]]
```

outer_shape()

Return the outer shape of the class of strong tableaux.

OUTPUT:

- a $k+1$-core

EXAMPLES:

```
sage: StrongTableaux( 2, [3,1] ).outer_shape()
[3, 1]
sage: type(StrongTableaux( 2, [3,1] ).outer_shape())
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
sage: StrongTableaux( 4, [[2,1], [1]] ).outer_shape()
[2, 1]
```

shape()

Return the shape of `self`.

If the `self` has an inner shape return a pair consisting of an inner and an outer shape. If the inner shape is empty then return only the outer shape.

OUTPUT:

- a $k+1$-core or a pair of $k+1$-cores

EXAMPLES:

```
sage: StrongTableaux( 2, [3,1] ).shape()
[3, 1]
sage: type(StrongTableaux( 2, [3,1] ).shape())
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
sage: StrongTableaux( 4, [[2,1], [1]] ).shape()
([2, 1], [1])
```

classmethod standard_marked_iterator(*k*, *size*, *outer_shape=None*, *inner_shape=[]*)

An iterator for generating standard strong marked tableaux.

An iterator which generates all standard marked k-tableaux of a given `size` which are contained in `outer_shape` and contain the `inner_shape`. If `outer_shape` is `None` then there is no restriction on the shape of the tableaux which are created.

INPUT:

- `k` - a positive integer
- `size` - a positive integer
- `outer_shape` - a list which is a $k+1$-core (default: `None`)
- `inner_shape` - a list which is a $k+1$-core (default: [])

OUTPUT:

- an iterator which returns the standard marked tableaux with `size` cells and that are contained in `outer_shape` and contain `inner_shape`

EXAMPLES:

```
sage: list(StrongTableaux.standard_marked_iterator(2, 3))
[[[-1, -2, 3], [-3]], [[-1, -2, -3], [3]], [[-1, -2], [-3], [3]], [[-1, 3, -3], [-2]], [[-1,
sage: list(StrongTableaux.standard_marked_iterator(2, 1, inner_shape=[1,1]))
[[[None, 1, -1], [None]], [[None, 1], [None], [-1]], [[None, -1], [None], [1]]]
sage: len(list(StrongTableaux.standard_marked_iterator(4,4)))
10
sage: len(list(StrongTableaux.standard_marked_iterator(4,6)))
140
sage: len(list(StrongTableaux.standard_marked_iterator(4,4, inner_shape=[2,2])))
200
sage: len(list(StrongTableaux.standard_marked_iterator(4,4, outer_shape=[5,2,2,1], inner_sha
24
```

TESTS:

```
sage: list(StrongTableaux.standard_marked_iterator(2,0,inner_shape=[3,1]))
[[[None, None, None], [None]]]
sage: list(StrongTableaux.standard_marked_iterator(4,0))
[[]]
```

classmethod standard_unmarked_iterator(*k*, *size*, *outer_shape=None*, *inner_shape=[]*)

An iterator for standard unmarked strong tableaux.

An iterator which generates all unmarked tableaux of a given `size` which are contained in `outer_shape` and which contain the `inner_shape`.

These are built recursively by building all standard marked strong tableaux of size `size` -1 and adding all possible covers.

If `outer_shape` is `None` then there is no restriction on the shape of the tableaux which are created.

INPUT:

- `k`, `size` - a positive integers
- `outer_shape` - a list representing a $k+1$-core (default: `None`)
- `inner_shape` - a list representing a $k+1$-core (default: [])

OUTPUT:

- an iterator which lists all standard strong unmarked tableaux with `size` cells and which are contained in `outer_shape` and contain `inner_shape`

EXAMPLES:

```
sage: list(StrongTableaux.standard_unmarked_iterator(2, 3))
[[[1, 2, 3], [3]], [[1, 2], [3], [3]], [[1, 3, 3], [2]], [[1, 3], [2], [3]]]
sage: list(StrongTableaux.standard_unmarked_iterator(2, 1, inner_shape=[1,1]))
[[[None, 1, 1], [None]], [[None, 1], [None], [1]]]
sage: len(list(StrongTableaux.standard_unmarked_iterator(4,4)))
10
sage: len(list(StrongTableaux.standard_unmarked_iterator(4,6)))
98
sage: len(list(StrongTableaux.standard_unmarked_iterator(4,4, inner_shape=[2,2])))
92
sage: len(list(StrongTableaux.standard_unmarked_iterator(4,4, outer_shape=[5,2,2,1], inner_s
10
```

TESTS:

```
sage: list(StrongTableaux.standard_unmarked_iterator(2,0, outer_shape=[3,1], inner_shape=[3,
[[[None, None, None], [None]]]
sage: list(StrongTableaux.standard_unmarked_iterator(4,0, outer_shape=[]))
[[]]
```

classmethod `transpositions_to_standard_strong` (*transeq*, *k*, *emptyTableau*=`[]`)

Return a strong tableau correponding to a sequence of transpositions.

This method returns the action by left multiplication on the empty strong tableau by transpositions specified by `transeq`.

INPUT:

- `transeq` – a sequence of transpositions t_{ij} (a list of pairs).
- `emptyTableau` – (default: `[]`) an empty list or a skew strong tableau possibly consisting of `None` entries

OUTPUT:

- a `StrongTableau` object

EXAMPLES:

```
sage: StrongTableaux.transpositions_to_standard_strong([[0,1]], 2)
[[-1]]
sage: StrongTableaux.transpositions_to_standard_strong([[-2,-1], [2,3]], 2, [[None, None]])
```

```
[[None, None, -1], [1], [-2]]
sage: StrongTableaux.transpositions_to_standard_strong([[2, 3], [1, 2], [0, 1]], 2)
[[-1, -2, -3], [3]]
sage: StrongTableaux.transpositions_to_standard_strong([[-1, 0], [1, 2], [0, 1]], 2)
[[-1, -2, 3], [-3]]
sage: StrongTableaux.transpositions_to_standard_strong([[3, 4], [-1, 0], [1, 2]], 2, [[None]
[[None, -1, 2, -3], [-2, 3]]
```

TESTS:

```
sage: StrongTableaux.transpositions_to_standard_strong([], 2, [[None, None, None], [None]])
[[None, None, None], [None]]
sage: StrongTableaux.transpositions_to_standard_strong([], 4, [])
[]
```

sage.combinat.k_tableau.**WeakTableau**(*t*, *k*, *inner_shape*=[ ], *representation*='core')

This is the dispatcher method for the element class of weak k-tableaux.

Standard weak k-tableaux correspond to saturated chains in the weak order. There are three formulations of weak tableaux, one in terms of cores, one in terms of k-bounded partitions, and one in terms of factorizations of affine Grassmannian elements. For semistandard weak k-tableaux, all letters of the same value have to satisfy the conditions of a horizontal strip. In the affine Grassmannian formulation this means that all factors are cyclically decreasing elements. For more information, see for example [LLMSSZ2013].

INPUT:

- `t` – a weak k-tableau in the specified representation:
  - for the 'core' representation `t` is a list of lists where each subtableaux should have a $k+1$-core shape; `None` is allowed as an entry for skew weak k-tableaux
  - for the 'bounded' representation `t` is a list of lists where each subtableaux should have a k-bounded shape; `None` is allowed as an entry for skew weak k-tableaux
  - for the 'factorized_permutation' representation `t` is either a list of cyclically decreasing Weyl group elements or a list of reduced words of cyclically decreasing Weyl group elements; to indicate a skew tableau in this representation, `inner_shape` should be the inner shape as a $(k+1)$-core
- `k` – positive integer
- `inner_shape` – this entry is only relevant for the 'factorized_permutation' representation and specifies the inner shape in case the tableau is skew (default: `[]`)
- `representation` – 'core', 'bounded', or 'factorized_permutation' (default: 'core')

EXAMPLES:

Here is an example of a weak 3-tableau in core representation:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: t.shape()
[5, 2, 1]
sage: t.weight()
(2, 2, 2)
sage: type(t)
<class 'sage.combinat.k_tableau.WeakTableaux_core_with_category.element_class'>
```

And now we give a skew weak 3-tableau in core representation:

```
sage: ts = WeakTableau([[None, 1, 1, 2, 2], [None, 2], [1]], 3)
sage: ts.shape()
([5, 2, 1], [1, 1])
```

```
sage: ts.weight()
(2, 2)
sage: type(ts)
<class 'sage.combinat.k_tableau.WeakTableaux_core_with_category.element_class'>
```

Next we create the analogue of the first example in bounded representation:

```
sage: tb = WeakTableau([[1,1,2],[2,3],[3]], 3, representation="bounded")
sage: tb.shape()
[3, 2, 1]
sage: tb.weight()
(2, 2, 2)
sage: type(tb)
<class 'sage.combinat.k_tableau.WeakTableaux_bounded_with_category.element_class'>
sage: tb.to_core_tableau()
[[1, 1, 2, 2, 3], [2, 3], [3]]
sage: t == tb.to_core_tableau()
True
```

And the analogue of the skew example in bounded representation:

```
sage: tbs = WeakTableau([[None, 1, 2], [None, 2], [1]], 3, representation = "bounded")
sage: tbs.shape()
([3, 2, 1], [1, 1])
sage: tbs.weight()
(2, 2)
sage: tbs.to_core_tableau()
[[None, 1, 1, 2, 2], [None, 2], [1]]
sage: ts.to_bounded_tableau() == tbs
True
```

Finally we do the same examples for the factorized permutation representation:

```
sage: tf = WeakTableau([[2,0],[3,2],[1,0]], 3, representation = "factorized_permutation")
sage: tf.shape()
[5, 2, 1]
sage: tf.weight()
(2, 2, 2)
sage: type(tf)
<class 'sage.combinat.k_tableau.WeakTableaux_factorized_permutation_with_category.element_class'
sage: tf.to_core_tableau() == t
True

sage: tfs = WeakTableau([[0,3],[2,1]], 3, inner_shape = [1,1], representation = 'factorized_perm
sage: tfs.shape()
([5, 2, 1], [1, 1])
sage: tfs.weight()
(2, 2)
sage: type(tfs)
<class 'sage.combinat.k_tableau.WeakTableaux_factorized_permutation_with_category.element_class'
sage: tfs.to_core_tableau()
[[None, 1, 1, 2, 2], [None, 2], [1]]
```

Another way to pass from one representation to another is as follows:

```
sage: ts
[[None, 1, 1, 2, 2], [None, 2], [1]]
sage: ts.parent()._representation
'core'
sage: ts.representation('bounded')
```

```
[[None, 1, 2], [None, 2], [1]]
```

To test whether a given semistandard tableau is a weak k-tableau in the bounded representation, one can ask:

```
sage: t = Tableau([[1,1,2],[2,3],[3]])
sage: t.is_k_tableau(3)
True
sage: t = SkewTableau([[None, 1, 2], [None, 2], [1]])
sage: t.is_k_tableau(3)
True
sage: t = SkewTableau([[None, 1, 1], [None, 2], [2]])
sage: t.is_k_tableau(3)
False
```

TESTS:

```
sage: t = WeakTableau([[2,0],[3,2],[1,0]], 3, representation = "bla")
Traceback (most recent call last):
...
NotImplementedError: The representation option needs to be 'core', 'bounded', or 'factorized_per
```

class sage.combinat.k_tableau.**WeakTableau_abstract**

Bases: sage.structure.list_clone.ClonableList

Abstract class for the various element classes of WeakTableau.

intermediate_shapes()

Return the intermediate shapes of self.

A (skew) tableau with letters $1, 2, \ldots, \ell$ can be viewed as a sequence of shapes, where the i-th shape is given by the shape of the subtableau on letters $1, 2, \ldots, i$. The output is the list of these shapes.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]],3)
sage: t.intermediate_shapes()
[[], [2], [4, 1], [5, 2, 1]]

sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: t.intermediate_shapes()
[[2], [2, 1], [3, 1, 1], [4, 1, 1], [5, 2, 1]]

sage: t = WeakTableau([[1,1,1],[2,2],[3]], 3, representation = 'bounded')
sage: t.intermediate_shapes()
[[], [3], [3, 2], [3, 2, 1]]

sage: t = WeakTableau([[None, None, 1], [2, 4], [3]], 3, representation = 'bounded')
sage: t.intermediate_shapes()
[[2], [3], [3, 1], [3, 1, 1], [3, 2, 1]]

sage: t = WeakTableau([[0],[3],[2],[3]], 3, inner_shape = [2], representation = 'factorized_
sage: t.intermediate_shapes()
[[2], [2, 1], [3, 1, 1], [4, 1, 1], [5, 2, 1]]
```

pp()

Return a pretty print string of the tableau.

EXAMPLES:

```
sage: t = WeakTableau([[None, 1, 1, 2, 2], [None, 2], [1]], 3)
sage: t.pp()
```

```
.  1  1  2  2
.  2
1
sage: t = WeakTableau([[2,0],[3,2]], 3, inner_shape = [2], representation = 'factorized_perm
sage: t.pp()
[s2*s0, s3*s2]
```

representation(*representation='core'*)

Return the analogue of `self` in the specified representation.

INPUT:

- `representation` – 'core', 'bounded', or 'factorized_permutation' (default: 'core')

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4, 7], [5, 6], [6],
sage: t.parent()._representation
'core'
sage: t.representation('bounded')
[[1, 1, 2, 4], [2, 3, 5], [3, 4], [5, 6], [6], [7]]
sage: t.representation('factorized_permutation')
[s0, s3*s1, s2*s1, s0*s4, s3*s0, s4*s2, s1*s0]

sage: tb = WeakTableau([[1, 1, 2, 4], [2, 3, 5], [3, 4], [5, 6], [6], [7]], 4, representatio
sage: tb.parent()._representation
'bounded'
sage: tb.representation('core') == t
True
sage: tb.representation('factorized_permutation')
[s0, s3*s1, s2*s1, s0*s4, s3*s0, s4*s2, s1*s0]

sage: tp = WeakTableau([[0],[3,1],[2,1],[0,4],[3,0],[4,2],[1,0]], 4, representation = 'facto
sage: tp.parent()._representation
'factorized_permutation'
sage: tp.representation('core') == t
True
sage: tp.representation('bounded') == tb
True
```

shape()

Return the shape of `self`.

When the tableau is straight, the outer shape is returned. When the tableau is skew, the tuple of the outer and inner shape is returned.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: t.shape()
[5, 2, 1]
sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: t.shape()
([5, 2, 1], [2])

sage: t = WeakTableau([[1,1,1],[2,2],[3]], 3, representation = 'bounded')
sage: t.shape()
[3, 2, 1]
sage: t = WeakTableau([[None, None, 1], [2, 4], [3]], 3, representation = 'bounded')
sage: t.shape()
```

```
([3, 2, 1], [2])

sage: t = WeakTableau([[2],[0,3],[2,1,0]], 3, representation = 'factorized_permutation')
sage: t.shape()
[5, 2, 1]
sage: t = WeakTableau([[2,0],[3,2]], 3, inner_shape = [2], representation = 'factorized_perm
sage: t.shape()
([5, 2, 1], [2])
```

size()

Return the size of the shape of `self`.

In the bounded representation, the size of the shape is the number of boxes in the outer shape minus the number of boxes in the inner shape. For the core and factorized permutation representation, the size is the length of the outer shape minus the length of the inner shape.

See also:

`sage.combinat.core.Core.length()`

EXAMPLES:

```
sage: t = WeakTableau([[None, 1, 1, 2, 2], [None, 2], [1]], 3)
sage: t.shape()
([5, 2, 1], [1, 1])
sage: t.size()
4
sage: t = WeakTableau([[1,1,2],[2,3],[3]], 3, representation="bounded")
sage: t.shape()
[3, 2, 1]
sage: t.size()
6
```

weight()

Return the weight of `self`.

The weight is a tuple whose i-th entry is the number of labels i in the bounded representation of `self`.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: t.weight()
(2, 2, 2)
sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: t.weight()
(1, 1, 1, 1)
sage: t = WeakTableau([[None,2,3],[3]],2)
sage: t.weight()
(0, 1, 1)

sage: t = WeakTableau([[1,1,1],[2,2],[3]], 3, representation = 'bounded')
sage: t.weight()
(3, 2, 1)
sage: t = WeakTableau([[1,1,2],[2,3],[3]], 3, representation = 'bounded')
sage: t.weight()
(2, 2, 2)
sage: t = WeakTableau([[None, None, 1], [2, 4], [3]], 3, representation = 'bounded')
sage: t.weight()
(1, 1, 1, 1)

sage: t = WeakTableau([[2],[0,3],[2,1,0]], 3, representation = 'factorized_permutation')
```

```
sage: t.weight()
(3, 2, 1)
sage: t = WeakTableau([[2,0],[3,2],[1,0]], 3, representation = 'factorized_permutation')
sage: t.weight()
(2, 2, 2)
sage: t = WeakTableau([[2,0],[3,2]], 3, inner_shape = [2], representation = 'factorized_perm
sage: t.weight()
(2, 2)
```

class sage.combinat.k_tableau.**WeakTableau_bounded**(*parent*, *t*)

Bases: sage.combinat.k_tableau.WeakTableau_abstract

A (skew) weak k-tableau represented in terms of k-bounded partitions.

check()

Check that `self` is a valid weak k-tableau.

EXAMPLES:

```
sage: t = WeakTableau([[1,1],[2]], 2, representation = 'bounded')
sage: t.check()

sage: t = WeakTableau([[None, None, 1], [2, 4], [3]], 3, representation = 'bounded')
sage: t.check()
```

TESTS:

```
sage: t = WeakTableau([[1,1,3],[2,2],[3]], 3, representation = 'bounded')
Traceback (most recent call last):
...
ValueError: This is not a proper weak 3-tableau

sage: T = WeakTableaux(3, [3,1], [2,1], representation = 'bounded')
sage: t = T([[None, 1,1], [2]])
Traceback (most recent call last):
...
ValueError: The inner shape of the parent does not agree with the inner shape of the tableau

sage: t = WeakTableau([[1,1],[1]], 3, representation = 'bounded')
Traceback (most recent call last):
...
ValueError: The tableaux is not semistandard!
```

classmethod from_core_tableau(*t*, *k*)

Construct weak k-bounded tableau from in k-core tableau.

EXAMPLES:

```
sage: from sage.combinat.k_tableau import WeakTableau_bounded
sage: WeakTableau_bounded.from_core_tableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
[[1, 1, 2], [2, 3], [3]]

sage: WeakTableau_bounded.from_core_tableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
[[None, None, 3], [1, 4], [2]]

sage: WeakTableau_bounded.from_core_tableau([[None,2,3],[3]], 2)
[[None, 2], [3]]
```

k_charge(*algorithm='I'*)

Return the k-charge of `self`.

INPUT:

- `algorithm` – (default: "I") if "I", computes k-charge using the I algorithm, otherwise uses the J-algorithm

OUTPUT:

- a nonnegative integer

For the definition of k-charge and the various algorithms to compute it see Section 3.3 of [LLMSSZ2013].

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2], [2, 3], [3]], 3, representation = 'bounded')
sage: t.k_charge()
2
sage: t = WeakTableau([[1, 3, 5], [2, 6], [4]], 3, representation = 'bounded')
sage: t.k_charge()
8
sage: t = WeakTableau([[1, 1, 2, 4], [2, 3, 5], [3, 4], [5, 6], [6], [7]], 4, representation
sage: t.k_charge()
12
```

shape_bounded()

Return the shape of `self` as k-bounded partition.

When the tableau is straight, the outer shape is returned as a k-bounded partition. When the tableau is skew, the tuple of the outer and inner shape is returned as k-bounded partitions.

EXAMPLES:

```
sage: t = WeakTableau([[1,1,1],[2,2],[3]], 3, representation = 'bounded')
sage: t.shape_bounded()
[3, 2, 1]

sage: t = WeakTableau([[None, None, 1], [2, 4], [3]], 3, representation = 'bounded')
sage: t.shape_bounded()
([3, 2, 1], [2])
```

shape_core()

Return the shape of `self` as $(k+1)$-core.

When the tableau is straight, the outer shape is returned as a $(k+1)$-core. When the tableau is skew, the tuple of the outer and inner shape is returned as $(k+1)$-cores.

EXAMPLES:

```
sage: t = WeakTableau([[1,1,1],[2,2],[3]], 3, representation = 'bounded')
sage: t.shape_core()
[5, 2, 1]

sage: t = WeakTableau([[None, None, 1], [2, 4], [3]], 3, representation = 'bounded')
sage: t.shape_core()
([5, 2, 1], [2])
```

to_core_tableau()

Return the weak k-tableau `self` where the shape of each restricted tableau is a $(k+1)$-core.

EXAMPLES:

```
sage: t = WeakTableau([[1,1,2,4],[2,3,5],[3,4],[5,6],[6],[7]], 4, representation = 'bounded'
sage: c = t.to_core_tableau(); c
[[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4, 7], [5, 6], [6], [7]]
sage: type(c)
```

```
<class 'sage.combinat.k_tableau.WeakTableaux_core_with_category.element_class'>
sage: t = WeakTableau([], 4, representation = 'bounded')
sage: t.to_core_tableau()
[]

sage: from sage.combinat.k_tableau import WeakTableau_bounded
sage: t = WeakTableau([[1,1,2],[2,3],[3]], 3, representation = 'bounded')
sage: WeakTableau_bounded.from_core_tableau(t.to_core_tableau(),3)
[[1, 1, 2], [2, 3], [3]]
sage: t == WeakTableau_bounded.from_core_tableau(t.to_core_tableau(),3)
True

sage: t = WeakTableau([[None, None, 1], [2, 4], [3]], 3, representation = 'bounded')
sage: t.to_core_tableau()
[[None, None, 1, 2, 4], [2, 4], [3]]
sage: t == WeakTableau_bounded.from_core_tableau(t.to_core_tableau(),3)
True
```

class sage.combinat.k_tableau.**WeakTableau_core**(*parent*, *t*)

Bases: sage.combinat.k_tableau.WeakTableau_abstract

A (skew) weak k-tableau represented in terms of $(k+1)$-cores.

check()

Check that self is a valid weak k-tableau.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2], [2]], 2)
sage: t.check()
sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: t.check()
```

TESTS:

```
sage: T = WeakTableaux(2, [3,1], [1,1,1,1])
sage: t = T([[1,2,3],[3]])
Traceback (most recent call last):
...
ValueError: The weight of the parent does not agree with the weight of the tableau!

sage: t = WeakTableau([[1, 2, 2], [1]], 2)
Traceback (most recent call last):
...
ValueError: The tableau is not semistandard!
```

dictionary_of_coordinates_at_residues(*v*)

Return a dictionary assigning to all residues of self with label v a list of cells with the given residue.

INPUT:

- v – a label of a cell in self

OUTPUT:

- dictionary assigning coordinates in self to residues

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]],3)
sage: t.dictionary_of_coordinates_at_residues(3)
{0: [(0, 4), (1, 1)], 2: [(2, 0)]}
```

```
sage: t = WeakTableau([[None, None, 1, 1, 4], [1, 4], [3]], 3)
sage: t.dictionary_of_coordinates_at_residues(1)
{2: [(0, 2)], 3: [(0, 3), (1, 0)]}

sage: t = WeakTableau([], 3)
sage: t.dictionary_of_coordinates_at_residues(1)
{}
```

k_charge(*algorithm='I'*)

Return the k-charge of `self`.

INPUT:

- `algorithm` – (default: "I") if "I", computes k-charge using the I algorithm, otherwise uses the J-algorithm

OUTPUT:

- a nonnegative integer

For the definition of k-charge and the various algorithms to compute it see Section 3.3 of [LLMSSZ2013].

See also:

`k_charge_I()` and `k_charge_J()`

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: t.k_charge()
2
sage: t = WeakTableau([[1, 3, 4, 5, 6], [2, 6], [4]], 3)
sage: t.k_charge()
8
sage: t = WeakTableau([[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4, 7], [5, 6], [6],
sage: t.k_charge()
12
```

TESTS:

```
sage: T = WeakTableaux(4, [13,9,5,3,2,1,1], [4,3,3,2,2,1,1,1])
sage: T.cardinality()
6
sage: all(t.k_charge_I() == t.k_charge_J() for t in T)
True
```

k_charge_I()

Return the k-charge of `self` using the I-algorithm.

For the definition of k-charge and the I-algorithm see Section 3.3 of [LLMSSZ2013].

OUTPUT:

- a nonnegative integer

See also:

`k_charge()` and `k_charge_J()`

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: t.k_charge_I()
2
```

```
sage: t = WeakTableau([[1, 3, 4, 5, 6], [2, 6], [4]], 3)
sage: t.k_charge_I()
8
sage: t = WeakTableau([[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4, 7], [5, 6], [6],
sage: t.k_charge_I()
12
```

TESTS:

```
sage: t = WeakTableau([[None, None, 1, 1, 4], [1, 4], [3]], 3)
sage: t.k_charge_I()
Traceback (most recent call last):
...
ValueError: k-charge is not defined for skew weak tableaux
```

k_charge_J()

Return the k-charge of `self` using the J-algorithm.

For the definition of k-charge and the J-algorithm see Section 3.3 of [LLMSSZ2013].

OUTPUT:

- a nonnegative integer

See also:

`k_charge()` and `k_charge_I()`

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: t.k_charge_J()
2
sage: t = WeakTableau([[1, 3, 4, 5, 6], [2, 6], [4]], 3)
sage: t.k_charge_J()
8
sage: t = WeakTableau([[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4, 7], [5, 6], [6],
sage: t.k_charge_J()
12
```

TESTS:

```
sage: t = WeakTableau([[None, None, 1, 1, 4], [1, 4], [3]], 3)
sage: t.k_charge_I()
Traceback (most recent call last):
...
ValueError: k-charge is not defined for skew weak tableaux

sage: t = WeakTableau([[1, 1, 2, 3], [2, 4, 4], [3, 6], [5]], 4, representation='bounded')
sage: t.k_charge() == t.k_charge(algorithm = 'J')
True
```

list_of_standard_cells()

Return a list of lists of the coordinates of the standard cells of `self`.

INPUT:

- `self` – a weak k-tableau in core representation with partition weight

OUTPUT:

- a list of lists of coordinates

Warning: This method currently only works for straight weak tableaux with partition weight.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: t.list_of_standard_cells()
[[(0, 1), (1, 0), (2, 0)], [(0, 0), (0, 2), (1, 1)]]
sage: t = WeakTableau([[1, 1, 1, 2], [2, 2, 3]], 5)
sage: t.list_of_standard_cells()
[[(0, 2), (1, 1), (1, 2)], [(0, 1), (1, 0)], [(0, 0), (0, 3)]]
sage: t = WeakTableau([[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4, 7], [5, 6], [6],
sage: t.list_of_standard_cells()
[[(0, 1), (1, 0), (2, 0), (0, 5), (3, 0), (4, 0), (5, 0)], [(0, 0), (0, 2), (1, 1), (2, 1),
```

TESTS:

```
sage: t = WeakTableau([],3)
sage: t.list_of_standard_cells()
[]

sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: t.list_of_standard_cells()
Traceback (most recent call last):
...
ValueError: This method only works for straight tableaux!

sage: t = WeakTableau([[1,2],[2]], 3)
sage: t.list_of_standard_cells()
Traceback (most recent call last):
...
ValueError: This method only works for weak tableaux with partition weight!
```

residues_of_entries(*v*)

Return a list of residues of cells of weak k-tableau `self` labeled by `v`.

INPUT:

- `v` – a label of a cell in `self`

OUTPUT:

- a list of residues

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]],3)
sage: t.residues_of_entries(1)
[0, 1]

sage: t = WeakTableau([[None, None, 1, 1, 4], [1, 4], [3]], 3)
sage: t.residues_of_entries(1)
[2, 3]
```

shape_bounded()

Return the shape of `self` as a k-bounded partition.

When the tableau is straight, the outer shape is returned as a k-bounded partition. When the tableau is skew, the tuple of the outer and inner shape is returned as k-bounded partitions.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]],3)
sage: t.shape_bounded()
[3, 2, 1]

sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: t.shape_bounded()
([3, 2, 1], [2])
```

shape_core()

Return the shape of `self` as a $(k+1)$-core.

When the tableau is straight, the outer shape is returned as a core. When the tableau is skew, the tuple of the outer and inner shape is returned as cores.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]],3)
sage: t.shape_core()
[5, 2, 1]

sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: t.shape_core()
([5, 2, 1], [2])
```

to_bounded_tableau()

Return the bounded representation of the weak k-tableau `self`.

Each restricted sutableaux of the output is a k-bounded partition.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: c = t.to_bounded_tableau(); c
[[1, 1, 2], [2, 3], [3]]
sage: type(c)
<class 'sage.combinat.k_tableau.WeakTableaux_bounded_with_category.element_class'>

sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: t.to_bounded_tableau()
[[None, None, 3], [1, 4], [2]]
sage: t.to_bounded_tableau().to_core_tableau() == t
True
```

to_factorized_permutation_tableau()

Return the factorized permutation representation of the weak k-tableau `self`.

EXAMPLES:

```
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]], 3)
sage: c = t.to_factorized_permutation_tableau(); c
[s2*s0, s3*s2, s1*s0]
sage: type(c)
<class 'sage.combinat.k_tableau.WeakTableaux_factorized_permutation_with_category.element_cl
sage: c.to_core_tableau() == t
True

sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: c = t.to_factorized_permutation_tableau(); c
[s0, s3, s2, s3]
sage: c._inner_shape
[2]
```

```
sage: c.to_core_tableau() == t
True
```

TESTS:

```
sage: t = WeakTableau([], 4)
sage: c = t.to_factorized_permutation_tableau(); c
[1]
sage: c._inner_shape
[]
sage: c.to_core_tableau() == t
True
```

class sage.combinat.k_tableau.**WeakTableau_factorized_permutation**(*parent*, *t*)

Bases: sage.combinat.k_tableau.WeakTableau_abstract

A weak (skew) k-tableau represented in terms of factorizations of affine permutations into cyclically decreasing elements.

check()

Check that `self` is a valid weak k-tableau.

EXAMPLES:

```
sage: t = WeakTableau([[2],[0,3],[2,1,0]], 3, representation = 'factorized_permutation')
sage: t.check()
```

TESTS:

```
sage: t = WeakTableau([[2,0],[3,2]], 3, representation = 'factorized_permutation')
Traceback (most recent call last):
...
ValueError: Error! this only works on type 'A' affine Grassmannian elements

sage: T = WeakTableaux(3, [4,1], [2,1], representation = 'factorized_permutation')
sage: t = T([[2],[1],[0]])
Traceback (most recent call last):
...
ValueError: The weight of the parent does not agree with the weight of the tableau!
```

classmethod from_core_tableau(*t*, *k*)

Construct weak factorized affine permutation tableau from a k-core tableau.

EXAMPLES:

```
sage: from sage.combinat.k_tableau import WeakTableau_factorized_permutation
sage: WeakTableau_factorized_permutation.from_core_tableau([[1, 1, 2, 2, 3], [2, 3], [3]],3)
[s2*s0, s3*s2, s1*s0]
sage: WeakTableau_factorized_permutation.from_core_tableau([[1, 1, 2, 3, 4, 4, 5, 5, 6], [2,
[s0, s3*s1, s2*s1, s0*s4, s3*s0, s4*s2, s1*s0]
sage: WeakTableau_factorized_permutation.from_core_tableau([[None, 1, 1, 2, 2], [None, 2], [
[s0*s3, s2*s1]
```

k_charge(*algorithm='I'*)

Return the k-charge of `self`.

OUTPUT:

- a nonnegative integer

EXAMPLES:

```
sage: t = WeakTableau([[2,0],[3,2],[1,0]], 3, representation = 'factorized_permutation')
sage: t.k_charge()
2
sage: t = WeakTableau([[0],[3],[2],[1],[3],[0]], 3, representation = 'factorized_permutation
sage: t.k_charge()
8
sage: t = WeakTableau([[0],[3,1],[2,1],[0,4],[3,0],[4,2],[1,0]], 4, representation = 'factor
sage: t.k_charge()
12
```

shape_bounded()

Return the shape of `self` as a k-bounded partition.

When the tableau is straight, the outer shape is returned as a k-bounded partition. When the tableau is skew, the tuple of the outer and inner shape is returned as k-bounded partitions.

EXAMPLES:

```
sage: t = WeakTableau([[2],[0,3],[2,1,0]], 3, representation = 'factorized_permutation')
sage: t.shape_bounded()
[3, 2, 1]

sage: t = WeakTableau([[2,0],[3,2]], 3, inner_shape = [2], representation = 'factorized_perm
sage: t.shape_bounded()
([3, 2, 1], [2])
```

shape_core()

Return the shape of `self` as a $(k+1)$-core.

When the tableau is straight, the outer shape is returned as a core. When the tableau is skew, the tuple of the outer and inner shape is returned as cores.

EXAMPLES:

```
sage: t = WeakTableau([[2],[0,3],[2,1,0]], 3, representation = 'factorized_permutation')
sage: t.shape_core()
[5, 2, 1]

sage: t = WeakTableau([[2,0],[3,2]], 3, inner_shape = [2], representation = 'factorized_perm
sage: t.shape()
([5, 2, 1], [2])
```

static straighten_input(t, k)

Straightens input.

INPUT:

- `t` – a list of reduced words or a list of elements in the Weyl group of type $A_k^{(1)}$
- `k` – a positive integer

EXAMPLES:

```
sage: from sage.combinat.k_tableau import WeakTableau_factorized_permutation
sage: WeakTableau_factorized_permutation.straighten_input([[2,0],[3,2],[1,0]], 3)
(s2*s0, s3*s2, s1*s0)
sage: W = WeylGroup(['A',4,1])
sage: WeakTableau_factorized_permutation.straighten_input([W.an_element(),W.an_element()], 4
(s0*s1*s2*s3*s4, s0*s1*s2*s3*s4)
```

TESTS:

```
sage: WeakTableau_factorized_permutation.straighten_input([W.an_element(),W.an_element()], 3
Traceback (most recent call last):
...
ValueError: a matrix from Full MatrixSpace of 5 by 5 dense matrices over Rational Field cann
```

to_core_tableau()

Return the weak k-tableau `self` where the shape of each restricted tableau is a $(k+1)$-core.

EXAMPLES:

```
sage: t = WeakTableau([[0], [3,1], [2,1], [0,4], [3,0], [4,2], [1,0]], 4, representation = '
[s0, s3*s1, s2*s1, s0*s4, s3*s0, s4*s2, s1*s0]
sage: c = t.to_core_tableau(); c
[[1, 1, 2, 3, 4, 4, 5, 5, 6], [2, 3, 5, 5, 6], [3, 4, 7], [5, 6], [6], [7]]
sage: type(c)
<class 'sage.combinat.k_tableau.WeakTableaux_core_with_category.element_class'>
sage: t = WeakTableau([[]], 4, representation = 'factorized_permutation'); t
[1]
sage: t.to_core_tableau()
[]

sage: from sage.combinat.k_tableau import WeakTableau_factorized_permutation
sage: t = WeakTableau([[2,0],[3,2],[1,0]], 3, representation = 'factorized_permutation')
sage: WeakTableau_factorized_permutation.from_core_tableau(t.to_core_tableau(), 3)
[s2*s0, s3*s2, s1*s0]
sage: t == WeakTableau_factorized_permutation.from_core_tableau(t.to_core_tableau(), 3)
True

sage: t = WeakTableau([[2,0],[3,2]], 3, inner_shape = [2], representation = 'factorized_perm
sage: t.to_core_tableau()
[[None, None, 1, 1, 2], [1, 2], [2]]
sage: t == WeakTableau_factorized_permutation.from_core_tableau(t.to_core_tableau(), 3)
True
```

sage.combinat.k_tableau.**WeakTableaux**(*k*, *shape*, *weight*, *representation='core'*)

This is the dispatcher method for the parent class of weak k-tableaux.

INPUT:

- `k` – positive integer
- `shape` – shape of the weak k-tableaux; for the 'core' and 'factorized_permutation' representation, the shape is inputted as a $(k+1)$-core; for the 'bounded' representation, the shape is inputted as a k-bounded partition; for skew tableaux, the shape is inputted as a tuple of the outer and inner shape
- `weight` – the weight of the weak k-tableaux as a list or tuple
- `representation` – 'core', 'bounded', or 'factorized_permutation' (default: 'core')

EXAMPLES:

```
sage: T = WeakTableaux(3, [5,2,1], [1,1,1,1,1,1])
sage: T.list()
[[[1, 3, 4, 5, 6], [2, 6], [4]],
[[1, 2, 4, 5, 6], [3, 6], [4]],
[[1, 2, 3, 4, 6], [4, 6], [5]],
[[1, 2, 3, 4, 5], [4, 5], [6]]]
sage: T.cardinality()
4

sage: T = WeakTableaux(3, [[5,2,1], [2]], [1,1,1,1])
```

```
sage: T.list()
[[[None, None, 2, 3, 4], [1, 4], [2]],
[[None, None, 1, 2, 4], [2, 4], [3]],
[[None, None, 1, 2, 3], [2, 3], [4]]]

sage: T = WeakTableaux(3, [3,2,1], [1,1,1,1,1,1], representation = 'bounded')
sage: T.list()
[[[1, 3, 5], [2, 6], [4]],
[[1, 2, 5], [3, 6], [4]],
[[1, 2, 3], [4, 6], [5]],
[[1, 2, 3], [4, 5], [6]]]

sage: T = WeakTableaux(3, [[3,2,1], [2]], [1,1,1,1], representation = 'bounded')
sage: T.list()
[[[None, None, 3], [1, 4], [2]],
[[None, None, 1], [2, 4], [3]],
[[None, None, 1], [2, 3], [4]]]

sage: T = WeakTableaux(3, [5,2,1], [1,1,1,1,1,1], representation = 'factorized_permutation')
sage: T.list()
[[s0, s3, s2, s1, s3, s0],
[s0, s3, s2, s3, s1, s0],
[s0, s2, s3, s2, s1, s0],
[s2, s0, s3, s2, s1, s0]]

sage: T = WeakTableaux(3, [[5,2,1], [2]], [1,1,1,1], representation = 'factorized_permutation')
sage: T.list()
[[s0, s3, s2, s3], [s0, s2, s3, s2], [s2, s0, s3, s2]]
```

class sage.combinat.k_tableau.**WeakTableaux_abstract**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Abstract class for the various parent classes of WeakTableaux.

representation (*representation='core'*)

Return the analogue of self in the specified representation.

INPUT:

- representation – 'core', 'bounded', or 'factorized_permutation' (default: 'core')

EXAMPLES:

```
sage: T = WeakTableaux(3, [5,2,1], [1,1,1,1,1,1])
sage: T._representation
'core'
sage: T.representation('bounded')
Bounded weak 3-Tableaux of (skew) 3-bounded shape [3, 2, 1] and weight (1, 1, 1, 1, 1, 1)
sage: T.representation('factorized_permutation')
Factorized permutation (skew) weak 3-Tableaux of shape [5, 2, 1] and weight (1, 1, 1, 1, 1,

sage: T = WeakTableaux(3, [3,2,1], [1,1,1,1,1,1], representation = 'bounded')
sage: T._representation
'bounded'
sage: T.representation('core')
Core weak 3-Tableaux of (skew) core shape [5, 2, 1] and weight (1, 1, 1, 1, 1, 1)
sage: T.representation('bounded')
Bounded weak 3-Tableaux of (skew) 3-bounded shape [3, 2, 1] and weight (1, 1, 1, 1, 1, 1)
sage: T.representation('bounded') == T
```

```
True
sage: T.representation('factorized_permutation')
Factorized permutation (skew) weak 3-Tableaux of shape [5, 2, 1] and weight (1, 1, 1, 1, 1,
sage: T.representation('factorized_permutation') == T
False

sage: T = WeakTableaux(3, [5,2,1], [1,1,1,1,1,1], representation = 'factorized_permutation')
sage: T._representation
'factorized_permutation'
sage: T.representation('core')
Core weak 3-Tableaux of (skew) core shape [5, 2, 1] and weight (1, 1, 1, 1, 1, 1)
sage: T.representation('bounded')
Bounded weak 3-Tableaux of (skew) 3-bounded shape [3, 2, 1] and weight (1, 1, 1, 1, 1, 1)
sage: T.representation('factorized_permutation')
Factorized permutation (skew) weak 3-Tableaux of shape [5, 2, 1] and weight (1, 1, 1, 1, 1,
```

shape()

Return the shape of the tableaux of `self`.

When `self` is the class of straight tableaux, the outer shape is returned. When `self` is the class of skew tableaux, the tuple of the outer and inner shape is returned.

Note that in the 'core' and 'factorized_permutation' representation, the shapes are $(k+1)$-cores. In the 'bounded' representation, the shapes are k-bounded partitions.

If the user wants to access the skew shape (even if the inner shape is empty), please use `self._shape`.

EXAMPLES:

```
sage: T = WeakTableaux(3, [5,2,2], [2,2,2,1])
sage: T.shape()
[5, 2, 2]
sage: T._shape
([5, 2, 2], [])
sage: T = WeakTableaux(3, [[5,2,2], [1]], [2,1,2,1])
sage: T.shape()
([5, 2, 2], [1])

sage: T = WeakTableaux(3, [3,2,2], [2,2,2,1], representation = 'bounded')
sage: T.shape()
[3, 2, 2]
sage: T._shape
([3, 2, 2], [])
sage: T = WeakTableaux(3, [[3,2,2], [1]], [2,1,2,1], representation = 'bounded')
sage: T.shape()
([3, 2, 2], [1])

sage: T = WeakTableaux(3, [4,1], [2,2], representation = 'factorized_permutation')
sage: T.shape()
[4, 1]
sage: T._shape
([4, 1], [])
sage: T = WeakTableaux(4, [[6,2,1], [2]], [2,1,1,1], representation = 'factorized_permutatic
sage: T.shape()
([6, 2, 1], [2])
```

size()

Return the size of the shape.

In the bounded representation, the size of the shape is the number of boxes in the outer shape minus the

number of boxes in the inner shape. For the core and factorized permutation representation, the size is the length of the outer shape minus the length of the inner shape.

EXAMPLES:

```
sage: T = WeakTableaux(3, [5,2,1], [1,1,1,1,1,1])
sage: T.size()
6
sage: T = WeakTableaux(3, [3,2,1], [1,1,1,1,1,1], representation = 'bounded')
sage: T.size()
6
sage: T = WeakTableaux(4, [[6,2,1], [2]], [2,1,1,1], 'factorized_permutation')
sage: T.size()
5
```

class sage.combinat.k_tableau.**WeakTableaux_bounded**(*k*, *shape*, *weight*)

Bases: sage.combinat.k_tableau.WeakTableaux_abstract

The class of (skew) weak k-tableaux in the bounded representation of shape `shape` (as k-bounded partition or tuple of k-bounded partitions in the skew case) and weight `weight`.

INPUT:

- `k` – positive integer
- `shape` – the shape of the k-tableaux represented as a k-bounded partition; if the tableaux are skew, the shape is a tuple of the outer and inner shape each represented as a k-bounded partition
- `weight` – the weight of the k-tableaux

EXAMPLES:

```
sage: T = WeakTableaux(3, [3,1], [2,2], representation = 'bounded')
sage: T.list()
[[[1, 1, 2], [2]]]

sage: T = WeakTableaux(3, [[3,2,1], [2]], [1,1,1,1], representation = 'bounded')
sage: T.list()
[[[None, None, 3], [1, 4], [2]],
[[None, None, 1], [2, 4], [3]],
[[None, None, 1], [2, 3], [4]]]
```

Element

alias of WeakTableau_bounded

class sage.combinat.k_tableau.**WeakTableaux_core**(*k*, *shape*, *weight*)

Bases: sage.combinat.k_tableau.WeakTableaux_abstract

The class of (skew) weak k-tableaux in the core representation of shape `shape` (as $k+1$-core) and weight `weight`.

INPUT:

- `k` – positive integer
- `shape` – the shape of the k-tableaux represented as a $(k+1)$-core; if the tableaux are skew, the shape is a tuple of the outer and inner shape (both as $(k+1)$-cores)
- `weight` – the weight of the k-tableaux

EXAMPLES:

```
sage: T = WeakTableaux(3, [4,1], [2,2])
sage: T.list()
```

```
[[[1, 1, 2, 2], [2]]]

sage: T = WeakTableaux(3, [[5,2,1], [2]], [1,1,1,1])
sage: T.list()
[[[None, None, 2, 3, 4], [1, 4], [2]],
[[None, None, 1, 2, 4], [2, 4], [3]],
[[None, None, 1, 2, 3], [2, 3], [4]]]
```

Element
alias of WeakTableau_core

circular_distance(*cr*, *r*)
Return the shortest counterclockwise distance between cr and r modulo $k+1$.

INPUT:

- cr, r – nonnegative integers between 0 and k

OUTPUT:

- a positive integer

EXAMPLES:

```
sage: T = WeakTableaux(10, [], [])
sage: T.circular_distance(8, 6)
2
sage: T.circular_distance(8, 8)
0
sage: T.circular_distance(8, 9)
10
```

diag(*c*, *ha*)
Return the number of diagonals strictly between cells c and ha of the same residue as c.

INPUT:

- c – a cell in the lattice
- ha – another cell in the lattice with bigger row and smaller column than c

OUTPUT:

- a nonnegative integer

EXAMPLES:

```
sage: T = WeakTableaux(4, [5,2,2], [2,2,2,1])
sage: T.diag((1,2),(4,0))
0
```

class sage.combinat.k_tableau.**WeakTableaux_factorized_permutation**(*k*, *shape*, *weight*)

Bases: sage.combinat.k_tableau.WeakTableaux_abstract

The class of (skew) weak k-tableaux in the factorized permutation representation of shape shape (as $k+1$-core or tuple of $(k+1)$-cores in the skew case) and weight weight.

INPUT:

- k – positive integer
- shape – the shape of the k-tableaux represented as a $(k+1)$-core; in the skew case the shape is a tuple of the outer and inner shape both as $(k+1)$-cores

•`weight` – the weight of the k-tableaux

EXAMPLES:

```
sage: T = WeakTableaux(3, [4,1], [2,2], representation = 'factorized_permutation')
sage: T.list()
[[s3*s2, s1*s0]]

sage: T = WeakTableaux(4, [[6,2,1], [2]], [2,1,1,1], representation = 'factorized_permutation')
sage: T.list()
[[s0, s4, s3, s4*s2], [s0, s3, s4, s3*s2], [s3, s0, s4, s3*s2]]
```

Element
alias of `WeakTableau_factorized_permutation`

`sage.combinat.k_tableau.`**`intermediate_shapes`**`(t)`
Return the intermediate shapes of tableau `t`.

A (skew) tableau with letters $1, 2, \ldots, \ell$ can be viewed as a sequence of shapes, where the i-th shape is given by the shape of the subtableau on letters $1, 2, \ldots, i$. The output is the list of these shapes.

OUTPUT:

•a list of lists representing partitions

EXAMPLES:

```
sage: from sage.combinat.k_tableau import intermediate_shapes
sage: t = WeakTableau([[1, 1, 2, 2, 3], [2, 3], [3]],3)
sage: intermediate_shapes(t)
[[], [2], [4, 1], [5, 2, 1]]

sage: t = WeakTableau([[None, None, 2, 3, 4], [1, 4], [2]], 3)
sage: intermediate_shapes(t)
[[2], [2, 1], [3, 1, 1], [4, 1, 1], [5, 2, 1]]
```

`sage.combinat.k_tableau.`**`nabs`**`(v)`
Return the absolute value of `v` or `None`.

INPUT:

•`v` – either an integer or `None`

OUTPUT:

•either a non-negative integer or `None`

EXAMPLES:

```
sage: from sage.combinat.k_tableau import nabs
sage: nabs(None)
sage: nabs(-3)
3
sage: nabs(None)
```

5.1.114 Kazhdan-Lusztig Polynomials

AUTHORS:

- Daniel Bump (2008): initial version
- Alan J.X. Guo (2014-03-18): `R_tilde()` method.

class sage.combinat.kazhdan_lusztig.**KazhdanLusztigPolynomial**(*W*, *q*, *trace=False*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.sage_object.SageObject

A Kazhdan-Lusztig polynomial.

INPUT:

- W – a Weyl Group
- q – an indeterminate

OPTIONAL:

- trace – if True, then this displays the trace: the intermediate results. This is instructive and fun.

The parent of q may be a PolynomialRing or a LaurentPolynomialRing.

REFERENCES:

EXAMPLES:

```
sage: W = WeylGroup("B3",prefix="s")
sage: [s1,s2,s3] = W.simple_reflections()
sage: R.<q> = LaurentPolynomialRing(QQ)
sage: KL = KazhdanLusztigPolynomial(W,q)
sage: KL.P(s2,s3*s2*s3*s1*s2)
1 + q
```

A faster implementation (using the optional package Coxeter 3) is given by:

```
sage: W = CoxeterGroup(['B', 3], implementation='coxeter3') # optional - coxeter3
sage: W.kazhdan_lusztig_polynomial([2], [3,2,3,1,2])        # optional - coxeter3
q + 1
```

P(*x*, *y*)

Return the Kazhdan-Lusztig P polynomial.

If the rank is large, this runs slowly at first but speeds up as you do repeated calculations due to the caching.

INPUT:

- x, y – elements of the underlying Coxeter group

See also:

kazhdan_lusztig_polynomial for a faster implementation using Fokko Ducloux's Coxeter3 C++ library.

EXAMPLES:

```
sage: R.<q> = QQ[]
sage: W = WeylGroup("A3", prefix="s")
sage: [s1,s2,s3] = W.simple_reflections()
sage: KL = KazhdanLusztigPolynomial(W, q)
sage: KL.P(s2,s2*s1*s3*s2)
q + 1
```

R(*x*, *y*)

Return the Kazhdan-Lusztig R polynomial.

INPUT:

- x, y – elements of the underlying Coxeter group

EXAMPLES:

```
sage: R.<q>=QQ[]
sage: W = WeylGroup("A2", prefix="s")
sage: [s1,s2]=W.simple_reflections()
sage: KL = KazhdanLusztigPolynomial(W, q)
sage: [KL.R(x,s2*s1) for x in [1,s1,s2,s1*s2]]
[q^2 - 2*q + 1, q - 1, q - 1, 0]
```

R_tilde(*x*, *y*)

Return the Kazhdan-Lusztig $\tilde{R}$ polynomial.

Information about the $\tilde{R}$ polynomials can be found in [Dy93] and [BB05].

INPUT:

- x, y – elements of the underlying Coxeter group

EXAMPLES:

```
sage: R.<q> = QQ[]
sage: W = WeylGroup("A2", prefix="s")
sage: [s1,s2] = W.simple_reflections()
sage: KL = KazhdanLusztigPolynomial(W, q)
sage: [KL.R_tilde(x,s2*s1) for x in [1,s1,s2,s1*s2]]
[q^2, q, q, 0]
```

5.1.115 Knutson-Tao Puzzles

This module implements a generic algorithm to solve Knutson-Tao puzzles. An instance of this class will be callable: the arguments are the labels of north-east and north-west sides of the puzzle boundary; the output is the list of the fillings of the puzzle with the specified pieces.

Acknowledgements

This code was written during Sage Days 45 at ICERM with Franco Saliola, Anne Schilling, and Avinash Dalal in discussions with Allen Knutson. The code was tested afterwards by Liz Beazley and Ed Richmond.

Todo

Functionality to add:

- plotter will not plot edge labels higher than 2; e.g. in BK puzzles, the labels are 1,..., n and so in 3-step examples, none of the edge labels with 3 appear
- we should also have a 3-step puzzle pieces constructor, taken from p22 of Arxiv math/0610538
- implement the bijection from puzzles to tableaux; see for example R. Vakil, A geometric Littlewood-Richardson rule, Arxiv math/0302294 or K. Purbhoo, Puzzles, Tableaux and Mosaics, Arxiv 0705.1184.

---

sage.combinat.knutson_tao_puzzles.**BK_pieces**(*max_letter*)

The puzzle pieces used in computing the Belkale-Kumar coefficients for any partial flag variety in type A.

There are two types of puzzle pieces:

- a triangle, with each edge labeled with the same letter;
- a rhombus, with edges labeled i, j, i, j in clockwise order with $i > j$.

Each of these is rotated by 60 degrees, but not reflected.

We model the rhombus pieces as two triangles: a delta piece north-west label i, north-east label j and south label $i(j)$; and a nabla piece with south-east label i, south-west label j and north label $i(j)$.

INPUT:

- `max_letter` – positive integer specifying the number of steps in the partial flag variety, equivalently, the number of elements in the alphabet for the edge labels. The smallest label is 1.

REFERENCES:

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import BK_pieces
sage: BK_pieces(3)
Nablas : [1\1/1, 1\2(1)/2, 1\3(1)/3, 2(1)\2/1, 2\1/2(1), 2\2/2, 2\3(2)/3, 3(1)\3/1, 3(2)\3/2, 3\
Deltas : [1/1\1, 1/2\2(1), 1/3\3(1), 2(1)/1\2, 2/2(1)\1, 2/2\2, 2/3\3(2), 3(1)/1\3, 3(2)/2\3, 3/
```

class sage.combinat.knutson_tao_puzzles.**DeltaPiece**(*south*, *north_west*, *north_east*)
Bases: sage.combinat.knutson_tao_puzzles.PuzzlePiece

Delta Piece takes as input three labels, inputted as strings. They label the South, Northwest and Northeast edges, respectively.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: DeltaPiece('a','b','c')
b/a\c
```

clockwise_rotation()
Rotates the Delta piece by 120 degree clockwise.

OUTPUT:

- Delta piece

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: delta = DeltaPiece('1','2','3')
sage: delta.clockwise_rotation()
1/3\2
```

edges()
Return the tuple of edge names.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: delta = DeltaPiece('1','2','3')
sage: delta.edges()
('south', 'north_west', 'north_east')
```

half_turn_rotation()
Rotates the Delta piece by 180 degree.

OUTPUT:

- Nabla piece

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: delta = DeltaPiece('1','2','3')
sage: delta.half_turn_rotation()
3\1/2
```

sage.combinat.knutson_tao_puzzles.**HT_grassmannian_pieces**()

Defines the puzzle pieces used in computing the torus-equivariant cohomology of the Grassmannian.

REFERENCES:

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import HT_grassmannian_pieces
sage: HT_grassmannian_pieces()
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1, 1\T0|1/0]
Deltas : [0/0\0, 0/1\10, 0/T0|1\1, 1/10\0, 1/1\1, 10/0\1]
```

sage.combinat.knutson_tao_puzzles.**HT_two_step_pieces**()

Defines the puzzle pieces used in computing the equivariant two step puzzle pieces.

For the puzzle pieces, see Figure 26 on page 22 of [CoskunVakil06].

REFERENCES:

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import HT_two_step_pieces
sage: HT_two_step_pieces()
Nablas : [(21)0\21/0, 0\(21)0/21, 0\0/0, 0\10/1, 0\20/2, 10\1/0, 10\2(10)/2,
1\0/10, 1\1/1, 1\21/2, 1\T0|1/0, 2(10)\2/10, 20\2/0, 21\0/(21)0, 21\2/1, 21\T0|21/0,
21\T10|21/10, 2\0/20, 2\1/21, 2\10/2(10), 2\2/2, 2\T0|2/0, 2\T10|2/10, 2\T1|2/1]
Deltas : [(21)0/0\21, 0/0\0, 0/1\10, 0/21\(21)0, 0/2\20, 0/T0|1\1, 0/T0|21\21, 0/T0|2\2,
1/10\0, 1/1\1, 1/2\21, 1/T1|2\2, 10/0\1, 10/2\2(10), 10/T10|21\21, 10/T10|2\2, 2(10)/10\2,
2/2(10)\10, 2/20\0, 2/21\1, 2/2\2, 20/0\2, 21/(21)0\0, 21/1\2]
```

sage.combinat.knutson_tao_puzzles.**H_grassmannian_pieces**()

Defines the puzzle pieces used in computing the cohomology of the Grassmannian.

REFERENCES:

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import H_grassmannian_pieces
sage: H_grassmannian_pieces()
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

sage.combinat.knutson_tao_puzzles.**H_two_step_pieces**()

Defines the puzzle pieces used in two step flags.

This rule is currently only conjecturally true. See [BuchKreschTamvakis03].

REFERENCES:

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import H_two_step_pieces
sage: H_two_step_pieces()
Nablas : [(21)0\21/0, 0\(21)0/21, 0\0/0, 0\10/1, 0\20/2, 10\1/0, 10\2(10)/2, 1\0/10, 1\1/1, 1\21
2(10)\2/10, 20\2/0, 21\0/(21)0, 21\2/1, 2\0/20, 2\1/21, 2\10/2(10), 2\2/2]
Deltas : [(21)0/0\21, 0/0\0, 0/1\10, 0/21\(21)0, 0/2\20, 1/10\0, 1/1\1, 1/2\21, 10/0\1, 10/2\2(1
2(10)/10\2, 2/2(10)\10, 2/20\0, 2/21\1, 2/2\2, 20/0\2, 21/(21)0\0, 21/1\2]
```

sage.combinat.knutson_tao_puzzles.**K_grassmannian_pieces**()
Defines the puzzle pieces used in computing the K-theory of the Grassmannian.

REFERENCES:

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import K_grassmannian_pieces
sage: K_grassmannian_pieces()
Nablas : [0\0/0, 0\10/1, 0\K/1, 10\1/0, 1\0/10, 1\0/K, 1\1/1, K\1/0]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1, K/K\K]
```

class sage.combinat.knutson_tao_puzzles.**KnutsonTaoPuzzleSolver**(*puzzle_pieces*)
Bases: sage.structure.unique_representation.UniqueRepresentation

Returns puzzle solver function used to create all puzzles with given boundary conditions.

This class implements a generic algorithm to solve Knutson-Tao puzzles. An instance of this class will be callable: the arguments are the labels of north-east and north-west sides of the puzzle boundary; the output is the list of the fillings of the puzzle with the specified pieces.

INPUT:

- puzzle_pieces – takes either a collection of puzzle pieces or a string indicating a pre-programmed collection of puzzle pieces:
  - H – cohomology of the Grassmannian
  - HT – equivariant cohomology of the Grassmannian
  - K – K-theory
  - H2step – cohomology of the *2-step* Grassmannian
  - HT2step – equivariant cohomology of the *2-step* Grassmannian
  - BK – Belkale-Kumar puzzle pieces
- max_letter – (default: None) None or a positive integer. This is only required only for Belkale-Kumar puzzles.

EXAMPLES:

Each puzzle piece is an edge-labelled triangle oriented in such a way that it has a south edge (called a *delta* piece) or a north edge (called a *nabla* piece). For example, the puzzle pieces corresponding to the cohomology of the Grassmannian are the following:

```
sage: from sage.combinat.knutson_tao_puzzles import H_grassmannian_pieces
sage: H_grassmannian_pieces()
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

In the string representation, the nabla pieces are depicted as c\a/b, where a is the label of the north edge, b is the label of the south-east edge, c is the label of the south-west edge. A similar string representation exists for the delta pieces.

To create a puzzle solver, one specifies a collection of puzzle pieces:

```
sage: KnutsonTaoPuzzleSolver(H_grassmannian_pieces())
Knutson-Tao puzzle solver with pieces:
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

The following shorthand to create the above puzzle solver is also supported:

```
sage: KnutsonTaoPuzzleSolver('H')
Knutson-Tao puzzle solver with pieces:
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

The solver will compute all fillings of the puzzle with the given puzzle pieces. The user specifies the labels of north-east and north-west sides of the puzzle boundary and the output is a list of the fillings of the puzzle with the specified pieces. For example, there is one solution to the puzzle whose north-west and north-east edges are both labeled '0':

```
sage: ps = KnutsonTaoPuzzleSolver('H')
sage: ps('0', '0')
[{(1, 1): 0/0\0}]
```

There are two solutions to the puzzle whose north-west and north-east edges are both labeled '0101':

```
sage: ps = KnutsonTaoPuzzleSolver('H')
sage: solns = ps('0101', '0101')
sage: len(solns)
2
sage: solns.sort(key=str)
sage: solns
[{(1, 2): 1/\0  0\/1, (1, 3): 0/\0  0\/0, (3, 3): 1/1\1, (4, 4): 10/0\1, (1, 4): 1/\0  0\/1, (1,
{(1, 2): 1/\1  10\/0, (1, 3): 0/\0  1\/10, (3, 3): 0/0\0, (4, 4): 1/1\1, (1, 4): 1/\0  0\/1, (1,
```

The pieces in a puzzle filling are indexed by pairs of non-negative integers (i, j) with $1 \leq i \leq j \leq n$, where n is the length of the word labelling the triangle edge. The pieces indexed by (i, i) are the triangles along the south edge of the puzzle.

```
sage: f = solns[0]
sage: [f[i, i] for i in range(1,5)]
[0/0\0, 1/1\1, 1/1\1, 10/0\1]
```

The pieces indexed by (i, j) for $j > i$ are a pair consisting of a delta piece and nabla piece glued together along the south edge and north edge, respectively (these pairs are called *rhombi*).

```
sage: f = solns[0]
sage: f[1, 2]
1/\0  0\/1
```

There are various methods and options to display puzzle solutions. A single puzzle can be displayed using the plot method of the puzzle:

```
sage: ps = KnutsonTaoPuzzleSolver("H")
sage: puzzle = ps('0101','1001')[0]
sage: puzzle.plot()  #not tested
sage: puzzle.plot(style='fill')  #not tested
sage: puzzle.plot(style='edges')  #not tested
```

To plot several puzzle solutions, use the plot method of the puzzle solver:

```
sage: ps = KnutsonTaoPuzzleSolver('K')
sage: solns = ps('0101', '0101')
sage: ps.plot(solns)        # not tested
```

The code can also generate a PDF of a puzzle (using LaTeX and *tikz*):

```
sage: latex.extra_preamble(r'''\usepackage{tikz}''')
sage: ps = KnutsonTaoPuzzleSolver('H')
sage: solns = ps('0101', '0101')
```

```
sage: view(solns[0], viewer='pdf', tightpage=True)  # not tested
```

Below are examples of using each of the currently supported puzzles.

Cohomology of the Grassmannian:

```
sage: ps = KnutsonTaoPuzzleSolver("H")
sage: solns = ps('0101', '0101')
sage: sorted(solns, key=str)
[{(1, 2): 1/\0  0\/1, (1, 3): 0/\0  0\/0, (3, 3): 1/1\1, (4, 4): 10/0\1, (1, 4): 1/\0  0\/1, (1,
{(1, 2): 1/\1  10\/0, (1, 3): 0/\0  1\/10, (3, 3): 0/0\0, (4, 4): 1/1\1, (1, 4): 1/\0  0\/1, (1,
```

Equivariant puzzles:

```
sage: ps = KnutsonTaoPuzzleSolver("HT")
sage: solns = ps('0101', '0101')
sage: sorted(solns, key=str)
[{(1, 2): 1/\0  0\/1, (1, 3): 0/\0  0\/0, (3, 3): 0/0\0, (4, 4): 1/1\1, (1, 4): 1/\0  0\/1, (1,
{(1, 2): 1/\0  0\/1, (1, 3): 0/\0  0\/0, (3, 3): 1/1\1, (4, 4): 10/0\1, (1, 4): 1/\0  0\/1, (1,
{(1, 2): 1/\1  10\/0, (1, 3): 0/\0  1\/10, (3, 3): 0/0\0, (4, 4): 1/1\1, (1, 4): 1/\0  0\/1, (1,
```

K-Theory puzzles:

```
sage: ps = KnutsonTaoPuzzleSolver("K")
sage: solns = ps('0101', '0101')
sage: sorted(solns, key=str)
[{(1, 2): 1/\0  0\/1, (1, 3): 0/\0  0\/0, (3, 3): 1/1\1, (4, 4): 10/0\1, (1, 4): 1/\0  0\/1, (1,
{(1, 2): 1/\1  10\/0, (1, 3): 0/\0  1\/10, (3, 3): 0/0\0, (4, 4): 1/1\1, (1, 4): 1/\0  0\/1, (1,
{(1, 2): 1/\1  10\/0, (1, 3): 0/\0  1\/K, (3, 3): 1/1\1, (4, 4): 10/0\1, (1, 4): 1/\0  0\/1, (1,
```

Two-step puzzles:

```
sage: ps = KnutsonTaoPuzzleSolver("H2step")
sage: solns = ps('01201', '01021')
sage: sorted(solns, key=str)
[{(1, 2): 1/\0  0\/1, (1, 3): 2/\0  0\/2, (3, 3): 1/1\1, (4, 5): 20/\2  1\/10, (4, 4): 1/1\1, (5
{(1, 2): 1/\1  10\/0, (1, 3): 2/\1  1\/2, (3, 3): 0/0\0, (4, 5): 2(10)/\2  0\/1, (4, 4): 0/0\0,
{(1, 2): 1/\21  20\/0, (1, 3): 2/\2  21\/1, (3, 3): 1/1\1, (4, 5): 21/\2  1\/1, (4, 4): 10/0\1,
```

Two-step equivariant puzzles:

```
sage: ps = KnutsonTaoPuzzleSolver("HT2step")
sage: solns = ps('10212', '12012')
sage: sorted(solns, key=str)
[{(1, 2): 0/\(21)0  1\/2, (1, 3): 2/\1  (21)0\/0, (3, 3): 0/0\0, (4, 5): 1/\1  2\/21, (4, 4): 2/
{(1, 2): 0/\(21)0  1\/2, (1, 3): 2/\1  (21)0\/0, (3, 3): 0/0\0, (4, 5): 2/\1  1\/2, (4, 4): 1/1\
{(1, 2): 0/\(21)0  1\/2, (1, 3): 2/\1  (21)0\/0, (3, 3): 2/2\2, (4, 5): 1/\1  2\/21, (4, 4): 20/
{(1, 2): 0/\1  1\/0, (1, 3): 2/\1  1\/2, (3, 3): 0/0\0, (4, 5): 1/\1  2\/21, (4, 4): 2/2\2, (5,
{(1, 2): 0/\1  1\/0, (1, 3): 2/\1  1\/2, (3, 3): 0/0\0, (4, 5): 2/\1  1\/2, (4, 4): 1/1\1, (5, 5
{(1, 2): 0/\10  1\/1, (1, 3): 2/\10  10\/2, (3, 3): 1/1\1, (4, 5): 10/\1  2\/20, (4, 4): 2/2\2,
{(1, 2): 0/\10  1\/1, (1, 3): 2/\10  10\/2, (3, 3): 1/1\1, (4, 5): 2/\1  1\/2, (4, 4): 10/0\1, (
{(1, 2): 0/\20  21\/1, (1, 3): 2/\2  20\/0, (3, 3): 0/0\0, (4, 5): 2/\1  1\/2, (4, 4): 1/1\1, (5
{(1, 2): 0/\20  21\/1, (1, 3): 2/\2  20\/0, (3, 3): 1/1\1, (4, 5): 2/\1  1\/2, (4, 4): 10/0\1, (
{(1, 2): 0/\21  21\/0, (1, 3): 2/\2  21\/1, (3, 3): 0/0\0, (4, 5): 2/\1  1\/2, (4, 4): 1/1\1, (5
```

Belkale-Kumar puzzles (the following example is Figure 2 of [KnutsonPurbhoo10]):

```
sage: ps = KnutsonTaoPuzzleSolver('BK', 3)
sage: solns = ps('12132', '23112')
sage: len(solns)
1
```

```
sage: solns[0].south_labels()
('3', '2', '1', '2', '1')
sage: solns
[{(1, 2): 2/\3(2)  3(1)\/1, (1, 3): 1/\3(1)  3(2)\/2, (3, 3): 1/1\1, (4, 5): 1/\1  2\/2(1), (4,
```

plot(*puzzles*)

Return plot of puzzles.

INPUT:

- `puzzles` – list of puzzles

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
sage: ps = KnutsonTaoPuzzleSolver('K')
sage: solns = ps('0101', '0101')
sage: ps.plot(solns)        # not tested
```

puzzle_pieces()

The puzzle pieces used for filling in the puzzles.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
sage: ps = KnutsonTaoPuzzleSolver('H')
sage: ps.puzzle_pieces()
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

solutions(*lamda*, *mu*, *algorithm='strips'*)

TESTS:

```
sage: from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
sage: ps = KnutsonTaoPuzzleSolver("H")
sage: ps('0101','1001')
[{(1, 2): 1/\1  10\/0, (1, 3): 0/\10  1\/1, (3, 3): 1/1\1, (4, 4): 10/0\1, (1, 4): 1/\1  10\
(1, 1): 0/1\10, (2, 3): 1/\0  0\/1, (2, 2): 0/0\0, (3, 4): 0/\0  1\/10, (2, 4): 0/\0  0\/0}]
sage: ps('0101','1001',algorithm='pieces')
[{(1, 2): 1/\1  10\/0, (1, 3): 0/\10  1\/1, (3, 3): 1/1\1, (4, 4): 10/0\1, (1, 4): 1/\1  10\
(2, 4): 0/\0  0\/0, (2, 3): 1/\0  0\/1, (2, 2): 0/0\0, (3, 4): 0/\0  1\/10, (1, 1): 0/1\10}]
```

structure_constants(*lamda*, *mu*, *nu=None*)

Compute cohomology structure coefficients from puzzles.

INPUT:

- `pieces` – puzzle pieces to be used
- `lambda`, `mu` – edge labels of puzzle for northwest and north east side
- `nu` – (default: `None`) If `nu` is not specified a dictionary is returned with the structure coefficients corresponding to all south labels; if `nu` is given, only the coefficients with the specified label is returned.

OUTPUT: dictionary

EXAMPLES:

Note: In order to standardize the output of the following examples, we output a sorted list of items from the dictionary instead of the dictionary itself.

Grassmannian cohomology:

```
sage: ps = KnutsonTaoPuzzleSolver('H')
sage: cp = ps.structure_constants('0101', '0101')
sage: sorted(cp.items(), key=str)
[(('0', '1', '1', '0'), 1), (('1', '0', '0', '1'), 1)]
sage: ps.structure_constants('001001', '001010', '010100')
1
```

Equivariant cohomology:

```
sage: ps = KnutsonTaoPuzzleSolver('HT')
sage: cp = ps.structure_constants('0101', '0101')
sage: sorted(cp.items(), key=str)
[(('0', '1', '0', '1'), y2 - y3),
(('0', '1', '1', '0'), 1),
(('1', '0', '0', '1'), 1)]
```

K-theory:

```
sage: ps = KnutsonTaoPuzzleSolver('K')
sage: cp = ps.structure_constants('0101', '0101')
sage: sorted(cp.items(), key=str)
[(('0', '1', '1', '0'), 1), (('1', '0', '0', '1'), 1), (('1', '0', '1', '0'), -1)]
```

Two-step:

```
sage: ps = KnutsonTaoPuzzleSolver('H2step')
sage: cp = ps.structure_constants('01122', '01122')
sage: sorted(cp.items(), key=str)
[(('0', '1', '1', '2', '2'), 1)]
sage: cp = ps.structure_constants('01201', '01021')
sage: sorted(cp.items(), key=str)
[(('0', '2', '1', '1', '0'), 1),
 (('1', '2', '0', '0', '1'), 1),
 (('2', '0', '1', '0', '1'), 1)]
```

Two-step equivariant:

```
sage: ps = KnutsonTaoPuzzleSolver('HT2step')
sage: cp = ps.structure_constants('10212', '12012')
sage: sorted(cp.items(), key=str)
[(('1', '2', '0', '1', '2'), y1*y2 - y2*y3 - y1*y4 + y3*y4),
 (('1', '2', '0', '2', '1'), y1 - y3),
 (('1', '2', '1', '0', '2'), y2 - y4),
 (('1', '2', '1', '2', '0'), 1),
 (('1', '2', '2', '0', '1'), 1),
 (('2', '1', '0', '1', '2'), y1 - y3),
 (('2', '1', '1', '0', '2'), 1)]
```

class sage.combinat.knutson_tao_puzzles.**NablaPiece**(*north*, *south_east*, *south_west*)

Bases: sage.combinat.knutson_tao_puzzles.PuzzlePiece

Nabla Piece takes as input three labels, inputted as strings. They label the North, Southeast and Southwest edges, respectively.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import NablaPiece
sage: NablaPiece('a','b','c')
c\a/b
```

clockwise_rotation()
Rotates the Nabla piece by 120 degree clockwise.

OUTPUT:

•Nabla piece

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import NablaPiece
sage: nabla = NablaPiece('1','2','3')
sage: nabla.clockwise_rotation()
2\3/1
```

edges()
Return the tuple of edge names.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import NablaPiece
sage: nabla = NablaPiece('1','2','3')
sage: nabla.edges()
('north', 'south_east', 'south_west')
```

half_turn_rotation()
Rotates the Nabla piece by 180 degree.

OUTPUT:

•Delta piece

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import NablaPiece
sage: nabla = NablaPiece('1','2','3')
sage: nabla.half_turn_rotation()
2/1\3
```

class sage.combinat.knutson_tao_puzzles.**PuzzleFilling**(*north_west_labels*, *north_east_labels*)
Bases: object

Create partial puzzles and provides methods to build puzzles from them.

add_piece(*piece*)
Adds piece to partial puzzle.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece, PuzzleFilling
sage: piece = DeltaPiece('0','1','0')
sage: P = PuzzleFilling('0101','0101'); P
{}
sage: P.add_piece(piece); P
{(1, 4): 1/0\0}
```

add_pieces(*pieces*)
Adds piece to partial puzzle.

INPUT:

•pieces – tuple of pieces

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece, PuzzleFilling
sage: P = PuzzleFilling('0101','0101'); P
{}
sage: piece = DeltaPiece('0','1','0')
sage: pieces = [piece,piece]
sage: P.add_pieces(pieces)
sage: P
{(2, 4): 1/0\0, (1, 4): 1/0\0}
```

contribution()

Return equivariant contributions from `self` in polynomial ring.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
sage: ps = KnutsonTaoPuzzleSolver("HT")
sage: puzzles = ps('0101','1001')
sage: sorted([p.contribution() for p in puzzles], key=str)
[1, y1 - y3]
```

copy()

Return copy of `self`.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece, PuzzleFilling
sage: piece = DeltaPiece('0','1','0')
sage: P = PuzzleFilling('0101','0101'); P
{}
sage: PP = P.copy()
sage: P.add_piece(piece); P
{(1, 4): 1/0\0}
sage: PP
{}
```

is_completed()

Whether partial puzzle is complete (completely filled) or not.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzleFilling
sage: P = PuzzleFilling('0101','0101')
sage: P.is_completed()
False

sage: from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
sage: ps = KnutsonTaoPuzzleSolver("H")
sage: puzzle = ps('0101','1001')[0]
sage: puzzle.is_completed()
True
```

is_in_south_edge()

Checks whether kink coordinates of partial puzzle is in south corner.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzleFilling
sage: P = PuzzleFilling('0101','0101')
sage: P.is_in_south_edge()
False
```

kink_coordinates()
Provides the coordinates of the kinks.

The kink coordinates are the coordinates up to which the puzzle has already been built. The kink starts in the north corner and then moves down the diagonals as the puzzles is built.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzleFilling
sage: P = PuzzleFilling('0101','0101')
sage: P
{}
sage: P.kink_coordinates()
(1, 4)
```

north_east_label_of_kink()
Return north east label of kink.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzleFilling
sage: P = PuzzleFilling('0101','0101')
sage: P.north_east_label_of_kink()
'0'
```

north_west_label_of_kink()
Return north-west label of kink.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzleFilling
sage: P = PuzzleFilling('0101','0101')
sage: P.north_west_label_of_kink()
'1'
```

plot(*labels=True*, *style='fill'*)
Plots completed puzzle.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
sage: ps = KnutsonTaoPuzzleSolver("H")
sage: puzzle = ps('0101','1001')[0]
sage: puzzle.plot()  #not tested
sage: puzzle.plot(style='fill')  #not tested
sage: puzzle.plot(style='edges')  #not tested
```

south_labels()
Return south labels for completed puzzle.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import KnutsonTaoPuzzleSolver
sage: ps = KnutsonTaoPuzzleSolver("H")
sage: ps('0101','1001')[0].south_labels()
('1', '0', '1', '0')
```

class sage.combinat.knutson_tao_puzzles.**PuzzlePiece**
Bases: object

Abstract class for puzzle pieces.

This abstract class contains information on how to test equality of puzzle pieces, and sets color and plotting options.

border()
Returns the border of `self`.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: delta = DeltaPiece('a','b','c')
sage: delta.border()
('a', 'b', 'c')
```

color()
Returns the color of `self`.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: delta = DeltaPiece('a','b','c')
sage: delta.color()
'white'
sage: delta = DeltaPiece('0','0','0')
sage: delta.color()
'red'
sage: delta = DeltaPiece('1','1','1')
sage: delta.color()
'blue'
sage: delta = DeltaPiece('2','2','2')
sage: delta.color()
'green'
sage: delta = DeltaPiece('2','K','2')
sage: delta.color()
'orange'
sage: delta = DeltaPiece('2','T1/2','2')
sage: delta.color()
'yellow'
```

edge_color(*edge*)
Color of the specified edge of `self` (to be used when plotting the piece).

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: delta = DeltaPiece('1','0','10')
sage: delta.edge_color('south')
'blue'
sage: delta.edge_color('north_west')
'red'
sage: delta.edge_color('north_east')
'white'
```

edge_label(*edge*)
Return the edge label of `edge`.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece
sage: delta = DeltaPiece('2','K','2')
sage: delta.edge_label('south')
'2'
sage: delta.edge_label('north_east')
```

```
'2'
sage: delta.edge_label('north_west')
'K'
```

class sage.combinat.knutson_tao_puzzles.**PuzzlePieces**(*forbidden_border_labels=None*)

Bases: object

Construct a valid set of puzzle pieces.

This class constructs the set of valid puzzle pieces. It can take a list of forbidden border labels as input. These labels are forbidden from appearing on the south edge of a puzzle filling. The user can add valid nabla or delta pieces and specify which rotations of these pieces are legal. For example, `rotations=0` does not add any additional pieces (only the piece itself), `rotations=60` adds six pieces (the pieces and its rotations by 60, 120, 180, 240, 300), etc..

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces, NablaPiece
sage: forbidden_border_labels = ['10']
sage: pieces = PuzzlePieces(forbidden_border_labels)
sage: pieces.add_piece(NablaPiece('0','0','0'), rotations=60)
sage: pieces.add_piece(NablaPiece('1','1','1'), rotations=60)
sage: pieces.add_piece(NablaPiece('1','0','10'), rotations=60)
sage: pieces
Nablas : [0\0/0, 0\10/1, 10\1/0, 1\0/10, 1\1/1]
Deltas : [0/0\0, 0/1\10, 1/10\0, 1/1\1, 10/0\1]
```

The user can obtain the list of valid rhombi pieces as follows:

```
sage: sorted([p for p in pieces.rhombus_pieces()], key=str)
[0/\0  0\/0, 0/\0  1\/10, 0/\10  10\/0, 0/\10  1\/1, 1/\0  0\/1,
1/\1  10\/0, 1/\1  1\/1, 10/\1  0\/0, 10/\1  1\/10]
```

add_T_piece(*label1*, *label2*)

Adds a nabla and delta piece with `label1` and `label2`.

This method adds a nabla piece with edges `label2`T‘‘label1‘‘|‘‘label2‘‘ / `label1`. and a delta piece with edges `label1`/ T‘‘label1‘‘|‘‘label2‘‘ `label2`. It also adds T‘‘label1‘‘|‘‘label2‘‘ to the forbidden list.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces
sage: pieces = PuzzlePieces()
sage: pieces.add_T_piece('1','3')
sage: pieces
Nablas : [3\T1|3/1]
Deltas : [1/T1|3\3]
sage: pieces._forbidden_border_labels
['T1|3']
```

add_forbidden_label(*label*)

Adds forbidden border labels.

INPUT:

- `label` – string specifying a new forbidden label

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces
sage: pieces = PuzzlePieces()
sage: pieces.add_forbidden_label('1')
sage: pieces._forbidden_border_labels
['1']
sage: pieces.add_forbidden_label('2')
sage: pieces._forbidden_border_labels
['1', '2']
```

add_piece(*piece*, *rotations=120*)

Adds `piece` to the list of pieces.

INPUT:

- `piece` – a nabla piece or a delta piece
- `rotations` – (default: 120) 0, 60, 120, 180

The user can add valid nabla or delta pieces and specify which rotations of these pieces are legal. For example, `rotations=0` does not add any additional pieces (only the piece itself), `rotations=60` adds six pieces (namely three delta and three nabla pieces), while `rotations=120` adds only delta or nabla (depending on which piece `self` is). `rotations=180` adds the piece and its 180 degree rotation, i.e. one delta and one nabla piece.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces, DeltaPiece
sage: delta = DeltaPiece('a','b','c')
sage: pieces = PuzzlePieces()
sage: pieces
Nablas : []
Deltas : []
sage: pieces.add_piece(delta)
sage: pieces
Nablas : []
Deltas : [a/c\b, b/a\c, c/b\a]

sage: pieces = PuzzlePieces()
sage: pieces.add_piece(delta,rotations=0)
sage: pieces
Nablas : []
Deltas : [b/a\c]

sage: pieces = PuzzlePieces()
sage: pieces.add_piece(delta,rotations=60)
sage: pieces
Nablas : [a\b/c, b\c/a, c\a/b]
Deltas : [a/c\b, b/a\c, c/b\a]
```

boundary_deltas()

Returns deltas with south edges not in the forbidden list.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces, DeltaPiece
sage: pieces = PuzzlePieces(['a'])
sage: delta = DeltaPiece('a','b','c')
sage: pieces.add_piece(delta,rotations=60)
sage: sorted([p for p in pieces.boundary_deltas()], key=str)
[a/c\b, c/b\a]
```

delta_pieces()
Returns the delta pieces as a set.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces, DeltaPiece
sage: pieces = PuzzlePieces()
sage: delta = DeltaPiece('a','b','c')
sage: pieces.add_piece(delta,rotations=60)
sage: sorted([p for p in pieces.delta_pieces()], key=str)
[a/c\b, b/a\c, c/b\a]
```

nabla_pieces()
Returns the nabla pieces as a set.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces, DeltaPiece
sage: pieces = PuzzlePieces()
sage: delta = DeltaPiece('a','b','c')
sage: pieces.add_piece(delta,rotations=60)
sage: sorted([p for p in pieces.nabla_pieces()], key=str)
[a\b/c, b\c/a, c\a/b]
```

rhombus_pieces()
Returns a list of all allowable rhombus pieces.

Allowable rhombus pieces are those where the south edge of the delta piece equals the north edge of the nabla piece.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import PuzzlePieces, DeltaPiece
sage: pieces = PuzzlePieces()
sage: delta = DeltaPiece('a','b','c')
sage: pieces.add_piece(delta,rotations=60)
sage: sorted([p for p in pieces.rhombus_pieces()], key=str)
[a/\b  b\/a, b/\c  c\/b, c/\a  a\/c]
```

class sage.combinat.knutson_tao_puzzles.**RhombusPiece**(*north_piece*, *south_piece*)
Bases: sage.combinat.knutson_tao_puzzles.PuzzlePiece

Class of rhombi pieces.

To construct a rhombus piece we input a delta and a nabla piece. The delta and nabla pieces are joined along the south and north edge, respectively.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece, NablaPiece, RhombusPiece
sage: delta = DeltaPiece('1','2','3')
sage: nabla = NablaPiece('4','5','6')
sage: RhombusPiece(delta,nabla)
2/\3  6\/5
```

edges()
Return the tuple of edge names.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece, NablaPiece, RhombusPiece
sage: delta = DeltaPiece('1','2','3')
sage: nabla = NablaPiece('4','5','6')
```

```
sage: RhombusPiece(delta,nabla).edges()
('north_west', 'north_east', 'south_east', 'south_west')
```

north_piece()
Return the north piece.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece, NablaPiece, RhombusPiece
sage: delta = DeltaPiece('1','2','3')
sage: nabla = NablaPiece('4','5','6')
sage: r = RhombusPiece(delta,nabla)
sage: r.north_piece()
2/1\3
```

south_piece()
Return the south piece.

EXAMPLES:

```
sage: from sage.combinat.knutson_tao_puzzles import DeltaPiece, NablaPiece, RhombusPiece
sage: delta = DeltaPiece('1','2','3')
sage: nabla = NablaPiece('4','5','6')
sage: r = RhombusPiece(delta,nabla)
sage: r.south_piece()
6\4/5
```

5.1.116 Lyndon words

sage.combinat.lyndon_word.**LyndonWord**(*data*, *check=True*)
Construction of a Lyndon word.

INPUT:

- data - list
- check - bool (optional, default: True) if True, a verification that the input data represent a Lyndon word.

OUTPUT:

A Lyndon word.

EXAMPLES:

```
sage: LyndonWord([1,2,2])
word: 122
sage: LyndonWord([1,2,3])
word: 123
sage: LyndonWord([2,1,2,3])
Traceback (most recent call last):
...
ValueError: not a Lyndon word
```

If check is False, then no verification is done:

```
sage: LyndonWord([2,1,2,3], check=False)
word: 2123
```

sage.combinat.lyndon_word.**LyndonWords**(*e=None*, *k=None*)
Returns the combinatorial class of Lyndon words.

A Lyndon word w is a word that is lexicographically less than all of its rotations. Equivalently, whenever w is split into two non-empty substrings, w is lexicographically less than the right substring.

INPUT:

- no input at all

or

- e - integer, size of alphabet
- k - integer, length of the words

or

- e - a composition

OUTPUT:

A combinatorial class of Lyndon words.

EXAMPLES:

```
sage: LyndonWords()
Lyndon words
```

If e is an integer, then e specifies the length of the alphabet; k must also be specified in this case:

```
sage: LW = LyndonWords(3,3); LW
Lyndon words from an alphabet of size 3 of length 3
sage: LW.first()
word: 112
sage: LW.last()
word: 233
sage: LW.random_element() # random
word: 112
sage: LW.cardinality()
8
```

If e is a (weak) composition, then it returns the class of Lyndon words that have evaluation e:

```
sage: LyndonWords([2, 0, 1]).list()
[word: 113]
sage: LyndonWords([2, 0, 1, 0, 1]).list()
[word: 1135, word: 1153, word: 1315]
sage: LyndonWords([2, 1, 1]).list()
[word: 1123, word: 1132, word: 1213]
```

class sage.combinat.lyndon_word.**LyndonWords_class**(*alphabet=None*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The set of all Lyndon words.

class sage.combinat.lyndon_word.**LyndonWords_evaluation**(*e*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The set of Lyndon words on a fixed multiset of letters.

EXAMPLES:

```
sage: L = LyndonWords([1,2,1])
sage: L
Lyndon words with evaluation [1, 2, 1]
```

```
sage: L.list()
[word: 1223, word: 1232, word: 1322]
```

cardinality()

Returns the number of Lyndon words with the evaluation e.

EXAMPLES:

```
sage: LyndonWords([]).cardinality()
0
sage: LyndonWords([2,2]).cardinality()
1
sage: LyndonWords([2,3,2]).cardinality()
30
```

Check to make sure that the count matches up with the number of Lyndon words generated.

```
sage: comps = [[],[2,2],[3,2,7],[4,2]]+Compositions(4).list()
sage: lws = [ LyndonWords(comp) for comp in comps]
sage: all( [ lw.cardinality() == len(lw.list()) for lw in lws] )
True
```

class sage.combinat.lyndon_word.**LyndonWords_nk**(n, k)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Lyndon words of fixed length n over the alphabet $1, 2, ..., k$.

EXAMPLES:

```
sage: L = LyndonWords(3, 4)
sage: L.list()
[word: 1112,
 word: 1113,
 word: 1122,
 word: 1123,
 ...
 word: 1333,
 word: 2223,
 word: 2233,
 word: 2333]
```

cardinality()

TESTS:

```
sage: [ LyndonWords(3,i).cardinality() for i in range(1, 11) ]
[3, 3, 8, 18, 48, 116, 312, 810, 2184, 5880]
```

sage.combinat.lyndon_word.**StandardBracketedLyndonWords**(n, k)

Returns the combinatorial class of standard bracketed Lyndon words from [1, ..., n] of length k. These are in one to one correspondence with the Lyndon words and form a basis for the subspace of degree k of the free Lie algebra of rank n.

EXAMPLES:

```
sage: SBLW33 = StandardBracketedLyndonWords(3,3); SBLW33
Standard bracketed Lyndon words from an alphabet of size 3 of length 3
sage: SBLW33.first()
[1, [1, 2]]
sage: SBLW33.last()
[[2, 3], 3]
```

```
sage: SBLW33.cardinality()
8
sage: SBLW33.random_element()
[1, [1, 2]]
```

class sage.combinat.lyndon_word.**StandardBracketedLyndonWords_nk**(*n*, *k*)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

TESTS:
```
sage: SBLW = StandardBracketedLyndonWords(3, 2)
sage: SBLW == loads(dumps(SBLW))
True
```

cardinality()
EXAMPLES:
```
sage: StandardBracketedLyndonWords(3, 3).cardinality()
8
sage: StandardBracketedLyndonWords(3, 4).cardinality()
18
```

sage.combinat.lyndon_word.**standard_bracketing**(*lw*)
Returns the standard bracketing of a Lyndon word lw.

EXAMPLES:
```
sage: import sage.combinat.lyndon_word as lyndon_word
sage: map( lyndon_word.standard_bracketing, LyndonWords(3,3) )
[[1, [1, 2]],
 [1, [1, 3]],
 [[1, 2], 2],
 [1, [2, 3]],
 [[1, 3], 2],
 [[1, 3], 3],
 [2, [2, 3]],
 [[2, 3], 3]]
```

5.1.117 Combinatorics on matrices

- *Dancing Links internal pyx code*
- *Dancing links C++ wrapper*
- *Hadamard matrices*
- *Latin Squares*

5.1.118 Combinatorics on matrix features that are imported by default in the interpreter namespace

5.1.119 Dancing Links internal pyx code

class sage.combinat.matrices.dancing_links.**dancing_linksWrapper**
Bases: object

A simple class that implements dancing links.

The main methods to list the solutions are `search()` and `get_solution()`. You can also use `number_of_solutions()` to count them.

This class simply wraps a C++ implementation of Carlo Hamalainen.

get_solution()
Return the current solution.

After a new solution is found using the method `search()` this method return the rows that make up the current solution.

TESTS:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2]]
sage: rows+= [[0,2]]
sage: rows+= [[1]]
sage: rows+= [[3]]
sage: x = dlx_solver(rows)
sage: print x.search()
1
sage: print x.get_solution()
[3, 0]
```

number_of_solutions(*ncpus=1*, *column=0*)
Return the number of distinct solutions.

INPUT:

- `ncpus` – integer (default: `1`), maximal number of subprocesses to use at the same time. If $ncpus > 1$ the dancing links problem is split into independent subproblems to allow parallel computation.
- `column` – integer (default: `0`), the column used to split the problem (ignored if `ncpus` is `1`)

OUPUT:

integer

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2]]
sage: rows += [[0,2]]
sage: rows += [[1]]
sage: rows += [[3]]
sage: x = dlx_solver(rows)
sage: x.number_of_solutions()
2

sage: rows = [[0,1,2], [3,4,5], [0,1], [2,3,4,5], [0], [1,2,3,4,5]]
sage: x = dlx_solver(rows)
sage: x.number_of_solutions(ncpus=2, column=3)
3
```

TESTS:

```
sage: dlx_solver([]).number_of_solutions()
0
```

rows()
Return the list of rows.

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [1,2], [0]]
sage: x = dlx_solver(rows)
sage: x.rows()
[[0, 1, 2], [1, 2], [0]]
```

search()

Search for a new solution.

Return `1` if a new solution is found and `0` otherwise. To recover the solution, use the method `get_solution()`.

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2]]
sage: rows+= [[0,2]]
sage: rows+= [[1]]
sage: rows+= [[3]]
sage: x = dlx_solver(rows)
sage: print x.search()
1
sage: print x.get_solution()
[3, 0]
```

TESTS:

Test that trac ticket #11814 is fixed:

```
sage: dlx_solver([]).search()
0
sage: dlx_solver([[]]).search()
0
```

If search is called once too often, it keeps returning 0:

```
sage: x = dlx_solver([[0]])
sage: x.search()
1
sage: x.search()
0
sage: x.search()
0
```

solutions_iterator()

Return an iterator of the solutions.

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [3,4,5], [0,1], [2,3,4,5], [0], [1,2,3,4,5]]
sage: d = dlx_solver(rows)
sage: list(d.solutions_iterator())
[[0, 1], [2, 3], [4, 5]]
```

split(*column*)

Return a dict of rows solving independent cases.

For each `i`-th row containing a `1` in the `column`, the dict associates the list of rows where each other row containing a `1` in the same `column` is replaced by the empty list.

This is used for parallel computations.

INPUT:

- `column` – integer, the column used to split the problem

OUTPUT:

dict where keys are row numbers and values are lists of rows

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2], [3,4,5], [0,1], [2,3,4,5], [0], [1,2,3,4,5]]
sage: d = dlx_solver(rows)
sage: d
Dancing links solver for 6 columns and 6 rows
sage: sorted(d.solutions_iterator())
[[0, 1], [2, 3], [4, 5]]
```

After the split each subproblem has the same number of columns and rows as the orginal one:

```
sage: D = d.split(0)
sage: D
{0: [[0, 1, 2], [3, 4, 5], [], [2, 3, 4, 5], [], [1, 2, 3, 4, 5]],
 2: [[], [3, 4, 5], [0, 1], [2, 3, 4, 5], [], [1, 2, 3, 4, 5]],
 4: [[], [3, 4, 5], [], [2, 3, 4, 5], [0], [1, 2, 3, 4, 5]]}
```

The (disjoint) union of the solutions of the subproblems is equal to the set of solutions shown above:

```
sage: for a in D.values(): list(dlx_solver(a).solutions_iterator())
[[0, 1]]
[[2, 3]]
[[4, 5]]
```

sage.combinat.matrices.dancing_links.**dlx_solver**(*rows*)

Internal-use wrapper for the dancing links C++ code.

EXAMPLES:

```
sage: from sage.combinat.matrices.dancing_links import dlx_solver
sage: rows = [[0,1,2]]
sage: rows+= [[0,2]]
sage: rows+= [[1]]
sage: rows+= [[3]]
sage: x = dlx_solver(rows)
sage: print x.search()
1
sage: print x.get_solution()
[3, 0]
sage: print x.search()
1
sage: print x.get_solution()
[3, 1, 2]
sage: print x.search()
0
```

sage.combinat.matrices.dancing_links.**make_dlxwrapper**(*s*)

Create a dlx wrapper from a Python *string* s.

This was historically used in unpickling and is kept for backwards compatibility. We expect s to be `dumps(rows)` where rows is the list of rows used to instantiate the object.

TESTS:

```
sage: from sage.combinat.matrices.dancing_links import make_dlxwrapper
sage: rows = [[0,1,2]]
sage: x = make_dlxwrapper(dumps(rows))
sage: print x.__str__()
Dancing links solver for 3 columns and 1 rows
```

5.1.120 Dancing links C++ wrapper

sage.combinat.matrices.dlxcpp.**AllExactCovers**(*M*)

Solves the exact cover problem on the matrix M (treated as a dense binary matrix).

EXAMPLES: No exact covers:

```
sage: M = Matrix([[1,1,0],[1,0,1],[0,1,1]])
sage: print [cover for cover in AllExactCovers(M)]
[]
```

Two exact covers:

```
sage: M = Matrix([[1,1,0],[1,0,1],[0,0,1],[0,1,0]])
sage: print [cover for cover in AllExactCovers(M)]
[[(1, 1, 0), (0, 0, 1)], [(1, 0, 1), (0, 1, 0)]]
```

sage.combinat.matrices.dlxcpp.**DLXCPP**(*rows*)

Solves the Exact Cover problem by using the Dancing Links algorithm described by Knuth.

Consider a matrix M with entries of 0 and 1, and compute a subset of the rows of this matrix which sum to the vector of all 1's.

The dancing links algorithm works particularly well for sparse matrices, so the input is a list of lists of the form:

```
[
 [i_11,i_12,...,i_1r]
 ...
 [i_m1,i_m2,...,i_ms]
]
```

where M[j][i_jk] = 1.

The first example below corresponds to the matrix:

```
1110
1010
0100
0001
```

which is exactly covered by:

```
1110
0001
```

and

```
1010
0100
0001
```

If soln is a solution given by DLXCPP(rows) then

[rows[soln[0]], rows[soln[1]], ... rows[soln[len(soln)-1]]]

is an exact cover.

Solutions are given as a list

EXAMPLES:

```
sage: rows = [[0,1,2]]
sage: rows+= [[0,2]]
sage: rows+= [[1]]
sage: rows+= [[3]]
sage: print [ x for x in DLXCPP(rows) ]
[[3, 0], [3, 1, 2]]
```

sage.combinat.matrices.dlxcpp.**OneExactCover**(*M*)

Solves the exact cover problem on the matrix M (treated as a dense binary matrix).

EXAMPLES:

```
sage: M = Matrix([[1,1,0],[1,0,1],[0,1,1]])  #no exact covers
sage: print OneExactCover(M)
None
sage: M = Matrix([[1,1,0],[1,0,1],[0,0,1],[0,1,0]]) #two exact covers
sage: print OneExactCover(M)
[(1, 1, 0), (0, 0, 1)]
```

5.1.121 Hadamard matrices

A Hadamard matrix is an $n \times n$ matrix H whose entries are either $+1$ or -1 and whose rows are mutually orthogonal. For example, the matrix H_2 defined by

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

is a Hadamard matrix. An $n \times n$ matrix H whose entries are either $+1$ or -1 is a Hadamard matrix if and only if:

1. $|det(H)| = n^{n/2}$ or
2. $H * H^t = n \cdot I_n$, where I_n is the identity matrix.

In general, the tensor product of an $m \times m$ Hadamard matrix and an $n \times n$ Hadamard matrix is an $(mn) \times (mn)$ matrix. In particular, if there is an $n \times n$ Hadamard matrix then there is a $(2n) \times (2n)$ Hadamard matrix (since one may tensor with H_2). This particular case is sometimes called the Sylvester construction.

The Hadamard conjecture (possibly due to Paley) states that a Hadamard matrix of order n exists if and only if $n = 1, 2$ or n is a multiple of 4.

The module below implements the Paley constructions (see for example [Hora]) and the Sylvester construction. It also allows you to pull a Hadamard matrix from the database at [HadaSloa].

AUTHORS:

- David Joyner (2009-05-17): initial version

REFERENCES:

sage.combinat.matrices.hadamard_matrix.**GS_skew_hadamard_smallcases**(*n*, *existence=False*, *check=True*)

Data for Williamson-Goethals-Seidel construction of skew Hadamard matrices

Here we keep the data for this construction. Namely, it needs 4 circulant matrices with extra properties, as described in sage.combinat.matrices.hadamard_matrix.williamson_goethals_seidel_skew_hadamard_ Matrices for $n = 36$ and 52 are given in [GS70s]. Matrices for $n = 92$ are given in [Wall71].

INPUT:

- n – the order of the matrix
- existence – if true (default), only check that we can do the construction
- check – if true (default), check the result.

TESTS:

```
sage: from sage.combinat.matrices.hadamard_matrix import GS_skew_hadamard_smallcases
sage: GS_skew_hadamard_smallcases(36)
36 x 36 dense matrix over Integer Ring...
sage: GS_skew_hadamard_smallcases(52)
52 x 52 dense matrix over Integer Ring...
sage: GS_skew_hadamard_smallcases(92)
92 x 92 dense matrix over Integer Ring...
sage: GS_skew_hadamard_smallcases(100)
```

sage.combinat.matrices.hadamard_matrix.**RSHCD_324**(*e*)

Return a size 324x324 Regular Symmetric Hadamard Matrix with Constant Diagonal.

We build the matrix M for the case $n = 324$, $\epsilon = 1$ directly from JankoKharaghaniTonchevGraph and for the case $\epsilon = -1$ from the "twist" M' of M, using Lemma 11 in [HX10]. Namely, it turns out that the matrix

$$M' = \begin{pmatrix} M_{12} & M_{11} \\ M_{11}^\top & M_{21} \end{pmatrix}, \quad \text{where} \quad M = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix},$$

and the M_{ij} are 162x162-blocks, also RSHCD, its diagonal blocks having zero row sums, as needed by [loc.cit.]. Interestingly, the corresponding $(324, 152, 70, 72)$-strongly regular graph has a vertex-transitive automorphism group of order 2592, twice the order of the (intransitive) automorphism group of the graph corresponding to M. Cf. [CP16].

INPUT:

- e – one of -1 or $+1$, equal to the value of ϵ

TESTS:

```
sage: from sage.combinat.matrices.hadamard_matrix import RSHCD_324, is_hadamard_matrix
sage: for e in [1,-1]: # long time
....:     M = RSHCD_324(e) # long time
....:     print M==M.T,is_hadamard_matrix(M),all([M[i,i]==1 for i in xrange(324)]) # long time
....:     print set(map(sum,M)) # long time
True True True
set([18])
True True True
set([-18])
```

REFERENCE:

sage.combinat.matrices.hadamard_matrix.**hadamard_matrix**(*n*, *existence=False*, *check=True*)

This function is available as hadamard_matrix(...) and matrix.hadamard(...).

Tries to construct a Hadamard matrix using a combination of Paley and Sylvester constructions.

INPUT:

- n (integer) – dimension of the matrix

- `existence` (boolean) – whether to build the matrix or merely query if a construction is available in Sage. When set to `True`, the function returns:
  - `True` – meaning that Sage knows how to build the matrix
  - `Unknown` – meaning that Sage does not know how to build the matrix, although the matrix may exist (see `sage.misc.unknown`).
  - `False` – meaning that the matrix does not exist.
- `check` (boolean) – whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.

EXAMPLES:

```
sage: hadamard_matrix(12).det()
2985984
sage: 12^6
2985984
sage: hadamard_matrix(1)
[1]
sage: hadamard_matrix(2)
[ 1  1]
[ 1 -1]
sage: hadamard_matrix(8) # random
[ 1  1  1  1  1  1  1  1]
[ 1 -1  1 -1  1 -1  1 -1]
[ 1  1 -1 -1  1  1 -1 -1]
[ 1 -1 -1  1  1 -1 -1  1]
[ 1  1  1  1 -1 -1 -1 -1]
[ 1 -1  1 -1 -1  1 -1  1]
[ 1  1 -1 -1 -1 -1  1  1]
[ 1 -1 -1  1 -1  1  1 -1]
sage: hadamard_matrix(8).det() == 8^4
True
```

We note that the method $hadamard_matrix()$ returns a normalised Hadamard matrix (the entries in the first row and column are all +1)

```
sage: hadamard_matrix(12) # random
[ 1  1| 1  1| 1  1| 1  1| 1  1| 1  1]
[ 1 -1|-1  1|-1  1|-1  1|-1  1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1 -1| 1  1|-1 -1|-1 -1| 1  1]
[ 1  1|-1 -1| 1 -1|-1  1|-1  1| 1 -1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1  1| 1 -1| 1  1|-1 -1|-1 -1]
[ 1  1| 1 -1|-1 -1| 1 -1|-1  1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1|-1 -1| 1  1| 1 -1| 1  1|-1 -1]
[ 1  1|-1  1| 1 -1|-1 -1| 1 -1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1|-1 -1|-1 -1| 1  1| 1 -1| 1  1]
[ 1  1|-1  1|-1  1| 1 -1|-1 -1| 1 -1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1  1|-1 -1|-1 -1| 1  1| 1 -1]
[ 1  1| 1 -1|-1  1|-1  1| 1 -1|-1 -1]
```

TESTS:

```
sage: matrix.hadamard(10,existence=True)
False
sage: matrix.hadamard(12,existence=True)
True
sage: matrix.hadamard(92,existence=True)
Unknown
sage: matrix.hadamard(10)
Traceback (most recent call last):
...
ValueError: The Hadamard matrix of order 10 does not exist
```

sage.combinat.matrices.hadamard_matrix.**hadamard_matrix_paleyI**(*n*, *normalize=True*)

Implements the Paley type I construction.

The Paley type I case corresponds to the case $p \cong 3 \mod 4$ for a prime p (see [Hora]).

INPUT:

- n – the matrix size
- normalize (boolean) – whether to normalize the result.

EXAMPLES:

We note that this method by default returns a normalised Hadamard matrix

```
sage: from sage.combinat.matrices.hadamard_matrix import hadamard_matrix_paleyI
sage: hadamard_matrix_paleyI(4)
[ 1  1  1  1]
[ 1 -1  1 -1]
[ 1 -1 -1  1]
[ 1  1 -1 -1]
```

Otherwise, it returns a skew Hadamard matrix H, i.e. $H = S + I$, with $S = -S^o p$

```
sage: M=hadamard_matrix_paleyI(4, normalize=False); M
[ 1  1  1  1]
[-1  1  1 -1]
[-1 -1  1  1]
[-1  1 -1  1]
sage: S=M-identity_matrix(4); -S==S.T
True
```

TESTS:

```
sage: from sage.combinat.matrices.hadamard_matrix import is_hadamard_matrix
sage: test_cases = [x+1 for x in range(100) if is_prime_power(x) and x%4==3]
sage: all(is_hadamard_matrix(hadamard_matrix_paleyI(n),normalized=True,verbose=True)
....:     for n in test_cases)
True
sage: all(is_hadamard_matrix(hadamard_matrix_paleyI(n,normalize=False),verbose=True)
....:     for n in test_cases)
True
```

sage.combinat.matrices.hadamard_matrix.**hadamard_matrix_paleyII**(*n*)

Implements the Paley type II construction.

The Paley type II case corresponds to the case $p \cong 1 \mod 4$ for a prime p (see [Hora]).

EXAMPLES:

```
sage: sage.combinat.matrices.hadamard_matrix.hadamard_matrix_paleyII(12).det()
2985984
sage: 12^6
2985984
```

We note that the method returns a normalised Hadamard matrix

```
sage: sage.combinat.matrices.hadamard_matrix.hadamard_matrix_paleyII(12)
[ 1  1| 1  1| 1  1| 1  1| 1  1| 1  1]
[ 1 -1|-1  1|-1  1|-1  1|-1  1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1 -1| 1  1|-1 -1|-1 -1| 1  1]
[ 1  1|-1 -1| 1 -1|-1  1|-1  1| 1 -1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1  1| 1 -1| 1  1|-1 -1|-1 -1]
[ 1  1| 1 -1|-1 -1| 1 -1|-1  1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1|-1 -1| 1  1| 1 -1| 1  1|-1 -1]
[ 1  1|-1  1| 1 -1|-1 -1| 1 -1|-1  1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1|-1 -1|-1 -1| 1  1| 1 -1| 1  1]
[ 1  1|-1  1|-1  1| 1 -1|-1 -1| 1 -1]
[-----+-----+-----+-----+-----+-----]
[ 1 -1| 1  1|-1 -1|-1 -1| 1  1| 1 -1]
[ 1  1| 1 -1|-1  1|-1  1| 1 -1|-1 -1]
```

TESTS:

```
sage: from sage.combinat.matrices.hadamard_matrix import (hadamard_matrix_paleyII, is_hadamard_m
sage: test_cases = [2*(x+1) for x in range(50) if is_prime_power(x) and x%4==1]
sage: all(is_hadamard_matrix(hadamard_matrix_paleyII(n),normalized=True,verbose=True)
....:     for n in test_cases)
True
```

sage.combinat.matrices.hadamard_matrix.**hadamard_matrix_www**(*url_file*, *comments=False*)

Pulls file from Sloane's database and returns the corresponding Hadamard matrix as a Sage matrix.

You must input a filename of the form "had.n.xxx.txt" as described on the webpage http://neilsloane.com/hadamard/, where "xxx" could be empty or a number of some characters.

If comments=True then the "Automorphism..." line of the had.n.xxx.txt file is printed if it exists. Otherwise nothing is done.

EXAMPLES:

```
sage: hadamard_matrix_www("had.4.txt")       # optional - internet
[ 1  1  1  1]
[ 1 -1  1 -1]
[ 1  1 -1 -1]
[ 1 -1 -1  1]
sage: hadamard_matrix_www("had.16.2.txt",comments=True)    # optional - internet
Automorphism group has order = 49152 = 2^14 * 3
[ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1]
[ 1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1]
[ 1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1  1 -1 -1]
[ 1 -1 -1  1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1]
[ 1  1  1  1 -1 -1 -1 -1  1  1  1  1 -1 -1 -1 -1]
[ 1 -1  1 -1 -1  1 -1  1  1 -1  1 -1 -1  1 -1  1]
[ 1  1 -1 -1 -1 -1  1  1  1  1 -1 -1 -1 -1  1  1]
```

```
[ 1 -1 -1  1 -1  1  1 -1  1 -1 -1  1 -1  1  1 -1]
[ 1  1  1  1  1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1]
[ 1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1  1  1  1  1]
[ 1  1 -1 -1  1 -1  1 -1 -1 -1  1  1 -1  1 -1  1]
[ 1  1 -1 -1 -1  1 -1  1 -1 -1  1  1  1 -1  1 -1]
[ 1 -1  1 -1  1 -1 -1  1 -1  1 -1  1 -1  1  1 -1]
[ 1 -1  1 -1 -1  1  1 -1 -1  1 -1  1  1 -1 -1  1]
[ 1 -1 -1  1  1  1 -1 -1 -1  1  1 -1 -1 -1  1  1]
[ 1 -1 -1  1 -1 -1  1  1 -1  1  1 -1  1  1 -1 -1]
```

sage.combinat.matrices.hadamard_matrix.**is_hadamard_matrix**(*M*, *normalized=False*, *skew=False*, *verbose=False*)

Test if M is a hadamard matrix.

INPUT:

- `M` – a matrix
- `normalized` (boolean) – whether to test if `M` is a normalized Hadamard matrix, i.e. has its first row/column filled with +1.
- `skew` (boolean) – whether to test if `M` is a skew Hadamard matrix, i.e. $M = S + I$ for $-S = S^\top$, and I the identity matrix.
- `verbose` (boolean) – whether to be verbose when the matrix is not Hadamard.

EXAMPLE:

```
sage: from sage.combinat.matrices.hadamard_matrix import is_hadamard_matrix
sage: h = matrix.hadamard(12)
sage: is_hadamard_matrix(h)
True
sage: from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix
sage: h=skew_hadamard_matrix(12)
sage: is_hadamard_matrix(h, skew=True)
True
sage: h = matrix.hadamard(12)
sage: h[0,0] = 2
sage: is_hadamard_matrix(h,verbose=True)
The matrix does not only contain +1 and -1 entries, e.g. 2
False
sage: h = matrix.hadamard(12)
sage: for i in range(12):
....:     h[i,2] = -h[i,2]
sage: is_hadamard_matrix(h,verbose=True,normalized=True)
The matrix is not normalized
False
```

TESTS:

```
sage: h = matrix.hadamard(12)
sage: is_hadamard_matrix(h, skew=True)
False
sage: is_hadamard_matrix(h, skew=True, verbose=True)
The matrix is not skew
False
sage: h=skew_hadamard_matrix(12)
sage: is_hadamard_matrix(h, skew=True, verbose=True)
True
sage: is_hadamard_matrix(h, skew=False, verbose=True)
```

```
True
sage: h=-h
sage: is_hadamard_matrix(h, skew=True, verbose=True)
The matrix is not skew - diagonal entries must be all 1
False
sage: is_hadamard_matrix(h, skew=False, verbose=True)
True
```

sage.combinat.matrices.hadamard_matrix.**normalise_hadamard**(*H*)

Return the normalised Hadamard matrix corresponding to H.

The normalised Hadamard matrix corresponding to a Hadamard matrix H is a matrix whose every entry in the first row and column is +1.

EXAMPLES:

```
sage: H = sage.combinat.matrices.hadamard_matrix.normalise_hadamard(hadamard_matrix(4))
sage: H == hadamard_matrix(4)
True
```

sage.combinat.matrices.hadamard_matrix.**regular_symmetric_hadamard_matrix_with_constant_diag**

Return a Regular Symmetric Hadamard Matrix with Constant Diagonal.

A Hadamard matrix is said to be *regular* if its rows all sum to the same value.

For $\epsilon \in \{-1, +1\}$, we say that M is a $(n, \epsilon) - RSHCD$ if M is a regular symmetric Hadamard matrix with constant diagonal $\delta \in \{-1, +1\}$ and row sums all equal to $\delta\epsilon\sqrt{(n)}$. For more information, see [HX10] or 10.5.1 in [BH12]. For the case $n = 324$, see RSHCD_324() and [CP16].

INPUT:

- n (integer) – side of the matrix
- e – one of -1 or $+1$, equal to the value of ϵ

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import regular_symmetric_hadamard_matrix_with_
sage: regular_symmetric_hadamard_matrix_with_constant_diagonal(4,1)
[ 1  1  1 -1]
[ 1  1 -1  1]
[ 1 -1  1  1]
[-1  1  1  1]
sage: regular_symmetric_hadamard_matrix_with_constant_diagonal(4,-1)
[ 1 -1 -1 -1]
[-1  1 -1 -1]
[-1 -1  1 -1]
[-1 -1 -1  1]
```

Other hardcoded values:

```
sage: for n,e in [(36,1),(36,-1),(100,1),(100,-1),(196, 1)]:
....:     print regular_symmetric_hadamard_matrix_with_constant_diagonal(n,e)
36 x 36 dense matrix over Integer Ring
36 x 36 dense matrix over Integer Ring
100 x 100 dense matrix over Integer Ring
100 x 100 dense matrix over Integer Ring
196 x 196 dense matrix over Integer Ring
```

```
sage: for n,e in [(324,1),(324,-1)]: # not tested - long time, tested in RSHCD_324
....:     print regular_symmetric_hadamard_matrix_with_constant_diagonal(n,e) # not tested - lon
324 x 324 dense matrix over Integer Ring
324 x 324 dense matrix over Integer Ring
```

From two close prime powers:

```
sage: print regular_symmetric_hadamard_matrix_with_constant_diagonal(64,-1)
64 x 64 dense matrix over Integer Ring
```

Recursive construction:

```
sage: print regular_symmetric_hadamard_matrix_with_constant_diagonal(144,-1)
144 x 144 dense matrix over Integer Ring
```

REFERENCE:

sage.combinat.matrices.hadamard_matrix.**rshcd_from_close_prime_powers**(*n*)

Return a $(n^2, 1)$-RSHCD when $n-1$ and $n+1$ are odd prime powers and $n = 0 \pmod 4$.

The construction implemented here appears in Theorem 4.3 from [GS70].

Note that the authors of [SWW72] claim in Corollary 5.12 (page 342) to have proved the same result without the $n = 0 \pmod 4$ restriction with a *very* similar construction. So far, however, I (Nathann Cohen) have not been able to make it work.

INPUT:

- n – an integer congruent to $0 \pmod 4$

See also:

regular_symmetric_hadamard_matrix_with_constant_diagonal()

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import rshcd_from_close_prime_powers
sage: rshcd_from_close_prime_powers(4)
[-1 -1  1 -1  1 -1 -1  1 -1  1 -1 -1  1 -1  1 -1]
[-1 -1  1  1 -1 -1 -1 -1 -1  1  1 -1 -1  1 -1  1]
[ 1  1 -1  1  1 -1 -1 -1 -1 -1  1 -1 -1 -1  1 -1]
[-1  1  1 -1  1  1 -1 -1 -1 -1 -1  1 -1 -1 -1  1]
[ 1 -1  1  1 -1  1  1 -1 -1 -1 -1 -1  1 -1 -1 -1]
[-1 -1 -1  1  1 -1  1  1 -1 -1 -1  1 -1  1 -1 -1]
[-1 -1 -1 -1  1  1 -1 -1  1 -1  1 -1  1  1 -1 -1]
[ 1 -1 -1 -1 -1  1 -1 -1 -1  1 -1  1 -1  1  1 -1]
[-1 -1 -1 -1 -1 -1  1 -1 -1 -1  1  1  1 -1  1  1]
[ 1  1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1  1  1 -1  1]
[-1  1  1 -1 -1 -1  1 -1  1 -1 -1 -1 -1  1  1 -1]
[-1 -1 -1  1 -1  1 -1  1  1 -1 -1 -1 -1 -1  1  1]
[ 1 -1 -1 -1  1 -1  1 -1  1  1 -1 -1 -1 -1 -1  1]
[-1  1 -1 -1 -1  1  1  1 -1  1  1 -1 -1 -1 -1 -1]
[ 1 -1  1 -1 -1 -1 -1  1  1 -1  1  1 -1 -1 -1 -1]
[-1  1 -1  1 -1 -1 -1 -1  1  1 -1  1  1 -1 -1 -1]
```

REFERENCE:

sage.combinat.matrices.hadamard_matrix.**skew_hadamard_matrix**(*n*, *existence=False*, *skew_normalize=True*, *check=True*)

Tries to construct a skew Hadamard matrix

A Hadamard matrix H is called skew if $H = S - I$, for I the identity matrix and $-S = S^\top$. Currently constructions from Section 14.1 of [Ha83] and few more exotic ones are implemented.

INPUT:

- `n` (integer) – dimension of the matrix
- `existence` (boolean) – whether to build the matrix or merely query if a construction is available in Sage. When set to `True`, the function returns:
  - `True` – meaning that Sage knows how to build the matrix
  - `Unknown` – meaning that Sage does not know how to build the matrix, but that the design may exist (see `sage.misc.unknown`).
  - `False` – meaning that the matrix does not exist.
- `skew_normalize` (boolean) – whether to make the 1st row all-one, and adjust the 1st column accordingly. Set to `True` by default.
- `check` (boolean) – whether to check that output is correct before returning it. As this is expected to be useless (but we are cautious guys), you may want to disable it whenever you want speed. Set to `True` by default.

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import skew_hadamard_matrix
sage: skew_hadamard_matrix(12).det()
2985984
sage: 12^6
2985984
sage: skew_hadamard_matrix(1)
[1]
sage: skew_hadamard_matrix(2)
[ 1  1]
[-1  1]
```

TESTS:

```
sage: skew_hadamard_matrix(10,existence=True)
False
sage: skew_hadamard_matrix(12,existence=True)
True
sage: skew_hadamard_matrix(784,existence=True)
True
sage: skew_hadamard_matrix(10)
Traceback (most recent call last):
...
ValueError: A skew Hadamard matrix of order 10 does not exist
sage: skew_hadamard_matrix(36)
36 x 36 dense matrix over Integer Ring...
sage: skew_hadamard_matrix(36)==skew_hadamard_matrix(36,skew_normalize=False)
False
sage: skew_hadamard_matrix(52)
52 x 52 dense matrix over Integer Ring...
sage: skew_hadamard_matrix(92)
92 x 92 dense matrix over Integer Ring...
sage: skew_hadamard_matrix(816)      # long time
816 x 816 dense matrix over Integer Ring...
sage: skew_hadamard_matrix(100)
Traceback (most recent call last):
...
```

```
ValueError: A skew Hadamard matrix of order 100 is not yet implemented.
sage: skew_hadamard_matrix(100,existence=True)
Unknown
```

REFERENCES:

sage.combinat.matrices.hadamard_matrix.**williamson_goethals_seidel_skew_hadamard_matrix**(*a*, *b*, *c*, *d*, *check*

Williamson-Goethals-Seidel construction of a skew Hadamard matrix

Given $n \times n$ (anti)circulant matrices A, B, C, D with 1,-1 entries, and satisfying $A + A^\top = 2I$, $AA^\top + BB^\top + CC^\top + DD^\top = 4nI$, one can construct a skew Hadamard matrix of order $4n$, cf. [GS70s].

INPUT:

- a – 1,-1 list specifying the 1st row of A
- b – 1,-1 list specifying the 1st row of B
- d – 1,-1 list specifying the 1st row of C
- c – 1,-1 list specifying the 1st row of D

EXAMPLES:

```
sage: from sage.combinat.matrices.hadamard_matrix import williamson_goethals_seidel_skew_hadamar
sage: a=[ 1,  1, 1, -1,  1, -1,  1, -1, -1]
sage: b=[ 1, -1, 1,  1, -1, -1,  1,  1, -1]
sage: c=[-1, -1]+[1]*6+[-1]
sage: d=[ 1,  1, 1, -1,  1,  1, -1,  1,  1]
sage: M=WGS(a,b,c,d,check=True)
```

REFERENCES:

5.1.122 Latin Squares

A *latin square* of order n is an $n \times n$ array such that each symbol $s \in \{0, 1, \ldots, n-1\}$ appears precisely once in each row, and precisely once in each column. A *partial latin square* of order n is an $n \times n$ array such that each symbol $s \in \{0, 1, \ldots, n-1\}$ appears at most once in each row, and at most once in each column. Empty cells are denoted by -1. A latin square L is a *completion* of a partial latin square P if $P \subseteq L$. If P completes to just L then P *has unique completion*.

A *latin bitrade* (T_1, T_2) is a pair of partial latin squares such that:

1. $\{(i, j) \mid (i, j, k) \in T_1$ for some symbol $k\} = \{(i, j) \mid (i, j, k') \in T_2$ for some symbol $k'\}$;
2. for each $(i, j, k) \in T_1$ and $(i, j, k') \in T_2$, $k \neq k'$;
3. the symbols appearing in row i of T_1 are the same as those of row i of T_2; the symbols appearing in column j of T_1 are the same as those of column j of T_2.

Intuitively speaking, a bitrade gives the difference between two latin squares, so if (T_1, T_2) is a bitrade for the pair of latin squares (L_1, L_2), then $L1 = (L2 \setminus T_1) \cup T_2$ and $L2 = (L1 \setminus T_2) \cup T_1$.

This file contains

1. LatinSquare class definition;
2. some named latin squares (back circulant, forward circulant, abelian 2-group);

3. functions is_partial_latin_square and is_latin_square to test if a LatinSquare object satisfies the definition of a latin square or partial latin square, respectively;
4. tests for completion and unique completion (these use the C++ implementation of Knuth's dancing links algorithm to solve the problem as a instance of $0-1$ matrix exact cover);
5. functions for calculating the τ_i representation of a bitrade and the genus of the associated hypermap embedding;
6. Markov chain of Jacobson and Matthews (1996) for generating latin squares uniformly at random (provides a generator interface);
7. a few examples of τ_i representations of bitrades constructed from the action of a group on itself by right multiplication, functions for converting to a pair of LatinSquare objects.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(5)
sage: B
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
[4 0 1 2 3]
sage: B.is_latin_square()
True
sage: B[0, 1] = 0
sage: B.is_latin_square()
False

sage: (a, b, c, G) = alternating_group_bitrade_generators(1)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1
[ 0 -1  3  1]
[-1  1  0  2]
[ 1  3  2 -1]
[ 2  0 -1  3]
sage: T2
[ 1 -1  0  3]
[-1  0  2  1]
[ 2  1  3 -1]
[ 0  3 -1  2]
sage: T1.nr_filled_cells()
12
sage: genus(T1, T2)
1
```

To do:

1. Latin squares with symbols from a ring instead of the integers $\{0, 1, \ldots, n-1\}$.
2. Isotopism testing of latin squares and bitrades via graph isomorphism (nauty?).
3. Combinatorial constructions for bitrades.

AUTHORS:

- Carlo Hamalainen (2008-03-23): initial version

TESTS:

```
sage: L = elementary_abelian_2group(3)
sage: L == loads(dumps(L))
True
```

class sage.combinat.matrices.latin.**LatinSquare**(**args*)

Latin squares.

This class implements a latin square of order n with rows and columns indexed by the set 0, 1, ..., n-1 and symbols from the same set. The underlying latin square is a matrix(ZZ, n, n). If L is a latin square, then the cell at row r, column c is empty if and only if L[r, c] < 0. In this way we allow partial latin squares and can speak of completions to latin squares, etc.

There are two ways to declare a latin square:

Empty latin square of order n:

```
sage: n = 3
sage: L = LatinSquare(n)
sage: L
[-1 -1 -1]
[-1 -1 -1]
[-1 -1 -1]
```

Latin square from a matrix:

```
sage: M = matrix(ZZ, [[0, 1], [2, 3]])
sage: LatinSquare(M)
[0 1]
[2 3]
```

actual_row_col_sym_sizes()

Bitrades sometimes end up in partial latin squares with unused rows, columns, or symbols. This function works out the actual number of used rows, columns, and symbols.

Warning: We assume that the unused rows/columns occur in the lower right of self, and that the used symbols are in the range {0, 1, ..., m} (no holes in that list).

EXAMPLE:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(3)
sage: B[0,2] = B[1,2] = B[2,2] = -1
sage: B[0,0] = B[2,1] = -1
sage: B
[-1  1 -1]
[ 1  2 -1]
[ 2 -1 -1]
sage: B.actual_row_col_sym_sizes()
(3, 2, 2)
```

apply_isotopism(*row_perm*, *col_perm*, *sym_perm*)

An isotopism is a permutation of the rows, columns, and symbols of a partial latin square self. Use isotopism() to convert a tuple (indexed from 0) to a Permutation object.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(5)
sage: B
[0 1 2 3 4]
```

```
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
[4 0 1 2 3]
sage: alpha = isotopism((0,1,2,3,4))
sage: beta  = isotopism((1,0,2,3,4))
sage: gamma = isotopism((2,1,0,3,4))
sage: B.apply_isotopism(alpha, beta, gamma)
[3 4 2 0 1]
[0 2 3 1 4]
[1 3 0 4 2]
[4 0 1 2 3]
[2 1 4 3 0]
```

clear_cells()

Mark every cell in self as being empty.

EXAMPLES:

```
sage: A = LatinSquare(matrix(ZZ, [[0, 1], [2, 3]]))
sage: A.clear_cells()
sage: A
[-1 -1]
[-1 -1]
```

column(*x*)

Returns column x of the latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(3).column(0)
(0, 1, 2)
```

contained_in(*Q*)

Returns True if self is a subset of Q?

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: P = elementary_abelian_2group(2)
sage: P[0, 0] = -1
sage: P.contained_in(elementary_abelian_2group(2))
True
sage: back_circulant(4).contained_in(elementary_abelian_2group(2))
False
```

disjoint_mate_dlxcpp_rows_and_map(*allow_subtrade*)

Internal function for find_disjoint_mates.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(4)
sage: B.disjoint_mate_dlxcpp_rows_and_map(allow_subtrade = True)
([[0, 16, 32],
  [1, 17, 32],
  [2, 18, 32],
  [3, 19, 32],
  [4, 16, 33],
  [5, 17, 33],
```

```
[6, 18, 33],
[7, 19, 33],
[8, 16, 34],
[9, 17, 34],
[10, 18, 34],
[11, 19, 34],
[12, 16, 35],
[13, 17, 35],
[14, 18, 35],
[15, 19, 35],
[0, 20, 36],
[1, 21, 36],
[2, 22, 36],
[3, 23, 36],
[4, 20, 37],
[5, 21, 37],
[6, 22, 37],
[7, 23, 37],
[8, 20, 38],
[9, 21, 38],
[10, 22, 38],
[11, 23, 38],
[12, 20, 39],
[13, 21, 39],
[14, 22, 39],
[15, 23, 39],
[0, 24, 40],
[1, 25, 40],
[2, 26, 40],
[3, 27, 40],
[4, 24, 41],
[5, 25, 41],
[6, 26, 41],
[7, 27, 41],
[8, 24, 42],
[9, 25, 42],
[10, 26, 42],
[11, 27, 42],
[12, 24, 43],
[13, 25, 43],
[14, 26, 43],
[15, 27, 43],
[0, 28, 44],
[1, 29, 44],
[2, 30, 44],
[3, 31, 44],
[4, 28, 45],
[5, 29, 45],
[6, 30, 45],
[7, 31, 45],
[8, 28, 46],
[9, 29, 46],
[10, 30, 46],
[11, 31, 46],
[12, 28, 47],
[13, 29, 47],
[14, 30, 47],
[15, 31, 47]],
```

```
{(0, 16, 32): (0, 0, 0),
 (0, 20, 36): (1, 0, 0),
 (0, 24, 40): (2, 0, 0),
 (0, 28, 44): (3, 0, 0),
 (1, 17, 32): (0, 0, 1),
 (1, 21, 36): (1, 0, 1),
 (1, 25, 40): (2, 0, 1),
 (1, 29, 44): (3, 0, 1),
 (2, 18, 32): (0, 0, 2),
 (2, 22, 36): (1, 0, 2),
 (2, 26, 40): (2, 0, 2),
 (2, 30, 44): (3, 0, 2),
 (3, 19, 32): (0, 0, 3),
 (3, 23, 36): (1, 0, 3),
 (3, 27, 40): (2, 0, 3),
 (3, 31, 44): (3, 0, 3),
 (4, 16, 33): (0, 1, 0),
 (4, 20, 37): (1, 1, 0),
 (4, 24, 41): (2, 1, 0),
 (4, 28, 45): (3, 1, 0),
 (5, 17, 33): (0, 1, 1),
 (5, 21, 37): (1, 1, 1),
 (5, 25, 41): (2, 1, 1),
 (5, 29, 45): (3, 1, 1),
 (6, 18, 33): (0, 1, 2),
 (6, 22, 37): (1, 1, 2),
 (6, 26, 41): (2, 1, 2),
 (6, 30, 45): (3, 1, 2),
 (7, 19, 33): (0, 1, 3),
 (7, 23, 37): (1, 1, 3),
 (7, 27, 41): (2, 1, 3),
 (7, 31, 45): (3, 1, 3),
 (8, 16, 34): (0, 2, 0),
 (8, 20, 38): (1, 2, 0),
 (8, 24, 42): (2, 2, 0),
 (8, 28, 46): (3, 2, 0),
 (9, 17, 34): (0, 2, 1),
 (9, 21, 38): (1, 2, 1),
 (9, 25, 42): (2, 2, 1),
 (9, 29, 46): (3, 2, 1),
 (10, 18, 34): (0, 2, 2),
 (10, 22, 38): (1, 2, 2),
 (10, 26, 42): (2, 2, 2),
 (10, 30, 46): (3, 2, 2),
 (11, 19, 34): (0, 2, 3),
 (11, 23, 38): (1, 2, 3),
 (11, 27, 42): (2, 2, 3),
 (11, 31, 46): (3, 2, 3),
 (12, 16, 35): (0, 3, 0),
 (12, 20, 39): (1, 3, 0),
 (12, 24, 43): (2, 3, 0),
 (12, 28, 47): (3, 3, 0),
 (13, 17, 35): (0, 3, 1),
 (13, 21, 39): (1, 3, 1),
 (13, 25, 43): (2, 3, 1),
 (13, 29, 47): (3, 3, 1),
 (14, 18, 35): (0, 3, 2),
 (14, 22, 39): (1, 3, 2),
```

```
    (14, 26, 43): (2, 3, 2),
    (14, 30, 47): (3, 3, 2),
    (15, 19, 35): (0, 3, 3),
    (15, 23, 39): (1, 3, 3),
    (15, 27, 43): (2, 3, 3),
    (15, 31, 47): (3, 3, 3)})
```

dlxcpp_has_unique_completion()

Check if the partial latin square self of order n can be embedded in precisely one latin square of order n.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(2).dlxcpp_has_unique_completion()
True
sage: P = LatinSquare(2)
sage: P.dlxcpp_has_unique_completion()
False
sage: P[0, 0] = 0
sage: P.dlxcpp_has_unique_completion()
True
```

dumps()

Since the latin square class doesn't hold any other private variables we just call dumps on self.square:

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(2) == loads(dumps(back_circulant(2)))
True
```

filled_cells_map()

Number the filled cells of self with integers from {1, 2, 3, ...}

INPUT:

- `self` - Partial latin square self (empty cells have negative values)

OUTPUT: A dictionary cells_map where cells_map[(i,j)] = m means that (i,j) is the m-th filled cell in P, while cells_map[m] = (i,j).

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: (a, b, c, G) = alternating_group_bitrade_generators(1)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1.filled_cells_map()
{1: (0, 0),
 2: (0, 2),
 3: (0, 3),
 4: (1, 1),
 5: (1, 2),
 6: (1, 3),
 7: (2, 0),
 8: (2, 1),
 9: (2, 2),
 10: (3, 0),
 11: (3, 1),
 12: (3, 3),
 (0, 0): 1,
 (0, 2): 2,
```

```
 (0, 3): 3,
 (1, 1): 4,
 (1, 2): 5,
 (1, 3): 6,
 (2, 0): 7,
 (2, 1): 8,
 (2, 2): 9,
 (3, 0): 10,
 (3, 1): 11,
 (3, 3): 12}
```

find_disjoint_mates(*nr_to_find=None*, *allow_subtrade=False*)

> **Warning:** If allow_subtrade is `True` then we may return a partial latin square that is *not* disjoint to `self`. In that case, use bitrade(P, Q) to get an actual bitrade.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(4)
sage: g = B.find_disjoint_mates(allow_subtrade = True)
sage: B1 = next(g)
sage: B0, B1 = bitrade(B, B1)
sage: assert is_bitrade(B0, B1)
sage: print B0, "\n,\n", B1
[-1  1  2 -1]
[-1  2 -1  0]
[-1 -1 -1 -1]
[-1  0  1  2]
,
[-1  2  1 -1]
[-1  0 -1  2]
[-1 -1 -1 -1]
[-1  1  2  0]
```

gcs()

A greedy critical set of a latin square self is found by successively removing elements in a row-wise (bottom-up) manner, checking for unique completion at each step.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: A = elementary_abelian_2group(3)
sage: G = A.gcs()
sage: A
[0 1 2 3 4 5 6 7]
[1 0 3 2 5 4 7 6]
[2 3 0 1 6 7 4 5]
[3 2 1 0 7 6 5 4]
[4 5 6 7 0 1 2 3]
[5 4 7 6 1 0 3 2]
[6 7 4 5 2 3 0 1]
[7 6 5 4 3 2 1 0]
sage: G
[ 0  1  2  3  4  5  6 -1]
[ 1  0  3  2  5  4 -1 -1]
[ 2  3  0  1  6 -1  4 -1]
[ 3  2  1  0 -1 -1 -1 -1]
[ 4  5  6 -1  0  1  2 -1]
```

```
[ 5  4 -1 -1  1  0 -1 -1]
[ 6 -1  4 -1  2 -1  0 -1]
[-1 -1 -1 -1 -1 -1 -1 -1]
```

is_completable()

Returns True if the partial latin square can be completed to a latin square.

EXAMPLES:

The following partial latin square has no completion because there is nowhere that we can place the symbol 0 in the third row:

```
sage: B = LatinSquare(3)

sage: B[0, 0] = 0
sage: B[1, 1] = 0
sage: B[2, 2] = 1

sage: B
[ 0 -1 -1]
[-1  0 -1]
[-1 -1  1]

sage: B.is_completable()
False

sage: B[2, 2] = 0
sage: B.is_completable()
True
```

is_empty_column(c)

Checks if column c of the partial latin square self is empty.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = back_circulant(4)
sage: L.is_empty_column(0)
False
sage: L[0,0] = L[1,0] = L[2,0] = L[3,0] = -1
sage: L.is_empty_column(0)
True
```

is_empty_row(r)

Checks if row r of the partial latin square self is empty.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = back_circulant(4)
sage: L.is_empty_row(0)
False
sage: L[0,0] = L[0,1] = L[0,2] = L[0,3] = -1
sage: L.is_empty_row(0)
True
```

is_latin_square()

self is a latin square if it is an n by n matrix, and each symbol in [0, 1, ..., n-1] appears exactly once in each row, and exactly once in each column.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: elementary_abelian_2group(4).is_latin_square()
True

sage: forward_circulant(7).is_latin_square()
True
```

is_partial_latin_square()

self is a partial latin square if it is an n by n matrix, and each symbol in [0, 1, ..., n-1] appears at most once in each row, and at most once in each column.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: LatinSquare(4).is_partial_latin_square()
True
sage: back_circulant(3).gcs().is_partial_latin_square()
True
sage: back_circulant(6).is_partial_latin_square()
True
```

is_uniquely_completable()

Returns True if the partial latin square self has exactly one completion to a latin square. This is just a wrapper for the current best-known algorithm, Dancing Links by Knuth. See dancing_links.spyx

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(4).gcs().is_uniquely_completable()
True

sage: G = elementary_abelian_2group(3).gcs()
sage: G.is_uniquely_completable()
True

sage: G[0, 0] = -1
sage: G.is_uniquely_completable()
False
```

latex()

Returns LaTeX code for the latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: print back_circulant(3).latex()
\begin{array}{|c|c|c|}\hline 0 & 1 & 2\\\hline 1 & 2 & 0\\\hline 2 & 0 & 1\\\hline\end{array
```

list()

Convert the latin square into a list, in a row-wise manner.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(3).list()
[0, 1, 2, 1, 2, 0, 2, 0, 1]
```

ncols()

Number of columns in the latin square.

EXAMPLES:

```
sage: LatinSquare(3).ncols()
3
```

nr_distinct_symbols()

Returns the number of distinct symbols in the partial latin square self.

EXAMPLE:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(5).nr_distinct_symbols()
5
sage: L = LatinSquare(10)
sage: L.nr_distinct_symbols()
0
sage: L[0, 0] = 0
sage: L[0, 1] = 1
sage: L.nr_distinct_symbols()
2
```

nr_filled_cells()

Returns the number of filled cells (i.e. cells with a positive value) in the partial latin square self.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: LatinSquare(matrix([[0, -1], [-1, 0]])).nr_filled_cells()
2
```

nrows()

Number of rows in the latin square.

EXAMPLES:

```
sage: LatinSquare(3).nrows()
3
```

permissable_values(*r*, *c*)

Find all values that do not appear in row r and column c of the latin square self. If self[r, c] is filled then we return the empty list.

INPUT:

- `self` - LatinSquare
- `r` - int; row of the latin square
- `c` - int; column of the latin square

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = back_circulant(5)
sage: L[0, 0] = -1
sage: L.permissable_values(0, 0)
[0]
```

random_empty_cell()

Find an empty cell of self, uniformly at random.

INPUT:

- `self` - LatinSquare

OUTPUT:

- `[r, c]` - cell such that self[r, c] is empty, or returns None if self is a (full) latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: P = back_circulant(2)
sage: P[1,1] = -1
sage: P.random_empty_cell()
[1, 1]
```

row(*x*)

Returns row x of the latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(3).row(0)
(0, 1, 2)
```

set_immutable()

A latin square is immutable if the underlying matrix is immutable.

EXAMPLES:

```
sage: L = LatinSquare(matrix(ZZ, [[0, 1], [2, 3]]))
sage: L.set_immutable()
sage: {L : 0}   # this would fail without set_immutable()
{[0 1]
[2 3]: 0}
```

top_left_empty_cell()

Returns the least [r, c] such that self[r, c] is an empty cell. If all cells are filled then we return None.

INPUT:

- `self` - LatinSquare

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(5)
sage: B[3, 4] = -1
sage: B.top_left_empty_cell()
[3, 4]
```

vals_in_col(*c*)

Returns a dictionary with key e if and only if column c of self has the symbol e.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(3)
sage: B[0, 0] = -1
sage: back_circulant(3).vals_in_col(0)
{0: True, 1: True, 2: True}
```

vals_in_row(*r*)

Returns a dictionary with key e if and only if row r of self has the symbol e.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(3)
sage: B[0, 0] = -1
sage: back_circulant(3).vals_in_row(0)
{0: True, 1: True, 2: True}
```

sage.combinat.matrices.latin.**LatinSquare_generator**(*L_start*, *check_assertions=False*)

Generator for a sequence of uniformly distributed latin squares, given L_start as the initial latin square. This code implements the Markov chain algorithm of Jacobson and Matthews (1996), see below for the BibTex entry. This generator will never throw the StopIteration exception, so it provides an infinite sequence of latin squares.

EXAMPLES:

Use the back circulant latin square of order 4 as the initial square and print the next two latin squares given by the Markov chain:

```
sage: from sage.combinat.matrices.latin import *
sage: g = LatinSquare_generator(back_circulant(4))
sage: next(g).is_latin_square()
True
```

REFERENCES:

sage.combinat.matrices.latin.**alternating_group_bitrade_generators**(*m*)

Construct generators a, b, c for the alternating group on 3m+1 points, such that a*b*c = 1.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: a, b, c, G = alternating_group_bitrade_generators(1)
sage: (a, b, c, G)
((1,2,3), (1,4,2), (2,4,3), Permutation Group with generators [(1,2,3), (1,4,2)])
sage: a*b*c
()

sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1
[ 0 -1  3  1]
[-1  1  0  2]
[ 1  3  2 -1]
[ 2  0 -1  3]
sage: T2
[ 1 -1  0  3]
[-1  0  2  1]
[ 2  1  3 -1]
[ 0  3 -1  2]
```

sage.combinat.matrices.latin.**back_circulant**(*n*)

The back-circulant latin square of order n is the Cayley table for (Z_n, +), the integers under addition modulo n.

INPUT:

- n - int; order of the latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(5)
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
```

```
[3 4 0 1 2]
[4 0 1 2 3]
```

sage.combinat.matrices.latin.**beta1**(*rce*, *T1*, *T2*)

Find the unique (x, c, e) in T2 such that (r, c, e) is in T1.

INPUT:

- `rce` - tuple (or list) (r, c, e) in T1
- `T1, T2` - latin bitrade

OUTPUT: (x, c, e) in T2.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: beta1([0, 0, 0], T1, T2)
(1, 0, 0)
```

sage.combinat.matrices.latin.**beta2**(*rce*, *T1*, *T2*)

Find the unique (r, x, e) in T2 such that (r, c, e) is in T1.

INPUT:

- `rce` - tuple (or list) (r, c, e) in T1
- `T1, T2` - latin bitrade

OUTPUT:

- (r, x, e) in T2.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: beta2([0, 0, 0], T1, T2)
(0, 1, 0)
```

sage.combinat.matrices.latin.**beta3**(*rce*, *T1*, *T2*)

Find the unique (r, c, x) in T2 such that (r, c, e) is in T1.

INPUT:

- `rce` - tuple (or list) (r, c, e) in T1
- `T1, T2` - latin bitrade

OUTPUT:

•(r, c, x) in T2.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: beta3([0, 0, 0], T1, T2)
(0, 0, 4)
```

sage.combinat.matrices.latin.**bitrade**(*T1*, *T2*)

Form the bitrade (Q1, Q2) from (T1, T2) by setting empty the cells (r, c) such that T1[r, c] == T2[r, c].

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B1 = back_circulant(5)
sage: alpha = isotopism((0,1,2,3,4))
sage: beta  = isotopism((1,0,2,3,4))
sage: gamma = isotopism((2,1,0,3,4))
sage: B2 = B1.apply_isotopism(alpha, beta, gamma)
sage: T1, T2 = bitrade(B1, B2)
sage: T1
[ 0  1 -1  3  4]
[ 1 -1 -1  4  0]
[ 2 -1  4  0  1]
[ 3  4  0  1  2]
[ 4  0  1  2  3]
sage: T2
[ 3  4 -1  0  1]
[ 0 -1 -1  1  4]
[ 1 -1  0  4  2]
[ 4  0  1  2  3]
[ 2  1  4  3  0]
```

sage.combinat.matrices.latin.**bitrade_from_group**(*a*, *b*, *c*, *G*)

Given group elements a, b, c in G such that abc = 1 and the subgroups a, b, c intersect (pairwise) only in the identity, construct a bitrade (T1, T2) where rows, columns, and symbols correspond to cosets of a, b, and c, respectively.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: a, b, c, G = alternating_group_bitrade_generators(1)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1
[ 0 -1  3  1]
[-1  1  0  2]
[ 1  3  2 -1]
[ 2  0 -1  3]
sage: T2
[ 1 -1  0  3]
[-1  0  2  1]
[ 2  1  3 -1]
[ 0  3 -1  2]
```

sage.combinat.matrices.latin.**cells_map_as_square**(*cells_map*, *n*)
Returns a LatinSquare with cells numbered from 1, 2, ... to given the dictionary cells_map.

Note: The value n should be the maximum of the number of rows and columns of the original partial latin square

---

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: (a, b, c, G) = alternating_group_bitrade_generators(1)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1
[ 0 -1  3  1]
[-1  1  0  2]
[ 1  3  2 -1]
[ 2  0 -1  3]
```

There are 12 filled cells in T:

```
sage: cells_map_as_square(T1.filled_cells_map(), max(T1.nrows(), T1.ncols()))
[ 1 -1  2  3]
[-1  4  5  6]
[ 7  8  9 -1]
[10 11 -1 12]
```

sage.combinat.matrices.latin.**check_bitrade_generators**(*a*, *b*, *c*)
Three group elements a, b, c will generate a bitrade if a*b*c = 1 and the subgroups a, b, c intersect (pairwise) in just the identity.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: a, b, c, G = p3_group_bitrade_generators(3)
sage: check_bitrade_generators(a, b, c)
True
sage: check_bitrade_generators(a, b, gap('()'))
False
```

sage.combinat.matrices.latin.**coin**()
Simulates a fair coin (returns True or False) using ZZ.random_element(2).

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: x = coin()
sage: x == 0 or x == 1
True
```

sage.combinat.matrices.latin.**column_containing_sym**(*L*, *r*, *x*)
Given an improper latin square L with L[r, c1] = L[r, c2] = x, return c1 or c2 with equal probability. This is an internal function and should only be used in LatinSquare_generator().

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = matrix([(1, 0, 2, 3), (0, 2, 3, 0), (2, 3, 0, 1), (3, 0, 1, 2)])
sage: L
[1 0 2 3]
[0 2 3 0]
[2 3 0 1]
[3 0 1 2]
```

```
sage: c = column_containing_sym(L, 1, 0)
sage: c == 0 or c == 3
True
```

sage.combinat.matrices.latin.**direct_product**(*L1*, *L2*, *L3*, *L4*)

The 'direct product' of four latin squares L1, L2, L3, L4 of order n is the latin square of order 2n consisting of

```
-----------
| L1 | L2 |
-----------
| L3 | L4 |
-----------
```

where the subsquares L2 and L3 have entries offset by n.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: direct_product(back_circulant(4), back_circulant(4), elementary_abelian_2group(2), element
[0 1 2 3 4 5 6 7]
[1 2 3 0 5 6 7 4]
[2 3 0 1 6 7 4 5]
[3 0 1 2 7 4 5 6]
[4 5 6 7 0 1 2 3]
[5 4 7 6 1 0 3 2]
[6 7 4 5 2 3 0 1]
[7 6 5 4 3 2 1 0]
```

sage.combinat.matrices.latin.**dlxcpp_find_completions**(*P*, *nr_to_find=None*)

Returns a list of all latin squares L of the same order as P such that P is contained in L. The optional parameter nr_to_find limits the number of latin squares that are found.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: dlxcpp_find_completions(LatinSquare(2))
[[0 1]
[1 0], [1 0]
[0 1]]

sage: dlxcpp_find_completions(LatinSquare(2), 1)
[[0 1]
[1 0]]
```

sage.combinat.matrices.latin.**dlxcpp_rows_and_map**(*P*)

Internal function for dlxcpp_find_completions. Given a partial latin square P we construct a list of rows of a 0-1 matrix M such that an exact cover of M corresponds to a completion of P to a latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: dlxcpp_rows_and_map(LatinSquare(2))
([[0, 4, 8],
  [1, 5, 8],
  [2, 4, 9],
  [3, 5, 9],
  [0, 6, 10],
  [1, 7, 10],
  [2, 6, 11],
  [3, 7, 11]],
 {(0, 4, 8): (0, 0, 0),
```

```
(0, 6, 10): (1, 0, 0),
(1, 5, 8): (0, 0, 1),
(1, 7, 10): (1, 0, 1),
(2, 4, 9): (0, 1, 0),
(2, 6, 11): (1, 1, 0),
(3, 5, 9): (0, 1, 1),
(3, 7, 11): (1, 1, 1)})
```

sage.combinat.matrices.latin.**elementary_abelian_2group**(*s*)

Returns the latin square based on the Cayley table for the elementary abelian 2-group of order 2s.

INPUT:

- s - int; order of the latin square will be 2s.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: elementary_abelian_2group(3)
[0 1 2 3 4 5 6 7]
[1 0 3 2 5 4 7 6]
[2 3 0 1 6 7 4 5]
[3 2 1 0 7 6 5 4]
[4 5 6 7 0 1 2 3]
[5 4 7 6 1 0 3 2]
[6 7 4 5 2 3 0 1]
[7 6 5 4 3 2 1 0]
```

sage.combinat.matrices.latin.**forward_circulant**(*n*)

The forward-circulant latin square of order n is the Cayley table for the operation r + c = (n-c+r) mod n.

INPUT:

- n - int; order of the latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: forward_circulant(5)
[0 4 3 2 1]
[1 0 4 3 2]
[2 1 0 4 3]
[3 2 1 0 4]
[4 3 2 1 0]
```

sage.combinat.matrices.latin.**genus**(*T1*, *T2*)

Returns the genus of hypermap embedding associated with the bitrade (T1, T2). Informally, we compute the [tau_1, tau_2, tau_3] permutation representation of the bitrade. Each cycle of tau_1, tau_2, and tau_3 gives a rotation scheme for a black, white, and star vertex (respectively). The genus then comes from Euler's formula. For more details see Carlo Hamalainen: Partitioning 3-homogeneous latin bitrades. To appear in Geometriae Dedicata, available at http://arxiv.org/abs/0710.0938

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: (a, b, c, G) = alternating_group_bitrade_generators(1)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: genus(T1, T2)
1
sage: (a, b, c, G) = pq_group_bitrade_generators(3, 7)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
```

```
sage: genus(T1, T2)
3
```

sage.combinat.matrices.latin.**group_to_LatinSquare**(*G*)

Construct a latin square on the symbols [0, 1, ..., n-1] for a group with an n by n Cayley table.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: group_to_LatinSquare(DihedralGroup(2))
[0 1 2 3]
[1 0 3 2]
[2 3 0 1]
[3 2 1 0]

sage: G = gap.Group(PermutationGroupElement((1,2,3)))
sage: group_to_LatinSquare(G)
[0 1 2]
[1 2 0]
[2 0 1]
```

sage.combinat.matrices.latin.**is_bitrade**(*T1*, *T2*)

Combinatorially, a pair (T1, T2) of partial latin squares is a bitrade if they are disjoint, have the same shape, and have row and column balance. For definitions of each of these terms see the relevant function in this file.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
```

sage.combinat.matrices.latin.**is_disjoint**(*T1*, *T2*)

The partial latin squares T1 and T2 are disjoint if T1[r, c] != T2[r, c] or T1[r, c] == T2[r, c] == -1 for each cell [r, c].

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import is_disjoint, back_circulant, isotopism
sage: is_disjoint(back_circulant(2), back_circulant(2))
False

sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_disjoint(T1, T2)
True
```

sage.combinat.matrices.latin.**is_primary_bitrade**(*a*, *b*, *c*, *G*)

A bitrade generated from elements a, b, c is primary if a, b, c = G.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: (a, b, c, G) = p3_group_bitrade_generators(5)
sage: is_primary_bitrade(a, b, c, G)
True
```

sage.combinat.matrices.latin.**is_row_and_col_balanced**(*T1*, *T2*)

Partial latin squares T1 and T2 are balanced if the symbols appearing in row r of T1 are the same as the symbols appearing in row r of T2, for each r, and if the same condition holds on columns.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = matrix([[0,1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1]])
sage: T2 = matrix([[0,1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1]])
sage: is_row_and_col_balanced(T1, T2)
True
sage: T2 = matrix([[0,3,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1]])
sage: is_row_and_col_balanced(T1, T2)
False
```

sage.combinat.matrices.latin.**is_same_shape**(*T1*, *T2*)

Two partial latin squares T1, T2 have the same shape if T1[r, c] = 0 if and only if T2[r, c] = 0.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: is_same_shape(elementary_abelian_2group(2), back_circulant(4))
True
sage: is_same_shape(LatinSquare(5), LatinSquare(5))
True
sage: is_same_shape(forward_circulant(5), LatinSquare(5))
False
```

sage.combinat.matrices.latin.**isotopism**(*p*)

Return a Permutation object that represents an isotopism (for rows, columns or symbols of a partial latin square).

Technically, all this function does is take as input a representation of a permutation of $0, ..., n-1$ and return a `Permutation` object defined on $1, ..., n$.

For a definition of isotopism, see the wikipedia section on isotopism.

INPUT:

According to the type of input (see examples below) :

- an integer n – the function returns the identity on $1, ..., n$.
- a string representing a permutation in disjoint cycles notation, e.g. $(0, 1, 2)(3, 4, 5)$ – the corresponding permutation is returned, shifted by 1 to act on $1, ..., n$.
- list/tuple of tuples – assumes disjoint cycle notation, see previous entry.
- a list of integers – the function adds 1 to each member of the list, and returns the corresponding permutation.
- a `PermutationGroupElement` `p` – returns a permutation describing `p` **without** any shift.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: isotopism(5) # identity on 5 points
[1, 2, 3, 4, 5]
```

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: g = G.gen(0)
sage: isotopism(g)
[2, 3, 1, 5, 4]

sage: isotopism([0,3,2,1]) # 0 goes to 0, 1 goes to 3, etc.
[1, 4, 3, 2]

sage: isotopism( (0,1,2) ) # single cycle, presented as a tuple
[2, 3, 1]

sage: x = isotopism( ((0,1,2), (3,4)) ) # tuple of cycles
sage: x
[2, 3, 1, 5, 4]
sage: x.to_cycles()
[(1, 2, 3), (4, 5)]
```

sage.combinat.matrices.latin.**next_conjugate**(*L*)

Permutes L[r, c] = e to the conjugate L[c, e] = r.

We assume that L is an n by n matrix and has values in the range 0, 1, ..., n-1.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = back_circulant(6)
sage: L
[0 1 2 3 4 5]
[1 2 3 4 5 0]
[2 3 4 5 0 1]
[3 4 5 0 1 2]
[4 5 0 1 2 3]
[5 0 1 2 3 4]
sage: next_conjugate(L)
[0 1 2 3 4 5]
[5 0 1 2 3 4]
[4 5 0 1 2 3]
[3 4 5 0 1 2]
[2 3 4 5 0 1]
[1 2 3 4 5 0]
sage: L == next_conjugate(next_conjugate(next_conjugate(L)))
True
```

sage.combinat.matrices.latin.**p3_group_bitrade_generators**(*p*)

Generators for a group of order p3 where p is a prime.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: p3_group_bitrade_generators(3)
((2,6,7)(3,8,9), (1,2,3)(4,7,8)(5,6,9), (1,9,2)(3,7,4)(5,8,6), Permutation Group with generators
```

sage.combinat.matrices.latin.**pq_group_bitrade_generators**(*p*, *q*)

Generators for a group of order pq where p and q are primes such that (q % p) == 1.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: pq_group_bitrade_generators(3,7)
((2,3,5)(4,7,6), (1,2,3,4,5,6,7), (1,4,2)(3,5,6), Permutation Group with generators [(2,3,5)(4,7
```

sage.combinat.matrices.latin.**row_containing_sym**(*L*, *c*, *x*)

Given an improper latin square L with L[r1, c] = L[r2, c] = x, return r1 or r2 with equal probability. This is an internal function and should only be used in LatinSquare_generator().

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = matrix([(0, 1, 0, 3), (3, 0, 2, 1), (1, 0, 3, 2), (2, 3, 1, 0)])
sage: L
[0 1 0 3]
[3 0 2 1]
[1 0 3 2]
[2 3 1 0]
sage: c = row_containing_sym(L, 1, 0)
sage: c == 1 or c == 2
True
```

sage.combinat.matrices.latin.**tau1**(*T1*, *T2*, *cells_map*)

The definition of τ_1 is

$$\tau_1 : T1 \to T1$$
$$\tau_1 = \beta_2^{-1}\beta_3$$

where the composition is left to right and $\beta_i : T2 \to T1$ changes just the i^{th} coordinate of a triple.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: (cells_map, t1, t2, t3) = tau123(T1, T2)
sage: t1 = tau1(T1, T2, cells_map)
sage: t1
[2, 3, 4, 5, 1, 7, 8, 9, 10, 6, 12, 13, 14, 15, 11, 17, 18, 19, 20, 16, 22, 23, 24, 25, 21]
sage: t1.to_cycles()
[(1, 2, 3, 4, 5), (6, 7, 8, 9, 10), (11, 12, 13, 14, 15), (16, 17, 18, 19, 20), (21, 22, 23, 24,
```

sage.combinat.matrices.latin.**tau123**(*T1*, *T2*)

Compute the tau_i representation for a bitrade (T1, T2). See the functions tau1, tau2, and tau3 for the mathematical definitions.

OUTPUT:

- (cells_map, t1, t2, t3)

where cells_map is a map to/from the filled cells of T1, and t1, t2, t3 are the tau1, tau2, tau3 permutations.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: (a, b, c, G) = pq_group_bitrade_generators(3, 7)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1
[ 0  1  3 -1 -1 -1 -1]
[ 1  2  4 -1 -1 -1 -1]
[ 2  3  5 -1 -1 -1 -1]
[ 3  4  6 -1 -1 -1 -1]
```

```
[ 4  5  0 -1 -1 -1 -1]
[ 5  6  1 -1 -1 -1 -1]
[ 6  0  2 -1 -1 -1 -1]
sage: T2
[ 1  3  0 -1 -1 -1 -1]
[ 2  4  1 -1 -1 -1 -1]
[ 3  5  2 -1 -1 -1 -1]
[ 4  6  3 -1 -1 -1 -1]
[ 5  0  4 -1 -1 -1 -1]
[ 6  1  5 -1 -1 -1 -1]
[ 0  2  6 -1 -1 -1 -1]
sage: (cells_map, t1, t2, t3) = tau123(T1, T2)
sage: cells_map
{1: (0, 0),
 2: (0, 1),
 3: (0, 2),
 4: (1, 0),
 5: (1, 1),
 6: (1, 2),
 7: (2, 0),
 8: (2, 1),
 9: (2, 2),
 10: (3, 0),
 11: (3, 1),
 12: (3, 2),
 13: (4, 0),
 14: (4, 1),
 15: (4, 2),
 16: (5, 0),
 17: (5, 1),
 18: (5, 2),
 19: (6, 0),
 20: (6, 1),
 21: (6, 2),
 (0, 0): 1,
 (0, 1): 2,
 (0, 2): 3,
 (1, 0): 4,
 (1, 1): 5,
 (1, 2): 6,
 (2, 0): 7,
 (2, 1): 8,
 (2, 2): 9,
 (3, 0): 10,
 (3, 1): 11,
 (3, 2): 12,
 (4, 0): 13,
 (4, 1): 14,
 (4, 2): 15,
 (5, 0): 16,
 (5, 1): 17,
 (5, 2): 18,
 (6, 0): 19,
 (6, 1): 20,
 (6, 2): 21}
sage: cells_map_as_square(cells_map, max(T1.nrows(), T1.ncols()))
[ 1  2  3 -1 -1 -1 -1]
[ 4  5  6 -1 -1 -1 -1]
```

```
[ 7  8  9 -1 -1 -1 -1]
[10 11 12 -1 -1 -1 -1]
[13 14 15 -1 -1 -1 -1]
[16 17 18 -1 -1 -1 -1]
[19 20 21 -1 -1 -1 -1]
sage: t1
[3, 1, 2, 6, 4, 5, 9, 7, 8, 12, 10, 11, 15, 13, 14, 18, 16, 17, 21, 19, 20]
sage: t2
[4, 8, 15, 7, 11, 18, 10, 14, 21, 13, 17, 3, 16, 20, 6, 19, 2, 9, 1, 5, 12]
sage: t3
[20, 18, 10, 2, 21, 13, 5, 3, 16, 8, 6, 19, 11, 9, 1, 14, 12, 4, 17, 15, 7]

sage: t1.to_cycles()
[(1, 3, 2), (4, 6, 5), (7, 9, 8), (10, 12, 11), (13, 15, 14), (16, 18, 17), (19, 21, 20)]
sage: t2.to_cycles()
[(1, 4, 7, 10, 13, 16, 19), (2, 8, 14, 20, 5, 11, 17), (3, 15, 6, 18, 9, 21, 12)]
sage: t3.to_cycles()
[(1, 20, 15), (2, 18, 4), (3, 10, 8), (5, 21, 7), (6, 13, 11), (9, 16, 14), (12, 19, 17)]
```

The product t1*t2*t3 is the identity, i.e. it fixes every point:

```
sage: len((t1*t2*t3).fixed_points()) == T1.nr_filled_cells()
True
```

sage.combinat.matrices.latin.**tau2**(*T1*, *T2*, *cells_map*)

The definition of τ_2 is

$$\tau_2 : T1 \to T1$$
$$\tau_2 = \beta_3^{-1}\beta_1$$

where the composition is left to right and $\beta_i : T2 \to T1$ changes just the i^{th} coordinate of a triple.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: (cells_map, t1, t2, t3) = tau123(T1, T2)
sage: t2 = tau2(T1, T2, cells_map)
sage: t2
[21, 22, 23, 24, 25, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
sage: t2.to_cycles()
[(1, 21, 16, 11, 6), (2, 22, 17, 12, 7), (3, 23, 18, 13, 8), (4, 24, 19, 14, 9), (5, 25, 20, 15,
```

sage.combinat.matrices.latin.**tau3**(*T1*, *T2*, *cells_map*)

The definition of τ_3 is

$$\tau_3 : T1 \to T1$$
$$\tau_3 = \beta_1^{-1}\beta_2$$

where the composition is left to right and $\beta_i : T2 \to T1$ changes just the i^{th} coordinate of a triple.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: (cells_map, t1, t2, t3) = tau123(T1, T2)
sage: t3 = tau3(T1, T2, cells_map)
sage: t3
[10, 6, 7, 8, 9, 15, 11, 12, 13, 14, 20, 16, 17, 18, 19, 25, 21, 22, 23, 24, 5, 1, 2, 3, 4]
sage: t3.to_cycles()
[(1, 10, 14, 18, 22), (2, 6, 15, 19, 23), (3, 7, 11, 20, 24), (4, 8, 12, 16, 25), (5, 9, 13, 17,
```

sage.combinat.matrices.latin.**tau_to_bitrade**(*t1*, *t2*, *t3*)

Given permutations t1, t2, t3 that represent a latin bitrade, convert them to an explicit latin bitrade (T1, T2). The result is unique up to isotopism.

EXAMPLE:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: _, t1, t2, t3 = tau123(T1, T2)
sage: U1, U2 = tau_to_bitrade(t1, t2, t3)
sage: assert is_bitrade(U1, U2)
sage: U1
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
[4 0 1 2 3]
sage: U2
[4 0 1 2 3]
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
```

5.1.123 Miscellaneous

class sage.combinat.misc.**DoublyLinkedList**(*l*)

A doubly linked list class that provides constant time hiding and unhiding of entries.

Note that this list's indexing is 1-based.

EXAMPLES:

```
sage: dll = sage.combinat.misc.DoublyLinkedList([1,2,3]); dll
Doubly linked list of [1, 2, 3]: [1, 2, 3]
sage: dll.hide(1); dll
Doubly linked list of [1, 2, 3]: [2, 3]
sage: dll.unhide(1); dll
Doubly linked list of [1, 2, 3]: [1, 2, 3]
```

```
sage: dll.hide(2); dll
Doubly linked list of [1, 2, 3]: [1, 3]
sage: dll.unhide(2); dll
Doubly linked list of [1, 2, 3]: [1, 2, 3]
```

head()

TESTS:

```
sage: dll = sage.combinat.misc.DoublyLinkedList([1,2,3])
sage: dll.head()
1
sage: dll.hide(1)
sage: dll.head()
2
```

hide(*i*)

TESTS:

```
sage: dll = sage.combinat.misc.DoublyLinkedList([1,2,3])
sage: dll.hide(1)
sage: list(dll)
[2, 3]
```

next(*j*)

TESTS:

```
sage: dll = sage.combinat.misc.DoublyLinkedList([1,2,3])
sage: dll.next(1)
2
sage: dll.hide(2)
sage: dll.next(1)
3
```

prev(*j*)

TESTS:

```
sage: dll = sage.combinat.misc.DoublyLinkedList([1,2,3])
sage: dll.prev(3)
2
sage: dll.hide(2)
sage: dll.prev(3)
1
```

unhide(*i*)

TESTS:

```
sage: dll = sage.combinat.misc.DoublyLinkedList([1,2,3])
sage: dll.hide(1); dll.unhide(1)
sage: list(dll)
[1, 2, 3]
```

class sage.combinat.misc.**IterableFunctionCall**(*f*, **args*, ***kwargs*)

This class wraps functions with a yield statement (generators) by an object that can be iterated over. For example,

EXAMPLES:

```
sage: def f(): yield 'a'; yield 'b'
```

This does not work:

```
sage: for z in f: print z
Traceback (most recent call last):
...
TypeError: 'function' object is not iterable
```

Use IterableFunctionCall if you want something like the above to work:

```
sage: from sage.combinat.misc import IterableFunctionCall
sage: g = IterableFunctionCall(f)
sage: for z in g: print z
a
b
```

If your function takes arguments, just put them after the function name. You needn't enclose them in a tuple or anything, just put them there:

```
sage: def f(n, m): yield 'a' * n; yield 'b' * m; yield 'foo'
sage: g = IterableFunctionCall(f, 2, 3)
sage: for z in g: print z
aa
bbb
foo
```

sage.combinat.misc.**check_integer_list_constraints**(*l*, ***kwargs*)

EXAMPLES:

```
sage: from sage.combinat.misc import check_integer_list_constraints
sage: cilc = check_integer_list_constraints
sage: l = [[2,1,3],[1,2],[3,3],[4,1,1]]
sage: cilc(l, min_part=2)
[[3, 3]]
sage: cilc(l, max_part=2)
[[1, 2]]
sage: cilc(l, length=2)
[[1, 2], [3, 3]]
sage: cilc(l, max_length=2)
[[1, 2], [3, 3]]
sage: cilc(l, min_length=3)
[[2, 1, 3], [4, 1, 1]]
sage: cilc(l, max_slope=0)
[[3, 3], [4, 1, 1]]
sage: cilc(l, min_slope=1)
[[1, 2]]
sage: cilc(l, outer=[2,2])
[[1, 2]]
sage: cilc(l, inner=[2,2])
[[3, 3]]

sage: cilc([1,2,3], length=3, singleton=True)
[1, 2, 3]
sage: cilc([1,2,3], length=2, singleton=True) is None
True
```

sage.combinat.misc.**umbral_operation**(*poly*)

Returns the umbral operation $\downarrow$ applied to poly.

The umbral operation replaces each instance of $x_i^{a_i}$ with $x_i * (x_i - 1) * \cdots * (x_i - a_i + 1)$.

EXAMPLES:

```
sage: P = PolynomialRing(QQ, 2, 'x')
sage: x = P.gens()
sage: from sage.combinat.misc import umbral_operation
sage: umbral_operation(x[0]^3) == x[0]*(x[0]-1)*(x[0]-2)
True
sage: umbral_operation(x[0]*x[1])
x0*x1
sage: umbral_operation(x[0]+x[1])
x0 + x1
sage: umbral_operation(x[0]^2*x[1]^2) == x[0]*(x[0]-1)*x[1]*(x[1]-1)
True
```

5.1.124 Low-level multichoose

class sage.combinat.multichoose_nk.**MultichooseNK**(n, k)

Bases: sage.combinat.combinat.CombinatorialClass

TESTS:

```
sage: a = MultichooseNK(3,2)
sage: a == loads(dumps(a))
True
```

cardinality()

Returns the number of multichoices of k things from a list of n things.

EXAMPLES:

```
sage: MultichooseNK(3,2).cardinality()
6
```

random_element()

Returns a random multichoice of k things from range(n).

EXAMPLES:

```
sage: MultichooseNK(5,2).random_element()
[0, 2]
sage: MultichooseNK(5,2).random_element()
[0, 1]
```

5.1.125 Non-Commutative Symmetric Functions and Quasi-Symmetric Functions

- *Introduction to Quasisymmetric Functions*
- *Non-Commutative Symmetric Functions (NCSF)*
- *Quasi-Symmetric Functions (QSym)*
- *Generic code for bases*

5.1.126 Features that are imported by default in the interpreter namespace

5.1.127 Common combinatorial tools

REFERENCES:

sage.combinat.ncsf_qsym.combinatorics.**coeff_dab**(I, J)

Return the number of standard composition tableaux of shape I with descent composition J.

INPUT:

- I, J – compositions

OUTPUT:

- An integer

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import coeff_dab
sage: coeff_dab(Composition([2,1]),Composition([2,1]))
1
sage: coeff_dab(Composition([1,1,2]),Composition([1,2,1]))
0
```

sage.combinat.ncsf_qsym.combinatorics.**coeff_ell**(J, I)

Returns the coefficient $\ell_{J,I}$ as defined in [NCSF].

INPUT:

- J – a composition
- I – a composition refining J

OUTPUT:

- integer

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import coeff_ell
sage: coeff_ell(Composition([1,1,1]), Composition([2,1]))
2
sage: coeff_ell(Composition([2,1]), Composition([3]))
2
```

sage.combinat.ncsf_qsym.combinatorics.**coeff_lp**(J, I)

Returns the coefficient $lp_{J,I}$ as defined in [NCSF].

INPUT:

- J – a composition
- I – a composition refining J

OUTPUT:

- integer

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import coeff_lp
sage: coeff_lp(Composition([1,1,1]), Composition([2,1]))
1
sage: coeff_lp(Composition([2,1]), Composition([3]))
1
```

sage.combinat.ncsf_qsym.combinatorics.**coeff_pi**(*J*, *I*)

Returns the coefficient $\pi_{J,I}$ as defined in [NCSF].

INPUT:

- J – a composition
- I – a composition refining J

OUTPUT:

- integer

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import coeff_pi
sage: coeff_pi(Composition([1,1,1]), Composition([2,1]))
2
sage: coeff_pi(Composition([2,1]), Composition([3]))
6
```

sage.combinat.ncsf_qsym.combinatorics.**coeff_sp**(*J*, *I*)

Returns the coefficient $sp_{J,I}$ as defined in [NCSF].

INPUT:

- J – a composition
- I – a composition refining J

OUTPUT:

- integer

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import coeff_sp
sage: coeff_sp(Composition([1,1,1]), Composition([2,1]))
2
sage: coeff_sp(Composition([2,1]), Composition([3]))
4
```

sage.combinat.ncsf_qsym.combinatorics.**compositions_order**(*n*)

Return the compositions of n ordered as defined in [QSCHUR].

Let $S(\gamma)$ return the composition γ after sorting. For compositions α and β, we order $\alpha \rhd \beta$ if

1. $S(\alpha) > S(\beta)$ lexicographically, or
2. $S(\alpha) = S(\beta)$ and $\alpha > \beta$ lexicographically.

INPUT:

- n – a positive integer

OUTPUT:

- A list of the compositions of n sorted into decreasing order by $\rhd$

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import compositions_order
sage: compositions_order(3)
[[3], [2, 1], [1, 2], [1, 1, 1]]
```

```
sage: compositions_order(4)
[[4], [3, 1], [1, 3], [2, 2], [2, 1, 1], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

sage.combinat.ncsf_qsym.combinatorics.**m_to_s_stat**(*R*, *I*, *K*)

Return the coefficient of the complete non-commutative symmetric function S^K in the expansion of the monomial non-commutative symmetric function M^I with respect to the complete basis over the ring R. This is the coefficient in formula (36) of Tevlin's paper [Tev2007].

INPUT:

- `R` – A ring, supposed to be a **Q**-algebra
- `I`, `K` – compositions

OUTPUT:

- The coefficient of S^K in the expansion of M^I in the complete basis of the non-commutative symmetric functions over `R`.

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import m_to_s_stat
sage: m_to_s_stat(QQ, Composition([2,1]), Composition([1,1,1]))
-1
sage: m_to_s_stat(QQ, Composition([3]), Composition([1,2]))
-2
sage: m_to_s_stat(QQ, Composition([2,1,2]), Composition([2,1,2]))
8/3
```

sage.combinat.ncsf_qsym.combinatorics.**number_of_SSRCT**(*content_comp*, *shape_comp*)

The number of semi-standard reverse composition tableaux.

The dual quasisymmetric-Schur functions satisfy a left Pieri rule where $S_n dQS_\gamma$ is a sum over dual quasisymmetric-Schur functions indexed by compositions which contain the composition γ. The definition of an SSRCT comes from this rule. The number of SSRCT of content β and shape α is equal to the number of SSRCT of content $(\beta_2, \ldots, \beta_\ell)$ and shape γ where dQS_α appears in the expansion of $S_{\beta_1} dQS_\gamma$.

In sage the recording tableau for these objects are called `CompositionTableaux`.

INPUT:

- `content_comp`, `shape_comp` – compositions

OUTPUT:

- An integer

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import number_of_SSRCT
sage: number_of_SSRCT(Composition([3,1]), Composition([1,3]))
0
sage: number_of_SSRCT(Composition([1,2,1]), Composition([1,3]))
1
sage: number_of_SSRCT(Composition([1,1,2,2]), Composition([3,3]))
2
sage: all(CompositionTableaux(be).cardinality()
....:     == sum(number_of_SSRCT(al,be)*binomial(4,len(al))
....:            for al in Compositions(4))
....:     for be in Compositions(4))
True
```

sage.combinat.ncsf_qsym.combinatorics.**number_of_fCT**(*content_comp*, *shape_comp*)
Return the number of Immaculate tableaux of shape `shape_comp` and content `content_comp`.

See [BBSSZ2012], Definition 3.9, for the notion of an immaculate tableau.

INPUT:

- `content_comp`, `shape_comp` – compositions

OUTPUT:

- An integer

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.combinatorics import number_of_fCT
sage: number_of_fCT(Composition([3,1]), Composition([1,3]))
0
sage: number_of_fCT(Composition([1,2,1]), Composition([1,3]))
1
sage: number_of_fCT(Composition([1,1,3,1]), Composition([2,1,3]))
2
```

5.1.128 Generic code for bases

This is a collection of code that is shared by bases of noncommutative symmetric functions and quasisymmetric functions.

AUTHORS:

- Jason Bandlow
- Franco Saliola
- Chris Berg

class sage.combinat.ncsf_qsym.generic_basis_code.**AlgebraMorphism**(*domain*, *on_generators*, *position=0*, *codomain=None*, *category=None*, *anti=False*)
Bases: sage.modules.with_basis.morphism.ModuleMorphismByLinearity

A class for algebra morphism defined on a free algebra from the image of the generators

class sage.combinat.ncsf_qsym.generic_basis_code.**BasesOfQSymOrNCSF**(*parent_with_realization*)
Bases: sage.categories.realizations.Category_realization_of_parent

TESTS:

```
sage: from sage.categories.realizations import Category_realization_of_parent
sage: A = Sets().WithRealizations().example(); A
The subset algebra of {1, 2, 3} over Rational Field
sage: C = A.Realizations(); C
Category of realizations of The subset algebra of {1, 2, 3} over Rational Field
sage: isinstance(C, Category_realization_of_parent)
True
sage: C.parent_with_realization
The subset algebra of {1, 2, 3} over Rational Field
sage: TestSuite(C).run(skip=["_test_category_over_bases"])
```

Todo

Fix the failing test by making `C` a singleton category. This will require some fiddling with the assertion in `Category_singleton.__classcall__()`

class ElementMethods

degree()

The maximum of the degrees of the homogeneous summands.

See also:

`homogeneous_degree()`

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: (x, y) = (S[2], S[3])
sage: x.degree()
2
sage: (x^3 + 4*y^2).degree()
6
sage: ((1 + x)^3).degree()
6

sage: F = QuasiSymmetricFunctions(QQ).F()
sage: (x, y) = (F[2], F[3])
sage: x.degree()
2
sage: (x^3 + 4*y^2).degree()
6
sage: ((1 + x)^3).degree()
6
```

TESTS:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.zero().degree()
Traceback (most recent call last):
...
ValueError: the zero element does not have a well-defined degree
sage: F = QuasiSymmetricFunctions(QQ).F()
sage: F.zero().degree()
Traceback (most recent call last):
...
ValueError: the zero element does not have a well-defined degree
```

degree_negation()

Return the image of `self` under the degree negation automorphism of the parent of `self`.

The degree negation is the automorphism which scales every homogeneous element of degree k by $(-1)^k$ (for all k).

Calling `degree_negation(self)` is equivalent to calling `self.parent().degree_negation(self)`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: f = 2*S[2,1] + 4*S[1,1] - 5*S[1,2] - 3*S[[]]
sage: f.degree_negation()
```

```
-3*S[] + 4*S[1, 1] + 5*S[1, 2] - 2*S[2, 1]

sage: QSym = QuasiSymmetricFunctions(QQ)
sage: dI = QSym.dualImmaculate()
sage: f = -3*dI[2,1] + 4*dI[2] + 2*dI[1]
sage: f.degree_negation()
-2*dI[1] + 4*dI[2] + 3*dI[2, 1]
```

TESTS:

The zero element behaves well:

```
sage: a = S.zero().degree_negation(); a
0
sage: parent(a)
Non-Commutative Symmetric Functions over the Integer Ring in the Complete basis
```

Todo

Generalize this to all graded vector spaces?

duality_pairing(*y*)

The duality pairing between elements of $NSym$ and elements of $QSym$.

The complete basis is dual to the monomial basis with respect to this pairing.

INPUT:

- `y` – an element of the dual Hopf algebra of `self`

OUTPUT:

- The result of pairing `self` with `y`.

EXAMPLES:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).Ribbon()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: R[1,1,2].duality_pairing(F[1,1,2])
1
sage: R[1,2,1].duality_pairing(F[1,1,2])
0

sage: L = NonCommutativeSymmetricFunctions(QQ).Elementary()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: L[1,2].duality_pairing(F[1,2])
0
sage: L[1,1,1].duality_pairing(F[1,2])
1
```

skew_by(*y*, *side='left'*)

The operation which is dual to multiplication by `y`, where `y` is an element of the dual space of `self`.

This is calculated through the coproduct of `self` and the expansion of `y` in the dual basis.

INPUT:

- `y` – an element of the dual Hopf algebra of `self`
- `side` – (Default='left') Either 'left' or 'right'

OUTPUT:

- The result of skewing `self` by `y`, on the side `side`

EXAMPLES:

Skewing an element of NCSF by an element of QSym:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
```

```
sage: R([2,2,2]).skew_by(F[1,1])
R[1, 1, 2] + R[1, 2, 1] + R[1, 3] + R[2, 1, 1] + 2*R[2, 2] + R[3, 1] + R[4]
sage: R([2,2,2]).skew_by(F[2])
R[1, 1, 2] + R[1, 2, 1] + R[1, 3] + R[2, 1, 1] + 3*R[2, 2] + R[3, 1] + R[4]
```

Skewing an element of QSym by an element of NCSF:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: R = NonCommutativeSymmetricFunctions(QQ).R()
sage: F = QuasiSymmetricFunctions(QQ).F()
sage: F[3,2].skew_by(R[1,1])
0
sage: F[3,2].skew_by(R[1,1], side='right')
0
sage: F[3,2].skew_by(S[1,1,1], side='right')
F[2]
sage: F[3,2].skew_by(S[1,2], side='right')
F[2]
sage: F[3,2].skew_by(S[2,1], side='right')
0
sage: F[3,2].skew_by(S[1,1,1])
F[2]
sage: F[3,2].skew_by(S[1,1])
F[1, 2]
sage: F[3,2].skew_by(S[1])
F[2, 2]
```

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: R = NonCommutativeSymmetricFunctions(QQ).R()
sage: M = QuasiSymmetricFunctions(QQ).M()
sage: M[3,2].skew_by(S[2])
0
sage: M[3,2].skew_by(S[2], side='right')
M[3]
sage: M[3,2].skew_by(S[3])
M[2]
sage: M[3,2].skew_by(S[3], side='right')
0
```

class BasesOfQSymOrNCSF.**ParentMethods**

alternating_sum_of_compositions(*n*)

Alternating sum over compositions of n.

Note that this differs from the method alternating_sum_of_finer_compositions() because the coefficient of the composition 1^n is positive. This method is used in the expansion of the elementary generators into the complete generators and vice versa.

INPUT:

- n – a positive integer

OUTPUT:

- The expansion of the complete generator indexed by n into the elementary basis.

EXAMPLES:

```
sage: L=NonCommutativeSymmetricFunctions(QQ).L()
sage: L.alternating_sum_of_compositions(0)
L[]
sage: L.alternating_sum_of_compositions(1)
L[1]
sage: L.alternating_sum_of_compositions(2)
```

```
L[1, 1] - L[2]
sage: L.alternating_sum_of_compositions(3)
L[1, 1, 1] - L[1, 2] - L[2, 1] + L[3]
sage: S=NonCommutativeSymmetricFunctions(QQ).S()
sage: S.alternating_sum_of_compositions(3)
S[1, 1, 1] - S[1, 2] - S[2, 1] + S[3]
```

alternating_sum_of_fatter_compositions(*composition*)

Return the alternating sum of fatter compositions in a basis of the non-commutative symmetric functions.

INPUT:

- `composition` – a composition

OUTPUT:

- The alternating sum of the compositions fatter than `composition`, in the basis `self`. The alternation is upon the length of the compositions, and is normalized so that `composition` has coefficient 1.

EXAMPLES:

```
sage: NCSF=NonCommutativeSymmetricFunctions(QQ)
sage: elementary = NCSF.elementary()
sage: elementary.alternating_sum_of_fatter_compositions(Composition([2,2,1]))
L[2, 2, 1] - L[2, 3] - L[4, 1] + L[5]
sage: elementary.alternating_sum_of_fatter_compositions(Composition([1,2]))
L[1, 2] - L[3]
```

TESTS:

```
sage: complete = NonCommutativeSymmetricFunctions(ZZ).complete()
sage: I = Composition([1,1])
sage: x = complete.alternating_sum_of_fatter_compositions(I)
sage: [c.parent() for c in x.coefficients()]
[Integer Ring, Integer Ring]
```

alternating_sum_of_finer_compositions(*composition*, *conjugate=False*)

Return the alternating sum of finer compositions in a basis of the non-commutative symmetric functions.

INPUT:

- `composition` – a composition
- `conjugate` – (default: `False`) a boolean

OUTPUT:

- The alternating sum of the compositions finer than `composition`, in the basis `self`. The alternation is upon the length of the compositions, and is normalized so that `composition` has coefficient 1. If the variable `conjugate` is set to `True`, then the conjugate of `composition` is used instead of `composition`.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: elementary = NCSF.elementary()
sage: elementary.alternating_sum_of_finer_compositions(Composition([2,2,1]))
L[1, 1, 1, 1, 1] - L[1, 1, 2, 1] - L[2, 1, 1, 1] + L[2, 2, 1]
sage: elementary.alternating_sum_of_finer_compositions(Composition([1,2]))
-L[1, 1, 1] + L[1, 2]
```

TESTS:

```
sage: complete = NonCommutativeSymmetricFunctions(ZZ).complete()
sage: I = Composition([2])
sage: x = complete.alternating_sum_of_finer_compositions(I)
sage: [c.parent() for c in x.coefficients()]
```

```
[Integer Ring, Integer Ring]
```

counit_on_basis(*I*)

The counit is defined by sending all elements of positive degree to zero.

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.counit_on_basis([1,3])
0
sage: M = QuasiSymmetricFunctions(QQ).M()
sage: M.counit_on_basis([1,3])
0
```

TESTS:

```
sage: S.counit_on_basis([])
1
sage: S.counit_on_basis(Composition([]))
1
sage: M.counit_on_basis([])
1
sage: M.counit_on_basis(Composition([]))
1
```

degree_negation(*element*)

Return the image of `element` under the degree negation automorphism of `self`.

The degree negation is the automorphism which scales every homogeneous element of degree k by $(-1)^k$ (for all k).

INPUT:

- `element` – element of `self`

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: f = 2*S[2,1] + 4*S[1,1] - 5*S[1,2] - 3*S[[]]
sage: S.degree_negation(f)
-3*S[] + 4*S[1, 1] + 5*S[1, 2] - 2*S[2, 1]

sage: QSym = QuasiSymmetricFunctions(QQ)
sage: dI = QSym.dualImmaculate()
sage: f = -3*dI[2,1] + 4*dI[2] + 2*dI[1]
sage: dI.degree_negation(f)
-2*dI[1] + 4*dI[2] + 3*dI[2, 1]
```

TESTS:

Using `degree_negation()` on an element of a different basis works correctly:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: S = NSym.S()
sage: Phi = NSym.Phi()
sage: S.degree_negation(Phi[2])
-S[1, 1] + 2*S[2]
sage: S.degree_negation(Phi[3])
-S[1, 1, 1] + 3/2*S[1, 2] + 3/2*S[2, 1] - 3*S[3]
sage: Phi.degree_negation(S[3])
-1/6*Phi[1, 1, 1] - 1/4*Phi[1, 2] - 1/4*Phi[2, 1] - 1/3*Phi[3]
```

The zero element behaves well:

```
sage: a = Phi.degree_negation(S.zero()); a
0
sage: parent(a)
Non-Commutative Symmetric Functions over the Rational Field in the Phi basis
```

Todo

Generalize this to all graded vector spaces?

---

degree_on_basis(*I*)

Return the degree of the basis element indexed by I.

INPUT:

- `I` – a composition

OUTPUT:

- The degree of the non-commutative symmetric function basis element of `self` indexed by `I`. By definition, this is the size of the composition `I`.

EXAMPLES:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: R.degree_on_basis(Composition([2,3]))
5
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: M.degree_on_basis(Composition([3,2]))
5
sage: M.degree_on_basis(Composition([]))
0
```

duality_pairing(*x*, *y*)

The duality pairing between elements of $NSym$ and elements of $QSym$.

This is a default implementation that uses `self.realizations_of().a_realization()` and its dual basis.

INPUT:

- `x` – an element of `self`
- `y` – an element in the dual basis of `self`

OUTPUT:

- The result of pairing the function `x` from `self` with the function `y` from the dual basis of `self`

EXAMPLES:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).Ribbon()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: R.duality_pairing(R[1,1,2], F[1,1,2])
1
sage: R.duality_pairing(R[1,2,1], F[1,1,2])
0
sage: F.duality_pairing(F[1,2,1], R[1,1,2])
0

sage: S = NonCommutativeSymmetricFunctions(QQ).Complete()
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: S.duality_pairing(S[1,1,2], M[1,1,2])
1
sage: S.duality_pairing(S[1,2,1], M[1,1,2])
0
sage: M.duality_pairing(M[1,1,2], S[1,1,2])
1
sage: M.duality_pairing(M[1,2,1], S[1,1,2])
0
```

```
sage: S = NonCommutativeSymmetricFunctions(QQ).Complete()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: S.duality_pairing(S[1,2], F[1,1,1])
0
sage: S.duality_pairing(S[1,1,1,1], F[4])
1
```

TESTS:

The result has the right parent even if the sum is empty:

```
sage: x = S.duality_pairing(S.zero(), F.zero()); x
0
sage: parent(x)
Rational Field
```

duality_pairing_by_coercion(*x*, *y*)

The duality pairing between elements of NSym and elements of QSym.

This is a default implementation that uses `self.realizations_of().a_realization()` and its dual basis.

INPUT:

- `x` – an element of `self`
- `y` – an element in the dual basis of `self`

OUTPUT:

- The result of pairing the function `x` from `self` with the function `y` from the dual basis of `self`

EXAMPLES:

```
sage: L = NonCommutativeSymmetricFunctions(QQ).Elementary()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: L.duality_pairing_by_coercion(L[1,2], F[1,2])
0
sage: F.duality_pairing_by_coercion(F[1,2], L[1,2])
0
sage: L.duality_pairing_by_coercion(L[1,1,1], F[1,2])
1
sage: F.duality_pairing_by_coercion(F[1,2], L[1,1,1])
1
```

TESTS:

The result has the right parent even if the sum is empty:

```
sage: x = F.duality_pairing_by_coercion(F.zero(), L.zero()); x
0
sage: parent(x)
Rational Field
```

duality_pairing_matrix(*basis*, *degree*)

The matrix of scalar products between elements of NSym and elements of QSym.

INPUT:

- `basis` – A basis of the dual Hopf algebra
- `degree` – a non-negative integer

OUTPUT:

- The matrix of scalar products between the basis `self` and the basis `basis` in the dual Hopf algebra in degree `degree`.

EXAMPLES:

The ribbon basis of NCSF is dual to the fundamental basis of QSym:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: R.duality_pairing_matrix(F, 3)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: F.duality_pairing_matrix(R, 3)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

The complete basis of NCSF is dual to the monomial basis of QSym:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).complete()
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: S.duality_pairing_matrix(M, 3)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: M.duality_pairing_matrix(S, 3)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

The matrix between the ribbon basis of NCSF and the monomial basis of QSym:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: R.duality_pairing_matrix(M, 3)
[ 1 -1 -1  1]
[ 0  1  0 -1]
[ 0  0  1 -1]
[ 0  0  0  1]
sage: M.duality_pairing_matrix(R, 3)
[ 1  0  0  0]
[-1  1  0  0]
[-1  0  1  0]
[ 1 -1 -1  1]
```

The matrix between the complete basis of NCSF and the fundamental basis of QSym:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).complete()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: S.duality_pairing_matrix(F, 3)
[1 1 1 1]
[0 1 0 1]
[0 0 1 1]
[0 0 0 1]
```

A base case test:

```
sage: R.duality_pairing_matrix(M,0)
[1]
```

one_basis()

Return the empty composition.

OUTPUT

•The empty composition.

EXAMPLES:

```
sage: L=NonCommutativeSymmetricFunctions(QQ).L()
sage: parent(L)
<class 'sage.combinat.ncsf_qsym.ncsf.NonCommutativeSymmetricFunctions.Elementary_with_ca
sage: parent(L).one_basis()
[]
```

skew(*x*, *y*, *side='left'*)

Return a function `x` in `self` skewed by a function `y` in the Hopf dual of `self`.

INPUT:

- `x` – a non-commutative or quasi-symmetric function; it is an element of `self`
- `y` – a quasi-symmetric or non-commutative symmetric function; it is an element of the dual algebra of `self`
- `side` – (default: `'left'`) either `'left'` or `'right'`

OUTPUT:

- The result of skewing the element `x` by the Hopf algebra element `y` (either from the left or from the right, as determined by `side`), written in the basis `self`.

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).complete()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: S.skew(S[2,2,2], F[1,1])
S[1, 1, 2] + S[1, 2, 1] + S[2, 1, 1]
sage: S.skew(S[2,2,2], F[2])
S[1, 1, 2] + S[1, 2, 1] + S[2, 1, 1] + 3*S[2, 2]

sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: R.skew(R[2,2,2], F[1,1])
R[1, 1, 2] + R[1, 2, 1] + R[1, 3] + R[2, 1, 1] + 2*R[2, 2] + R[3, 1] + R[4]
sage: R.skew(R[2,2,2], F[2])
R[1, 1, 2] + R[1, 2, 1] + R[1, 3] + R[2, 1, 1] + 3*R[2, 2] + R[3, 1] + R[4]

sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: R = NonCommutativeSymmetricFunctions(QQ).R()
sage: M = QuasiSymmetricFunctions(QQ).M()
sage: M.skew(M[3,2], S[2])
0
sage: M.skew(M[3,2], S[2], side='right')
M[3]
sage: M.skew(M[3,2], S[3])
M[2]
sage: M.skew(M[3,2], S[3], side='right')
0
```

TESTS:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).R()
sage: R.skew([2,1], [1])
Traceback (most recent call last):
...
AssertionError: x must be an element of Non-Commutative Symmetric Functions over the Rat
sage: R([2,1]).skew_by([1])
Traceback (most recent call last):
...
AssertionError: y must be an element of Quasisymmetric functions over the Rational Field
sage: F = QuasiSymmetricFunctions(QQ).F()
sage: F([2,1]).skew_by([1])
Traceback (most recent call last):
```

```
...
AssertionError: y must be an element of Non-Commutative Symmetric Functions over the Rat
```

sum_of_fatter_compositions(*composition*)

Return the sum of all fatter compositions.

INPUT:

- composition – a composition

OUTPUT:

- the sum of all basis elements which are indexed by compositions fatter (coarser?) than composition.

EXAMPLES:

```
sage: L=NonCommutativeSymmetricFunctions(QQ).L()
sage: L.sum_of_fatter_compositions(Composition([2,1]))
L[2, 1] + L[3]
sage: R=NonCommutativeSymmetricFunctions(QQ).R()
sage: R.sum_of_fatter_compositions(Composition([1,3]))
R[1, 3] + R[4]
```

sum_of_finer_compositions(*composition*)

Return the sum of all finer compositions.

INPUT:

- composition – a composition

OUTPUT:

- The sum of all basis self elements which are indexed by compositions finer than composition.

EXAMPLES:

```
sage: L=NonCommutativeSymmetricFunctions(QQ).L()
sage: L.sum_of_finer_compositions(Composition([2,1]))
L[1, 1, 1] + L[2, 1]
sage: R=NonCommutativeSymmetricFunctions(QQ).R()
sage: R.sum_of_finer_compositions(Composition([1,3]))
R[1, 1, 1, 1] + R[1, 1, 2] + R[1, 2, 1] + R[1, 3]
```

sum_of_partition_rearrangements(*par*)

Return the sum of all basis elements indexed by compositions which can be sorted to obtain a given partition.

INPUT:

- par – a partition

OUTPUT:

- The sum of all self basis elements indexed by compositions which are permutations of par (without multiplicity).

EXAMPLES:

```
sage: NCSF=NonCommutativeSymmetricFunctions(QQ)
sage: elementary = NCSF.elementary()
sage: elementary.sum_of_partition_rearrangements(Partition([2,2,1]))
L[1, 2, 2] + L[2, 1, 2] + L[2, 2, 1]
sage: elementary.sum_of_partition_rearrangements(Partition([3,2,1]))
L[1, 2, 3] + L[1, 3, 2] + L[2, 1, 3] + L[2, 3, 1] + L[3, 1, 2] + L[3, 2, 1]
sage: elementary.sum_of_partition_rearrangements(Partition([]))
L[]
```

BasesOfQSymOrNCSF.**super_categories**()

TESTS:

```
sage: from sage.combinat.ncsf_qsym.generic_basis_code import BasesOfQSymOrNCSF
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: BasesOfQSymOrNCSF(QSym).super_categories()
[Category of realizations of Quasisymmetric functions over the Rational Field,
 Category of graded hopf algebras with basis over Rational Field,
 Join of Category of realizations of hopf algebras over Rational Field and
 Category of graded algebras over Rational Field]
```

class sage.combinat.ncsf_qsym.generic_basis_code.**GradedModulesWithInternalProduct**(*base*, *name=None*)

Bases: sage.categories.category_types.Category_over_base_ring

Constructs the class of modules with internal product. This is used to give an internal product structure to the non-commutative symmetric functions.

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.generic_basis_code import GradedModulesWithInternalProduct
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: R in GradedModulesWithInternalProduct(QQ)
True
```

class ElementMethods

internal_product(*other*)

Return the internal product of two non-commutative symmetric functions.

The internal product on the algebra of non-commutative symmetric functions is adjoint to the internal coproduct on the algebra of quasisymmetric functions with respect to the duality pairing between these two algebras. This means, explicitly, that any two non-commutative symmetric functions f and g and any quasi-symmetric function h satisfy

$$\langle f * g, h \rangle = \sum_i \langle f, h'_i \rangle \langle g, h''_i \rangle ,$$

where we write $\Delta^{\times}(h)$ as $\sum_i h'_i \otimes h''_i$. Here, $f * g$ denotes the internal product of the non-commutative symmetric functions f and g.

If f and g are two homogeneous elements of $NSym$ having distinct degrees, then the internal product $f * g$ is zero.

Explicit formulas can be given for internal products of elements of the complete and the Psi bases. First, the formula for the Complete basis ([NCSF1] Proposition 5.1): If I and J are two compositions of lengths p and q, respectively, then the corresponding Complete homogeneous non-commutative symmetric functions S^I and S^J have internal product

$$S^I * S^J = \sum S^{\operatorname{comp} M},$$

where the sum ranges over all $p \times q$-matrices $M \in \mathbf{N}^{p \times q}$ (with nonnegative integers as entries) whose row sum vector is I (that is, the sum of the entries of the r-th row is the r-th part of I for all r) and whose column sum vector is J (that is, the sum of all entries of the s-th row is the s-th part of J for all s). Here, for any $M \in \mathbf{N}^{p \times q}$, we denote by $\operatorname{comp} M$ the composition obtained by reading the entries of the matrix M in the usual order (row by row, proceeding left to right in each row, traversing the rows from top to bottom).

The formula on the Psi basis ([NCSF2] Lemma 3.10) is more complicated. Let I and J be two compositions of lengths p and q, respectively, having the same size $|I| = |J|$. We denote by Ψ^K the element of the Psi basis corresponding to any composition K.

- If $p > q$, then $\Psi^I * \Psi^J$ is plainly 0.
- Assume that $p = q$. Let $\widetilde{\delta}_{I,J}$ denote the integer 1 if the compositions I and J are permutations of each other, and the integer 0 otherwise. For every positive integer i, let m_i denote the number of parts of I equal to i. Then, $\Psi^I * \Psi^J$ equals $\widetilde{\delta}_{I,J} \prod_{i>0} i^{m_i} m_i! \Psi^I$.
- Now assume that $p < q$. Write the composition I as $I = (i_1, i_2, \ldots, i_p)$. For every nonempty composition $K = (k_1, k_2, \ldots, k_s)$, denote by Γ_K the non-commutative symmetric function $k_1[\ldots[[\Psi_{k_1}, \Psi_{k_2}], \Psi_{k_3}], \ldots \Psi_{k_s}]$. For any subset A of $\{1, 2, \ldots, q\}$, let J_A be the composition obtained from J by removing the r-th parts for all $r \notin A$ (while keeping the r-th parts for all $r \in A$ in order). Then, $\Psi^I * \Psi^J$ equals the sum of $\Gamma_{J_{K_1}} \Gamma_{J_{K_2}} \cdots \Gamma_{J_{K_p}}$ over all ordered set partitions $(K_1, K_2, \ldots, K_p)$ of $\{1, 2, \ldots, q\}$ into p parts such that each $1 \leq k \leq p$ satisfies $|J_{K_k}| = i_k$. (See `OrderedSetPartition()` for the meaning of "ordered set partition".)

Aliases for `internal_product()` are `itensor()` and `kronecker_product()`.

INPUT:

- `other` – another non-commutative symmetric function

OUTPUT:

- The result of taking the internal product of `self` with `other`.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: x = S.an_element(); x
2*S[] + 2*S[1] + 3*S[1, 1]
sage: x.internal_product(S[2])
3*S[1, 1]
sage: x.internal_product(S[1])
2*S[1]
sage: S[1,2].internal_product(S[1,2])
S[1, 1, 1] + S[1, 2]
```

Let us check the duality between the inner product and the inner coproduct in degree 4:

```
sage: M = QuasiSymmetricFunctions(FiniteField(29)).M()
sage: S = NonCommutativeSymmetricFunctions(FiniteField(29)).S()
sage: def tensor_incopr(f, g, h):  # computes \sum_i \left< f, h'_i \right> \left< g, h'
....:     result = h.base_ring().zero()
....:     h_parent = h.parent()
....:     for partition_pair, coeff in h.internal_coproduct().monomial_coefficients().it
....:         result += coeff * f.duality_pairing(h_parent[partition_pair[0]]) * g.duali
....:     return result
sage: def testall(n):
....:     return all( all( all( tensor_incopr(S[u], S[v], M[w]) == (S[u].itensor(S[v])).
....:                           for w in Compositions(n) )
....:                     for v in Compositions(n) )
....:                 for u in Compositions(n) )
sage: testall(2)
True
sage: testall(3)  # long time
True
sage: testall(4)  # long time
True
```

The internal product on the algebra of non-commutative symmetric functions commutes with the canonical commutative projection on the symmetric functions:

```
sage: S = NonCommutativeSymmetricFunctions(ZZ).S()
sage: e = SymmetricFunctions(ZZ).e()
sage: def int_pr_of_S_in_e(I, J):
....:     return (S[I].internal_product(S[J])).to_symmetric_function()
sage: all( all( int_pr_of_S_in_e(I, J)
```

```
....:           == S[I].to_symmetric_function().internal_product(S[J].to_symmetric_funct
....:           for I in Compositions(3) )
....:      for J in Compositions(3) )
True
```

itensor(*other*)

Return the internal product of two non-commutative symmetric functions.

The internal product on the algebra of non-commutative symmetric functions is adjoint to the internal coproduct on the algebra of quasisymmetric functions with respect to the duality pairing between these two algebras. This means, explicitly, that any two non-commutative symmetric functions f and g and any quasi-symmetric function h satisfy

$$\langle f * g, h \rangle = \sum_i \langle f, h'_i \rangle \langle g, h''_i \rangle ,$$

where we write $\Delta^{\times}(h)$ as $\sum_i h'_i \otimes h''_i$. Here, $f*g$ denotes the internal product of the non-commutative symmetric functions f and g.

If f and g are two homogeneous elements of $NSym$ having distinct degrees, then the internal product $f * g$ is zero.

Explicit formulas can be given for internal products of elements of the complete and the Psi bases. First, the formula for the Complete basis ([NCSF1] Proposition 5.1): If I and J are two compositions of lengths p and q, respectively, then the corresponding Complete homogeneous non-commutative symmetric functions S^I and S^J have internal product

$$S^I * S^J = \sum S^{\operatorname{comp} M},$$

where the sum ranges over all $p \times q$-matrices $M \in \mathbf{N}^{p \times q}$ (with nonnegative integers as entries) whose row sum vector is I (that is, the sum of the entries of the r-th row is the r-th part of I for all r) and whose column sum vector is J (that is, the sum of all entries of the s-th row is the s-th part of J for all s). Here, for any $M \in \mathbf{N}^{p \times q}$, we denote by $\operatorname{comp} M$ the composition obtained by reading the entries of the matrix M in the usual order (row by row, proceeding left to right in each row, traversing the rows from top to bottom).

The formula on the Psi basis ([NCSF2] Lemma 3.10) is more complicated. Let I and J be two compositions of lengths p and q, respectively, having the same size $|I| = |J|$. We denote by Ψ^K the element of the Psi basis corresponding to any composition K.

- If $p > q$, then $\Psi^I * \Psi^J$ is plainly 0.
- Assume that $p = q$. Let $\widetilde{\delta}_{I,J}$ denote the integer 1 if the compositions I and J are permutations of each other, and the integer 0 otherwise. For every positive integer i, let m_i denote the number of parts of I equal to i. Then, $\Psi^I * \Psi^J$ equals $\widetilde{\delta}_{I,J} \prod_{i>0} i^{m_i} m_i! \Psi^I$.
- Now assume that $p < q$. Write the composition I as $I = (i_1, i_2, \ldots, i_p)$. For every nonempty composition $K = (k_1, k_2, \ldots, k_s)$, denote by Γ_K the non-commutative symmetric function $k_1[\ldots[[\Psi_{k_1}, \Psi_{k_2}], \Psi_{k_3}], \ldots \Psi_{k_s}]$. For any subset A of $\{1, 2, \ldots, q\}$, let J_A be the composition obtained from J by removing the r-th parts for all $r \notin A$ (while keeping the r-th parts for all $r \in A$ in order). Then, $\Psi^I * \Psi^J$ equals the sum of $\Gamma_{J_{K_1}} \Gamma_{J_{K_2}} \cdots \Gamma_{J_{K_p}}$ over all ordered set partitions $(K_1, K_2, \ldots, K_p)$ of $\{1, 2, \ldots, q\}$ into p parts such that each $1 \leq k \leq p$ satisfies $|J_{K_k}| = i_k$. (See `OrderedSetPartition()` for the meaning of "ordered set partition".)

Aliases for `internal_product()` are `itensor()` and `kronecker_product()`.

INPUT:

- `other` – another non-commutative symmetric function

OUTPUT:

- The result of taking the internal product of `self` with `other`.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: x = S.an_element(); x
2*S[] + 2*S[1] + 3*S[1, 1]
sage: x.internal_product(S[2])
3*S[1, 1]
sage: x.internal_product(S[1])
2*S[1]
sage: S[1,2].internal_product(S[1,2])
S[1, 1, 1] + S[1, 2]
```

Let us check the duality between the inner product and the inner coproduct in degree 4:

```
sage: M = QuasiSymmetricFunctions(FiniteField(29)).M()
sage: S = NonCommutativeSymmetricFunctions(FiniteField(29)).S()
sage: def tensor_incopr(f, g, h):  # computes \sum_i \left< f, h'_i \right> \left< g, h'
....:     result = h.base_ring().zero()
....:     h_parent = h.parent()
....:     for partition_pair, coeff in h.internal_coproduct().monomial_coefficients().it
....:         result += coeff * f.duality_pairing(h_parent[partition_pair[0]]) * g.duali
....:     return result
sage: def testall(n):
....:     return all( all( all( tensor_incopr(S[u], S[v], M[w]) == (S[u].itensor(S[v])).
....:                           for w in Compositions(n) )
....:                       for v in Compositions(n) )
....:                   for u in Compositions(n) )
sage: testall(2)
True
sage: testall(3)  # long time
True
sage: testall(4)  # long time
True
```

The internal product on the algebra of non-commutative symmetric functions commutes with the canonical commutative projection on the symmetric functions:

```
sage: S = NonCommutativeSymmetricFunctions(ZZ).S()
sage: e = SymmetricFunctions(ZZ).e()
sage: def int_pr_of_S_in_e(I, J):
....:     return (S[I].internal_product(S[J])).to_symmetric_function()
sage: all( all( int_pr_of_S_in_e(I, J)
....:           == S[I].to_symmetric_function().internal_product(S[J].to_symmetric_funct
....:           for I in Compositions(3) )
....:      for J in Compositions(3) )
True
```

kronecker_product (*other*)

Return the internal product of two non-commutative symmetric functions.

The internal product on the algebra of non-commutative symmetric functions is adjoint to the internal coproduct on the algebra of quasisymmetric functions with respect to the duality pairing between these two algebras. This means, explicitly, that any two non-commutative symmetric functions f and g and any quasi-symmetric function h satisfy

$$\langle f * g, h \rangle = \sum_i \langle f, h'_i \rangle \langle g, h''_i \rangle ,$$

where we write $\Delta^{\times}(h)$ as $\sum_i h'_i \otimes h''_i$. Here, $f * g$ denotes the internal product of the non-commutative symmetric functions f and g.

If f and g are two homogeneous elements of $NSym$ having distinct degrees, then the internal product

$f * g$ is zero.

Explicit formulas can be given for internal products of elements of the complete and the Psi bases. First, the formula for the Complete basis ([NCSF1] Proposition 5.1): If I and J are two compositions of lengths p and q, respectively, then the corresponding Complete homogeneous non-commutative symmetric functions S^I and S^J have internal product

$$S^I * S^J = \sum S^{\operatorname{comp} M},$$

where the sum ranges over all $p \times q$-matrices $M \in \mathbf{N}^{p \times q}$ (with nonnegative integers as entries) whose row sum vector is I (that is, the sum of the entries of the r-th row is the r-th part of I for all r) and whose column sum vector is J (that is, the sum of all entries of the s-th row is the s-th part of J for all s). Here, for any $M \in \mathbf{N}^{p \times q}$, we denote by $\operatorname{comp} M$ the composition obtained by reading the entries of the matrix M in the usual order (row by row, proceeding left to right in each row, traversing the rows from top to bottom).

The formula on the Psi basis ([NCSF2] Lemma 3.10) is more complicated. Let I and J be two compositions of lengths p and q, respectively, having the same size $|I| = |J|$. We denote by Ψ^K the element of the Psi basis corresponding to any composition K.

- If $p > q$, then $\Psi^I * \Psi^J$ is plainly 0.
- Assume that $p = q$. Let $\widetilde{\delta}_{I,J}$ denote the integer 1 if the compositions I and J are permutations of each other, and the integer 0 otherwise. For every positive integer i, let m_i denote the number of parts of I equal to i. Then, $\Psi^I * \Psi^J$ equals $\widetilde{\delta}_{I,J} \prod_{i>0} i^{m_i} m_i! \Psi^I$.
- Now assume that $p < q$. Write the composition I as $I = (i_1, i_2, \ldots, i_p)$. For every nonempty composition $K = (k_1, k_2, \ldots, k_s)$, denote by Γ_K the non-commutative symmetric function $k_1[\ldots[[\Psi_{k_1}, \Psi_{k_2}], \Psi_{k_3}], \ldots \Psi_{k_s}]$. For any subset A of $\{1, 2, \ldots, q\}$, let J_A be the composition obtained from J by removing the r-th parts for all $r \notin A$ (while keeping the r-th parts for all $r \in A$ in order). Then, $\Psi^I * \Psi^J$ equals the sum of $\Gamma_{J_{K_1}} \Gamma_{J_{K_2}} \cdots \Gamma_{J_{K_p}}$ over all ordered set partitions $(K_1, K_2, \ldots, K_p)$ of $\{1, 2, \ldots, q\}$ into p parts such that each $1 \leq k \leq p$ satisfies $|J_{K_k}| = i_k$. (See `OrderedSetPartition()` for the meaning of "ordered set partition".)

Aliases for `internal_product()` are `itensor()` and `kronecker_product()`.

INPUT:

- `other` – another non-commutative symmetric function

OUTPUT:

- The result of taking the internal product of `self` with `other`.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: x = S.an_element(); x
2*S[] + 2*S[1] + 3*S[1, 1]
sage: x.internal_product(S[2])
3*S[1, 1]
sage: x.internal_product(S[1])
2*S[1]
sage: S[1,2].internal_product(S[1,2])
S[1, 1, 1] + S[1, 2]
```

Let us check the duality between the inner product and the inner coproduct in degree 4:

```
sage: M = QuasiSymmetricFunctions(FiniteField(29)).M()
sage: S = NonCommutativeSymmetricFunctions(FiniteField(29)).S()
sage: def tensor_incopr(f, g, h):  # computes \sum_i \left< f, h'_i \right> \left< g, h'
....:     result = h.base_ring().zero()
....:     h_parent = h.parent()
....:     for partition_pair, coeff in h.internal_coproduct().monomial_coefficients().it
....:         result += coeff * f.duality_pairing(h_parent[partition_pair[0]]) * g.duali
....:     return result
```

```
sage: def testall(n):
....:     return all( all( all( tensor_incopr(S[u], S[v], M[w]) == (S[u].itensor(S[v])).
....:                           for w in Compositions(n) )
....:                       for v in Compositions(n) )
....:                   for u in Compositions(n) )
sage: testall(2)
True
sage: testall(3)  # long time
True
sage: testall(4)  # long time
True
```

The internal product on the algebra of non-commutative symmetric functions commutes with the canonical commutative projection on the symmetric functions:

```
sage: S = NonCommutativeSymmetricFunctions(ZZ).S()
sage: e = SymmetricFunctions(ZZ).e()
sage: def int_pr_of_S_in_e(I, J):
....:     return (S[I].internal_product(S[J])).to_symmetric_function()
sage: all( all( int_pr_of_S_in_e(I, J)
....:           == S[I].to_symmetric_function().internal_product(S[J].to_symmetric_funct
....:           for I in Compositions(3) )
....:       for J in Compositions(3) )
True
```

class GradedModulesWithInternalProduct.**ParentMethods**

internal_product()

The bilinear product inherited from the isomorphism with the descent algebra.

This is constructed by extending the method internal_product_on_basis() bilinearly, if available, or using the method internal_product_by_coercion().

OUTPUT:

•The internal product map of the algebra the non-commutative symmetric functions.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: S.internal_product
Generic endomorphism of Non-Commutative Symmetric Functions over the Rational Field in t
sage: S.internal_product(S[2,2], S[1,2,1])
2*S[1, 1, 1, 1] + S[1, 1, 2] + S[2, 1, 1]
sage: S.internal_product(S[2,2], S[1,2])
0
```

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: R.internal_product
<bound method ....internal_product_by_coercion ...>
sage: R.internal_product_by_coercion(R[1, 1], R[1,1])
R[2]
sage: R.internal_product(R[2,2], R[1,2])
0
```

internal_product_on_basis(*I*, *J*)

The internal product of the two basis elements indexed by I and J (optional)

INPUT:

•I, J – compositions indexing two elements of the basis of self

Returns the internal product of the corresponding basis elements. If this method is implemented, the internal product is defined from it by linearity.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: S.internal_product_on_basis([2,2], [1,2,1])
2*S[1, 1, 1, 1] + S[1, 1, 2] + S[2, 1, 1]
sage: S.internal_product_on_basis([2,2], [2,1])
0
```

itensor()

The bilinear product inherited from the isomorphism with the descent algebra.

This is constructed by extending the method internal_product_on_basis() bilinearly, if available, or using the method internal_product_by_coercion().

OUTPUT:

•The internal product map of the algebra the non-commutative symmetric functions.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: S.internal_product
Generic endomorphism of Non-Commutative Symmetric Functions over the Rational Field in t
sage: S.internal_product(S[2,2], S[1,2,1])
2*S[1, 1, 1, 1] + S[1, 1, 2] + S[2, 1, 1]
sage: S.internal_product(S[2,2], S[1,2])
0
```

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: R.internal_product
<bound method ....internal_product_by_coercion ...>
sage: R.internal_product_by_coercion(R[1, 1], R[1,1])
R[2]
sage: R.internal_product(R[2,2], R[1,2])
0
```

kronecker_product()

The bilinear product inherited from the isomorphism with the descent algebra.

This is constructed by extending the method internal_product_on_basis() bilinearly, if available, or using the method internal_product_by_coercion().

OUTPUT:

•The internal product map of the algebra the non-commutative symmetric functions.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: S.internal_product
Generic endomorphism of Non-Commutative Symmetric Functions over the Rational Field in t
sage: S.internal_product(S[2,2], S[1,2,1])
2*S[1, 1, 1, 1] + S[1, 1, 2] + S[2, 1, 1]
sage: S.internal_product(S[2,2], S[1,2])
0
```

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: R.internal_product
<bound method ....internal_product_by_coercion ...>
sage: R.internal_product_by_coercion(R[1, 1], R[1,1])
```

```
R[2]
sage: R.internal_product(R[2,2], R[1,2])
0
```

class GradedModulesWithInternalProduct.**Realizations**(*category*, **args*)

Bases: sage.categories.realizations.RealizationsCategory

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class ParentMethods

internal_product_by_coercion(*left*, *right*)

Internal product of `left` and `right`.

This is a default implementation that computes the internal product in the realization specified by `self.realization_of().a_realization()`.

INPUT:

- `left` – an element of the non-commutative symmetric functions
- `right` – an element of the non-commutative symmetric functions

OUTPUT:

- The internal product of `left` and `right`.

EXAMPLES:

```
sage: S=NonCommutativeSymmetricFunctions(QQ).S()
sage: S.internal_product_by_coercion(S[2,1], S[3])
S[2, 1]
sage: S.internal_product_by_coercion(S[2,1], S[4])
0
```

GradedModulesWithInternalProduct.**super_categories**()

EXAMPLES:

```
sage: from sage.combinat.ncsf_qsym.generic_basis_code import GradedModulesWithInternalProduc
sage: GradedModulesWithInternalProduct(ZZ).super_categories()
[Category of graded modules over Integer Ring]
```

5.1.129 Non-Commutative Symmetric Functions

class sage.combinat.ncsf_qsym.ncsf.**NonCommutativeSymmetricFunctions**(*R*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The abstract algebra of non-commutative symmetric functions.

We construct the abstract algebra of non-commutative symmetric functions over the rational numbers:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: NCSF
Non-Commutative Symmetric Functions over the Rational Field
sage: S = NCSF.complete()
sage: R = NCSF.ribbon()
sage: S[2,1]*R[1,2]
S[2, 1, 1, 2] - S[2, 1, 3]
```

NCSF is the unique free (non-commutative!) graded connected algebra with one generator in each degree:

```
sage: NCSF.category()
Join of Category of hopf algebras over Rational Field
    and Category of graded algebras over Rational Field
    and Category of monoids with realizations
    and Category of coalgebras over Rational Field with realizations

sage: [S[i].degree() for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We use the Sage standard renaming idiom to get shorter outputs:

```
sage: NCSF.rename("NCSF")
sage: NCSF
NCSF
```

NCSF has many representations as a concrete algebra. Each of them has a distinguished basis, and its elements are expanded in this basis. Here is the Ψ (`Psi`) representation:

```
sage: Psi = NCSF.Psi()
sage: Psi
NCSF in the Psi basis
```

Elements of `Psi` are linear combinations of basis elements indexed by compositions:

```
sage: Psi.an_element()
2*Psi[] + 2*Psi[1] + 3*Psi[1, 1]
```

The basis itself is accessible through:

```
sage: Psi.basis()
Lazy family (Term map from Compositions of non-negative integers...
sage: Psi.basis().keys()
Compositions of non-negative integers
```

To construct an element one can therefore do:

```
sage: Psi.basis()[Composition([2,1,3])]
Psi[2, 1, 3]
```

As this is rather cumbersome, the following abuses of notation are allowed:

```
sage: Psi[Composition([2, 1, 3])]
Psi[2, 1, 3]
sage: Psi[[2, 1, 3]]
Psi[2, 1, 3]
sage: Psi[2, 1, 3]
Psi[2, 1, 3]
```

or even:

```
sage: Psi[(i for i in [2, 1, 3])]
Psi[2, 1, 3]
```

Unfortunately, due to a limitation in Python syntax, one cannot use:

```
sage: Psi[]       # not implemented
```

Instead, you can use:

```
sage: Psi[[]]
Psi[]
```

Now, we can construct linear combinations of basis elements:

```
sage: Psi[2,1,3] + 2 * (Psi[4] + Psi[2,1])
2*Psi[2, 1] + Psi[2, 1, 3] + 2*Psi[4]
```

Algebra structure

To start with, `Psi` is a graded algebra, the grading being induced by the size of compositions. The one is the basis element indexed by the empty composition:

```
sage: Psi.one()
Psi[]
sage: S.one()
S[]
sage: R.one()
R[]
```

As we have seen above, the `Psi` basis is multiplicative; that is multiplication is induced by linearity from the concatenation of compositions:

```
sage: Psi[1,3] * Psi[2,1]
Psi[1, 3, 2, 1]
sage: (Psi.one() + 2 * Psi[1,3]) * Psi[2, 4]
2*Psi[1, 3, 2, 4] + Psi[2, 4]
```

Hopf algebra structure

`Psi` is further endowed with a coalgebra structure. The coproduct is an algebra morphism, and therefore determined by its values on the generators; those are primitive:

```
sage: Psi[1].coproduct()
Psi[] # Psi[1] + Psi[1] # Psi[]
sage: Psi[2].coproduct()
Psi[] # Psi[2] + Psi[2] # Psi[]
```

The coproduct, being cocommutative on the generators, is cocommutative everywhere:

```
sage: Psi[1,2].coproduct()
Psi[] # Psi[1, 2] + Psi[1] # Psi[2] + Psi[1, 2] # Psi[] + Psi[2] # Psi[1]
```

The algebra and coalgebra structures on `Psi` combine to form a bialgebra structure, which cooperates with the grading to form a connected graded bialgebra. Thus, as any connected graded bialgebra, `Psi` is a Hopf algebra. Over `QQ` (or any other $\mathbf{Q}$-algebra), this Hopf algebra `Psi` is isomorphic to the tensor algebra of its space of primitive elements.

The antipode is an anti-algebra morphism; in the `Psi` basis, it sends the generators to their opposites and changes their sign if they are of odd degree:

```
sage: Psi[3].antipode()
-Psi[3]
sage: Psi[1,3,2].antipode()
-Psi[2, 3, 1]
sage: Psi[1,3,2].coproduct().apply_multilinear_morphism(lambda be,ga: Psi(be)*Psi(ga).antipode()
0
```

The counit is defined by sending all elements of positive degree to zero:

```
sage: S[3].degree(), S[3,1,2].degree(), S.one().degree()
(3, 6, 0)
sage: S[3].counit()
0
sage: S[3,1,2].counit()
0
sage: S.one().counit()
1
sage: (S[3] - 2*S[3,1,2] + 7).counit()
7
sage: (R[3] - 2*R[3,1,2] + 7).counit()
7
```

It is possible to change the prefix used to display the basis elements using the method `print_options()`. Say that for instance one wanted to display the `Complete` basis as having a prefix `H` instead of the default `S`:

```
sage: H = NCSF.complete()
sage: H.an_element()
2*S[] + 2*S[1] + 3*S[1, 1]
sage: H.print_options(prefix='H')
sage: H.an_element()
2*H[] + 2*H[1] + 3*H[1, 1]
sage: H.print_options(prefix='S') #restore to 'S'
```

Concrete representations

NCSF admits the concrete realizations defined in [NCSF1]:

```
sage: Phi        = NCSF.Phi()
sage: Psi        = NCSF.Psi()
sage: ribbon     = NCSF.ribbon()
sage: complete   = NCSF.complete()
sage: elementary = NCSF.elementary()
```

To change from one basis to another, one simply does:

```
sage: Phi(Psi[1])
Phi[1]
sage: Phi(Psi[3])
-1/4*Phi[1, 2] + 1/4*Phi[2, 1] + Phi[3]
```

In general, one can mix up different bases in computations:

```
sage: Phi[1] * Psi[1]
Phi[1, 1]
```

Some of the changes of basis are easy to guess:

```
sage: ribbon(complete[1,3,2])
R[1, 3, 2] + R[1, 5] + R[4, 2] + R[6]
```

This is the sum of all fatter compositions. Using the usual Möbius function for the boolean lattice, the inverse change of basis is given by the alternating sum of all fatter compositions:

```
sage: complete(ribbon[1,3,2])
S[1, 3, 2] - S[1, 5] - S[4, 2] + S[6]
```

The analogue of the elementary basis is the sum over all finer compositions than the 'complement' of the composition in the ribbon basis:

```
sage: Composition([1,3,2]).complement()
[2, 1, 2, 1]
sage: ribbon(elementary([1,3,2]))
R[1, 1, 1, 1, 1, 1] + R[1, 1, 1, 2, 1] + R[2, 1, 1, 1, 1] + R[2, 1, 2, 1]
```

By Möbius inversion on the composition poset, the ribbon basis element corresponding to a composition I is then the alternating sum over all compositions fatter than the complement composition of I in the elementary basis:

```
sage: elementary(ribbon[2,1,2,1])
L[1, 3, 2] - L[1, 5] - L[4, 2] + L[6]
```

The Φ (`Phi`) and Ψ bases are computed by changing to and from the `Complete` basis. The expansion of Ψ basis is given in Proposition 4.5 of [NCSF1] by the formulae

$$S^I = \sum_{J \geq I} \frac{1}{\pi_u(J, I)} \Psi^J$$

and

$$\Psi^I = \sum_{J \geq I} (-1)^{\ell(J)-\ell(I)} lp(J, I) S^J$$

where the coefficients $\pi_u(J, I)$ and $lp(J, I)$ are coefficients in the methods `coeff_pi()` and `coeff_lp()` respectively. For example:

```
sage: Psi(complete[3])
1/6*Psi[1, 1, 1] + 1/3*Psi[1, 2] + 1/6*Psi[2, 1] + 1/3*Psi[3]
sage: complete(Psi[3])
S[1, 1, 1] - 2*S[1, 2] - S[2, 1] + 3*S[3]
```

The `Phi` basis is another analogue of the power sum basis from the algebra of symmetric functions and the expansion in the Complete basis is given in Proposition 4.9 of [NCSF1] by the formulae

$$S^I = \sum_{J \geq I} \frac{1}{sp(J, I)} \Phi^J$$

and

$$\Phi^I = \sum_{J \geq I} (-1)^{\ell(J)-\ell(I)} \frac{\prod_i I_i}{\ell(J, I)} S^J$$

where the coefficients $sp(J, I)$ and $\ell(J, I)$ are coefficients in the methods `coeff_sp()` and `coeff_ell()` respectively. For example:

```
sage: Phi(complete[3])
1/6*Phi[1, 1, 1] + 1/4*Phi[1, 2] + 1/4*Phi[2, 1] + 1/3*Phi[3]
sage: complete(Phi[3])
S[1, 1, 1] - 3/2*S[1, 2] - 3/2*S[2, 1] + 3*S[3]
```

Here is how to fetch the conversion morphisms:

```
sage: f = complete.coerce_map_from(elementary); f
Generic morphism:
  From: NCSF in the Elementary basis
  To:   NCSF in the Complete basis
sage: g = elementary.coerce_map_from(complete); g
Generic morphism:
  From: NCSF in the Complete basis
  To:   NCSF in the Elementary basis
sage: f.category()
Category of homsets of unital magmas and right modules over Rational Field and
  left modules over Rational Field
sage: f(elementary[1,2,2])
S[1, 1, 1, 1, 1] - S[1, 1, 1, 2] - S[1, 2, 1, 1] + S[1, 2, 2]
sage: g(complete[1,2,2])
L[1, 1, 1, 1, 1] - L[1, 1, 1, 2] - L[1, 2, 1, 1] + L[1, 2, 2]
sage: h = f*g; h
Composite map:
  From: NCSF in the Complete basis
  To:   NCSF in the Complete basis
  Defn:   Generic morphism:
          From: NCSF in the Complete basis
          To:   NCSF in the Elementary basis
        then
          Generic morphism:
          From: NCSF in the Elementary basis
          To:   NCSF in the Complete basis
sage: h(complete[1,3,2])
S[1, 3, 2]
```

Additional concrete representations

NCSF has some additional bases which appear in the literature:

```
sage: Monomial                 = NCSF.Monomial()
sage: Immaculate               = NCSF.Immaculate()
sage: dualQuasisymmetric_Schur = NCSF.dualQuasisymmetric_Schur()
```

The `Monomial` basis was introduced in [Tev2007] and the `Immaculate` basis was introduced in [BBSSZ2012]. The `Quasisymmetric_Schur` were defined in [QSCHUR] and the dual basis is implemented here as `dualQuasisymmetric_Schur`. Refer to the documentation for the use and definition of these bases.

Todo

•implement fundamental, forgotten, and simple (coming from the simple modules of HS_n) bases.

We revert back to the original name from our custom short name NCSF:

```
sage: NCSF
NCSF
```

```
sage: NCSF.rename()
sage: NCSF
Non-Commutative Symmetric Functions over the Rational Field
```

TESTS:

```
sage: TestSuite(Phi).run()
sage: TestSuite(Psi).run()
sage: TestSuite(complete).run()
```

class Bases (*parent_with_realization*)

Bases: `sage.categories.realizations.Category_realization_of_parent`

Category of bases of non-commutative symmetric functions.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: N.Bases()
Category of bases of Non-Commutative Symmetric Functions over the Rational Field
sage: R = N.Ribbon()
sage: R in N.Bases()
True
```

class ElementMethods

bernstein_creation_operator(*n*)

Return the image of `self` under the n-th Bernstein creation operator.

Let n be an integer. The n-th Bernstein creation operator $\mathbb{B}_n$ is defined as the endomorphism of the space $NSym$ of noncommutative symmetric functions which sends every f to

$$\sum_{i \geq 0} (-1)^i H_{n+i} F_{1^i}^{\perp},$$

where usual notations are in place (the letter H stands for the complete basis of $NSym$, the letter F stands for the fundamental basis of the algebra $QSym$ of quasisymmetric functions, and $F_{1^i}^{\perp}$ means skewing (`skew_by()`) by F_{1^i}). Notice that F_{1^i} is nothing other than the elementary symmetric function e_i.

This has been introduced in [BBSSZ2012], section 3.1, in analogy to the Bernstein creation operators on the symmetric functions (`bernstein_creation_operator()`), and studied further in [BBSSZ2012], mainly in the context of immaculate functions (`Immaculate`). In fact, if $(\alpha_1, \alpha_2, \ldots, \alpha_m)$ is an m-tuple of integers, then

$$\mathbb{B}_n I_{(\alpha_1, \alpha_2, \ldots, \alpha_m)} = I_{(n, \alpha_1, \alpha_2, \ldots, \alpha_m)},$$

where $I_{(\alpha_1, \alpha_2, \ldots, \alpha_m)}$ is the immaculate function associated to the m-tuple $(\alpha_1, \alpha_2, \ldots, \alpha_m)$ (see `immaculate_function()`).

EXAMPLES:

We get the immaculate functions by repeated application of Bernstein creation operators:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: I = NSym.I()
sage: S = NSym.S()
sage: def immaculate_by_bernstein(xs):
....:     # immaculate function corresponding to integer
....:     # tuple ``xs``, computed by iterated application
```

```
....:     # of Bernstein creation operators.
....:     res = S.one()
....:     for i in reversed(xs):
....:         res = res.bernstein_creation_operator(i)
....:     return res
sage: import itertools
sage: all( immaculate_by_bernstein(p) == I.immaculate_function(p)
....:      for p in itertools.product(range(-1, 3), repeat=3))
True
```

Some examples:

```
sage: S[3,2].bernstein_creation_operator(-2)
S[2, 1]
sage: S[3,2].bernstein_creation_operator(-1)
S[1, 2, 1] - S[2, 2] - S[3, 1]
sage: S[3,2].bernstein_creation_operator(0)
-S[1, 2, 2] - S[1, 3, 1] + S[2, 2, 1] + S[3, 2]
sage: S[3,2].bernstein_creation_operator(1)
S[1, 3, 2] - S[2, 2, 2] - S[2, 3, 1] + S[3, 2, 1]
sage: S[3,2].bernstein_creation_operator(2)
S[2, 3, 2] - S[3, 2, 2] - S[3, 3, 1] + S[4, 2, 1]
```

chi()

Return the commutative image of a non-commutative symmetric function.

OUTPUT:

•The commutative image of `self`. This will be a symmetric function.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: x = R.an_element(); x
2*R[] + 2*R[1] + 3*R[1, 1]
sage: x.to_symmetric_function()
2*s[] + 2*s[1] + 3*s[1, 1]
sage: y = N.Phi()[1,3]
sage: y.to_symmetric_function()
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 3*h[3, 1]
```

expand(*n*, *alphabet='x'*)

Expand the noncommutative symmetric function into an element of a free algebra in `n` indeterminates of an alphabet, which by default is `'x'`.

INPUT:

•`n` – a nonnegative integer; the number of variables in the expansion

•`alphabet` – (default: `'x'`); the alphabet in which `self` is to be expanded

OUTPUT:

•An expansion of `self` into the `n` variables specified by `alphabet`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: S = NSym.S()
sage: S[3].expand(3)
x0^3 + x0^2*x1 + x0^2*x2 + x0*x1^2 + x0*x1*x2
 + x0*x2^2 + x1^3 + x1^2*x2 + x1*x2^2 + x2^3
sage: L = NSym.L()
sage: L[3].expand(3)
x2*x1*x0
sage: L[2].expand(3)
x1*x0 + x2*x0 + x2*x1
```

```
sage: L[3].expand(4)
x2*x1*x0 + x3*x1*x0 + x3*x2*x0 + x3*x2*x1
sage: Psi = NSym.Psi()
sage: Psi[2, 1].expand(3)
x0^3 + x0^2*x1 + x0^2*x2 + x0*x1*x0 + x0*x1^2 + x0*x1*x2
 + x0*x2*x0 + x0*x2*x1 + x0*x2^2 - x1*x0^2 - x1*x0*x1
 - x1*x0*x2 + x1^2*x0 + x1^3 + x1^2*x2 + x1*x2*x0
 + x1*x2*x1 + x1*x2^2 - x2*x0^2 - x2*x0*x1 - x2*x0*x2
 - x2*x1*x0 - x2*x1^2 - x2*x1*x2 + x2^2*x0 + x2^2*x1 + x2^3
```

One can use a different set of variables by adding an optional argument `alphabet=...`:

```
sage: L[3].expand(4, alphabet="y")
y2*y1*y0 + y3*y1*y0 + y3*y2*y0 + y3*y2*y1
```

TESTS:

```
sage: (3*S([])).expand(2)
3
sage: L[4,2].expand(0)
0
sage: S([]).expand(0)
1
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: S[3].expand(3)
x0^3 + x0^2*x1 + x0^2*x2 + x0*x1^2 + x0*x1*x2
 + x0*x2^2 + x1^3 + x1^2*x2 + x1*x2^2 + x2^3
```

Todo

So far this is only implemented on the elementary basis, and everything else goes through coercion. Maybe it is worth shortcutting some of the other bases?

---

left_padded_kronecker_product(*x*)

Return the left-padded Kronecker product of `self` and `x` in the basis of `self`.

The left-padded Kronecker product is a bilinear map mapping two non-commutative symmetric functions to another, not necessarily preserving degree. It can be defined as follows: Let $*$ denote the internal product (`internal_product()`) on the space of non-commutative symmetric functions. For any composition I, let S^I denote the complete homogeneous symmetric function indexed by I. For any compositions α, β, γ, let $g^{\gamma}_{\alpha,\beta}$ denote the coefficient of S^γ in the internal product $S^\alpha * S^\beta$. For every composition $I = (i_1, i_2, \ldots, i_k)$ and every integer $n > |I|$, define the *'n'-completion of 'I'* to be the composition $(n - |I|, i_1, i_2, \ldots, i_k)$; this n-completion is denoted by $I[n]$. Then, for any compositions α and β and every integer $n > |\alpha| + |\beta|$, we can write the internal product $S^{\alpha[n]} * S^{\beta[n]}$ in the form

$$S^{\alpha[n]} * S^{\beta[n]} = \sum_\gamma g^{\gamma[n]}_{\alpha[n],\beta[n]} S^{\gamma[n]}$$

with γ ranging over all compositions. The coefficients $g^{\gamma[n]}_{\alpha[n],\beta[n]}$ are independent on n. These coefficients $g^{\gamma[n]}_{\alpha[n],\beta[n]}$ are denoted by $\widetilde{g}^{\gamma}_{\alpha,\beta}$, and the non-commutative symmetric function

$$\sum_\gamma \widetilde{g}^{\gamma}_{\alpha,\beta} S^\gamma$$

is said to be the *left-padded Kronecker product* of S^α and S^β. By bilinearity, this extends to a definition of a left-padded Kronecker product of any two non-commutative symmetric functions.

The left-padded Kronecker product on the non-commutative symmetric functions lifts the left-padded Kronecker product on the symmetric functions. More precisely: Let π denote the canonical projection (`to_symmetric_function()`) from the non-commutative symmetric functions to the symmetric functions. Then, any two non-commutative symmetric functions f and g satisfy

$$\pi(f \overline{*} g) = \pi(f) \overline{*} \pi(g),$$

where the $\overline{*}$ on the left-hand side denotes the left-padded Kronecker product on the non-commutative symmetric functions, and the $\overline{*}$ on the right-hand side denotes the left-padded Kronecker product on the symmetric functions.

INPUT:

- `x` – element of the ring of non-commutative symmetric functions over the same base ring as `self`

OUTPUT:

- the left-padded Kronecker product of `self` with `x` (an element of the ring of non-commutative symmetric functions in the same basis as `self`)

AUTHORS:

- Darij Grinberg (15 Mar 2014)

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: S = NSym.S()
sage: S[2,1].left_padded_kronecker_product(S[3])
S[1, 1, 1, 1] + S[1, 2, 1] + S[2, 1] + S[2, 1, 1, 1] + S[2, 2, 1] + S[3, 2, 1]
sage: S[2,1].left_padded_kronecker_product(S[1])
S[1, 1, 1] + S[1, 2, 1] + S[2, 1]
sage: S[1].left_padded_kronecker_product(S[2,1])
S[1, 1, 1] + S[2, 1] + S[2, 1, 1]
sage: S[1,1].left_padded_kronecker_product(S[2])
S[1, 1] + 2*S[1, 1, 1] + S[2, 1, 1]
sage: S[1].left_padded_kronecker_product(S[1,2,1])
S[1, 1, 1, 1] + S[1, 2, 1] + S[1, 2, 1, 1] + S[2, 1, 1]
sage: S[2].left_padded_kronecker_product(S[3])
S[1, 2] + S[2, 1, 1] + S[3, 2]
```

Taking the left-padded Kronecker product with $1 = S$ is the identity map on the ring of non-commutative symmetric functions:

```
sage: all( S[Composition([])].left_padded_kronecker_product(S[lam])
....:      == S[lam].left_padded_kronecker_product(S[Composition([])])
....:      == S[lam] for i in range(4)
....:      for lam in Compositions(i) )
True
```

Here is a rule for the left-padded Kronecker product of S_1 (this is the same as $S^{(1)}$) with any complete homogeneous function: Let I be a composition. Then, the left-padded Kronecker product of S_1 and S^I is $\sum_K a_K S^K$, where the sum runs over all compositions K, and the coefficient a_K is defined as the number of ways to obtain K from I by one of the following two operations:

- Insert a 1 at the end of I.
- Subtract 1 from one of the entries of I (and remove the entry if it thus becomes 0), and insert a 1 at the end of I.

We check this for compositions of size ≤ 4:

```
sage: def mults1(I):
....:     # Left left-padded Kronecker multiplication by S[1].
....:     res = S[I[:] + [1]]
....:     for k in range(len(I)):
....:         I2 = I[:]
```

```
....:             if I2[k] == 1:
....:                 I2 = I2[:k] + I2[k+1:]
....:             else:
....:                 I2[k] -= 1
....:             res += S[I2 + [1]]
....:         return res
sage: all( mults1(I) == S[1].left_padded_kronecker_product(S[I])
....:       for i in range(5) for I in Compositions(i) )
True
```

A similar rule can be made for the left-padded Kronecker product of any complete homogeneous function with S_1: Let I be a composition. Then, the left-padded Kronecker product of S^I and S_1 is $\sum_K b_K S^K$, where the sum runs over all compositions K, and the coefficient b_K is defined as the number of ways to obtain K from I by one of the following two operations:

- Insert a 1 at the front of I.
- Subtract 1 from one of the entries of I (and remove the entry if it thus becomes 0), and insert a 1 right after this entry.

We check this for compositions of size ≤ 4:

```
sage: def mults2(I):
....:     # Left left-padded Kronecker multiplication by S[1].
....:     res = S[[1] + I[:]]
....:     for k in range(len(I)):
....:         I2 = I[:]
....:         i2k = I2[k]
....:         if i2k != 1:
....:             I2 = I2[:k] + [i2k-1, 1] + I2[k+1:]
....:         res += S[I2]
....:     return res
sage: all( mults2(I) == S[I].left_padded_kronecker_product(S[1])
....:       for i in range(5) for I in Compositions(i) )
True
```

Checking the $\pi(f \overline{*} g) = \pi(f) \overline{*} \pi(g)$ equality:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: R = NSym.R()
sage: def testpi(n):
....:     for I in Compositions(n):
....:         for J in Compositions(n):
....:             a = R[I].to_symmetric_function()
....:             b = R[J].to_symmetric_function()
....:             x = a.left_padded_kronecker_product(b)
....:             y = R[I].left_padded_kronecker_product(R[J])
....:             y = y.to_symmetric_function()
....:             if x != y:
....:                 return False
....:     return True
sage: testpi(3)
True
```

TESTS:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: (2*S([])).left_padded_kronecker_product(3*S([]))
6*S[]
```

Different bases and base rings:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
```

```
sage: L = NSym.L()
sage: L(S[2].left_padded_kronecker_product(L[2]))
L[1, 1, 1] + L[2] + L[2, 1, 1] - L[2, 2]
sage: S(L[2].left_padded_kronecker_product(S[1,1]))
S[1, 1] + 2*S[1, 1, 1] + S[1, 1, 1, 1] - S[1, 1, 2]

sage: NSym = NonCommutativeSymmetricFunctions(CyclotomicField(12))
sage: S = NSym.S()
sage: L = NSym.L()
sage: v = L[2].left_padded_kronecker_product(L[2]); v
L[1, 1] + L[1, 1, 1] + (-1)*L[2] + L[2, 2]
sage: parent(v)
Non-Commutative Symmetric Functions over the Cyclotomic Field of order 12 and degree

sage: NSym = NonCommutativeSymmetricFunctions(Zmod(14))
sage: S = NSym.S()
sage: L = NSym.L()
sage: v = L[2].left_padded_kronecker_product(L[2]); v
L[1, 1] + L[1, 1, 1] + 13*L[2] + L[2, 2]
sage: parent(v)
Non-Commutative Symmetric Functions over the Ring of integers modulo 14 in the Elemen

sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: R = NSym.R()
sage: v = R[1].left_padded_kronecker_product(R[1]); parent(v)
Non-Commutative Symmetric Functions over the Integer Ring in the Ribbon basis
```

omega_involution()

Return the image of the noncommutative symmetric function `self` under the omega involution.

The omega involution is defined as the algebra antihomomorphism $NCSF \to NCSF$ which, for every positive integer n, sends the n-th complete non-commutative symmetric function S_n to the n-th elementary non-commutative symmetric function Λ_n. This omega involution is denoted by ω. It can be shown that every composition I satisfies

$$\omega(S^I) = \Lambda^{I^r}, \quad \omega(\Lambda^I) = S^{I^r}, \quad \omega(R_I) = R_{I^t}, \quad \omega(\Phi^I) = (-1)^{|I|-\ell(I)}\Phi^{I^r}, \omega(\Psi^I) = (-1)^{|I|-\ell(I)}\Psi^{I^r},$$

where I^r denotes the reversed composition of I, and I^t denotes the conjugate composition of I, and $\ell(I)$ denotes the length of the composition I, and standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, R for the ribbon basis, Φ for that of the power-sums of the second kind, and Ψ for that of the power-sums of the first kind). More generally, if f is a homogeneous element of $NCSF$ of degree n, then

$$\omega(f) = (-1)^n S(f),$$

where S denotes the antipode of $NCSF$.

The omega involution ω is an involution and a coalgebra automorphism of $NCSF$. It is an automorphism of the graded vector space $NCSF$. If π denotes the projection from $NCSF$ to the ring of symmetric functions (`to_symmetric_function()`), then $\pi(\omega(f)) = \omega(\pi(f))$ for every $f \in NCSF$, where the ω on the right hand side denotes the omega automorphism of Sym.

The omega involution on $NCSF$ is adjoint to the omega involution on $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The omega involution has been denoted by ω in [LMvW13], section 3.6. See [NCSF1], section 3.1 for the properties of this map.

See also:

omega involution of QSym, psi involution of NCSF, star involution of NCSF.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: L = NSym.L()
sage: L(S[3,2].omega_involution())
L[2, 3]
sage: L(S[6,3].omega_involution())
L[3, 6]
sage: L(S[1,3].omega_involution())
L[3, 1]
sage: L((S[9,1] - S[8,2] + 2*S[6,4] - 3*S[3] + 4*S[[]]).omega_involution()) # long ti
4*L[] + L[1, 9] - L[2, 8] - 3*L[3] + 2*L[4, 6]
sage: L((S[3,3] - 2*S[2]).omega_involution())
-2*L[2] + L[3, 3]
sage: L(S([4,2]).omega_involution())
L[2, 4]
sage: R = NSym.R()
sage: R([4,2]).omega_involution()
R[1, 2, 1, 1, 1]
sage: R.zero().omega_involution()
0
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: Phi = NSym.Phi()
sage: Phi([2,1]).omega_involution()
-Phi[1, 2]
sage: Psi = NSym.Psi()
sage: Psi([2,1]).omega_involution()
-Psi[1, 2]
sage: Psi([3,1]).omega_involution()
Psi[1, 3]
```

Testing the $\pi(\omega(f)) = \omega(\pi(f))$ relation noticed above:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: R = NSym.R()
sage: all( R(I).omega_involution().to_symmetric_function()
....:      == R(I).to_symmetric_function().omega_involution()
....:      for I in Compositions(4) )
True
```

The omega involution on $QSym$ is adjoint to the omega involution on $NSym$ with respect to the duality pairing:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.M()
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: S = NSym.S()
sage: all( all( M(I).omega_involution().duality_pairing(S(J))
....:           == M(I).duality_pairing(S(J).omega_involution())
....:           for I in Compositions(2) )
....:      for J in Compositions(3) )
True
```

psi_involution()

Return the image of the noncommutative symmetric function `self` under the involution ψ.

The involution ψ is defined as the linear map $NCSF \to NCSF$ which, for every composition I, sends the complete noncommutative symmetric function S^I to the elementary noncommutative

symmetric function Λ^I. It can be shown that every composition I satisfies

$$\psi(R_I) = R_{I^c}, \quad \psi(S^I) = \Lambda^I, \quad \psi(\Lambda^I) = S^I, \quad \psi(\Phi^I) = (-1)^{|I|-\ell(I)}\Phi^I$$

where I^c denotes the complement of the composition I, and $\ell(I)$ denotes the length of I, and where standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, Φ for the basis of the power sums of the second kind, and R for the ribbon basis). The map ψ is an involution and a graded Hopf algebra automorphism of $NCSF$. If π denotes the projection from $NCSF$ to the ring of symmetric functions (`to_symmetric_function()`), then $\pi(\psi(f)) = \omega(\pi(f))$ for every $f \in NCSF$, where the ω on the right hand side denotes the omega automorphism of Sym.

The involution ψ of $NCSF$ is adjoint to the involution ψ of $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The involution ψ has been denoted by ψ in [LMvW13], section 3.6.

See also:

`psi involution of QSym`, `star involution of NCSF`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: R = NSym.R()
sage: R[3,2].psi_involution()
R[1, 1, 2, 1]
sage: R[6,3].psi_involution()
R[1, 1, 1, 1, 1, 2, 1, 1]
sage: (R[9,1] - R[8,2] + 2*R[2,4] - 3*R[3] + 4*R[[]]).psi_involution()
4*R[] - 3*R[1, 1, 1] + R[1, 1, 1, 1, 1, 1, 1, 1, 2] - R[1, 1, 1, 1, 1, 1, 1, 2, 1] +
sage: (R[3,3] - 2*R[2]).psi_involution()
-2*R[1, 1] + R[1, 1, 2, 1, 1]
sage: R([2,1,1]).psi_involution()
R[1, 3]
sage: S = NSym.S()
sage: S([2,1]).psi_involution()
S[1, 1, 1] - S[2, 1]
sage: S.zero().psi_involution()
0
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: Phi = NSym.Phi()
sage: Phi([2,1]).psi_involution()
-Phi[2, 1]
sage: Phi([3,1]).psi_involution()
Phi[3, 1]
```

The Psi basis doesn't behave as nicely:

```
sage: Psi = NSym.Psi()
sage: Psi([2,1]).psi_involution()
-Psi[2, 1]
sage: Psi([3,1]).psi_involution()
1/2*Psi[1, 2, 1] - 1/2*Psi[2, 1, 1] + Psi[3, 1]
```

The involution ψ commutes with the antipode:

```
sage: all( R(I).psi_involution().antipode()
....:      == R(I).antipode().psi_involution()
....:      for I in Compositions(4) )
True
```

Testing the $\pi(\psi(f)) = \omega(\pi(f))$ relation noticed above:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: R = NSym.R()
sage: all( R(I).psi_involution().to_symmetric_function()
....:      == R(I).to_symmetric_function().omega()
....:      for I in Compositions(4) )
True
```

The involution ψ of $QSym$ is adjoint to the involution ψ of $NSym$ with respect to the duality pairing:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.M()
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: S = NSym.S()
sage: all( all( M(I).psi_involution().duality_pairing(S(J))
....:           == M(I).duality_pairing(S(J).psi_involution())
....:           for I in Compositions(2) )
....:      for J in Compositions(3) )
True
```

star_involution()

Return the image of the noncommutative symmetric function `self` under the star involution.

The star involution is defined as the algebra antihomomorphism $NCSF \to NCSF$ which, for every positive integer n, sends the n-th complete non-commutative symmetric function S_n to S_n. Denoting by f^* the image of an element $f \in NCSF$ under this star involution, it can be shown that every composition I satisfies

$$(S^I)^* = S^{I^r}, \quad (\Lambda^I)^* = \Lambda^{I^r}, \quad R_I^* = R_{I^r}, \quad (\Phi^I)^* = \Phi^{I^r},$$

where I^r denotes the reversed composition of I, and standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, R for the ribbon basis, and Φ for that of the power-sums of the second kind). The star involution is an involution and a coalgebra automorphism of $NCSF$. It is an automorphism of the graded vector space $NCSF$. Under the canonical isomorphism between the n-th graded component of $NCSF$ and the descent algebra of the symmetric group S_n (see `to_descent_algebra()`), the star involution (restricted to the n-th graded component) corresponds to the automorphism of the descent algebra given by $x \mapsto \omega_n x \omega_n$, where ω_n is the permutation $(n, n-1, \ldots, 1) \in S_n$ (written here in one-line notation). If π denotes the projection from $NCSF$ to the ring of symmetric functions (`to_symmetric_function()`), then $\pi(f^*) = \pi(f)$ for every $f \in NCSF$.

The star involution on $NCSF$ is adjoint to the star involution on $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The star involution has been denoted by ρ in [LMvW13], section 3.6. See [NCSF2], section 2.3 for the properties of this map.

See also:

`star involution of QSym`, `psi involution of NCSF`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: S[3,2].star_involution()
S[2, 3]
sage: S[6,3].star_involution()
S[3, 6]
sage: (S[9,1] - S[8,2] + 2*S[6,4] - 3*S[3] + 4*S[[]]).star_involution()
4*S[] + S[1, 9] - S[2, 8] - 3*S[3] + 2*S[4, 6]
```

```
sage: (S[3,3] - 2*S[2]).star_involution()
-2*S[2] + S[3, 3]
sage: S([4,2]).star_involution()
S[2, 4]
sage: R = NSym.R()
sage: R([4,2]).star_involution()
R[2, 4]
sage: R.zero().star_involution()
0
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: Phi = NSym.Phi()
sage: Phi([2,1]).star_involution()
Phi[1, 2]
```

The Psi basis doesn't behave as nicely:

```
sage: Psi = NSym.Psi()
sage: Psi([2,1]).star_involution()
Psi[1, 2]
sage: Psi([3,1]).star_involution()
1/2*Psi[1, 1, 2] - 1/2*Psi[1, 2, 1] + Psi[1, 3]
```

The star involution commutes with the antipode:

```
sage: all( R(I).star_involution().antipode()
....:      == R(I).antipode().star_involution()
....:      for I in Compositions(4) )
True
```

Checking the relation with the descent algebra described above:

```
sage: def descent_test(n):
....:     DA = DescentAlgebra(QQ, n)
....:     NSym = NonCommutativeSymmetricFunctions(QQ)
....:     S = NSym.S()
....:     DAD = DA.D()
....:     w_n = DAD(set(range(1, n)))
....:     for I in Compositions(n):
....:         if not (S[I].star_involution()
....:                 == w_n * S[I].to_descent_algebra(n) * w_n):
....:             return False
....:         return True
sage: all( descent_test(i) for i in range(4) )
True
sage: all( descent_test(i) for i in range(6) ) # long time
True
```

Testing the $\pi(f^*) = \pi(f)$ relation noticed above:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: R = NSym.R()
sage: all( R(I).star_involution().to_symmetric_function()
....:      == R(I).to_symmetric_function()
....:      for I in Compositions(4) )
True
```

The star involution on $QSym$ is adjoint to the star involution on $NSym$ with respect to the duality pairing:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.M()
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: S = NSym.S()
```

```
sage: all( all( M(I).star_involution().duality_pairing(S(J))
....:           == M(I).duality_pairing(S(J).star_involution())
....:           for I in Compositions(2) )
....:       for J in Compositions(3) )
True
```

to_descent_algebra(*n=None*)

Return the image of the `n`-th degree homogeneous component of `self` in the descent algebra of S_n over the same base ring as `self`.

This is based upon the canonical isomorphism from the n-th degree homogeneous component of the algebra of noncommutative symmetric functions to the descent algebra of S_n. This isomorphism maps the inner product of noncommutative symmetric functions either to the product in the descent algebra of S_n or to its opposite (depending on how the latter is defined).

If `n` is not specified, it will be taken to be the highest homogeneous component of `self`.

OUTPUT:

- The image of the `n`-th homogeneous component of `self` under the isomorphism into the descent algebra of S_n over the same base ring as `self`.

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(ZZ).S()
sage: S[2,1].to_descent_algebra(3)
B[2, 1]
sage: (S[1,2,1] - 3 * S[1,1,2]).to_descent_algebra(4)
-3*B[1, 1, 2] + B[1, 2, 1]
sage: S[2,1].to_descent_algebra(2)
0
sage: S[2,1].to_descent_algebra()
B[2, 1]
sage: S.zero().to_descent_algebra().parent()
Descent algebra of 0 over Integer Ring in the subset basis
sage: (S[1,2,1] - 3 * S[1,1,2]).to_descent_algebra(1)
0
```

to_ncsym()

Return the image of `self` in the symmetric functions in non-commuting variables under the map described in `to_ncsym_on_basis()`.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: x = R.an_element(); x
2*R[] + 2*R[1] + 3*R[1, 1]
sage: x.to_ncsym()
2*m{} + 2*m{{1}} + 3/2*m{{1}, {2}}
sage: y = N.Phi()[1,2]
sage: y.to_ncsym()
m{{1}, {2, 3}} + m{{1, 2, 3}}
```

to_symmetric_function()

Return the commutative image of a non-commutative symmetric function.

OUTPUT:

- The commutative image of `self`. This will be a symmetric function.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: x = R.an_element(); x
```

```
2*R[] + 2*R[1] + 3*R[1, 1]
sage: x.to_symmetric_function()
2*s[] + 2*s[1] + 3*s[1, 1]
sage: y = N.Phi()[1,3]
sage: y.to_symmetric_function()
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 3*h[3, 1]
```

to_symmetric_group_algebra()

Return the image of a non-commutative symmetric function into the symmetric group algebra where the ribbon basis element indexed by a composition is associated with the sum of all permutations which have descent set equal to said composition. In compliance with the anti- isomorphism between the descent algebra and the non-commutative symmetric functions, we index descent positions by the reversed composition.

OUTPUT:

- The image of `self` under the embedding of the n-th degree homogeneous component of the non-commutative symmetric functions in the symmetric group algebra of S_n. This can behave unexpectedly when `self` is not homogeneous.

EXAMPLES:

```
sage: R=NonCommutativeSymmetricFunctions(QQ).R()
sage: R[2,1].to_symmetric_group_algebra()
[1, 3, 2] + [2, 3, 1]
sage: R([]).to_symmetric_group_algebra()
[]
```

TESTS:

Sending a noncommutative symmetric function to the symmetric group algebra directly has the same result as going through the descent algebra:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: SGA4 = SymmetricGroupAlgebra(QQ, 4)
sage: D4 = DescentAlgebra(QQ, 4).D()
sage: all( S[C].to_symmetric_group_algebra()
....:      == SGA4(D4(S[C].to_descent_algebra(4)))
....:      for C in Compositions(4) )
True
```

verschiebung(n)

Return the image of the noncommutative symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the map from the $\mathbf{k}$-algebra of noncommutative symmetric functions to itself that sends the complete function S^I indexed by a composition $I = (i_1, i_2, \ldots, i_k)$ to $S^{(i_1/n, i_2/n, \ldots, i_k/n)}$ if all of the numbers $i_1, i_2, \ldots, i_k$ are divisible by n, and to 0 otherwise. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every positive integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(S_r) = S_{r/n}, \quad \mathbf{V}_n(\Lambda_r) = (-1)^{r-r/n}\Lambda_{r/n}, \quad \mathbf{V}_n(\Psi_r) = n\Psi_{r/n}, \quad \mathbf{V}_n(\Phi_r) = n\Phi_{r/n}$$

(where S_r denotes the r-th complete non-commutative symmetric function, Λ_r denotes the r-th elementary non-commutative symmetric function, Ψ_r denotes the r-th power-sum non-commutative symmetric function of the first kind, and Φ_r denotes the r-th power-sum non-commutative symmetric function of the second kind). For every positive integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(S_r) = \mathbf{V}_n(\Lambda_r) = \mathbf{V}_n(\Psi_r) = \mathbf{V}_n(\Phi_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism.

It is a lift of the n-th Verschiebung operator on the ring of symmetric functions (`verschiebung()`) to the ring of noncommutative symmetric functions.

The action of the n-th Verschiebung operator can also be described on the ribbon Schur functions. Namely, every composition I of size $n\ell$ satisfies

$$\mathbf{V}_n(R_I) = (-1)^{\ell(I)-\ell(J)} \cdot R_{J/n},$$

where J denotes the meet of the compositions I and $(\underbrace{n, n, \ldots, n}_{|I|/n \text{ times}})$, where $\ell(I)$ is the length of I, and where J/n denotes the composition obtained by dividing every entry of J by n. For a composition I of size not divisible by n, we have $\mathbf{V}_n(R_I) = 0$.

See also:

`frobenius method of QSym`, `verschiebung method of Sym`

INPUT:

- n – a positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of noncommutative symmetric functions) to `self`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: S[3,2].verschiebung(2)
0
sage: S[6,4].verschiebung(2)
S[3, 2]
sage: (S[9,1] - S[8,2] + 2*S[6,4] - 3*S[3] + 4*S[[]]).verschiebung(2)
4*S[] + 2*S[3, 2] - S[4, 1]
sage: (S[3,3] - 2*S[2]).verschiebung(3)
S[1, 1]
sage: S([4,2]).verschiebung(1)
S[4, 2]
sage: R = NSym.R()
sage: R([4,2]).verschiebung(2)
R[2, 1]
```

Being Hopf algebra endomorphisms, the Verschiebung operators commute with the antipode:

```
sage: all( R(I).verschiebung(2).antipode()
....:      == R(I).antipode().verschiebung(2)
....:      for I in Compositions(4) )
True
```

They lift the Verschiebung operators of the ring of symmetric functions:

```
sage: all( S(I).verschiebung(2).to_symmetric_function()
....:      == S(I).to_symmetric_function().verschiebung(2)
....:      for I in Compositions(4) )
True
```

The Frobenius operators on $QSym$ are adjoint to the Verschiebung operators on $NSym$ with respect to the duality pairing:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: M = QSym.M()
sage: all( all( M(I).frobenius(3).duality_pairing(S(J))
....:           == M(I).duality_pairing(S(J).verschiebung(3))
```

```
....:           for I in Compositions(2) )
....:       for J in Compositions(3) )
True
```

class NonCommutativeSymmetricFunctions.Bases.**ParentMethods**

immaculate_function(*xs*)

Return the immaculate function corresponding to the integer vector `xs`, written in the basis `self`.

If α is any integer vector – i.e., an element of $\mathbf{Z}^m$ for some $m \in \mathbf{N}$ –, the *immaculate function corresponding to* α is a non-commutative symmetric function denoted by $\mathfrak{S}_\alpha$. One way to define this function is by setting

$$\mathfrak{S}_\alpha = \sum_{\sigma \in S_m} (-1)^\sigma S_{\alpha_1+\sigma(1)-1} S_{\alpha_2+\sigma(2)-2} \cdots S_{\alpha_m+\sigma(m)-m},$$

where α is written in the form $(\alpha_1, \alpha_2, \ldots, \alpha_m)$, and where S stands for the complete basis (`Complete`).

The immaculate function $\mathfrak{S}_\alpha$ first appeared in [BBSSZ2012] (where it was defined differently, but the definition we gave above appeared as Theorem 3.27).

The immaculate functions $\mathfrak{S}_\alpha$ for α running over all compositions (i.e., finite sequences of positive integers) form a basis of $NCSF$. This is the *immaculate basis* (`Immaculate`).

INPUT:

- `xs` – list (or tuple or any iterable – possibly a composition) of integers

OUTPUT:

The immaculate function $\mathfrak{S}_{xs}$ written in the basis `self`.

EXAMPLES:

Let us first check that, for `xs` a composition, we get the same as the result of `self.realization_of().I()[xs]`:

```
sage: def test_comp(xs):
....:     NSym = NonCommutativeSymmetricFunctions(QQ)
....:     I = NSym.I()
....:     return I[xs] == I.immaculate_function(xs)
sage: def test_allcomp(n):
....:     return all( test_comp(c) for c in Compositions(n) )
sage: test_allcomp(1)
True
sage: test_allcomp(2)
True
sage: test_allcomp(3)
True
```

Now some examples of non-composition immaculate functions:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: I = NSym.I()
sage: I.immaculate_function([0, 1])
0
sage: I.immaculate_function([0, 2])
-I[1, 1]
sage: I.immaculate_function([-1, 1])
-I[]
sage: I.immaculate_function([2, -1])
0
```

```
sage: I.immaculate_function([2, 0])
I[2]
sage: I.immaculate_function([2, 0, 1])
0
sage: I.immaculate_function([1, 0, 2])
-I[1, 1, 1]
sage: I.immaculate_function([2, 0, 2])
-I[2, 1, 1]
sage: I.immaculate_function([0, 2, 0, 2])
I[1, 1, 1, 1] + I[1, 2, 1]
sage: I.immaculate_function([2, 0, 2, 0, 2])
I[2, 1, 1, 1, 1] + I[2, 1, 2, 1]
```

TESTS:

Basis-independence:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: L = NSym.L()
sage: S = NSym.S()
sage: L(S.immaculate_function([0, 2, 0, 2])) == L.immaculate_function([0, 2, 0, 2])
True
```

to_ncsym()

Morphism κ of `self` to the algebra of symmetric functions in non-commuting variables that for the natural maps $\chi : NCSym \to Sym$ and $\rho : NSym \to Sym$, we have $\chi \circ \kappa = \rho$.

This is constructed by extending the method `to_ncsym_on_basis()` linearly.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: x = R.an_element(); x
2*R[] + 2*R[1] + 3*R[1, 1]
sage: R.to_ncsym(x)
2*m{} + 2*m{{1}} + 3/2*m{{1}, {2}}
sage: S = N.complete()
sage: S.to_ncsym(S[1,2])
1/2*m{{1}, {2}, {3}} + m{{1}, {2, 3}} + 1/2*m{{1, 2}, {3}}
 + m{{1, 2, 3}} + 1/2*m{{1, 3}, {2}}
sage: Phi = N.Phi()
sage: Phi.to_ncsym(Phi[1,3])
-1/4*m{{1}, {2}, {3, 4}} - 1/4*m{{1}, {2, 3}, {4}} + m{{1}, {2, 3, 4}}
 + 1/2*m{{1}, {2, 4}, {3}} - 1/4*m{{1, 2}, {3, 4}} - 1/4*m{{1, 2, 3}, {4}}
 + m{{1, 2, 3, 4}} + 1/2*m{{1, 2, 4}, {3}} + 1/2*m{{1, 3}, {2, 4}}
 - 1/4*m{{1, 3, 4}, {2}} - 1/4*m{{1, 4}, {2, 3}}
sage: R.to_ncsym
Generic morphism:
  From: Non-Commutative Symmetric Functions over the Rational Field in the Ribbon bas
  To:   Symmetric functions in non-commuting variables over the Rational Field in the
```

to_ncsym_on_basis(I)

The image of the basis element indexed by `I` under the map κ to the symmetric functions in non-commuting variables such that for the natural maps $\chi : NCSym \to Sym$ and $\rho : NSym \to Sym$, we have $\chi \circ \kappa = \rho$.

This default implementation does a change of basis and computes the image in the complete basis.

See also:

`to_ncsym_on_basis()`

INPUT:

- I – a composition

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.to_ncsym(S[2,1])
1/2*m{{1}, {2}, {3}} + 1/2*m{{1}, {2, 3}} + m{{1, 2}, {3}}
 + m{{1, 2, 3}} + 1/2*m{{1, 3}, {2}}
sage: R = NonCommutativeSymmetricFunctions(QQ).R()
sage: R.to_ncsym_on_basis(Composition([2,1]))
1/3*m{{1}, {2}, {3}} + 1/6*m{{1}, {2, 3}} + 2/3*m{{1, 2}, {3}} + 1/6*m{{1, 3}, {2}}
```

to_symmetric_function()

Morphism to the algebra of symmetric functions.

This is constructed by extending the computation on the basis or by coercion to the complete basis.

OUTPUT:

- The module morphism from the basis `self` to the symmetric functions which corresponds to taking a commutative image.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: x = R.an_element(); x
2*R[] + 2*R[1] + 3*R[1, 1]
sage: R.to_symmetric_function(x)
2*s[] + 2*s[1] + 3*s[1, 1]
sage: nM = N.Monomial()
sage: nM.to_symmetric_function(nM[3,1])
h[1, 1, 1, 1] - 7/2*h[2, 1, 1] + h[2, 2] + 7/2*h[3, 1] - 2*h[4]
```

to_symmetric_function_on_basis(*I*)

The image of the basis element indexed by `I` under the map to the symmetric functions.

This default implementation does a change of basis and computes the image in the complete basis.

INPUT:

- I – a composition

OUTPUT:

- The image of the non-commutative basis element of `self` indexed by the composition `I` under the map from non-commutative symmetric functions to the symmetric functions. This will be a symmetric function.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: I = N.Immaculate()
sage: I.to_symmetric_function(I[1,3])
-h[2, 2] + h[3, 1]
sage: I.to_symmetric_function(I[1,2])
0
sage: Phi = N.Phi()
sage: Phi.to_symmetric_function_on_basis([3,1,2])==Phi.to_symmetric_function(Phi[3,1,
True
sage: Phi.to_symmetric_function_on_basis([])
h[]
```

NonCommutativeSymmetricFunctions.Bases.**super_categories**()

Return the super categories of the category of bases of the non-commutative symmetric functions.

OUTPUT:

•list

TESTS:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: N.Bases().super_categories()
[Category of bases of Non-Commutative Symmetric Functions or Quasisymmetric functions ov
 Category of realizations of graded modules with internal product over Rational Field]
```

class NonCommutativeSymmetricFunctions.**Complete**(*NCSF*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The Hopf algebra of non-commutative symmetric functions in the Complete basis.

The Complete basis is defined in Definition 3.4 of [NCSF1], where it is denoted by $(S^I)_I$. It is a multiplicative basis, and is connected to the elementary generators Λ_i of the ring of non-commutative symmetric functions by the following relation: Define a non-commutative symmetric function S_n for every nonnegative integer n by the power series identity

$$\sum_{k\geq 0} t^k S_k = \left(\sum_{k\geq 0} (-t)^k \Lambda_k\right)^{-1},$$

with Λ_0 denoting 1. For every composition $(i_1, i_2, \ldots, i_k)$, we have $S^{(i_1,i_2,\ldots,i_k)} = S_{i_1} S_{i_2} \cdots S_{i_k}$.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: S = NCSF.Complete(); S
Non-Commutative Symmetric Functions over the Rational Field in the Complete basis
sage: S.an_element()
2*S[] + 2*S[1] + 3*S[1, 1]
```

The following are aliases for this basis:

```
sage: NCSF.complete()
Non-Commutative Symmetric Functions over the Rational Field in the Complete basis
sage: NCSF.S()
Non-Commutative Symmetric Functions over the Rational Field in the Complete basis
```

class Element(*M*, *x*)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

An element in the Complete basis.

psi_involution()

Return the image of the noncommutative symmetric function `self` under the involution ψ.

The involution ψ is defined as the linear map $NCSF \to NCSF$ which, for every composition I, sends the complete noncommutative symmetric function S^I to the elementary noncommutative symmetric function Λ^I. It can be shown that every composition I satisfies

$$\psi(R_I) = R_{I^c}, \quad \psi(S^I) = \Lambda^I, \quad \psi(\Lambda^I) = S^I, \quad \psi(\Phi^I) = (-1)^{|I|-\ell(I)} \Phi^I$$

where I^c denotes the complement of the composition I, and $\ell(I)$ denotes the length of I, and where standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, Φ for the basis of the power sums of the second kind, and R for the ribbon basis). The map ψ is an involution and a graded Hopf algebra automorphism of $NCSF$. If π denotes the projection from $NCSF$ to the ring of symmetric functions (`to_symmetric_function()`), then $\pi(\psi(f)) = \omega(\pi(f))$ for every $f \in NCSF$, where the ω on the right hand side denotes the omega automorphism of Sym.

The involution ψ of $NCSF$ is adjoint to the involution ψ of $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The involution ψ has been denoted by ψ in [LMvW13], section 3.6.

See also:

psi involution of NCSF, psi involution of QSym, star involution of NCSF.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: S = NSym.S()
sage: L = NSym.L()
sage: S[3,1].psi_involution()
S[1, 1, 1, 1] - S[1, 2, 1] - S[2, 1, 1] + S[3, 1]
sage: L(S[3,1].psi_involution())
L[3, 1]
sage: S[[]].psi_involution()
S[]
sage: S[1,1].psi_involution()
S[1, 1]
sage: (S[2,1] - 2*S[2]).psi_involution()
-2*S[1, 1] + S[1, 1, 1] + 2*S[2] - S[2, 1]
```

The implementation at hand is tailored to the complete basis. It is equivalent to the generic implementation via the ribbon basis:

```
sage: R = NSym.R()
sage: all( R(S[I].psi_involution()) == R(S[I]).psi_involution()
....:      for I in Compositions(4) )
True
```

NonCommutativeSymmetricFunctions.Complete.**dual**()

Return the dual basis to the complete basis of non-commutative symmetric functions. This is the Monomial basis of quasi-symmetric functions.

OUTPUT:

- The Monomial basis of quasi-symmetric functions.

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.dual()
Quasisymmetric functions over the Rational Field in the Monomial basis
```

NonCommutativeSymmetricFunctions.Complete.**internal_product_on_basis**(*I*, *J*)

The internal product of two non-commutative symmetric complete functions.

See internal_product() for a thorough documentation of this operation.

INPUT:

- I, J – compositions

OUTPUT:

- The internal product of the complete non-commutative symmetric function basis elements indexed by I and J, expressed in the complete basis.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: S.internal_product_on_basis([2,2],[1,2,1])
2*S[1, 1, 1, 1] + S[1, 1, 2] + S[2, 1, 1]
```

```
sage: S.internal_product_on_basis([2,2],[1,2])
0
```

NonCommutativeSymmetricFunctions.Complete.**to_ncsym_on_basis**(*I*)

Return the image of the complete non-commutative symmetric function in the symmetric functions in non-commuting variables under the embedding $\mathcal{I}$ which fixes the symmetric functions.

This map is defined by

$$S_n \mapsto \sum_{A \vdash [n]} \frac{\lambda(A)!\lambda(A)^!}{n!} \mathbf{m}_A$$

and extended as an algebra homomorphism.

Note: A remark in [BRRZ08] makes it clear that the embedding of NCSF into NCSym that preserves the projection into the symmetric functions is not unique. While this seems to be a natural embedding, any set of algebraic generators can be sent to a set of free elements in NCSym is also an embedding.

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.to_ncsym_on_basis(Composition([2]))
1/2*m{{1}, {2}} + m{{1, 2}}
sage: S.to_ncsym_on_basis(Composition([1,2,1]))
1/2*m{{1}, {2}, {3}, {4}} + 1/2*m{{1}, {2}, {3, 4}} + m{{1}, {2, 3}, {4}}
 + m{{1}, {2, 3, 4}} + 1/2*m{{1}, {2, 4}, {3}} + 1/2*m{{1, 2}, {3}, {4}}
 + 1/2*m{{1, 2}, {3, 4}} + m{{1, 2, 3}, {4}} + m{{1, 2, 3, 4}}
 + 1/2*m{{1, 2, 4}, {3}} + 1/2*m{{1, 3}, {2}, {4}} + 1/2*m{{1, 3}, {2, 4}}
 + 1/2*m{{1, 3, 4}, {2}} + 1/2*m{{1, 4}, {2}, {3}} + m{{1, 4}, {2, 3}}
sage: S.to_ncsym_on_basis(Composition([]))
m{}
```

TESTS:

Check that the image under $\mathcal{I}$ fixes the symmetric functions:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
sage: mon = SymmetricFunctions(QQ).monomial()
sage: all(S[c].to_ncsym().to_symmetric_function() == S[c].to_symmetric_function()
....:     for i in range(5) for c in Compositions(i))
True
```

We also check that the $NCSym$ monomials agree on the homogeneous and complete basis:

```
sage: h = SymmetricFunctions(QQ).h()
sage: all(m.from_symmetric_function(h[i]) == S[i].to_ncsym() for i in range(6))
True
```

NonCommutativeSymmetricFunctions.Complete.**to_symmetric_function**()

Morphism to the algebra of symmetric functions.

This is constructed by extending the computation on the complete basis.

OUTPUT:

- The module morphism from the basis `self` to the symmetric functions which corresponds to taking a commutative image.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: S.to_symmetric_function(S[3,1,2])
```

```
h[3, 2, 1]
sage: S.to_symmetric_function(S[[]])
h[]
```

NonCommutativeSymmetricFunctions.Complete.**to_symmetric_function_on_basis**(*I*)

The commutative image of a complete element

The commutative image of a basis element is obtained by sorting the indexing composition of the basis element and the output is in the complete basis of the symmetric functions.

INPUT:

- I – a composition

OUTPUT:

- The commutative image of the complete basis element indexed by I. The result is the complete symmetric function indexed by the partition obtained by sorting I.

EXAMPLES:

```
sage: S=NonCommutativeSymmetricFunctions(QQ).complete()
sage: S.to_symmetric_function_on_basis([2,1,3])
h[3, 2, 1]
sage: S.to_symmetric_function_on_basis([])
h[]
```

class NonCommutativeSymmetricFunctions.**Elementary**(*NCSF*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The Hopf algebra of non-commutative symmetric functions in the Elementary basis.

The Elementary basis is defined in Definition 3.4 of [NCSF1], where it is denoted by $(\Lambda^I)_I$. It is a multiplicative basis, and is obtained from the elementary generators Λ_i of the ring of non-commutative symmetric functions through the formula $\Lambda^{(i_1,i_2,\ldots,i_k)} = \Lambda_{i_1}\Lambda_{i_2}\cdots\Lambda_{i_k}$ for every composition $(i_1, i_2, \ldots, i_k)$.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: L = NCSF.Elementary(); L
Non-Commutative Symmetric Functions over the Rational Field in the Elementary basis
sage: L.an_element()
2*L[] + 2*L[1] + 3*L[1, 1]
```

The following are aliases for this basis:

```
sage: NCSF.elementary()
Non-Commutative Symmetric Functions over the Rational Field in the Elementary basis
sage: NCSF.L()
Non-Commutative Symmetric Functions over the Rational Field in the Elementary basis
```

class **Element**(*M*, *x*)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

psi_involution()

Return the image of the noncommutative symmetric function `self` under the involution ψ.

The involution ψ is defined as the linear map $NCSF \to NCSF$ which, for every composition I, sends the complete noncommutative symmetric function S^I to the elementary noncommutative symmetric function Λ^I. It can be shown that every composition I satisfies

$$\psi(R_I) = R_{I^c}, \quad \psi(S^I) = \Lambda^I, \quad \psi(\Lambda^I) = S^I, \quad \psi(\Phi^I) = (-1)^{|I|-\ell(I)}\Phi^I$$

where I^c denotes the complement of the composition I, and $\ell(I)$ denotes the length of I, and where standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, Φ for the basis of the power sums of the second kind, and R for the ribbon basis). The map ψ is an involution and a graded Hopf algebra automorphism of $NCSF$. If π denotes the projection from $NCSF$ to the ring of symmetric functions (`to_symmetric_function()`), then $\pi(\psi(f)) = \omega(\pi(f))$ for every $f \in NCSF$, where the ω on the right hand side denotes the omega automorphism of Sym.

The involution ψ of $NCSF$ is adjoint to the involution ψ of $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The involution ψ has been denoted by ψ in [LMvW13], section 3.6.

See also:

`psi involution of NCSF`, `psi involution of QSym`, `star involution of NCSF`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: S = NSym.S()
sage: L = NSym.L()
sage: L[3,1].psi_involution()
L[1, 1, 1, 1] - L[1, 2, 1] - L[2, 1, 1] + L[3, 1]
sage: S(L[3,1].psi_involution())
S[3, 1]
sage: L[[]].psi_involution()
L[]
sage: L[1,1].psi_involution()
L[1, 1]
sage: (L[2,1] - 2*L[2]).psi_involution()
-2*L[1, 1] + L[1, 1, 1] + 2*L[2] - L[2, 1]
```

The implementation at hand is tailored to the elementary basis. It is equivalent to the generic implementation via the ribbon basis:

```
sage: R = NSym.R()
sage: all( R(L[I].psi_involution()) == R(L[I]).psi_involution()
....:      for I in Compositions(3) )
True
sage: all( R(L[I].psi_involution()) == R(L[I]).psi_involution()
....:      for I in Compositions(4) )
True
```

star_involution()

Return the image of the noncommutative symmetric function `self` under the star involution.

The star involution is defined as the algebra antihomomorphism $NCSF \to NCSF$ which, for every positive integer n, sends the n-th complete non-commutative symmetric function S_n to S_n. Denoting by f^* the image of an element $f \in NCSF$ under this star involution, it can be shown that every composition I satisfies

$$(S^I)^* = S^{I^r}, \quad (\Lambda^I)^* = \Lambda^{I^r}, \quad R_I^* = R_{I^r}, \quad (\Phi^I)^* = \Phi^{I^r},$$

where I^r denotes the reversed composition of I, and standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, R for the ribbon basis, and Φ for that of the power-sums of the second kind). The star involution is an involution and a coalgebra automorphism of $NCSF$. It is an automorphism of the graded vector space $NCSF$. Under the canonical isomorphism between the n-th graded component of $NCSF$ and the descent algebra of the symmetric group S_n (see `to_descent_algebra()`), the star involution (restricted to the n-th graded component) corresponds to the automorphism of the descent algebra given by $x \mapsto \omega_n x \omega_n$, where ω_n is the permutation $(n, n-1, \ldots, 1) \in S_n$ (written here in one-line notation). If π denotes the projection from $NCSF$ to the ring of symmetric functions (`to_symmetric_function()`), then $\pi(f^*) = \pi(f)$ for every $f \in NCSF$.

The star involution on $NCSF$ is adjoint to the star involution on $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The star involution has been denoted by ρ in [LMvW13], section 3.6. See [NCSF2], section 2.3 for the properties of this map.

See also:

`star involution of NCSF`, `psi involution of NCSF`, `star involution of QSym`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: L = NSym.L()
sage: L[3,3,2,3].star_involution()
L[3, 2, 3, 3]
sage: L[6,3,3].star_involution()
L[3, 3, 6]
sage: (L[1,9,1] - L[8,2] + 2*L[6,4] - 3*L[3] + 4*L[[]]).star_involution()
4*L[] + L[1, 9, 1] - L[2, 8] - 3*L[3] + 2*L[4, 6]
sage: (L[3,3] - 2*L[2]).star_involution()
-2*L[2] + L[3, 3]
sage: L([4,1]).star_involution()
L[1, 4]
```

The implementation at hand is tailored to the elementary basis. It is equivalent to the generic implementation via the complete basis:

```
sage: S = NSym.S()
sage: all( S(L[I].star_involution()) == S(L[I]).star_involution()
....:      for I in Compositions(4) )
True
```

verschiebung(n)

Return the image of the noncommutative symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the map from the $\mathbf{k}$-algebra of noncommutative symmetric functions to itself that sends the complete function S^I indexed by a composition $I = (i_1, i_2, \ldots, i_k)$ to $S^{(i_1/n, i_2/n, \ldots, i_k/n)}$ if all of the numbers $i_1, i_2, \ldots, i_k$ are divisible by n, and to 0 otherwise. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every positive integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(S_r) = S_{r/n}, \quad \mathbf{V}_n(\Lambda_r) = (-1)^{r-r/n}\Lambda_{r/n}, \quad \mathbf{V}_n(\Psi_r) = n\Psi_{r/n}, \quad \mathbf{V}_n(\Phi_r) = n\Phi_{r/n}$$

(where S_r denotes the r-th complete non-commutative symmetric function, Λ_r denotes the r-th elementary non-commutative symmetric function, Ψ_r denotes the r-th power-sum non-commutative symmetric function of the first kind, and Φ_r denotes the r-th power-sum non-commutative symmetric function of the second kind). For every positive integer r with $n \nmid r$,

it satisfes

$$\mathbf{V}_n(S_r) = \mathbf{V}_n(\Lambda_r) = \mathbf{V}_n(\Psi_r) = \mathbf{V}_n(\Phi_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism.

It is a lift of the n-th Verschiebung operator on the ring of symmetric functions (`verschiebung()`) to the ring of noncommutative symmetric functions.

The action of the n-th Verschiebung operator can also be described on the ribbon Schur functions. Namely, every composition I of size $n\ell$ satisfies

$$\mathbf{V}_n(R_I) = (-1)^{\ell(I)-\ell(J)} \cdot R_{J/n},$$

where J denotes the meet of the compositions I and $(\underbrace{n, n, \ldots, n}_{|I|/n \text{ times}})$, where $\ell(I)$ is the length of I, and where J/n denotes the composition obtained by dividing every entry of J by n. For a composition I of size not divisible by n, we have $\mathbf{V}_n(R_I) = 0$.

See also:

`verschiebung method of NCSF`, `frobenius method of QSym`, `verschiebung method of Sym`

INPUT:

- n – a positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of noncommutative symmetric functions) to `self`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: L = NSym.L()
sage: L([4,2]).verschiebung(2)
-L[2, 1]
sage: L([2,4]).verschiebung(2)
-L[1, 2]
sage: L([6]).verschiebung(2)
-L[3]
sage: L([2,1]).verschiebung(3)
0
sage: L([3]).verschiebung(2)
0
sage: L([]).verschiebung(2)
L[]
sage: L([5, 1]).verschiebung(3)
0
sage: L([5, 1]).verschiebung(6)
0
sage: L([5, 1]).verschiebung(2)
0
sage: L([1, 2, 3, 1]).verschiebung(7)
0
sage: L([7]).verschiebung(7)
L[1]
sage: L([1, 2, 3, 1]).verschiebung(5)
0
sage: (L[1] - L[2] + 2*L[3]).verschiebung(1)
L[1] - L[2] + 2*L[3]
```

TESTS:

The current implementation on the Elementary basis gives the same results as the default implementation:

```
sage: S = NSym.S()
sage: def test_L(N, n):
....:     for I in Compositions(N):
....:         if S(L[I].verschiebung(n)) != S(L[I]).verschiebung(n):
....:             return False
....:     return True
sage: test_L(4, 2)
True
sage: test_L(6, 2)
True
sage: test_L(6, 3)
True
sage: test_L(8, 4)      # long time
True
```

class NonCommutativeSymmetricFunctions.**Immaculate**(*NCSF*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The immaculate basis of the non-commutative symmetric functions.

The immaculate basis first appears in Berg, Bergeron, Saliola, Serrano and Zabrocki's [BBSSZ2012]. It can be described as the family $(\mathfrak{S}_\alpha)$, where α runs over all compositions, and $\mathfrak{S}_\alpha$ denotes the immaculate function corresponding to α (see immaculate_function()).

If α is a composition $(\alpha_1, \alpha_2, \ldots, \alpha_m)$, then

$$\mathfrak{S}_\alpha = \sum_{\sigma \in S_m} (-1)^\sigma S_{\alpha_1+\sigma(1)-1} S_{\alpha_2+\sigma(2)-2} \cdots S_{\alpha_m+\sigma(m)-m}.$$

Warning: This *basis* contains only the immaculate functions indexed by compositions (i.e., finite sequences of positive integers). To obtain the remaining immaculate functions (sensu lato), use the immaculate_function() method. Calling the immaculate *basis* with a list which is not a composition will currently return garbage!

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: I = NCSF.I()
sage: I([1,3,2])*I([1])
I[1, 3, 2, 1] + I[1, 3, 3] + I[1, 4, 2] + I[2, 3, 2]
sage: I([1])*I([1,3,2])
I[1, 1, 3, 2] - I[2, 2, 1, 2] - I[2, 2, 2, 1] - I[2, 2, 3] - I[3, 2, 2]
sage: I([1,3])*I([1,1])
I[1, 3, 1, 1] + I[1, 4, 1] + I[2, 3, 1] + I[2, 4]
sage: I([3,1])*I([2,1])
I[3, 1, 2, 1] + I[3, 2, 1, 1] + I[3, 2, 2] + I[3, 3, 1] + I[4, 1, 1, 1] + I[4, 1, 2] + 2*I[4
sage: R = NCSF.ribbon()
sage: I(R[1,3,1])
I[1, 3, 1] + I[2, 2, 1] + I[2, 3] + I[3, 1, 1] + I[3, 2]
sage: R(I(R([2,1,3])))
R[2, 1, 3]
```

class **Element**(*M*, *x*)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

An element in the Immaculate basis.

bernstein_creation_operator(n)

Return the image of `self` under the n-th Bernstein creation operator.

Let n be an integer. The n-th Bernstein creation operator $\mathbb{B}_n$ is defined as the endomorphism of the space $NSym$ of noncommutative symmetric functions given by

$$\mathbb{B}_n I_{(\alpha_1,\alpha_2,\ldots,\alpha_m)} = I_{(n,\alpha_1,\alpha_2,\ldots,\alpha_m)},$$

where $I_{(\alpha_1,\alpha_2,\ldots,\alpha_m)}$ is the immaculate function associated to the m-tuple $(\alpha_1, \alpha_2, \ldots, \alpha_m) \in \mathbf{Z}^m$.

This has been introduced in [BBSSZ2012], section 3.1, in analogy to the Bernstein creation operators on the symmetric functions.

For more information on the n-th Bernstein creation operator, see `bernstein_creation_operator()`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: I = NSym.I()
sage: b = I[1,3,2,1]
sage: b.bernstein_creation_operator(3)
I[3, 1, 3, 2, 1]
sage: b.bernstein_creation_operator(5)
I[5, 1, 3, 2, 1]
sage: elt = b + 3*I[4,1,2]
sage: elt.bernstein_creation_operator(1)
I[1, 1, 3, 2, 1] + 3*I[1, 4, 1, 2]
```

We check that this agrees with the definition on the Complete basis:

```
sage: S = NSym.S()
sage: S(elt).bernstein_creation_operator(1) == S(elt.bernstein_creation_operator(1))
True
```

Check on non-positive values of n:

```
sage: I[2,2,2].bernstein_creation_operator(-1)
I[1, 1, 1, 2] + I[1, 1, 2, 1] + I[1, 2, 1, 1] - I[1, 2, 2]
sage: I[2,3,2].bernstein_creation_operator(0)
-I[1, 1, 3, 2] - I[1, 2, 2, 2] - I[1, 2, 3, 1] + I[2, 3, 2]
```

NonCommutativeSymmetricFunctions.Immaculate.**dual**()

Return the dual basis to the Immaculate basis of NCSF.

The basis returned is the dualImmaculate basis of QSym.

OUTPUT:

- The dualImmaculate basis of the quasi-symmetric functions.

EXAMPLES:

```
sage: I=NonCommutativeSymmetricFunctions(QQ).Immaculate()
sage: I.dual()
Quasisymmetric functions over the Rational Field in the dualImmaculate
basis
```

class NonCommutativeSymmetricFunctions.**Monomial**(*NCSF*)

Bases: `sage.combinat.free_module.CombinatorialFreeModule`, `sage.misc.bindable_class.BindableClass`

The monomial basis defined in Tevlin's paper [Tev2007].

The monomial basis is well-defined only when the base ring is a $\mathbf{Q}$-algebra. It is the basis denoted by $(M^I)_I$ in [Tev2007].

TESTS:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: nM = NCSF.monomial(); nM
Non-Commutative Symmetric Functions over the Rational Field in the Monomial basis
sage: nM([1,1])*nM([2])
3*nM[1, 1, 2] + nM[1, 3] + nM[2, 2]
sage: R = NCSF.ribbon()
sage: nM(R[1,3,1])
11*nM[1, 1, 1, 1, 1] + 8*nM[1, 1, 2, 1] + 8*nM[1, 2, 1, 1] + 5*nM[1, 3, 1] + 8*nM[2, 1, 1, 1
```

class NonCommutativeSymmetricFunctions.**MultiplicativeBases**(*parent_with_realization*)

Bases: sage.categories.realizations.Category_realization_of_parent

Category of multiplicative bases of non-commutative symmetric functions.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: N.MultiplicativeBases()
Category of multiplicative bases of Non-Commutative Symmetric Functions over the Rational Fi
```

The complete basis is a multiplicative basis, but the ribbon basis is not:

```
sage: N.Complete() in N.MultiplicativeBases()
True
sage: N.Ribbon() in N.MultiplicativeBases()
False
```

class ParentMethods

algebra_generators()

Return the algebra generators of a given multiplicative basis of non-commutative symmetric functions.

OUTPUT:

- The family of generators of the multiplicative basis self.

EXAMPLES:

```
sage: Psi = NonCommutativeSymmetricFunctions(QQ).Psi()
sage: f = Psi.algebra_generators()
sage: f
Lazy family (<lambda>(i))_{i in Positive integers}
sage: f[1], f[2], f[3]
(Psi[1], Psi[2], Psi[3])
```

algebra_morphism(*on_generators*, ***keywords*)

Given a map defined on the generators of the multiplicative basis self, return the algebra morphism that extends this map to the whole algebra of non-commutative symmetric functions.

INPUT:

- on_generators – a function defined on the index set of the generators (that is, on the positive integers)
- anti – a boolean; defaults to False
- category – a category; defaults to None

OUTPUT:

- The algebra morphism of self which is defined by on_generators in the basis self.

When `anti` is set to `True`, an algebra anti-morphism is computed instead of an algebra morphism.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: Psi = NCSF.Psi()
sage: double = lambda i: Psi[i,i]
sage: f = Psi.algebra_morphism(double, codomain = Psi)
sage: f
Generic endomorphism of Non-Commutative Symmetric Functions over the Rational Field i
sage: f(2*Psi[[]] + 3 * Psi[1,3,2] + Psi[2,4] )
2*Psi[] + 3*Psi[1, 1, 3, 3, 2, 2] + Psi[2, 2, 4, 4]
sage: f.category()
Category of endsets of unital magmas and right modules over Rational Field and left m
```

When extra properties about the morphism are known, one can specify the category of which it is a morphism:

```
sage: negate = lambda i: -Psi[i]
sage: f = Psi.algebra_morphism(negate, codomain = Psi, category = GradedHopfAlgebrasW
sage: f
Generic endomorphism of Non-Commutative Symmetric Functions over the Rational Field i
sage: f(2*Psi[[]] + 3 * Psi[1,3,2] + Psi[2,4] )
2*Psi[] - 3*Psi[1, 3, 2] + Psi[2, 4]
sage: f.category()
Category of endsets of hopf algebras over Rational Field and graded modules over Rati
```

If `anti` is true, this returns an anti-algebra morphism:

```
sage: f = Psi.algebra_morphism(double, codomain = Psi, anti=True)
sage: f
Generic endomorphism of Non-Commutative Symmetric Functions over the Rational Field i
sage: f(2*Psi[[]] + 3 * Psi[1,3,2] + Psi[2,4] )
2*Psi[] + 3*Psi[2, 2, 3, 3, 1, 1] + Psi[4, 4, 2, 2]
sage: f.category()
Category of endsets of modules with basis over Rational Field
```

antipode()

Return the antipode morphism on the basis `self`.

The antipode of $NSym$ is closely related to the omega involution; see `omega_involution()` for the latter.

OUTPUT:

•The antipode module map from non-commutative symmetric functions on basis `self`.

EXAMPLES:

```
sage: S=NonCommutativeSymmetricFunctions(QQ).S()
sage: S.antipode
Generic endomorphism of Non-Commutative Symmetric Functions over the Rational Field i
```

coproduct()

Return the coproduct morphism in the basis `self`.

OUTPUT:

•The coproduct module map from non-commutative symmetric functions on basis `self`.

EXAMPLES:

```
sage: S=NonCommutativeSymmetricFunctions(QQ).S()
sage: S.coproduct
Generic morphism:
  From: Non-Commutative Symmetric Functions over the Rational Field in the Complete b
  To:   Non-Commutative Symmetric Functions over the Rational Field in the Complete b
```

product_on_basis(*composition1*, *composition2*)

Return the product of two basis elements from the multiplicative basis. Multiplication is just concatenation on compositions.

INPUT:

- `composition1`, `composition2` – integer compositions

OUTPUT:

- The product of the two non-commutative symmetric functions indexed by `composition1` and `composition2` in the multiplicative basis `self`. This will be again a non-commutative symmetric function.

EXAMPLES:

```
sage: Psi = NonCommutativeSymmetricFunctions(QQ).Psi()
sage: Psi[3,1,2] * Psi[4,2] == Psi[3,1,2,4,2]
True
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.product_on_basis(Composition([2,1]), Composition([1,2]))
S[2, 1, 1, 2]
```

to_symmetric_function()

Morphism to the algebra of symmetric functions.

This is constructed by extending the algebra morphism by the image of the generators.

OUTPUT:

- The module morphism from the basis `self` to the symmetric functions which corresponds to taking a commutative image.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: S.to_symmetric_function(S[1,3])
h[3, 1]
sage: Phi = N.Phi()
sage: Phi.to_symmetric_function(Phi[1,3])
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 3*h[3, 1]
sage: Psi = N.Psi()
sage: Psi.to_symmetric_function(Psi[1,3])
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 3*h[3, 1]
```

to_symmetric_function_on_generators(*i*)

Morphism of the generators to symmetric functions.

This is constructed by coercion to the complete basis and applying the morphism.

OUTPUT:

- The module morphism from the basis `self` to the symmetric functions which corresponds to taking a commutative image.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: Phi = N.Phi()
sage: Phi.to_symmetric_function_on_generators(3)
h[1, 1, 1] - 3*h[2, 1] + 3*h[3]
sage: Phi.to_symmetric_function_on_generators(0)
h[]
sage: Psi = N.Psi()
sage: Psi.to_symmetric_function_on_generators(3)
h[1, 1, 1] - 3*h[2, 1] + 3*h[3]
sage: L = N.elementary()
sage: L.to_symmetric_function_on_generators(3)
h[1, 1, 1] - 2*h[2, 1] + h[3]
```

NonCommutativeSymmetricFunctions.MultiplicativeBases.**super_categories**()
Return the super categories of the category of multiplicative bases of the non-commutative symmetric functions.

OUTPUT:
- list

TESTS:
```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: N.MultiplicativeBases().super_categories()
[Category of bases of Non-Commutative Symmetric Functions over the Rational Field]
```

class NonCommutativeSymmetricFunctions.**MultiplicativeBasesOnGroupLikeElements**(*parent_with_re*
Bases: sage.categories.realizations.Category_realization_of_parent

Category of multiplicative bases on grouplike elements of non-commutative symmetric functions.

Here, a "multiplicative basis on grouplike elements" means a multiplicative basis whose generators $(f_1, f_2, f_3, \ldots)$ satisfy

$$\Delta(f_i) = \sum_{j=0}^{i} f_j \otimes f_{i-j}$$

with $f_0 = 1$. (In other words, the generators are to form a divided power sequence in the sense of a coalgebra.) This does not mean that the generators are grouplike, but means that the element $1 + f_1 + f_2 + f_3 + \cdots$ in the completion of the ring of non-commutative symmetric functions with respect to the grading is grouplike.

EXAMPLES:
```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: N.MultiplicativeBasesOnGroupLikeElements()
Category of multiplicative bases on group like elements of Non-Commutative Symmetric Functio
```

The complete basis is a multiplicative basis, but the ribbon basis is not:
```
sage: N.Complete() in N.MultiplicativeBasesOnGroupLikeElements()
True
sage: N.Ribbon() in N.MultiplicativeBasesOnGroupLikeElements()
False
```

class ParentMethods

antipode_on_basis(*composition*)
Return the application of the antipode to a basis element.

INPUT:
- composition – a composition

OUTPUT:
- The image of the basis element indexed by composition under the antipode map.

EXAMPLES:
```
sage: S = NonCommutativeSymmetricFunctions(QQ).complete()
sage: S.antipode_on_basis(Composition([2,1]))
-S[1, 1, 1] + S[1, 2]
sage: S[1].antipode() # indirect doctest
-S[1]
sage: S[2].antipode() # indirect doctest
S[1, 1] - S[2]
sage: S[3].antipode() # indirect docttest
```

```
-S[1, 1, 1] + S[1, 2] + S[2, 1] - S[3]
sage: S[2,3].coproduct().apply_multilinear_morphism(lambda be,ga: S(be)*S(ga).antipod
0
sage: S[2,3].coproduct().apply_multilinear_morphism(lambda be,ga: S(be).antipode()*S(
0
```

coproduct_on_generators(*i*)

Return the image of the i^{th} generator of the algebra under the coproduct.

INPUT:

- `i` – a positive integer

OUTPUT:

- The result of applying the coproduct to the i^{th} generator of `self`.

EXAMPLES:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).complete()
sage: S.coproduct_on_generators(3)
S[] # S[3] + S[1] # S[2] + S[2] # S[1] + S[3] # S[]
```

TESTS:

```
sage: S.coproduct_on_generators(0)
Traceback (most recent call last):
...
ValueError: Not a positive integer: 0
```

NonCommutativeSymmetricFunctions.MultiplicativeBasesOnGroupLikeElements.**super_categ**

Return the super categories of the category of multiplicative bases of group-like elements of the non-commutative symmetric functions.

OUTPUT:

- list

TESTS:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: N.MultiplicativeBasesOnGroupLikeElements().super_categories()
[Category of multiplicative bases of Non-Commutative Symmetric Functions over the Ration
```

class NonCommutativeSymmetricFunctions.**MultiplicativeBasesOnPrimitiveElements**(*parent_with_re*

Bases: sage.categories.realizations.Category_realization_of_parent

Category of multiplicative bases of the non-commutative symmetric functions whose generators are primitive elements.

An element x of a bialgebra is *primitive* if $\Delta(x) = x \otimes 1 + 1 \otimes x$, where Δ is the coproduct of the bialgebra.

Given a multiplicative basis and knowing the coproducts and antipodes of its generators, one can compute the coproduct and the antipode of any element, since they are respectively algebra morphisms and anti-morphisms. See `antipode_on_generators()` and `coproduct_on_generators()`.

Todo

this could be generalized to any free algebra.

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: N.MultiplicativeBasesOnPrimitiveElements()
Category of multiplicative bases on primitive elements of Non-Commutative Symmetric Functio
```

The Phi and Psi bases are multiplicative bases whose generators are primitive elements, but the complete and ribbon bases are not:

```
sage: N.Phi() in N.MultiplicativeBasesOnPrimitiveElements()
True
sage: N.Psi() in N.MultiplicativeBasesOnPrimitiveElements()
True
sage: N.Complete() in N.MultiplicativeBasesOnPrimitiveElements()
False
sage: N.Ribbon() in N.MultiplicativeBasesOnPrimitiveElements()
False
```

class ParentMethods

antipode_on_generators(i)

Return the image of a generator of a primitive basis of the non-commutative symmetric functions under the antipode map.

INPUT:

- `i` – a positive integer

OUTPUT:

- The image of the i-th generator of the multiplicative basis `self` under the antipode of the algebra of non-commutative symmetric functions.

EXAMPLES:

```
sage: Psi=NonCommutativeSymmetricFunctions(QQ).Psi()
sage: Psi.antipode_on_generators(2)
-Psi[2]
```

TESTS:

```
sage: Psi.antipode_on_generators(0)
Traceback (most recent call last):
...
ValueError: Not a positive integer: 0
```

coproduct_on_generators(i)

Return the image of the i^{th} generator of the multiplicative basis `self` under the coproduct.

INPUT:

- `i` – a positive integer

OUTPUT:

- The result of applying the coproduct to the i^{th} generator of `self`.

EXAMPLES:

```
sage: Psi = NonCommutativeSymmetricFunctions(QQ).Psi()
sage: Psi.coproduct_on_generators(3)
Psi[] # Psi[3] + Psi[3] # Psi[]
```

TESTS:

```
sage: Psi.coproduct_on_generators(0)
Traceback (most recent call last):
...
ValueError: Not a positive integer: 0
```

NonCommutativeSymmetricFunctions.MultiplicativeBasesOnPrimitiveElements.**super_categ**

Return the super categories of the category of multiplicative bases of primitive elements of the non-commutative symmetric functions.

OUTPUT:

- list

TESTS:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: N.MultiplicativeBasesOnPrimitiveElements().super_categories()
[Category of multiplicative bases of Non-Commutative Symmetric Functions over the Rationa
```

class NonCommutativeSymmetricFunctions.**Phi**(*NCSF*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The Hopf algebra of non-commutative symmetric functions in the Phi basis.

The Phi basis is defined in Definition 3.4 of [NCSF1], where it is denoted by $(\Phi^I)_I$. It is a multiplicative basis, and is connected to the elementary generators Λ_i of the ring of non-commutative symmetric functions by the following relation: Define a non-commutative symmetric function Φ_n for every positive integer n by the power series identity

$$\sum_{k\geq 1} t^k \frac{1}{k}\Phi_k = -\log\left(\sum_{k\geq 0}(-t)^k\Lambda_k\right),$$

with Λ_0 denoting 1. For every composition $(i_1, i_2, \ldots, i_k)$, we have $\Phi^{(i_1,i_2,\ldots,i_k)} = \Phi_{i_1}\Phi_{i_2}\cdots\Phi_{i_k}$.

The Φ-basis is well-defined only when the base ring is a $\mathbf{Q}$-algebra. The elements of the Φ-basis are known as the "power-sum non-commutative symmetric functions of the second kind".

The generators Φ_n are related to the (first) Eulerian idempotents in the descent algebras of the symmetric groups (see [NCSF1], 5.4 for details).

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: Phi = NCSF.Phi(); Phi
Non-Commutative Symmetric Functions over the Rational Field in the Phi basis
sage: Phi.an_element()
2*Phi[] + 2*Phi[1] + 3*Phi[1, 1]
```

class Element(*M*, *x*)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

psi_involution()

Return the image of the noncommutative symmetric function self under the involution ψ.

The involution ψ is defined as the linear map $NCSF \to NCSF$ which, for every composition I, sends the complete noncommutative symmetric function S^I to the elementary noncommutative symmetric function Λ^I. It can be shown that every composition I satisfies

$$\psi(R_I) = R_{I^c}, \quad \psi(S^I) = \Lambda^I, \quad \psi(\Lambda^I) = S^I, \quad \psi(\Phi^I) = (-1)^{|I|-\ell(I)}\Phi^I$$

where I^c denotes the complement of the composition I, and $\ell(I)$ denotes the length of I, and where standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, Φ for the basis of the power sums of the second kind, and

R for the ribbon basis). The map ψ is an involution and a graded Hopf algebra automorphism of $NCSF$. If π denotes the projection from $NCSF$ to the ring of symmetric functions (to_symmetric_function()), then $\pi(\psi(f)) = \omega(\pi(f))$ for every $f \in NCSF$, where the ω on the right hand side denotes the omega automorphism of Sym.

The involution ψ of $NCSF$ is adjoint to the involution ψ of $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The involution ψ has been denoted by ψ in [LMvW13], section 3.6.

See also:

psi involution of NCSF, psi involution of QSym, star involution of NCSF.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: Phi = NSym.Phi()
sage: Phi[3,2].psi_involution()
-Phi[3, 2]
sage: Phi[2,2].psi_involution()
Phi[2, 2]
sage: Phi[[]].psi_involution()
Phi[]
sage: (Phi[2,1] - 2*Phi[2]).psi_involution()
2*Phi[2] - Phi[2, 1]
sage: Phi(0).psi_involution()
0
```

The implementation at hand is tailored to the Phi basis. It is equivalent to the generic implementation via the ribbon basis:

```
sage: R = NSym.R()
sage: all( R(Phi[I].psi_involution()) == R(Phi[I]).psi_involution()
....:      for I in Compositions(4) )
True
```

star_involution()

Return the image of the noncommutative symmetric function self under the star involution.

The star involution is defined as the algebra antihomomorphism $NCSF \to NCSF$ which, for every positive integer n, sends the n-th complete non-commutative symmetric function S_n to S_n. Denoting by f^* the image of an element $f \in NCSF$ under this star involution, it can be shown that every composition I satisfies

$$(S^I)^* = S^{I^r}, \quad (\Lambda^I)^* = \Lambda^{I^r}, \quad R_I^* = R_{I^r}, \quad (\Phi^I)^* = \Phi^{I^r},$$

where I^r denotes the reversed composition of I, and standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, R for the ribbon basis, and Φ for that of the power-sums of the second kind). The star involution is an involution and a coalgebra automorphism of $NCSF$. It is an automorphism of the graded vector space $NCSF$. Under the canonical isomorphism between the n-th graded component of $NCSF$ and the descent algebra of the symmetric group S_n (see to_descent_algebra()), the star involution (restricted to the n-th graded component) corresponds to the automorphism of the descent algebra given by $x \mapsto \omega_n x \omega_n$, where ω_n is the permutation $(n, n-1, \ldots, 1) \in S_n$ (written here in one-line notation). If π denotes the projection from $NCSF$ to the ring of symmetric functions (to_symmetric_function()), then $\pi(f^*) = \pi(f)$ for every $f \in NCSF$.

The star involution on $NCSF$ is adjoint to the star involution on $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The star involution has been denoted by ρ in [LMvW13], section 3.6. See [NCSF2], section 2.3 for the properties of this map.

See also:

`star involution of NCSF`, `psi involution of NCSF`, `star involution of QSym`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: Phi = NSym.Phi()
sage: Phi[3,1,1,4].star_involution()
Phi[4, 1, 1, 3]
sage: Phi[4,2,1].star_involution()
Phi[1, 2, 4]
sage: (Phi[1,4] - Phi[2,3] + 2*Phi[5,4] - 3*Phi[3] + 4*Phi[[]]).star_involution()
4*Phi[] - 3*Phi[3] - Phi[3, 2] + Phi[4, 1] + 2*Phi[4, 5]
sage: (Phi[3,3] + 3*Phi[1]).star_involution()
3*Phi[1] + Phi[3, 3]
sage: Phi([2,1]).star_involution()
Phi[1, 2]
```

The implementation at hand is tailored to the Phi basis. It is equivalent to the generic implementation via the complete basis:

```
sage: S = NSym.S()
sage: all( S(Phi[I].star_involution()) == S(Phi[I]).star_involution()
....:      for I in Compositions(4) )
True
```

verschiebung(*n*)

Return the image of the noncommutative symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the map from the $\mathbf{k}$-algebra of noncommutative symmetric functions to itself that sends the complete function S^I indexed by a composition $I = (i_1, i_2, \ldots, i_k)$ to $S^{(i_1/n, i_2/n, \ldots, i_k/n)}$ if all of the numbers $i_1, i_2, \ldots, i_k$ are divisible by n, and to 0 otherwise. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every positive integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(S_r) = S_{r/n}, \quad \mathbf{V}_n(\Lambda_r) = (-1)^{r-r/n}\Lambda_{r/n}, \quad \mathbf{V}_n(\Psi_r) = n\Psi_{r/n}, \quad \mathbf{V}_n(\Phi_r) = n\Phi_{r/n}$$

(where S_r denotes the r-th complete non-commutative symmetric function, Λ_r denotes the r-th elementary non-commutative symmetric function, Ψ_r denotes the r-th power-sum non-commutative symmetric function of the first kind, and Φ_r denotes the r-th power-sum non-commutative symmetric function of the second kind). For every positive integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(S_r) = \mathbf{V}_n(\Lambda_r) = \mathbf{V}_n(\Psi_r) = \mathbf{V}_n(\Phi_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism.

It is a lift of the n-th Verschiebung operator on the ring of symmetric functions (`verschiebung()`) to the ring of noncommutative symmetric functions.

The action of the n-th Verschiebung operator can also be described on the ribbon Schur functions. Namely, every composition I of size $n\ell$ satisfies

$$\mathbf{V}_n(R_I) = (-1)^{\ell(I)-\ell(J)} \cdot R_{J/n},$$

where J denotes the meet of the compositions I and $(\underbrace{n, n, \ldots, n}_{|I|/n \text{ times}})$, where $\ell(I)$ is the length of I, and where J/n denotes the composition obtained by dividing every entry of J by n. For a composition I of size not divisible by n, we have $\mathbf{V}_n(R_I) = 0$.

See also:

`verschiebung method of NCSF`, `frobenius method of QSym`, `verschiebung method of Sym`

INPUT:

- n – a positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of noncommutative symmetric functions) to `self`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: Phi = NSym.Phi()
sage: Phi([4,2]).verschiebung(2)
4*Phi[2, 1]
sage: Phi([2,4]).verschiebung(2)
4*Phi[1, 2]
sage: Phi([6]).verschiebung(2)
2*Phi[3]
sage: Phi([2,1]).verschiebung(3)
0
sage: Phi([3]).verschiebung(2)
0
sage: Phi([]).verschiebung(2)
Phi[]
sage: Phi([5, 1]).verschiebung(3)
0
sage: Phi([5, 1]).verschiebung(6)
0
sage: Phi([5, 1]).verschiebung(2)
0
sage: Phi([1, 2, 3, 1]).verschiebung(7)
0
sage: Phi([7]).verschiebung(7)
7*Phi[1]
sage: Phi([1, 2, 3, 1]).verschiebung(5)
0
sage: (Phi[1] - Phi[2] + 2*Phi[3]).verschiebung(1)
Phi[1] - Phi[2] + 2*Phi[3]
```

TESTS:

The current implementation on the Phi basis gives the same results as the default implementation:

```
sage: NSym = NonCommutativeSymmetricFunctions(QQ)
sage: S = NSym.S()
sage: Phi = NSym.Phi()
sage: def test_phi(N, n):
....:     for I in Compositions(N):
....:         if S(Phi[I].verschiebung(n)) != S(Phi[I]).verschiebung(n):
....:             return False
....:     return True
sage: test_phi(4, 2)
```

```
True
sage: test_phi(6, 2)
True
sage: test_phi(6, 3)
True
sage: test_phi(8, 4)      # long time
True
```

class NonCommutativeSymmetricFunctions.**Psi**(*NCSF*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The Hopf algebra of non-commutative symmetric functions in the Psi basis.

The Psi basis is defined in Definition 3.4 of [NCSF1], where it is denoted by $(\Psi^I)_I$. It is a multiplicative basis, and is connected to the elementary generators Λ_i of the ring of non-commutative symmetric functions by the following relation: Define a non-commutative symmetric function Ψ_n for every positive integer n by the power series identity

$$\frac{d}{dt}\sigma(t) = \sigma(t) \cdot \left(\sum_{k \geq 1} t^{k-1} \Psi_k \right),$$

where

$$\sigma(t) = \left(\sum_{k \geq 0} (-t)^k \Lambda_k \right)^{-1}$$

and where Λ_0 denotes 1. For every composition $(i_1, i_2, \ldots, i_k)$, we have $\Psi^{(i_1,i_2,\ldots,i_k)} = \Psi_{i_1} \Psi_{i_2} \cdots \Psi_{i_k}$.

The Ψ-basis is a basis only when the base ring is a $\mathbf{Q}$-algebra (although the Ψ^I can be defined over any base ring). The elements of the Ψ-basis are known as the "power-sum non-commutative symmetric functions of the first kind". The generators Ψ_n correspond to the Dynkin (quasi-)idempotents in the descent algebras of the symmetric groups (see [NCSF1], 5.2 for details).

Another (equivalent) definition of Ψ_n is

$$\Psi_n = \sum_{i=0}^{n-1} (-1)^i R_{1^i, n-i},$$

where R denotes the ribbon basis of $NCSF$, and where 1^i stands for i repetitions of the integer 1.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: Psi = NCSF.Psi(); Psi
Non-Commutative Symmetric Functions over the Rational Field in the Psi basis
sage: Psi.an_element()
2*Psi[] + 2*Psi[1] + 3*Psi[1, 1]
```

Checking the equivalent definition of Ψ_n:

```
sage: def test_psi(n):
....:     NCSF = NonCommutativeSymmetricFunctions(ZZ)
....:     R = NCSF.R()
....:     Psi = NCSF.Psi()
....:     a = R.sum([(-1) ** i * R[[1]*i + [n-i]]
....:                for i in range(n)])
....:     return a == R(Psi[n])
```

```
sage: test_psi(2)
True
sage: test_psi(3)
True
sage: test_psi(4)
True
```

class Element(M, x)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

verschiebung(n)

Return the image of the noncommutative symmetric function self under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the map from the $\mathbf{k}$-algebra of noncommutative symmetric functions to itself that sends the complete function S^I indexed by a composition $I = (i_1, i_2, \ldots, i_k)$ to $S^{(i_1/n, i_2/n, \ldots, i_k/n)}$ if all of the numbers $i_1, i_2, \ldots, i_k$ are divisible by n, and to 0 otherwise. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every positive integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(S_r) = S_{r/n}, \quad \mathbf{V}_n(\Lambda_r) = (-1)^{r-r/n}\Lambda_{r/n}, \quad \mathbf{V}_n(\Psi_r) = n\Psi_{r/n}, \quad \mathbf{V}_n(\Phi_r) = n\Phi_{r/n}$$

(where S_r denotes the r-th complete non-commutative symmetric function, Λ_r denotes the r-th elementary non-commutative symmetric function, Ψ_r denotes the r-th power-sum non-commutative symmetric function of the first kind, and Φ_r denotes the r-th power-sum non-commutative symmetric function of the second kind). For every positive integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(S_r) = \mathbf{V}_n(\Lambda_r) = \mathbf{V}_n(\Psi_r) = \mathbf{V}_n(\Phi_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism.

It is a lift of the n-th Verschiebung operator on the ring of symmetric functions (verschiebung()) to the ring of noncommutative symmetric functions.

The action of the n-th Verschiebung operator can also be described on the ribbon Schur functions. Namely, every composition I of size $n\ell$ satisfies

$$\mathbf{V}_n(R_I) = (-1)^{\ell(I)-\ell(J)} \cdot R_{J/n},$$

where J denotes the meet of the compositions I and $\underbrace{(n, n, \ldots, n)}_{|I|/n \text{ times}}$, where $\ell(I)$ is the length of I, and where J/n denotes the composition obtained by dividing every entry of J by n. For a composition I of size not divisible by n, we have $\mathbf{V}_n(R_I) = 0$.

See also:

verschiebung method of NCSF, frobenius method of QSym, verschiebung method of Sym

INPUT:
- n – a positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of noncommutative symmetric functions) to `self`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: Psi = NSym.Psi()
sage: Psi([4,2]).verschiebung(2)
4*Psi[2, 1]
sage: Psi([2,4]).verschiebung(2)
4*Psi[1, 2]
sage: Psi([6]).verschiebung(2)
2*Psi[3]
sage: Psi([2,1]).verschiebung(3)
0
sage: Psi([3]).verschiebung(2)
0
sage: Psi([]).verschiebung(2)
Psi[]
sage: Psi([5, 1]).verschiebung(3)
0
sage: Psi([5, 1]).verschiebung(6)
0
sage: Psi([5, 1]).verschiebung(2)
0
sage: Psi([1, 2, 3, 1]).verschiebung(7)
0
sage: Psi([7]).verschiebung(7)
7*Psi[1]
sage: Psi([1, 2, 3, 1]).verschiebung(5)
0
sage: (Psi[1] - Psi[2] + 2*Psi[3]).verschiebung(1)
Psi[1] - Psi[2] + 2*Psi[3]
```

TESTS:

The current implementation on the Psi basis gives the same results as the default implementation:

```
sage: S = NSym.S()
sage: def test_psi(N, n):
....:     for I in Compositions(N):
....:         if S(Psi[I].verschiebung(n)) != S(Psi[I]).verschiebung(n):
....:             return False
....:     return True
sage: test_psi(4, 2)
True
sage: test_psi(6, 2)
True
sage: test_psi(6, 3)
True
sage: test_psi(8, 4)      # long time
True
```

NonCommutativeSymmetricFunctions.Psi.**internal_product_on_basis_by_bracketing**(*I*, *J*)

The internal product of two elements of the Psi basis.

See `internal_product()` for a thorough documentation of this operation.

This is an implementation using [NCSF2] Lemma 3.10. It is fast when the length of I is small, but can get very slow otherwise. Therefore it is not being used by default for internally multiplying Psi functions.

INPUT:

- I, J – compositions

OUTPUT:

- The internal product of the elements of the Psi basis of $NSym$ indexed by I and J, expressed in the Psi basis.

AUTHORS:

- Travis Scrimshaw, 29 Mar 2014

EXAMPLES:

```
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: Psi = N.Psi()
sage: Psi.internal_product_on_basis_by_bracketing([2,2],[1,2,1])
0
sage: Psi.internal_product_on_basis_by_bracketing([1,2,1],[2,1,1])
4*Psi[1, 2, 1]
sage: Psi.internal_product_on_basis_by_bracketing([2,1,1],[1,2,1])
4*Psi[2, 1, 1]
sage: Psi.internal_product_on_basis_by_bracketing([1,2,1], [1,1,1,1])
0
sage: Psi.internal_product_on_basis_by_bracketing([3,1], [1,2,1])
-Psi[1, 2, 1] + Psi[2, 1, 1]
sage: Psi.internal_product_on_basis_by_bracketing([1,2,1], [3,1])
0
sage: Psi.internal_product_on_basis_by_bracketing([2,2],[1,2])
0
sage: Psi.internal_product_on_basis_by_bracketing([4], [1,2,1])
-Psi[1, 1, 2] + 2*Psi[1, 2, 1] - Psi[2, 1, 1]
```

TESTS:

The internal product computed by this method is identical with the one obtained by coercion to the complete basis:

```
sage: S = N.S()
sage: def psi_int_test(n):
....:     for I in Compositions(n):
....:         for J in Compositions(n):
....:             a = S(Psi.internal_product_on_basis_by_bracketing(I, J))
....:             b = S(Psi[I]).internal_product(S(Psi[J]))
....:             if a != b:
....:                 return False
....:     return True
sage: all( psi_int_test(i) for i in range(4) )
True
sage: psi_int_test(4)   # long time
True
sage: psi_int_test(5)   # long time
True
```

class NonCommutativeSymmetricFunctions.**Ribbon**(*NCSF*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The Hopf algebra of non-commutative symmetric functions in the Ribbon basis.

The Ribbon basis is defined in Definition 3.12 of [NCSF1], where it is denoted by $(R_I)_I$. It is connected to the complete basis of the ring of non-commutative symmetric functions by the following relation: For

every composition I, we have

$$R_I = \sum_J (-1)^{\ell(I)-\ell(J)} S^J,$$

where the sum is over all compositions J which are coarser than I and $\ell(I)$ denotes the length of I. (See the proof of Proposition 4.13 in [NCSF1].)

The elements of the Ribbon basis are commonly referred to as the ribbon Schur functions.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: R = NCSF.Ribbon(); R
Non-Commutative Symmetric Functions over the Rational Field in the Ribbon basis
sage: R.an_element()
2*R[] + 2*R[1] + 3*R[1, 1]
```

The following are aliases for this basis:

```
sage: NCSF.ribbon()
Non-Commutative Symmetric Functions over the Rational Field in the Ribbon basis
sage: NCSF.R()
Non-Commutative Symmetric Functions over the Rational Field in the Ribbon basis
```

class Element(M, x)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

star_involution()

Return the image of the noncommutative symmetric function self under the star involution.

The star involution is defined as the algebra antihomomorphism $NCSF \to NCSF$ which, for every positive integer n, sends the n-th complete non-commutative symmetric function S_n to S_n. Denoting by f^* the image of an element $f \in NCSF$ under this star involution, it can be shown that every composition I satisfies

$$(S^I)^* = S^{I^r}, \quad (\Lambda^I)^* = \Lambda^{I^r}, \quad R_I^* = R_{I^r}, \quad (\Phi^I)^* = \Phi^{I^r},$$

where I^r denotes the reversed composition of I, and standard notations for classical bases of $NCSF$ are being used (S for the complete basis, Λ for the elementary basis, R for the ribbon basis, and Φ for that of the power-sums of the second kind). The star involution is an involution and a coalgebra automorphism of $NCSF$. It is an automorphism of the graded vector space $NCSF$. Under the canonical isomorphism between the n-th graded component of $NCSF$ and the descent algebra of the symmetric group S_n (see to_descent_algebra()), the star involution (restricted to the n-th graded component) corresponds to the automorphism of the descent algebra given by $x \mapsto \omega_n x \omega_n$, where ω_n is the permutation $(n, n-1, \ldots, 1) \in S_n$ (written here in one-line notation). If π denotes the projection from $NCSF$ to the ring of symmetric functions (to_symmetric_function()), then $\pi(f^*) = \pi(f)$ for every $f \in NCSF$.

The star involution on $NCSF$ is adjoint to the star involution on $QSym$ by the standard adjunction between $NCSF$ and $QSym$.

The star involution has been denoted by ρ in [LMvW13], section 3.6. See [NCSF2], section 2.3 for the properties of this map.

See also:

`star involution of NCSF`, `star involution of QSym`, `psi involution of NCSF`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: R = NSym.R()
sage: R[3,1,4,2].star_involution()
R[2, 4, 1, 3]
sage: R[4,1,2].star_involution()
R[2, 1, 4]
sage: (R[1] - R[2] + 2*R[5,4] - 3*R[3] + 4*R[[]]).star_involution()
4*R[] + R[1] - R[2] - 3*R[3] + 2*R[4, 5]
sage: (R[3,3] - 21*R[1]).star_involution()
-21*R[1] + R[3, 3]
sage: R([14,1]).star_involution()
R[1, 14]
```

The implementation at hand is tailored to the ribbon basis. It is equivalent to the generic implementation via the complete basis:

```
sage: S = NSym.S()
sage: all( S(R[I].star_involution()) == S(R[I]).star_involution()
....:      for I in Compositions(4) )
True
```

verschiebung(*n*)

Return the image of the noncommutative symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the map from the $\mathbf{k}$-algebra of noncommutative symmetric functions to itself that sends the complete function S^I indexed by a composition $I = (i_1, i_2, \ldots, i_k)$ to $S^{(i_1/n, i_2/n, \ldots, i_k/n)}$ if all of the numbers $i_1, i_2, \ldots, i_k$ are divisible by n, and to 0 otherwise. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every positive integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(S_r) = S_{r/n}, \quad \mathbf{V}_n(\Lambda_r) = (-1)^{r-r/n}\Lambda_{r/n}, \quad \mathbf{V}_n(\Psi_r) = n\Psi_{r/n}, \quad \mathbf{V}_n(\Phi_r) = n\Phi_{r/n}$$

(where S_r denotes the r-th complete non-commutative symmetric function, Λ_r denotes the r-th elementary non-commutative symmetric function, Ψ_r denotes the r-th power-sum non-commutative symmetric function of the first kind, and Φ_r denotes the r-th power-sum non-commutative symmetric function of the second kind). For every positive integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(S_r) = \mathbf{V}_n(\Lambda_r) = \mathbf{V}_n(\Psi_r) = \mathbf{V}_n(\Phi_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism.

It is a lift of the n-th Verschiebung operator on the ring of symmetric functions (`verschiebung()`) to the ring of noncommutative symmetric functions.

The action of the n-th Verschiebung operator can also be described on the ribbon Schur functions. Namely, every composition I of size $n\ell$ satisfies

$$\mathbf{V}_n(R_I) = (-1)^{\ell(I)-\ell(J)} \cdot R_{J/n},$$

where J denotes the meet of the compositions I and $(\underbrace{n, n, \ldots, n}_{|I|/n \text{ times}})$, where $\ell(I)$ is the length of I, and where J/n denotes the composition obtained by dividing every entry of J by n. For a composition I of size not divisible by n, we have $\mathbf{V}_n(R_I) = 0$.

See also:

`verschiebung method of NCSF`, `frobenius method of QSym`, `verschiebung method of Sym`

INPUT:

- n – a positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of noncommutative symmetric functions) to `self`.

EXAMPLES:

```
sage: NSym = NonCommutativeSymmetricFunctions(ZZ)
sage: R = NSym.R()
sage: R([4,2]).verschiebung(2)
R[2, 1]
sage: R([2,1]).verschiebung(3)
-R[1]
sage: R([3]).verschiebung(2)
0
sage: R([]).verschiebung(2)
R[]
sage: R([5, 1]).verschiebung(3)
-R[2]
sage: R([5, 1]).verschiebung(6)
-R[1]
sage: R([5, 1]).verschiebung(2)
-R[3]
sage: R([1, 2, 3, 1]).verschiebung(7)
-R[1]
sage: R([1, 2, 3, 1]).verschiebung(5)
0
sage: (R[1] - R[2] + 2*R[3]).verschiebung(1)
R[1] - R[2] + 2*R[3]
```

TESTS:

The current implementation on the ribbon basis gives the same results as the default implementation:

```
sage: S = NSym.S()
sage: def test_ribbon(N, n):
....:     for I in Compositions(N):
....:         if S(R[I].verschiebung(n)) != S(R[I]).verschiebung(n):
....:             return False
....:     return True
sage: test_ribbon(4, 2)
True
sage: test_ribbon(6, 2)
True
sage: test_ribbon(6, 3)
True
sage: test_ribbon(8, 4)      # long time
True
```

NonCommutativeSymmetricFunctions.Ribbon.**antipode_on_basis**(*composition*)

Return the application of the antipode to a basis element of the ribbon basis `self`.

INPUT:

- `composition` – a composition

OUTPUT:

- The image of the basis element indexed by `composition` under the antipode map.

EXAMPLES:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: R.antipode_on_basis(Composition([2,1]))
-R[2, 1]
sage: R[3,1].antipode() # indirect doctest
R[2, 1, 1]
sage: R[[]].antipode() # indirect doctest
R[]
```

We check that the implementation of the antipode at hand does not contradict the generic one:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: all( S(R[I].antipode()) == S(R[I]).antipode()
....:      for I in Compositions(4) )
True
```

NonCommutativeSymmetricFunctions.Ribbon.**dual**()

Return the dual basis to the ribbon basis of the non-commutative symmetric functions. This is the Fundamental basis of the quasi-symmetric functions.

OUTPUT:

- The fundamental basis of the quasi-symmetric functions.

EXAMPLES:

```
sage: R=NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: R.dual()
Quasisymmetric functions over the Rational Field in the Fundamental basis
```

NonCommutativeSymmetricFunctions.Ribbon.**product_on_basis**(*I*, *J*)

Return the product of two ribbon basis elements of the non-commutative symmetric functions.

INPUT:

- `I`, `J` – compositions

OUTPUT:

- The product of the ribbon functions indexed by `I` and `J`.

EXAMPLES:

```
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: R[1,2,1] * R[3,1]
R[1, 2, 1, 3, 1] + R[1, 2, 4, 1]
sage: ( R[1,2] + R[3] ) * ( R[3,1] + R[1,2,1] )
R[1, 2, 1, 2, 1] + R[1, 2, 3, 1] + R[1, 3, 2, 1] + R[1, 5, 1] + R[3, 1, 2, 1] + R[3, 3,
```

TESTS:

```
sage: R[[]] * R[3,1]
R[3, 1]
sage: R[1,2,1] * R[[]]
R[1, 2, 1]
sage: R.product_on_basis(Composition([2,1]), Composition([1]))
R[2, 1, 1] + R[2, 2]
```

NonCommutativeSymmetricFunctions.Ribbon.**to_symmetric_function_on_basis**(*I*)

Return the commutative image of a ribbon basis element of the non-commutative symmetric functions.

INPUT:

- I – a composition

OUTPUT:

- The commutative image of the ribbon basis element indexed by I. This will be expressed as a symmetric function in the Schur basis.

EXAMPLES:

```
sage: R=NonCommutativeSymmetricFunctions(QQ).R()
sage: R.to_symmetric_function_on_basis(Composition([3,1,1]))
s[3, 1, 1]
sage: R.to_symmetric_function_on_basis(Composition([4,2,1]))
s[4, 2, 1] + s[5, 1, 1] + s[5, 2]
sage: R.to_symmetric_function_on_basis(Composition([]))
s[]
```

NonCommutativeSymmetricFunctions.**a_realization**()

Gives a realization of the algebra of non-commutative symmetric functions. This particular realization is the complete basis of non-commutative symmetric functions.

OUTPUT:

- The realization of the non-commutative symmetric functions in the complete basis.

EXAMPLES:

```
sage: NonCommutativeSymmetricFunctions(ZZ).a_realization()
Non-Commutative Symmetric Functions over the Integer Ring in the Complete basis
```

NonCommutativeSymmetricFunctions.**dual**()

Return the dual to the non-commutative symmetric functions.

OUTPUT:

- The dual of the non-commutative symmetric functions over a ring. This is the algebra of quasi-symmetric functions over the ring.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: NCSF.dual()
Quasisymmetric functions over the Rational Field
```

class NonCommutativeSymmetricFunctions.**dualQuasisymmetric_Schur**(*NCSF*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The basis of NCSF dual to the Quasisymmetric-Schur basis of QSym.

The Quasisymmetric_Schur functions are defined in [QSCHUR] (see also Definition 5.1.1 of [LMvW13]). The dual basis in the algebra of non-commutative symmetric functions is defined by the following formula:

$$R_\alpha = \sum_T dQS_{shape(T)},$$

where the sum is over all standard composition tableaux with descent composition equal to α. The Quasisymmetric_Schur basis QS_α has the property that

$$s_\lambda = \sum_{sort(\alpha)=\lambda} QS_\alpha.$$

As a consequence the commutative image of a dual Quasisymmetric-Schur element in the algebra of symmetric functions (the map defined in the method to_symmetric_function()) is equal to the Schur function indexed by the decreasing sort of the indexing composition.

See also:

CompositionTableaux, CompositionTableau.

EXAMPLES:

```
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: dQS = NCSF.dQS()
sage: dQS([1,3,2])*dQS([1])
dQS[1, 2, 4] + dQS[1, 3, 2, 1] + dQS[1, 3, 3] + dQS[3, 2, 2]
sage: dQS([1])*dQS([1,3,2])
dQS[1, 1, 3, 2] + dQS[1, 3, 3] + dQS[1, 4, 2] + dQS[2, 3, 2]
sage: dQS([1,3])*dQS([1,1])
dQS[1, 3, 1, 1] + dQS[1, 4, 1] + dQS[3, 2, 1] + dQS[4, 2]
sage: dQS([3,1])*dQS([2,1])
dQS[1, 1, 4, 1] + dQS[1, 4, 2] + dQS[1, 5, 1] + dQS[2, 4, 1] + dQS[3, 1,
2, 1] + dQS[3, 2, 2] + dQS[3, 3, 1] + dQS[4, 3] + dQS[5, 2]
sage: dQS([1,1]).coproduct()
dQS[] # dQS[1, 1] + dQS[1] # dQS[1] + dQS[1, 1] # dQS[]
sage: dQS([3,3]).coproduct().monomial_coefficients()[(Composition([1,2]),Composition([1,2]))
-1
sage: S = NCSF.complete()
sage: dQS(S[1,3,1])
dQS[1, 3, 1] + dQS[1, 4] + dQS[3, 2] + dQS[4, 1] + dQS[5]
sage: S(dQS[1,3,1])
S[1, 3, 1] - S[3, 2] - S[4, 1] + S[5]
sage: s = SymmetricFunctions(QQ).s()
sage: s(S(dQS([2,1,3])).to_symmetric_function())
s[3, 2, 1]
```

dual()

The dual basis to the dual Quasisymmetric-Schur basis of NCSF.

The basis returned is the Quasisymmetric_Schur basis of QSym.

OUTPUT:

- the Quasisymmetric-Schur basis of the quasi-symmetric functions

EXAMPLES:

```
sage: dQS=NonCommutativeSymmetricFunctions(QQ).dualQuasisymmetric_Schur()
sage: dQS.dual()
Quasisymmetric functions over the Rational Field in the Quasisymmetric
Schur basis
sage: dQS.duality_pairing_matrix(dQS.dual(),3)
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

to_symmetric_function_on_basis(*I*)

The commutative image of a dual quasi-symmetric Schur element

The commutative image of a basis element is obtained by sorting the indexing composition of the basis element.

INPUT:

- I – a composition

OUTPUT:

- The commutative image of the dual quasi-Schur basis element indexed by I. The result is the Schur symmetric function indexed by the partition obtained by sorting I.

EXAMPLES:

```
sage: dQS=NonCommutativeSymmetricFunctions(QQ).dQS()
sage: dQS.to_symmetric_function_on_basis([2,1,3])
s[3, 2, 1]
sage: dQS.to_symmetric_function_on_basis([])
s[]
```

5.1.130 Quasisymmetric functions

REFERENCES:

AUTHOR:

- Jason Bandlow
- Franco Saliola
- Chris Berg
- Darij Grinberg

class sage.combinat.ncsf_qsym.qsym.**QuasiSymmetricFunctions**(R)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The Hopf algebra of quasisymmetric functions.

Let R be a commutative ring with unity. The R-algebra of quasi-symmetric functions may be realized as an R-subalgebra of the ring of power series in countably many variables $R[[x_1, x_2, x_3, \ldots]]$. It consists of those formal power series p which are degree-bounded (i. e., the degrees of all monomials occuring with nonzero coefficient in p are bounded from above, although the bound can depend on p) and satisfy the following condition: For every tuple $(a_1, a_2, \ldots, a_m)$ of positive integers, the coefficient of the monomial $x_{i_1}^{a_1} x_{i_2}^{a_2} \cdots x_{i_m}^{a_m}$ in p is the same for all strictly increasing sequences $(i_1 < i_2 < \cdots < i_m)$ of positive integers. (In other words, the coefficient of a monomial in p depends only on the sequence of nonzero exponents in the monomial. If "sequence" were to be replaced by "multiset" here, we would obtain the definition of a symmetric function.)

The R-algebra of quasi-symmetric functions is commonly called QSym_R or occasionally just QSym (when R is clear from the context or $\mathbf{Z}$ or $\mathbf{Q}$). It is graded by the total degree of the power series. Its homogeneous elements of degree k form a free R-submodule of rank equal to the number of compositions of k (that is, 2^{k-1} if $k \geq 1$, else 1).

The two classical bases of QSym, the monomial basis $(M_I)_I$ and the fundamental basis $(F_I)_I$, are indexed by compositions $I = (I_1, I_2, \cdots, I_\ell)$ and defined by the formulas:

$$M_I = \sum_{1 \leq i_1 < i_2 < \cdots < i_\ell} x_{i_1}^{I_1} x_{i_2}^{I_2} \cdots x_{i_\ell}^{I_\ell}$$

and

$$F_I = \sum_{(j_1, j_2, \ldots, j_n)} x_{j_1} x_{j_2} \cdots x_{j_n}$$

where in the second equation the sum runs over all weakly increasing n-tuples $(j_1, j_2, \ldots, j_n)$ of positive integers (where n is the size of I) which increase strictly from j_r to j_{r+1} if r is a descent of the composition I.

These bases are related by the formula

$F_I = \sum_{J \leq I} M_J$

where the inequality $J \leq I$ indicates that J is finer than I.

The R-algebra of quasi-symmetric functions is a Hopf algebra, with the coproduct satisfying

$$\Delta M_I = \sum_{k=0}^{\ell} M_{(I_1,I_2,\cdots,I_k)} \otimes M_{(I_{k+1},I_{k+2},\cdots,I_\ell)}$$

for every composition $I = (I_1, I_2, \cdots, I_\ell)$.

It is possible to define an R-algebra of quasi-symmetric functions in a finite number of variables as well (but it is not a bialgebra). These quasi-symmetric functions are actual polynomials then, not just power series.

Chapter 5 of [GriRei2014] and Section 11 of [HazWitt1] are devoted to quasi-symmetric functions, as are Malvenuto's thesis [Mal1993] and part of Chapter 7 of [Sta-EC2].

The implementation of the quasi-symmetric function Hopf algebra

We realize the R-algebra of quasi-symmetric functions in Sage as a graded Hopf algebra with basis elements indexed by compositions:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: QSym.category()
Join of Category of hopf algebras over Rational Field
    and Category of graded algebras over Rational Field
    and Category of monoids with realizations
    and Category of coalgebras over Rational Field with realizations
```

The most standard two bases for this R-algebra are the monomial and fundamental bases, and are accessible by the `M()` and `F()` methods:

```
sage: M = QSym.M()
sage: F = QSym.F()
sage: M(F[2,1,2])
M[1, 1, 1, 1, 1] + M[1, 1, 1, 2] + M[2, 1, 1, 1] + M[2, 1, 2]
sage: F(M[2,1,2])
F[1, 1, 1, 1, 1] - F[1, 1, 1, 2] - F[2, 1, 1, 1] + F[2, 1, 2]
```

The product on this space is commutative and is inherited from the product on the realization within the ring of power series:

```
sage: M[3]*M[1,1] == M[1,1]*M[3]
True
sage: M[3]*M[1,1]
M[1, 1, 3] + M[1, 3, 1] + M[1, 4] + M[3, 1, 1] + M[4, 1]
sage: F[3]*F[1,1]
F[1, 1, 3] + F[1, 2, 2] + F[1, 3, 1] + F[1, 4] + F[2, 1, 2]
 + F[2, 2, 1] + F[2, 3] + F[3, 1, 1] + F[3, 2] + F[4, 1]
sage: M[3]*F[2]
M[1, 1, 3] + M[1, 3, 1] + M[1, 4] + M[2, 3] + M[3, 1, 1] + M[3, 2]
 + M[4, 1] + M[5]
sage: F[2]*M[3]
F[1, 1, 1, 2] - F[1, 2, 2] + F[2, 1, 1, 1] - F[2, 1, 2] - F[2, 2, 1]
 + F[5]
```

There is a coproduct on QSym as well, which in the Monomial basis acts by cutting the composition into a left half and a right half. The coproduct is not co-commutative:

```
sage: M[1,3,1].coproduct()
M[] # M[1, 3, 1] + M[1] # M[3, 1] + M[1, 3] # M[1] + M[1, 3, 1] # M[]
```

```
sage: F[1,3,1].coproduct()
F[] # F[1, 3, 1] + F[1] # F[3, 1] + F[1, 1] # F[2, 1]
 + F[1, 2] # F[1, 1] + F[1, 3] # F[1] + F[1, 3, 1] # F[]
```

The duality pairing with non-commutative symmetric functions

These two operations endow the quasi-symmetric functions QSym with the structure of a Hopf algebra. It is the graded dual Hopf algebra of the non-commutative symmetric functions $NCSF$. Under this duality, the Monomial basis of QSym is dual to the Complete basis of $NCSF$, and the Fundamental basis of QSym is dual to the Ribbon basis of $NCSF$ (see [MR]).

```
sage: S = M.dual(); S
Non-Commutative Symmetric Functions over the Rational Field in the Complete basis
sage: M[1,3,1].duality_pairing( S[1,3,1] )
1
sage: M.duality_pairing_matrix( S, degree=4 )
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
sage: F.duality_pairing_matrix( S, degree=4 )
[1 0 0 0 0 0 0 0]
[1 1 0 0 0 0 0 0]
[1 0 1 0 0 0 0 0]
[1 1 1 1 0 0 0 0]
[1 0 0 0 1 0 0 0]
[1 1 0 0 1 1 0 0]
[1 0 1 0 1 0 1 0]
[1 1 1 1 1 1 1 1]
sage: NCSF = M.realization_of().dual()
sage: R = NCSF.Ribbon()
sage: F.duality_pairing_matrix( R, degree=4 )
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
sage: M.duality_pairing_matrix( R, degree=4 )
[ 1  0  0  0  0  0  0  0]
[-1  1  0  0  0  0  0  0]
[-1  0  1  0  0  0  0  0]
[ 1 -1 -1  1  0  0  0  0]
[-1  0  0  0  1  0  0  0]
[ 1 -1  0  0 -1  1  0  0]
[ 1  0 -1  0 -1  0  1  0]
[-1  1  1 -1  1 -1 -1  1]
```

Let H and G be elements of QSym, and h an element of $NCSF$. Then, if we represent the duality pairing with the mathematical notation $[\cdot, \cdot]$,

$[HG, h] = [H \otimes G, \Delta(h)]$.

For example, the coefficient of `M[2,1,4,1]` in `M[1,3]*M[2,1,1]` may be computed with the duality pairing:

```
sage: I, J = Composition([1,3]), Composition([2,1,1])
sage: (M[I]*M[J]).duality_pairing(S[2,1,4,1])
1
```

And the coefficient of `S[1,3] # S[2,1,1]` in `S[2,1,4,1].coproduct()` is equal to this result:

```
sage: S[2,1,4,1].coproduct()
S[] # S[2, 1, 4, 1] + ... + S[1, 3] # S[2, 1, 1] + ... + S[4, 1] # S[2, 1]
```

The duality pairing on the tensor space is another way of getting this coefficient, but currently the method `duality_pairing` is not defined on the tensor squared space. However, we can extend this functionality by applying a linear morphism to the terms in the coproduct, as follows:

```
sage: X = S[2,1,4,1].coproduct()
sage: def linear_morphism(x, y):
....:     return x.duality_pairing(M[1,3]) * y.duality_pairing(M[2,1,1])
sage: X.apply_multilinear_morphism(linear_morphism, codomain=ZZ)
1
```

Similarly, if H is an element of QSym and g and h are elements of $NCSF$, then

$$[H, gh] = [\Delta(H), g \otimes h].$$

For example, the coefficient of `R[2,3,1]` in `R[2,1]*R[2,1]` is computed with the duality pairing by the following command:

```
sage: (R[2,1]*R[2,1]).duality_pairing(F[2,3,1])
1
sage: R[2,1]*R[2,1]
R[2, 1, 2, 1] + R[2, 3, 1]
```

This coefficient should then be equal to the coefficient of `F[2,1] # F[2,1]` in `F[2,3,1].coproduct()`:

```
sage: F[2,3,1].coproduct()
F[] # F[2, 3, 1] + ... + F[2, 1] # F[2, 1]  + ... + F[2, 3, 1] # F[]
```

This can also be computed by the duality pairing on the tensor space, as above:

```
sage: X = F[2,3,1].coproduct()
sage: def linear_morphism(x, y):
....:     return x.duality_pairing(R[2,1]) * y.duality_pairing(R[2,1])
sage: X.apply_multilinear_morphism(linear_morphism, codomain=ZZ)
1
```

The operation dual to multiplication by a non-commutative symmetric function

Let $g \in NCSF$ and consider the linear endomorphism of $NCSF$ defined by left (respectively, right) multiplication by g. Since there is a duality between QSym and $NCSF$, this linear transformation induces an operator $g^{\perp}$ on QSym satisfying

$$[g^{\perp}(H), h] = [H, gh].$$

for any non-commutative symmetric function h.

This is implemented by the method `skew_by()`. Explicitly, if `H` is a quasi-symmetric function and `g` a non-commutative symmetric function, then `H.skew_by(g)` and `H.skew_by(g, side='right')` are expressions that satisfy, for any non-commutative symmetric function `h`, the following equalities:

```
H.skew_by(g).duality_pairing(h) == H.duality_pairing(g*h)
H.skew_by(g, side='right').duality_pairing(h) == H.duality_pairing(h*g)
```

For example, `M[J].skew_by(S[I])` is 0 unless the composition `J` begins with `I` and `M(J).skew_by(S(I), side='right')` is 0 unless the composition `J` ends with `I`. For example:

```
sage: M[3,2,2].skew_by(S[3])
M[2, 2]
sage: M[3,2,2].skew_by(S[2])
0
sage: M[3,2,2].coproduct().apply_multilinear_morphism( lambda x,y: x.duality_pairing(S[3])*y )
M[2, 2]
sage: M[3,2,2].skew_by(S[3], side='right')
0
sage: M[3,2,2].skew_by(S[2], side='right')
M[3, 2]
```

The counit

The counit is defined by sending all elements of positive degree to zero:

```
sage: M[3].degree(), M[3,1,2].degree(), M.one().degree()
(3, 6, 0)
sage: M[3].counit()
0
sage: M[3,1,2].counit()
0
sage: M.one().counit()
1
sage: (M[3] - 2*M[3,1,2] + 7).counit()
7
sage: (F[3] - 2*F[3,1,2] + 7).counit()
7
```

The antipode

The antipode sends the Fundamental basis element indexed by the composition I to $(-1)^{|I|}$ times the Fundamental basis element indexed by the conjugate composition to I (where $|I|$ stands for the size of I, that is, the sum of all entries of I).

```
sage: F[3,2,2].antipode()
-F[1, 2, 2, 1, 1]
sage: Composition([3,2,2]).conjugate()
[1, 2, 2, 1, 1]
```

The antipodes of the Monomial quasisymmetric functions can also be computed easily: Every composition I satisfies

$$\omega(M_I) = (-1)^{\ell(I)} \sum M_J,$$

where the sum ranges over all compositions J of $|I|$ which are coarser than the reversed composition I^r of I. Here, $\ell(I)$ denotes the length of the composition I (that is, the number of its parts).

```
sage: M[3,2,1].antipode()
-M[1, 2, 3] - M[1, 5] - M[3, 3] - M[6]
sage: M[3,2,2].antipode()
-M[2, 2, 3] - M[2, 5] - M[4, 3] - M[7]
```

We demonstrate here the defining relation of the antipode:

```
sage: X = F[3,2,2].coproduct()
sage: X.apply_multilinear_morphism(lambda x,y: x*y.antipode())
0
sage: X.apply_multilinear_morphism(lambda x,y: x.antipode()*y)
0
```

The relation with symmetric functions

The quasi-symmetric functions are a ring which contain the symmetric functions as a subring. The Monomial quasi-symmetric functions are related to the monomial symmetric functions by

$$m_\lambda = \sum_{\mathrm{sort}(I)=\lambda} M_I$$

(where $\mathrm{sort}(I)$ denotes the result of sorting the entries of I in decreasing order).

There are methods to test if an expression in the quasi-symmetric functions is a symmetric function and, if it is, send it to an expression in the symmetric functions:

```
sage: f = M[1,1,2] + M[1,2,1]
sage: f.is_symmetric()
False
sage: g = M[3,1] + M[1,3]
sage: g.is_symmetric()
True
sage: g.to_symmetric_function()
m[3, 1]
```

The expansion of the Schur function in terms of the Fundamental quasi-symmetric functions is due to [Ges]. There is one term in the expansion for each standard tableau of shape equal to the partition indexing the Schur function.

```
sage: f = F[3,2] + F[2,2,1] + F[2,3] + F[1,3,1] + F[1,2,2]
sage: f.is_symmetric()
True
sage: f.to_symmetric_function()
5*m[1, 1, 1, 1, 1] + 3*m[2, 1, 1, 1] + 2*m[2, 2, 1] + m[3, 1, 1] + m[3, 2]
sage: s = SymmetricFunctions(QQ).s()
sage: s(f.to_symmetric_function())
s[3, 2]
```

It is also possible to convert a symmetric function to a quasi-symmetric function:

```
sage: m = SymmetricFunctions(QQ).m()
sage: M( m[3,1,1] )
M[1, 1, 3] + M[1, 3, 1] + M[3, 1, 1]
sage: F( s[2,2,1] )
F[1, 1, 2, 1] + F[1, 2, 1, 1] + F[1, 2, 2] + F[2, 1, 2] + F[2, 2, 1]
```

It is possible to experiment with the quasi-symmetric function expansion of other bases, but it is important that the base ring be the same for both algebras.

```
sage: R = QQ['t']
sage: Qp = SymmetricFunctions(R).hall_littlewood().Qp()
sage: QSymt = QuasiSymmetricFunctions(R)
sage: Ft = QSymt.F()
sage: Ft( Qp[2,2] )
F[1, 2, 1] + t*F[1, 3] + (t+1)*F[2, 2] + t*F[3, 1] + t^2*F[4]

sage: K = QQ['q','t'].fraction_field()
sage: Ht = SymmetricFunctions(K).macdonald().Ht()
sage: Fqt = QuasiSymmetricFunctions(Ht.base_ring()).F()
sage: Fqt(Ht[2,1])
q*t*F[1, 1, 1] + (q+t)*F[1, 2] + (q+t)*F[2, 1] + F[3]
```

The following will raise an error because the base ring of `F` is not equal to the base ring of `Ht`:

```
sage: F(Ht[2,1])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= McdHt[2, 1]) an element of self (=Quasisymmetric functic
```

The map to the ring of polynomials

The quasi-symmetric functions can be seen as an inverse limit of a subring of a polynomial ring as the number of variables increases. Indeed, there exists a projection from the quasi-symmetric functions onto the polynomial ring $R[x_1, x_2, \ldots, x_n]$. This projection is defined by sending the variables $x_{n+1}, x_{n+2}, \cdots$ to 0, while the remaining n variables remain fixed. Note that this projection sends M_I to 0 if the length of the composition I is higher than n.

```
sage: M[1,3,1].expand(4)
x0*x1^3*x2 + x0*x1^3*x3 + x0*x2^3*x3 + x1*x2^3*x3
sage: F[1,3,1].expand(4)
x0*x1^3*x2 + x0*x1^3*x3 + x0*x1^2*x2*x3 + x0*x1*x2^2*x3 + x0*x2^3*x3 + x1*x2^3*x3
sage: M[1,3,1].expand(2)
0
```

TESTS:

```
sage: QSym = QuasiSymmetricFunctions(QQ); QSym
Quasisymmetric functions over the Rational Field
sage: QSym.base_ring()
Rational Field
```

class Bases (*parent_with_realization*)

Bases: `sage.categories.realizations.Category_realization_of_parent`

Category of bases of quasi-symmetric functions.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: QSym.Bases()
Category of bases of Quasisymmetric functions over the Rational Field
```

class ElementMethods

Methods common to all elements of `QuasiSymmetricFunctions`.

adams_operation(**args*, ***opts*)

dendriform_leq(*other*)

Return the result of applying the dendriform smaller-or-equal operation to the two quasi-symmetric functions `self` and `other`.

The dendriform smaller-or-equal operation is a binary operation, denoted by $\preceq$ and written infix, on the ring of quasi-symmetric functions. It can be defined as a restriction of a binary operation (denoted by $\preceq$ and written infix as well) on the ring of formal power series $R[[x_1, x_2, x_3, \ldots]]$, which is defined as follows: If m and n are two monomials in $x_1, x_2, x_3, \ldots$, then we let $m \preceq n$ be the product mn if the smallest positive integer i for which x_i occurs in m is smaller or equal to the smallest positive integer j for which x_j occurs in n (this is understood to be false when $m = 1$ and $n \neq 1$, and true when $n = 1$), and we let $m \preceq n$ be 0 otherwise. Having thus defined $\preceq$ on monomials, we extend $\preceq$ to a binary operation on $R[[x_1, x_2, x_3, \ldots]]$ by requiring it to be continuous (in both inputs) and R-bilinear. It is easily seen that $QSym \preceq QSym \subseteq QSym$, so that $\preceq$ restricts to a binary operation on $QSym$.

This operation $\preceq$ is related to the dendriform smaller relation $\prec$ (`dendriform_less()`). Namely, if we define a binary operation $\succ$ on $QSym$ by $a \succ b = b \prec a$, then $(QSym, \preceq, \succ)$ is a dendriform R-algebra. Thus, any $a, b \in QSym$ satisfy $a \preceq b = ab - b \prec a$.

See also:

`dendriform_less()`

INPUT:

- `other` – a quasi-symmetric function over the same base ring as `self`

OUTPUT:

The quasi-symmetric function `self` $\preceq$ `other`, written in the basis of `self`.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.M()
sage: M[2, 1].dendriform_leq(M[1, 2])
2*M[2, 1, 1, 2] + M[2, 1, 2, 1] + M[2, 1, 3] + M[2, 2, 2]
 + M[3, 1, 2] + M[3, 2, 1] + M[3, 3]
sage: F = QSym.F()
sage: F[2, 1].dendriform_leq(F[1, 2])
F[2, 1, 1, 2] + F[2, 1, 2, 1] + F[2, 1, 3] + F[2, 2, 1, 1]
 + 2*F[2, 2, 2] + F[2, 3, 1] + F[3, 1, 2] + F[3, 2, 1] + F[3, 3]
```

dendriform_less(*other*)

Return the result of applying the dendriform smaller operation to the two quasi-symmetric functions `self` and `other`.

The dendriform smaller operation is a binary operation, denoted by $\prec$ and written infix, on the ring of quasi-symmetric functions. It can be defined as a restriction of a binary operation (denoted by $\prec$ and written infix as well) on the ring of formal power series $R[[x_1, x_2, x_3, \ldots]]$, which is defined as follows: If m and n are two monomials in $x_1, x_2, x_3, \ldots$, then we let $m \prec n$ be the product mn if the smallest positive integer i for which x_i occurs in m is smaller than the smallest positive integer j for which x_j occurs in n (this is understood to be false when $m = 1$, and true when $m \neq 1$ and $n = 1$), and we let $m \prec n$ be 0 otherwise. Having thus defined $\prec$ on monomials, we extend $\prec$ to a binary operation on $R[[x_1, x_2, x_3, \ldots]]$ by requiring it to be continuous (in both inputs) and R-bilinear. It is easily seen that $QSym \prec QSym \subseteq QSym$, so that $\prec$ restricts to a binary operation on $QSym$.

See also:

`dendriform_leq()`

INPUT:

•other – a quasi-symmetric function over the same base ring as self

OUTPUT:

The quasi-symmetric function self $\prec$ other, written in the basis of self.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.M()
sage: M[2, 1].dendriform_less(M[1, 2])
2*M[2, 1, 1, 2] + M[2, 1, 2, 1] + M[2, 1, 3] + M[2, 2, 2]
sage: F = QSym.F()
sage: F[2, 1].dendriform_less(F[1, 2])
F[1, 1, 2, 1, 1] + F[1, 1, 2, 2] + F[1, 1, 3, 1]
 + F[1, 2, 1, 2] + F[1, 2, 2, 1] + F[1, 2, 3]
 + F[2, 1, 1, 2] + F[2, 1, 2, 1] + F[2, 1, 3] + F[2, 2, 2]
```

The operation $\prec$ can be used to recursively construct the dual immaculate basis: For every positive integer m and every composition I, the dual immaculate function $\mathrm{dI}_{[m,I]}$ of the composition $[m, I]$ (this composition is I with m prepended to it) is $F_{[m]} \prec \mathrm{dI}_I$.

```
sage: dI = QSym.dI()
sage: dI(F[2]).dendriform_less(dI[1, 2])
dI[2, 1, 2]
```

We check with the identity element:

```
sage: M.one().dendriform_less(M[1,2])
0
sage: M[1,2].dendriform_less(M.one())
M[1, 2]
```

The operation $\prec$ is not symmetric, nor if $a \prec b \neq 0$, then $b \prec a = 0$ (as it would be for a single monomial):

```
sage: M[1,2,1].dendriform_less(M[1,2])
M[1, 1, 2, 1, 2] + 2*M[1, 1, 2, 2, 1] + M[1, 1, 2, 3]
 + M[1, 1, 4, 1] + 2*M[1, 2, 1, 1, 2] + M[1, 2, 1, 2, 1]
 + M[1, 2, 1, 3] + M[1, 2, 2, 2] + M[1, 3, 1, 2]
 + M[1, 3, 2, 1] + M[1, 3, 3]
sage: M[1,2].dendriform_less(M[1,2,1])
M[1, 1, 2, 1, 2] + 2*M[1, 1, 2, 2, 1] + M[1, 1, 2, 3]
 + M[1, 1, 4, 1] + M[1, 2, 1, 2, 1] + M[1, 3, 2, 1]
```

expand(*n*, *alphabet='x'*)

Expand the quasi-symmetric function into n variables in an alphabet, which by default is 'x'.

INPUT:

•n – A nonnegative integer; the number of variables in the expansion

•alphabet – (default: 'x'); the alphabet in which self is to be expanded

OUTPUT:

•An expansion of self into the n variables specified by alphabet.

EXAMPLES:

```
sage: F=QuasiSymmetricFunctions(QQ).Fundamental()
sage: F[3].expand(3)
x0^3 + x0^2*x1 + x0*x1^2 + x1^3 + x0^2*x2 + x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2 +
sage: F[2,1].expand(3)
x0^2*x1 + x0^2*x2 + x0*x1*x2 + x1^2*x2
```

One can use a different set of variable by adding an optional argument alphabet=...

```
sage: F=QuasiSymmetricFunctions(QQ).Fundamental()
sage: F[3].expand(2,alphabet='y')
```

```
y0^3 + y0^2*y1 + y0*y1^2 + y1^3
```

TESTS:

```
sage: (3*F([])).expand(2)
3
sage: F[4,2].expand(0)
0
sage: F([]).expand(0)
1
```

frobenius(*n*)

Return the image of the quasi-symmetric function `self` under the n-th Frobenius operator.

The n-th Frobenius operator $\mathbf{f}_n$ is defined to be the map from the R-algebra of quasi-symmetric functions to itself that sends every symmetric function $P(x_1, x_2, x_3, \ldots)$ to $P(x_1^n, x_2^n, x_3^n, \ldots)$. This operator $\mathbf{f}_n$ is a Hopf algebra endomorphism, and satisfies

$$f_n M_{(i_1,i_2,i_3,\ldots)} = M_{(ni_1,ni_2,ni_3,\ldots)}$$

for every composition $(i_1, i_2, i_3, \ldots)$ (where M means the monomial basis).

The n-th Frobenius operator is also called the n-th Frobenius endomorphism. It is not related to the Frobenius map which connects the ring of symmetric functions with the representation theory of the symmetric group.

The n-th Frobenius operator is also the n-th Adams operator of the Λ-ring of quasi-symmetric functions over the integers.

The restriction of the n-th Frobenius operator to the subring formed by all symmetric functions is, not unexpectedly, the n-th Frobenius operator of the ring of symmetric functions.

See also:

`Symmetric functions plethsym`

INPUT:

- n – a positive integer

OUTPUT:

The result of applying the n-th Frobenius operator (on the ring of quasi-symmetric functions) to `self`.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: M = QSym.M()
sage: F = QSym.F()
sage: M[3,2].frobenius(2)
M[6, 4]
sage: (M[2,1] - 2*M[3]).frobenius(4)
M[8, 4] - 2*M[12]
sage: M([]).frobenius(3)
M[]
sage: F[1,1].frobenius(2)
F[1, 1, 1, 1] - F[1, 1, 2] - F[2, 1, 1] + F[2, 2]
```

The Frobenius endomorphisms are multiplicative:

```
sage: all( all( M(I).frobenius(3) * M(J).frobenius(3)
....:           == (M(I) * M(J)).frobenius(3)
....:           for I in Compositions(3) )
....:      for J in Compositions(2) )
True
```

Being Hopf algebra endomorphisms, the Frobenius operators commute with the antipode:

```
sage: all( M(I).frobenius(4).antipode()
....:      == M(I).antipode().frobenius(4)
....:      for I in Compositions(3) )
True
```

The restriction of the Frobenius operators to the subring of symmetric functions are the Frobenius operators of the latter:

```
sage: e = SymmetricFunctions(ZZ).e()
sage: all( M(e(lam)).frobenius(3)
....:      == M(e(lam).frobenius(3))
....:      for lam in Partitions(3) )
True
```

internal_coproduct()

Return the inner coproduct of `self` in the basis of `self`.

The inner coproduct (also known as the Kronecker coproduct, or as the second comultiplication on the R-algebra of quasi-symmetric functions) is an R-algebra homomorphism $\Delta^{\times}$ from the R-algebra of quasi-symmetric functions to the tensor square (over R) of quasi-symmetric functions. It can be defined in the following two ways:

1. If I is a composition, then a $(0, I)$-matrix will mean a matrix whose entries are nonnegative integers such that no row and no column of this matrix is zero, and such that if all the non-zero entries of the matrix are read (row by row, starting at the topmost row, reading every row from left to right), then the reading word obtained is I. If A is a $(0, I)$-matrix, then $\mathrm{row}(A)$ will denote the vector of row sums of A (regarded as a composition), and $\mathrm{column}(A)$ will denote the vector of column sums of A (regarded as a composition).

   For every composition I, the internal coproduct $\Delta^{\times}(M_I)$ of the I-th monomial quasisymmetric function M_I is the sum

   $$\sum_{A \text{ is a } (0,I)\text{-matrix}} M_{\mathrm{row}(A)} \otimes M_{\mathrm{column}(A)}.$$

   See Section 11.39 of [HazWitt1].

2. For every permutation w, let $C(w)$ denote the descent composition of w. Then, for any composition I of size n, the internal coproduct $\Delta^{\times}(F_I)$ of the I-th fundamental quasisymmetric function F_I is the sum

   $$\sum_{\substack{\sigma \in S_n, \\ \tau \in S_n, \\ \tau\sigma = \pi}} F_{C(\sigma)} \otimes F_{C(\tau)},$$

   where π is any permutation in S_n having descent composition I and where permutations act from the left and multiply accordingly, so $\tau\sigma$ means first applying σ and then τ. See Theorem 4.23 in [Mal1993], but beware of the notations which are apparently different from those in [HazWitt1].

The restriction of the internal coproduct to the R-algebra of symmetric functions is the well-known internal coproduct on the symmetric functions.

The method `kronecker_coproduct()` is a synonym of this one.

EXAMPLES:

Let us compute the internal coproduct of M_{21} (which is short for $M_{[2,1]}$). The $(0, [2, 1])$-matrices are

$$\begin{bmatrix} 2 & 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}, \text{ and } \begin{bmatrix} 0 & 2 \\ 1 & 0 \end{bmatrix}$$

so

$$\Delta^{\times}(M_{21}) = M_3 \otimes M_{21} + M_{21} \otimes M_3 + M_{21} \otimes M_{21} + M_{21} \otimes M_{12}.$$

This is confirmed by the following Sage computation (incidentally demonstrating the non-cocommutativity of the internal coproduct):

```
sage: M = QuasiSymmetricFunctions(ZZ).M()
sage: a = M([2,1])
sage: a.internal_coproduct()
M[2, 1] # M[1, 2] + M[2, 1] # M[2, 1] + M[2, 1] # M[3] + M[3] # M[2, 1]
```

Further examples:

```
sage: all( M([i]).internal_coproduct() == tensor([M([i]), M([i])])
....:      for i in range(1, 4) )
True

sage: M([1, 2]).internal_coproduct()
M[1, 2] # M[1, 2] + M[1, 2] # M[2, 1] + M[1, 2] # M[3] + M[3] # M[1, 2]
```

The definition of $\Delta^{\times}(M_I)$ in terms of $(0, I)$-matrices is not suitable for computation in cases where the length of I is large, but we can use it as a doctest. Here is a naive implementation:

```
sage: def naive_internal_coproduct_on_M(I):
....:     # INPUT: composition I
....:     #        (not quasi-symmetric function)
....:     # OUTPUT: interior coproduct of M_I
....:     M = QuasiSymmetricFunctions(ZZ).M()
....:     M2 = M.tensor(M)
....:     res = M2.zero()
....:     l = len(I)
....:     n = I.size()
....:     for S in Subsets(range(l**2), l):
....:         M_list = sorted(S)
....:         row_M = [sum([I[M_list.index(l * i + j)]
....:                       for j in range(l) if
....:                       l * i + j in S])
....:                  for i in range(l)]
....:         col_M = [sum([I[M_list.index(l * i + j)]
....:                       for i in range(l) if
....:                       l * i + j in S])
....:                  for j in range(l)]
....:         if 0 in row_M:
....:             first_zero = row_M.index(0)
....:             row_M = row_M[:first_zero]
....:             if sum(row_M) != n:
....:                 continue
....:         if 0 in col_M:
....:             first_zero = col_M.index(0)
....:             col_M = col_M[:first_zero]
....:             if sum(col_M) != n:
....:                 continue
....:         res += tensor([M(Compositions(n)(row_M)),
....:                        M(Compositions(n)(col_M))])
....:     return res
sage: all( naive_internal_coproduct_on_M(I)
....:      == M(I).internal_coproduct()
....:      for I in Compositions(3) )
True
```

TESTS:

Border cases:

```
sage: M = QuasiSymmetricFunctions(ZZ).M()
sage: F = QuasiSymmetricFunctions(ZZ).F()
sage: M([]).internal_coproduct()
M[] # M[]
sage: F([]).internal_coproduct()
F[] # F[]
```

The implementations on the `F` and `M` bases agree with each other:

```
sage: M = QuasiSymmetricFunctions(ZZ).M()
sage: F = QuasiSymmetricFunctions(ZZ).F()
sage: def int_copr_on_F_via_M(I):
....:     result = tensor([F.zero(), F.zero()])
....:     w = M(F(I)).internal_coproduct()
....:     for lam, a in w:
....:         (U, V) = lam
....:         result += a * tensor([F(M(U)), F(M(V))])
....:     return result
sage: all( int_copr_on_F_via_M(I) == F(I).internal_coproduct()
....:      for I in Compositions(3) )
True
sage: all( int_copr_on_F_via_M(I) == F(I).internal_coproduct()
....:      for I in Compositions(4) )
True
```

Restricting to the subring of symmetric functions gives the standard internal coproduct on the latter:

```
sage: M = QuasiSymmetricFunctions(ZZ).M()
sage: e = SymmetricFunctions(ZZ).e()
sage: def int_copr_of_e_in_M(mu):
....:     result = tensor([M.zero(), M.zero()])
....:     w = e(mu).internal_coproduct()
....:     for lam, a in w:
....:         (nu, kappa) = lam
....:         result += a * tensor([M(e(nu)), M(e(kappa))])
....:     return result
sage: all( int_copr_of_e_in_M(mu) == M(e(mu)).internal_coproduct()
....:      for mu in Partitions(3) )
True
sage: all( int_copr_of_e_in_M(mu) == M(e(mu)).internal_coproduct()
....:      for mu in Partitions(4) )
True
```

Todo

Implement this directly on the monomial basis maybe? The $(0, I)$-matrices are a pain to generate from their definition, but maybe there is a good algorithm. If so, the above "further examples" should be moved to the M-method.

---

is_symmetric()

Return `True` if `self` is an element of the symmetric functions.

This is being tested by looking at the expansion in the Monomial basis and checking if the coefficients are the same if the indexing compositions are permutations of each other.

OUTPUT:

- `True` if `self` is symmetric. `False` if `self` is not symmetric.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: F = QSym.Fundamental()
sage: (F[3,2] + F[2,3]).is_symmetric()
False
sage: (F[1, 1, 1, 2] + F[1, 1, 3] + F[1, 3, 1] + F[2, 1, 1, 1] + F[3, 1, 1]).is_symme
True
sage: F([]).is_symmetric()
True
```

kronecker_coproduct()

Return the inner coproduct of `self` in the basis of `self`.

The inner coproduct (also known as the Kronecker coproduct, or as the second comultiplication on the R-algebra of quasi-symmetric functions) is an R-algebra homomorphism $\Delta^\times$ from the R-algebra of quasi-symmetric functions to the tensor square (over R) of quasi-symmetric functions. It can be defined in the following two ways:

1. If I is a composition, then a $(0, I)$-matrix will mean a matrix whose entries are nonnegative integers such that no row and no column of this matrix is zero, and such that if all the non-zero entries of the matrix are read (row by row, starting at the topmost row, reading every row from left to right), then the reading word obtained is I. If A is a $(0, I)$-matrix, then $\mathrm{row}(A)$ will denote the vector of row sums of A (regarded as a composition), and $\mathrm{column}(A)$ will denote the vector of column sums of A (regarded as a composition).

   For every composition I, the internal coproduct $\Delta^\times(M_I)$ of the I-th monomial quasisymmetric function M_I is the sum

   $$\sum_{A \text{ is a } (0,I)\text{-matrix}} M_{\mathrm{row}(A)} \otimes M_{\mathrm{column}(A)}.$$

   See Section 11.39 of [HazWitt1].

2. For every permutation w, let $C(w)$ denote the descent composition of w. Then, for any composition I of size n, the internal coproduct $\Delta^\times(F_I)$ of the I-th fundamental quasisymmetric function F_I is the sum

   $$\sum_{\substack{\sigma \in S_n, \\ \tau \in S_n, \\ \tau\sigma = \pi}} F_{C(\sigma)} \otimes F_{C(\tau)},$$

   where π is any permutation in S_n having descent composition I and where permutations act from the left and multiply accordingly, so $\tau\sigma$ means first applying σ and then τ. See Theorem 4.23 in [Mal1993], but beware of the notations which are apparently different from those in [HazWitt1].

The restriction of the internal coproduct to the R-algebra of symmetric functions is the well-known internal coproduct on the symmetric functions.

The method `kronecker_coproduct()` is a synonym of this one.

EXAMPLES:

Let us compute the internal coproduct of M_{21} (which is short for $M_{[2,1]}$). The $(0, [2, 1])$-matrices are

$$\begin{bmatrix} 2 & 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}, \text{ and } \begin{bmatrix} 0 & 2 \\ 1 & 0 \end{bmatrix}$$

so

$$\Delta^\times(M_{21}) = M_3 \otimes M_{21} + M_{21} \otimes M_3 + M_{21} \otimes M_{21} + M_{21} \otimes M_{12}.$$

This is confirmed by the following Sage computation (incidentally demonstrating the non-cocommutativity of the internal coproduct):

```
sage: M = QuasiSymmetricFunctions(ZZ).M()
sage: a = M([2,1])
sage: a.internal_coproduct()
M[2, 1] # M[1, 2] + M[2, 1] # M[2, 1] + M[2, 1] # M[3] + M[3] # M[2, 1]
```

Further examples:

```
sage: all( M([i]).internal_coproduct() == tensor([M([i]), M([i])])
....:      for i in range(1, 4) )
True

sage: M([1, 2]).internal_coproduct()
M[1, 2] # M[1, 2] + M[1, 2] # M[2, 1] + M[1, 2] # M[3] + M[3] # M[1, 2]
```

The definition of $\Delta^{\times}(M_I)$ in terms of $(0, I)$-matrices is not suitable for computation in cases where the length of I is large, but we can use it as a doctest. Here is a naive implementation:

```
sage: def naive_internal_coproduct_on_M(I):
....:     # INPUT: composition I
....:     #        (not quasi-symmetric function)
....:     # OUTPUT: interior coproduct of M_I
....:     M = QuasiSymmetricFunctions(ZZ).M()
....:     M2 = M.tensor(M)
....:     res = M2.zero()
....:     l = len(I)
....:     n = I.size()
....:     for S in Subsets(range(l**2), l):
....:         M_list = sorted(S)
....:         row_M = [sum([I[M_list.index(l * i + j)]
....:                       for j in range(l) if
....:                       l * i + j in S])
....:                  for i in range(l)]
....:         col_M = [sum([I[M_list.index(l * i + j)]
....:                       for i in range(l) if
....:                       l * i + j in S])
....:                  for j in range(l)]
....:         if 0 in row_M:
....:             first_zero = row_M.index(0)
....:             row_M = row_M[:first_zero]
....:             if sum(row_M) != n:
....:                 continue
....:         if 0 in col_M:
....:             first_zero = col_M.index(0)
....:             col_M = col_M[:first_zero]
....:             if sum(col_M) != n:
....:                 continue
....:         res += tensor([M(Compositions(n)(row_M)),
....:                        M(Compositions(n)(col_M))])
....:     return res
sage: all( naive_internal_coproduct_on_M(I)
....:      == M(I).internal_coproduct()
....:      for I in Compositions(3) )
True
```

TESTS:

Border cases:

```
sage: M = QuasiSymmetricFunctions(ZZ).M()
sage: F = QuasiSymmetricFunctions(ZZ).F()
sage: M([]).internal_coproduct()
```

```
M[] # M[]
sage: F([]).internal_coproduct()
F[] # F[]
```

The implementations on the `F` and `M` bases agree with each other:

```
sage: M = QuasiSymmetricFunctions(ZZ).M()
sage: F = QuasiSymmetricFunctions(ZZ).F()
sage: def int_copr_on_F_via_M(I):
....:     result = tensor([F.zero(), F.zero()])
....:     w = M(F(I)).internal_coproduct()
....:     for lam, a in w:
....:         (U, V) = lam
....:         result += a * tensor([F(M(U)), F(M(V))])
....:     return result
sage: all( int_copr_on_F_via_M(I) == F(I).internal_coproduct()
....:      for I in Compositions(3) )
True
sage: all( int_copr_on_F_via_M(I) == F(I).internal_coproduct()
....:      for I in Compositions(4) )
True
```

Restricting to the subring of symmetric functions gives the standard internal coproduct on the latter:

```
sage: M = QuasiSymmetricFunctions(ZZ).M()
sage: e = SymmetricFunctions(ZZ).e()
sage: def int_copr_of_e_in_M(mu):
....:     result = tensor([M.zero(), M.zero()])
....:     w = e(mu).internal_coproduct()
....:     for lam, a in w:
....:         (nu, kappa) = lam
....:         result += a * tensor([M(e(nu)), M(e(kappa))])
....:     return result
sage: all( int_copr_of_e_in_M(mu) == M(e(mu)).internal_coproduct()
....:      for mu in Partitions(3) )
True
sage: all( int_copr_of_e_in_M(mu) == M(e(mu)).internal_coproduct()
....:      for mu in Partitions(4) )
True
```

Todo

Implement this directly on the monomial basis maybe? The $(0, I)$-matrices are a pain to generate from their definition, but maybe there is a good algorithm. If so, the above "further examples" should be moved to the M-method.

---

omega_involution()

Return the image of the quasisymmetric function `self` under the omega involution.

The omega involution is defined as the linear map $QSym \to QSym$ which, for every composition I, sends the fundamental quasisymmetric function F_I to F_{I^t}, where I^t denotes the conjugate (`conjugate()`) of the composition I. This map is commonly denoted by ω. It is an algebra homomorphism and a coalgebra antihomomorphism; it also is an involution and an automorphism of the graded vector space $QSym$. Also, every composition I satisfies

$$\omega(M_I) = (-1)^{|I|-\ell(I)} \sum M_J,$$

where the sum ranges over all compositions J of $|I|$ which are coarser than the reversed composition I^r of I. Here, $\ell(I)$ denotes the length of the composition I (that is, the number of parts of

I).

If f is a homogeneous element of $NCSF$ of degree n, then

$$\omega(f) = (-1)^n S(f),$$

where S denotes the antipode of $QSym$.

The restriction of ω to the ring of symmetric functions (which is a subring of $QSym$) is precisely the omega involution (`omega()`) of said ring.

The omega involution on $QSym$ is adjoint to the omega involution on $NCSF$ by the standard adjunction between $NCSF$ and $QSym$.

The omega involution has been denoted by ω in [LMvW13], section 3.6.

See also:

`omega involution on NCSF`, `psi involution on QSym`, `star involution on QSym`.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: F = QSym.F()
sage: F[3,2].omega_involution()
F[1, 2, 1, 1]
sage: F[6,3].omega_involution()
F[1, 1, 2, 1, 1, 1, 1, 1]
sage: (F[9,1] - F[8,2] + 2*F[2,4] - 3*F[3] + 4*F[[]]).omega_involution()
4*F[] - 3*F[1, 1, 1] + 2*F[1, 1, 1, 2, 1] - F[1, 2, 1, 1, 1, 1, 1, 1, 1] + F[2, 1, 1,
sage: (F[3,3] - 2*F[2]).omega_involution()
-2*F[1, 1] + F[1, 1, 2, 1, 1]
sage: F([2,1,1]).omega_involution()
F[3, 1]
sage: M = QSym.M()
sage: M([2,1]).omega_involution()
-M[1, 2] - M[3]
sage: M.zero().omega_involution()
0
```

Testing the fact that the restriction of ω to Sym is the omega automorphism of Sym:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: e = Sym.e()
sage: all( F(e[lam]).omega_involution() == F(e[lam].omega())
....:      for lam in Partitions(4) )
True
```

psi_involution()

Return the image of the quasisymmetric function `self` under the involution ψ.

The involution ψ is defined as the linear map $QSym \to QSym$ which, for every composition I, sends the fundamental quasisymmetric function F_I to F_{I^c}, where I^c denotes the complement of the composition I. The map ψ is an involution and a graded Hopf algebra automorphism of $QSym$. Its restriction to the ring of symmetric functions coincides with the omega automorphism of the latter ring.

The involution ψ of $QSym$ is adjoint to the involution ψ of $NCSF$ by the standard adjunction between $NCSF$ and $QSym$.

The involution ψ has been denoted by ψ in [LMvW13], section 3.6.

See also:

psi involution on NCSF, star involution on QSym.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: F = QSym.F()
sage: F[3,2].psi_involution()
F[1, 1, 2, 1]
sage: F[6,3].psi_involution()
F[1, 1, 1, 1, 1, 2, 1, 1]
sage: (F[9,1] - F[8,2] + 2*F[2,4] - 3*F[3] + 4*F[[]]).psi_involution()
4*F[] - 3*F[1, 1, 1] + F[1, 1, 1, 1, 1, 1, 1, 1, 2] - F[1, 1, 1, 1, 1, 1, 1, 2, 1] +
sage: (F[3,3] - 2*F[2]).psi_involution()
-2*F[1, 1] + F[1, 1, 2, 1, 1]
sage: F([2,1,1]).psi_involution()
F[1, 3]
sage: M = QSym.M()
sage: M([2,1]).psi_involution()
-M[2, 1] - M[3]
sage: M.zero().psi_involution()
0
```

The involution ψ commutes with the antipode:

```
sage: all( F(I).psi_involution().antipode()
....:      == F(I).antipode().psi_involution()
....:      for I in Compositions(4) )
True
```

Testing the fact that the restriction of ψ to Sym is the omega automorphism of Sym:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: e = Sym.e()
sage: all( F(e[lam]).psi_involution() == F(e[lam].omega())
....:      for lam in Partitions(4) )
True
```

star_involution()

Return the image of the quasisymmetric function `self` under the star involution.

The star involution is defined as the linear map $QSym \to QSym$ which, for every composition I, sends the monomial quasisymmetric function M_I to M_{I^r}. Here, if I is a composition, we denote by I^r the reversed composition of I. Denoting by f^* the image of an element $f \in QSym$ under the star involution, it can be shown that every composition I satisfies

$$(M_I)^* = M_{I^r}, \quad (F_I)^* = F_{I^r},$$

where F_I denotes the fundamental quasisymmetric function corresponding to the composition I. The star involution is an involution, an algebra automorphism and a coalgebra anti-automorphism of $QSym$. It also is an automorphism of the graded vector space $QSym$, and is the identity on the subspace Sym of $QSym$. It is adjoint to the star involution on $NCSF$ by the standard adjunction between $NCSF$ and $QSym$.

The star involution has been denoted by ρ in [LMvW13], section 3.6.

See also:

star involution on NCSF.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: M = QSym.M()
sage: M[3,2].star_involution()
```

```
M[2, 3]
sage: M[6,3].star_involution()
M[3, 6]
sage: (M[9,1] - M[6,2] + 2*M[6,4] - 3*M[3] + 4*M[[]]).star_involution()
4*M[] + M[1, 9] - M[2, 6] - 3*M[3] + 2*M[4, 6]
sage: (M[3,3] - 2*M[2]).star_involution()
-2*M[2] + M[3, 3]
sage: M([4,2]).star_involution()
M[2, 4]
sage: dI = QSym.dI()
sage: dI([1,2]).star_involution()
-dI[1, 2] + dI[2, 1]
sage: dI.zero().star_involution()
0
```

The star involution commutes with the antipode:

```
sage: all( M(I).star_involution().antipode()
....:      == M(I).antipode().star_involution()
....:      for I in Compositions(4) )
True
```

The star involution is the identity on Sym:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: e = Sym.e()
sage: all( M(e(lam)).star_involution() == M(e(lam))
....:      for lam in Partitions(4) )
True
```

to_symmetric_function()

Convert a quasi-symmetric function to a symmetric function.

OUTPUT:

- If `self` is a symmetric function, then return the expansion in the monomial basis. Otherwise raise an error.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: F = QSym.Fundamental()
sage: (F[3,2] + F[2,3]).to_symmetric_function()
Traceback (most recent call last):
...
ValueError: F[2, 3] + F[3, 2] is not a symmetric function
sage: m = SymmetricFunctions(QQ).m()
sage: s = SymmetricFunctions(QQ).s()
sage: F(s[3,1,1]).to_symmetric_function()
6*m[1, 1, 1, 1, 1] + 3*m[2, 1, 1, 1] + m[2, 2, 1] + m[3, 1, 1]
sage: m(s[3,1,1])
6*m[1, 1, 1, 1, 1] + 3*m[2, 1, 1, 1] + m[2, 2, 1] + m[3, 1, 1]
```

class QuasiSymmetricFunctions.Bases.**ParentMethods**

Methods common to all bases of `QuasiSymmetricFunctions`.

Eulerian(*n*, *j*, *k=None*)

Return the Eulerian (quasi)symmetric function $Q_{n,j}$ in terms of `self`.

INPUT:

- `n` – the value n or a partition
- `j` – the number of excedances
- `k` – (optional) if specified, determines the number of fixed points of the permtutation

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.M()
sage: M.Eulerian(3, 1)
4*M[1, 1, 1] + 3*M[1, 2] + 3*M[2, 1] + 2*M[3]
sage: M.Eulerian(4, 1, 2)
6*M[1, 1, 1, 1] + 4*M[1, 1, 2] + 4*M[1, 2, 1]
 + 2*M[1, 3] + 4*M[2, 1, 1] + 3*M[2, 2] + 2*M[3, 1] + M[4]
sage: QS = QSym.QS()
sage: QS.Eulerian(4, 2)
2*QS[1, 3] + QS[2, 2] + 2*QS[3, 1] + 3*QS[4]
sage: QS.Eulerian([2, 2, 1], 2)
QS[1, 2, 2] + QS[1, 4] + QS[2, 1, 2] + QS[2, 2, 1]
 + QS[2, 3] + QS[3, 2] + QS[4, 1] + QS[5]
sage: dI = QSym.dI()
sage: dI.Eulerian(5, 2)
-dI[1, 3, 1] - 5*dI[1, 4] + dI[2, 2, 1] + dI[3, 1, 1]
 + 5*dI[3, 2] + 6*dI[4, 1] + 6*dI[5]
```

from_polynomial (*f*, *check=True*)

The quasi-symmetric function expanded in this basis corresponding to the quasi-symmetric polynomial `f`.

This is a default implementation that computes the expansion in the Monomial basis and converts to this basis.

INPUT:

- `f` – a polynomial in finitely many variables over the same base ring as `self`. It is assumed that this polynomial is quasi-symmetric.
- `check` – boolean (default: `True`), checks whether the polynomial is indeed quasi-symmetric.

OUTPUT:

- quasi-symmetric function

EXAMPLES:

```
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: P = PolynomialRing(QQ, 'x', 3)
sage: x = P.gens()
sage: f = x[0] + x[1] + x[2]
sage: M.from_polynomial(f)
M[1]
sage: F.from_polynomial(f)
F[1]
sage: f = x[0]**2+x[1]**2+x[2]**2
sage: M.from_polynomial(f)
M[2]
sage: F.from_polynomial(f)
-F[1, 1] + F[2]
```

If the polynomial is not quasi-symmetric, an error is raised:

```
sage: f = x[0]^2+x[1]
sage: M.from_polynomial(f)
Traceback (most recent call last):
...
ValueError: x0^2 + x1 is not a quasi-symmetric polynomial
sage: F.from_polynomial(f)
Traceback (most recent call last):
...
ValueError: x0^2 + x1 is not a quasi-symmetric polynomial
```

TESTS:

We convert some quasi-symmetric functions to quasi-symmetric polynomials and back:

```
sage: f = (M[1,2] + M[1,1]).expand(3); f
x0*x1^2 + x0*x2^2 + x1*x2^2 + x0*x1 + x0*x2 + x1*x2
sage: M.from_polynomial(f)
M[1, 1] + M[1, 2]
sage: f = (2*M[2,1]+M[1,1]+3*M[3]).expand(3)
sage: M.from_polynomial(f)
M[1, 1] + 2*M[2, 1] + 3*M[3]
sage: f = (F[1,2] + F[1,1]).expand(3); f
x0*x1^2 + x0*x1*x2 + x0*x2^2 + x1*x2^2 + x0*x1 + x0*x2 + x1*x2
sage: F.from_polynomial(f)
F[1, 1] + F[1, 2]
sage: f = (2*F[2,1]+F[1,1]+3*F[3]).expand(3)
sage: F.from_polynomial(f)
F[1, 1] + 2*F[2, 1] + 3*F[3]
```

QuasiSymmetricFunctions.Bases.**super_categories**()

Return the super categories of bases of the Quasi-symmetric functions.

OUTPUT:

- a list of categories

TESTS:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: QSym.Bases().super_categories()
[Category of commutative bases of Non-Commutative Symmetric Functions or Quasisymmetric
```

class QuasiSymmetricFunctions.**Fundamental**(*QSym*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The Hopf algebra of quasi-symmetric functions in the Fundamental basis.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: F = QSym.F()
sage: M = QSym.M()
sage: F(M[2,2])
F[1, 1, 1, 1] - F[1, 1, 2] - F[2, 1, 1] + F[2, 2]
sage: s = SymmetricFunctions(QQ).s()
sage: F(s[3,2])
F[1, 2, 2] + F[1, 3, 1] + F[2, 2, 1] + F[2, 3] + F[3, 2]
sage: (1+F[1])^3
F[] + 3*F[1] + 3*F[1, 1] + F[1, 1, 1] + 2*F[1, 2] + 3*F[2] + 2*F[2, 1] + F[3]
sage: F[1,2,1].coproduct()
F[] # F[1, 2, 1] + F[1] # F[2, 1] + F[1, 1] # F[1, 1] + F[1, 2] # F[1] + F[1, 2, 1] # F[]
```

The following is an alias for this basis:

```
sage: QSym.Fundamental()
Quasisymmetric functions over the Rational Field in the Fundamental basis
```

TESTS:

```
sage: F(M([]))
F[]
sage: F(M(0))
0
sage: F(s([]))
```

```
F[]
sage: F(s(0))
0
```

class Element(*M*, *x*)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

internal_coproduct()

Return the inner coproduct of self in the Fundamental basis.

The inner coproduct (also known as the Kronecker coproduct, or as the second comultiplication on the R-algebra of quasi-symmetric functions) is an R-algebra homomorphism $\Delta^{\times}$ from the R-algebra of quasi-symmetric functions to the tensor square (over R) of quasi-symmetric functions. It can be defined in the following two ways:

1. If I is a composition, then a $(0, I)$-matrix will mean a matrix whose entries are nonnegative integers such that no row and no column of this matrix is zero, and such that if all the non-zero entries of the matrix are read (row by row, starting at the topmost row, reading every row from left to right), then the reading word obtained is I. If A is a $(0, I)$-matrix, then $\mathrm{row}(A)$ will denote the vector of row sums of A (regarded as a composition), and $\mathrm{column}(A)$ will denote the vector of column sums of A (regarded as a composition).

   For every composition I, the internal coproduct $\Delta^{\times}(M_I)$ of the I-th monomial quasisymmetric function M_I is the sum

   $$\sum_{A \text{ is a } (0,I)\text{-matrix}} M_{\mathrm{row}(A)} \otimes M_{\mathrm{column}(A)}.$$

   See Section 11.39 of [HazWitt1].

2. For every permutation w, let $C(w)$ denote the descent composition of w. Then, for any composition I of size n, the internal coproduct $\Delta^{\times}(F_I)$ of the I-th fundamental quasisymmetric function F_I is the sum

   $$\sum_{\substack{\sigma \in S_n, \\ \tau \in S_n, \\ \tau\sigma = \pi}} F_{C(\sigma)} \otimes F_{C(\tau)},$$

   where π is any permutation in S_n having descent composition I and where permutations act from the left and multiply accordingly, so $\tau\sigma$ means first applying σ and then τ. See Theorem 4.23 in [Mal1993], but beware of the notations which are apparently different from those in [HazWitt1].

The restriction of the internal coproduct to the R-algebra of symmetric functions is the well-known internal coproduct on the symmetric functions.

The method kronecker_coproduct() is a synonym of this one.

EXAMPLES:

Let us compute the internal coproduct of M_{21} (which is short for $M_{[2,1]}$). The $(0, [2, 1])$-matrices are

$$\begin{bmatrix} 2 & 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}, \text{ and } \begin{bmatrix} 0 & 2 \\ 1 & 0 \end{bmatrix}$$

so

$$\Delta^{\times}(M_{21}) = M_3 \otimes M_{21} + M_{21} \otimes M_3 + M_{21} \otimes M_{21} + M_{21} \otimes M_{12}.$$

This is confirmed by the following Sage computation (incidentally demonstrating the non-cocommutativity of the internal coproduct):

```
sage: M = QuasiSymmetricFunctions(ZZ).M()
sage: a = M([2,1])
sage: a.internal_coproduct()
M[2, 1] # M[1, 2] + M[2, 1] # M[2, 1] + M[2, 1] # M[3] + M[3] # M[2, 1]
```

Some examples on the Fundamental basis:

```
sage: F = QuasiSymmetricFunctions(ZZ).F()
sage: F([1,1]).internal_coproduct()
F[1, 1] # F[2] + F[2] # F[1, 1]
sage: F([2]).internal_coproduct()
F[1, 1] # F[1, 1] + F[2] # F[2]
sage: F([3]).internal_coproduct()
F[1, 1, 1] # F[1, 1, 1] + F[1, 2] # F[1, 2] + F[1, 2] # F[2, 1]
 + F[2, 1] # F[1, 2] + F[2, 1] # F[2, 1] + F[3] # F[3]
sage: F([1,2]).internal_coproduct()
F[1, 1, 1] # F[1, 2] + F[1, 2] # F[2, 1] + F[1, 2] # F[3]
 + F[2, 1] # F[1, 1, 1] + F[2, 1] # F[2, 1] + F[3] # F[1, 2]
```

kronecker_coproduct()

Return the inner coproduct of `self` in the Fundamental basis.

The inner coproduct (also known as the Kronecker coproduct, or as the second comultiplication on the R-algebra of quasi-symmetric functions) is an R-algebra homomorphism $\Delta^{\times}$ from the R-algebra of quasi-symmetric functions to the tensor square (over R) of quasi-symmetric functions. It can be defined in the following two ways:

1. If I is a composition, then a $(0, I)$-matrix will mean a matrix whose entries are nonnegative integers such that no row and no column of this matrix is zero, and such that if all the non-zero entries of the matrix are read (row by row, starting at the topmost row, reading every row from left to right), then the reading word obtained is I. If A is a $(0, I)$-matrix, then $\mathrm{row}(A)$ will denote the vector of row sums of A (regarded as a composition), and $\mathrm{column}(A)$ will denote the vector of column sums of A (regarded as a composition).

   For every composition I, the internal coproduct $\Delta^{\times}(M_I)$ of the I-th monomial quasisymmetric function M_I is the sum

   $$\sum_{A \text{ is a } (0,I)\text{-matrix}} M_{\mathrm{row}(A)} \otimes M_{\mathrm{column}(A)}.$$

   See Section 11.39 of [HazWitt1].

2. For every permutation w, let $C(w)$ denote the descent composition of w. Then, for any composition I of size n, the internal coproduct $\Delta^{\times}(F_I)$ of the I-th fundamental quasisymmetric function F_I is the sum

   $$\sum_{\substack{\sigma \in S_n, \\ \tau \in S_n, \\ \tau\sigma = \pi}} F_{C(\sigma)} \otimes F_{C(\tau)},$$

where π is any permutation in S_n having descent composition I and where permutations act from the left and multiply accordingly, so $\tau\sigma$ means first applying σ and then τ. See Theorem 4.23 in [Mal1993], but beware of the notations which are apparently different from those in [HazWitt1].

The restriction of the internal coproduct to the R-algebra of symmetric functions is the well-known internal coproduct on the symmetric functions.

The method `kronecker_coproduct()` is a synonym of this one.

EXAMPLES:

Let us compute the internal coproduct of M_{21} (which is short for $M_{[2,1]}$). The $(0, [2, 1])$-matrices are

$$\begin{bmatrix} 2 & 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}, \text{ and } \begin{bmatrix} 0 & 2 \\ 1 & 0 \end{bmatrix}$$

so

$$\Delta^{\times}(M_{21}) = M_3 \otimes M_{21} + M_{21} \otimes M_3 + M_{21} \otimes M_{21} + M_{21} \otimes M_{12}.$$

This is confirmed by the following Sage computation (incidentally demonstrating the non-cocommutativity of the internal coproduct):

```
sage: M = QuasiSymmetricFunctions(ZZ).M()
sage: a = M([2,1])
sage: a.internal_coproduct()
M[2, 1] # M[1, 2] + M[2, 1] # M[2, 1] + M[2, 1] # M[3] + M[3] # M[2, 1]
```

Some examples on the Fundamental basis:

```
sage: F = QuasiSymmetricFunctions(ZZ).F()
sage: F([1,1]).internal_coproduct()
F[1, 1] # F[2] + F[2] # F[1, 1]
sage: F([2]).internal_coproduct()
F[1, 1] # F[1, 1] + F[2] # F[2]
sage: F([3]).internal_coproduct()
F[1, 1, 1] # F[1, 1, 1] + F[1, 2] # F[1, 2] + F[1, 2] # F[2, 1]
 + F[2, 1] # F[1, 2] + F[2, 1] # F[2, 1] + F[3] # F[3]
sage: F([1,2]).internal_coproduct()
F[1, 1, 1] # F[1, 2] + F[1, 2] # F[2, 1] + F[1, 2] # F[3]
 + F[2, 1] # F[1, 1, 1] + F[2, 1] # F[2, 1] + F[3] # F[1, 2]
```

star_involution()

Return the image of the quasisymmetric function `self` under the star involution.

The star involution is defined as the linear map $QSym \to QSym$ which, for every composition I, sends the monomial quasisymmetric function M_I to M_{I^r}. Here, if I is a composition, we denote by I^r the reversed composition of I. Denoting by f^* the image of an element $f \in QSym$ under the star involution, it can be shown that every composition I satisfies

$$(M_I)^* = M_{I^r}, \quad (F_I)^* = F_{I^r},$$

where F_I denotes the fundamental quasisymmetric function corresponding to the composition I. The star involution is an involution, an algebra automorphism and a coalgebra anti-automorphism of $QSym$. It also is an automorphism of the graded vector space $QSym$, and is the identity on the subspace Sym of $QSym$. It is adjoint to the star involution on $NCSF$ by the standard adjunction between $NCSF$ and $QSym$.

The star involution has been denoted by ρ in [LMvW13], section 3.6.

See also:

star involution on QSym, star involution on NCSF.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: F = QSym.F()
sage: F[3,1].star_involution()
F[1, 3]
sage: F[5,3].star_involution()
F[3, 5]
sage: (F[9,1] - F[6,2] + 2*F[6,4] - 3*F[3] + 4*F[[]]).star_involution()
4*F[] + F[1, 9] - F[2, 6] - 3*F[3] + 2*F[4, 6]
sage: (F[3,3] - 2*F[2]).star_involution()
-2*F[2] + F[3, 3]
sage: F([4,2]).star_involution()
F[2, 4]
sage: dI = QSym.dI()
sage: dI([1,2]).star_involution()
-dI[1, 2] + dI[2, 1]
sage: dI.zero().star_involution()
0
```

QuasiSymmetricFunctions.Fundamental.**Eulerian**(*n*, *j*, *k=None*)

Return the Eulerian (quasi)symmetric function $Q_{n,j}$ (with n either an integer or a partition) defined in [SW2010] in terms of the fundamental quasisymmetric functions. Or, if the optional argument k is specified, return the function $Q_{n,j,k}$ defined ibidem.

If n and j are nonnegative integers, then the Eulerian quasisymmetric function $Q_{n,j}$ is defined as

$$Q_{n,j} := \sum_{\sigma} F_{\mathrm{Dex}(\sigma)},$$

where we sum over all permutations $\sigma \in S_n$ such that the number of excedances of σ is j, and where $\mathrm{Dex}(\sigma)$ is a composition of n defined as follows: Let S be the set of all $i \in \{1, 2, \ldots, n-1\}$ such that either $\sigma_i > \sigma_{i+1} > i+1$ or $i \geq \sigma_i > \sigma_{i+1}$ or $\sigma_{i+1} > i+1 > \sigma_i$. Then, $\mathrm{Dex}(\sigma)$ is set to be the composition of n whose descent set is S.

Here, an excedance of a permutation $\sigma \in S_n$ means an element $i \in \{1, 2, \ldots, n-1\}$ satisfying $\sigma_i > i$.

Similarly we can define a quasisymmetric function $Q_{\lambda,j}$ for every partition λ and every nonnegative integer j. This differs from $Q_{n,j}$ only in that the sum is restricted to all permutations $\sigma \in S_n$ whose cycle type is λ (where $n = |\lambda|$, and where we still require the number of excedances to be j). The method at hand allows computing these functions by passing λ as the n parameter.

Analogously we can define a quasisymmetric function $Q_{n,j,k}$ for any nonnegative integers n, j and k by restricting the sum to all permutations $\sigma \in S_n$ that have exactly k fixed points (and j excedances). This can be obtained by specifying the optional k argument in this method.

All three versions of Eulerian quasisymmetric functions ($Q_{n,j}$, $Q_{\lambda,j}$ and $Q_{n,j,k}$) are actually symmetric functions. See Eulerian().

INPUT:

- n – the nonnegative integer n or a partition
- j – the number of excedances
- k – (optional) if specified, determines the number of fixed points of the permutations which are being summed over

REFERENCES:

EXAMPLES:

```
sage: F = QuasiSymmetricFunctions(QQ).F()
sage: F.Eulerian(3, 1)
F[1, 2] + F[2, 1] + 2*F[3]
sage: F.Eulerian(4, 2)
F[1, 2, 1] + 2*F[1, 3] + 3*F[2, 2] + 2*F[3, 1] + 3*F[4]
sage: F.Eulerian(5, 2)
F[1, 1, 2, 1] + F[1, 1, 3] + F[1, 2, 1, 1] + 7*F[1, 2, 2] + 6*F[1, 3, 1] + 6*F[1, 4] + 2
sage: F.Eulerian(4, 0)
F[4]
sage: F.Eulerian(4, 3)
F[4]
sage: F.Eulerian(4, 1, 2)
F[1, 2, 1] + F[1, 3] + 2*F[2, 2] + F[3, 1] + F[4]
sage: F.Eulerian(Partition([2, 2, 1]), 2)
F[1, 1, 2, 1] + F[1, 2, 1, 1] + 2*F[1, 2, 2] + F[1, 3, 1]
 + F[1, 4] + F[2, 1, 2] + 2*F[2, 2, 1] + 2*F[2, 3]
 + 2*F[3, 2] + F[4, 1] + F[5]
sage: F.Eulerian(0, 0)
F[]
sage: F.Eulerian(0, 1)
0
sage: F.Eulerian(1, 0)
F[1]
sage: F.Eulerian(1, 1)
0
```

TESTS:

```
sage: F = QuasiSymmetricFunctions(QQ).F()
sage: F.Eulerian(Partition([3, 1]), 1, 1)
Traceback (most recent call last):
...
ValueError: invalid input, k cannot be specified
```

QuasiSymmetricFunctions.Fundamental.**antipode_on_basis**(*compo*)

Return the antipode to a Fundamental quasi-symmetric basis element.

INPUT:

- compo – composition

OUTPUT:

- The result of the antipode applied to the quasi-symmetric Fundamental basis element indexed by compo.

EXAMPLES:

```
sage: F = QuasiSymmetricFunctions(QQ).F()
sage: F.antipode_on_basis(Composition([2,1]))
-F[2, 1]
```

QuasiSymmetricFunctions.Fundamental.**coproduct_on_basis**(*compo*)

Return the coproduct to a Fundamental quasi-symmetric basis element.

Combinatorial rule: quasi-deconcatenation.

INPUT:

- compo – composition

OUTPUT:

- The application of the coproduct to the Fundamental quasi-symmetric function indexed by the composition compo.

EXAMPLES:

```
sage: F = QuasiSymmetricFunctions(QQ).Fundamental()
sage: F[4].coproduct()
F[] # F[4] + F[1] # F[3] + F[2] # F[2] + F[3] # F[1] + F[4] # F[]
sage: F[2,1,3].coproduct()
F[] # F[2, 1, 3] + F[1] # F[1, 1, 3] + F[2] # F[1, 3] + F[2, 1] # F[3] + F[2, 1, 1] # F[
```

TESTS:

```
sage: F.coproduct_on_basis(Composition([2,1,3]))
F[] # F[2, 1, 3] + F[1] # F[1, 1, 3] + F[2] # F[1, 3] + F[2, 1] # F[3] + F[2, 1, 1] # F[
sage: F.one().coproduct()           # generic for graded / graded connected
F[] # F[]
```

QuasiSymmetricFunctions.Fundamental.**dual**()

Return the dual basis to the Fundamental basis. This is the ribbon basis of the non-commutative symmetric functions.

OUTPUT:

•The ribbon basis of the non-commutative symmetric functions.

EXAMPLES:

```
sage: F = QuasiSymmetricFunctions(QQ).F()
sage: F.dual()
Non-Commutative Symmetric Functions over the Rational Field in the Ribbon basis
```

class QuasiSymmetricFunctions.**HazewinkelLambda**(*QSym*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The Hazewinkel lambda basis of the quasi-symmetric functions.

This basis goes back to [Haz2004], albeit it is indexed in a different way here. It is a multiplicative basis in a weak sense of this word (the product of any two basis elements is a basis element, but of course not the one obtained by concatenating the indexing compositions).

In [Haz2004], Hazewinkel showed that the $\mathbf{k}$-algebra QSym is a polynomial algebra. (The proof is correct but rests upon an unproven claim that the lexicographically largest term of the n-th shuffle power of a Lyndon word is the n-fold concatenation of this Lyndon word with itself, occurring $n!$ times in that shuffle power. But this can be deduced from Section 2 of [Rad1979]. See also Chapter 6 of [GriRei2014], specifically Theorem 6.99, for a complete proof.) More precisely, he showed that QSym is generated, as a free commutative $\mathbf{k}$-algebra, by the elements $\lambda^n(M_I)$, where n ranges over the positive integers, and I ranges over all compositions which are Lyndon words and whose entries have gcd 1. Here, λ^n denotes the n-th lambda operation as explained in lambda_of_monomial().

Thus, products of these generators form a $\mathbf{k}$-module basis of QSym. We index this basis by compositions here. More precisely, we define the Hazewinkel lambda basis $(\mathrm{HWL}_I)_I$ (with I ranging over all compositions) as follows:

Let I be a composition. Let $I = I_1 I_2 \ldots I_k$ be the Chen-Fox-Lyndon factorization of I (see lyndon_factorization()). For every $j \in \{1, 2, \ldots, k\}$, let g_j be the gcd of the entries of the Lyndon word I_j, and let J_j be the result of dividing the entries of I_j by this gcd. Then, HWL_I is defined to be $\prod_{j=1}^{k} \lambda^{g_j}(M_{J_j})$.

Todo

The conversion from the M basis to the HWL basis is currently implemented in the naive way (inverting the base-change matrix in the other direction). This matrix is not triangular (not even after any permutations of the bases), and there could very well be a faster method (the one given by Hazewinkel?).

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: HWL = QSym.HazewinkelLambda()
sage: M = QSym.M()
sage: M(HWL([2]))
M[1, 1]
sage: M(HWL([1,1]))
2*M[1, 1] + M[2]
sage: M(HWL([1,2]))
M[1, 2]
sage: M(HWL([2,1]))
3*M[1, 1, 1] + M[1, 2] + M[2, 1]
sage: M(HWL(Composition([])))
M[]
sage: HWL(M([1,1]))
HWL[2]
sage: HWL(M(Composition([2])))
HWL[1, 1] - 2*HWL[2]
sage: HWL(M([1]))
HWL[1]
```

TESTS:

Transforming from the M-basis into the HWL-basis and back returns us to where we started:

```
sage: all( M(HWL(M[I])) == M[I] for I in Compositions(3) )
True
sage: all( HWL(M(HWL[I])) == HWL[I] for I in Compositions(4) )
True
```

Checking the HWL basis elements corresponding to Lyndon words:

```
sage: all( M(HWL[Composition(I)])
....:      == M.lambda_of_monomial([i // gcd(I) for i in I], gcd(I))
....:      for I in LyndonWords(e=3, k=2) )
True
```

product_on_basis(*I*, *J*)

The product on Hazewinkel Lambda basis elements.

The product of the basis elements indexed by two compositions I and J is the basis element obtained by concatenating the Lyndon factorizations of the words I and J, then reordering the Lyndon factors in nonincreasing order, and finally concatenating them in this order (giving a new composition).

INPUT:

- I, J – compositions

OUTPUT:

- The product of the Hazewinkel Lambda quasi-symmetric functions indexed by I and J, expressed in the Hazewinkel Lambda basis.

EXAMPLES:

```
sage: HWL = QuasiSymmetricFunctions(QQ).HazewinkelLambda()
sage: c1 = Composition([1, 2, 1])
sage: c2 = Composition([2, 1, 3, 2])
sage: HWL.product_on_basis(c1, c2)
HWL[2, 1, 3, 2, 1, 2, 1]
sage: HWL.product_on_basis(c1, Composition([]))
HWL[1, 2, 1]
sage: HWL.product_on_basis(Composition([]), Composition([]))
HWL[]
```

TESTS:

```
sage: M = QuasiSymmetricFunctions(QQ).M()
sage: all( all( M(HWL[I] * HWL[J]) == M(HWL[I]) * M(HWL[J])  # long time
....:           for I in Compositions(3) )
....:       for J in Compositions(3) )
True
```

class QuasiSymmetricFunctions.**Monomial**(*QSym*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The Hopf algebra of quasi-symmetric function in the Monomial basis.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.M()
sage: F = QSym.F()
sage: M(F[2,2])
M[1, 1, 1, 1] + M[1, 1, 2] + M[2, 1, 1] + M[2, 2]
sage: m = SymmetricFunctions(QQ).m()
sage: M(m[3,1,1])
M[1, 1, 3] + M[1, 3, 1] + M[3, 1, 1]
sage: (1+M[1])^3
M[] + 3*M[1] + 6*M[1, 1] + 6*M[1, 1, 1] + 3*M[1, 2] + 3*M[2] + 3*M[2, 1] + M[3]
sage: M[1,2,1].coproduct()
M[] # M[1, 2, 1] + M[1] # M[2, 1] + M[1, 2] # M[1] + M[1, 2, 1] # M[]
```

The following is an alias for this basis:

```
sage: QSym.Monomial()
Quasisymmetric functions over the Rational Field in the Monomial basis
```

TESTS:

```
sage: M(F([]))
M[]
sage: M(F(0))
0
sage: M(m([]))
M[]
```

class **Element**(*M*, *x*)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Element methods of the Monomial basis of QuasiSymmetricFunctions.

expand(*n*, *alphabet='x'*)

Expand the quasi-symmetric function written in the monomial basis in n variables.

INPUT:

- n – an integer
- alphabet – (default: 'x') a string

OUTPUT:

- The quasi-symmetric function self expressed in the n variables described by alphabet.

Todo

accept an *alphabet* as input

EXAMPLES:

```
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: M[4,2].expand(3)
x0^4*x1^2 + x0^4*x2^2 + x1^4*x2^2
```

One can use a different set of variables by using the optional argument `alphabet`:

```
sage: M=QuasiSymmetricFunctions(QQ).Monomial()
sage: M[2,1,1].expand(4,alphabet='y')
y0^2*y1*y2 + y0^2*y1*y3 + y0^2*y2*y3 + y1^2*y2*y3
```

TESTS:

```
sage: (3*M([])).expand(2)
3
sage: M[4,2].expand(0)
0
sage: M([]).expand(0)
1
```

is_symmetric()

Determine if a quasi-symmetric function, written in the Monomial basis, is symmetric.

This is being tested by looking at the expansion in the Monomial basis and checking if the coefficients are the same if the indexing compositions are permutations of each other.

OUTPUT:

- `True` if `self` is an element of the symmetric functions and `False` otherwise.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.Monomial()
sage: (M[3,2] + M[2,3] + M[4,1]).is_symmetric()
False
sage: (M[3,2] + M[2,3]).is_symmetric()
True
sage: (M[1,2,1] + M[1,1,2]).is_symmetric()
False
sage: (M[1,2,1] + M[1,1,2] + M[2,1,1]).is_symmetric()
True
```

psi_involution()

Return the image of the quasisymmetric function `self` under the involution ψ.

The involution ψ is defined as the linear map $QSym \to QSym$ which, for every composition I, sends the fundamental quasisymmetric function F_I to F_{I^c}, where I^c denotes the complement of the composition I. The map ψ is an involution and a graded Hopf algebra automorphism of $QSym$. Its restriction to the ring of symmetric functions coincides with the omega automorphism of the latter ring.

The involution ψ of $QSym$ is adjoint to the involution ψ of $NCSF$ by the standard adjunction between $NCSF$ and $QSym$.

The involution ψ has been denoted by ψ in [LMvW13], section 3.6.

See also:

`psi involution on QSym`, `psi involution on NCSF`, `star involution on QSym`.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: M = QSym.M()
sage: M[3,2].psi_involution()
```

```
-M[3, 2] - M[5]
sage: M[3,1].psi_involution()
M[3, 1] + M[4]
sage: M[3,1,1].psi_involution()
M[3, 1, 1] + M[3, 2] + M[4, 1] + M[5]
sage: M[1,1,1].psi_involution()
M[1, 1, 1] + M[1, 2] + M[2, 1] + M[3]
sage: M[[]].psi_involution()
M[]
sage: M(0).psi_involution()
0
sage: (2*M[[]] - M[3,1] + 4*M[2]).psi_involution()
2*M[] - 4*M[2] - M[3, 1] - M[4]
```

This particular implementation is tailored to the monomial basis. It is semantically equivalent to the generic implementation it overshadows:

```
sage: F = QSym.F()
sage: all( F(M[I].psi_involution()) == F(M[I]).psi_involution()
....:      for I in Compositions(3) )
True

sage: F = QSym.F()
sage: all( F(M[I].psi_involution()) == F(M[I]).psi_involution()
....:      for I in Compositions(4) )
True
```

to_symmetric_function()

Take a quasi-symmetric function, expressed in the monomial basis, and return its symmetric realization, when possible, expressed in the monomial basis of symmetric functions.

OUTPUT:

- If `self` is a symmetric function, then the expansion in the monomial basis of the symmetric functions is returned. Otherwise an error is raised.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: M = QSym.Monomial()
sage: (M[3,2] + M[2,3] + M[4,1]).to_symmetric_function()
Traceback (most recent call last):
...
ValueError: M[2, 3] + M[3, 2] + M[4, 1] is not a symmetric function
sage: (M[3,2] + M[2,3] + 2*M[4,1] + 2*M[1,4]).to_symmetric_function()
m[3, 2] + 2*m[4, 1]
sage: m = SymmetricFunctions(QQ).m()
sage: M(m[3,1,1]).to_symmetric_function()
m[3, 1, 1]
sage: (M(m[2,1])*M(m[2,1])).to_symmetric_function()-m[2,1]*m[2,1]
0
```

TESTS:

```
sage: (M(0)).to_symmetric_function()
0
sage: (M([])).to_symmetric_function()
m[]
sage: (2*M([])).to_symmetric_function()
2*m[]
```

We check that the result is indexed by partitions:

```
sage: M([]).to_symmetric_function().leading_support().parent()
Partitions
```

QuasiSymmetricFunctions.Monomial.**antipode_on_basis**(*compo*)

Return the result of the antipode applied to a quasi-symmetric Monomial basis element.

INPUT:

- compo – composition

OUTPUT:

- The result of the antipode applied to the composition compo, expressed in the Monomial basis.

EXAMPLES:

```
sage: M = QuasiSymmetricFunctions(QQ).M()
sage: M.antipode_on_basis(Composition([2,1]))
M[1, 2] + M[3]
sage: M.antipode_on_basis(Composition([]))
M[]
```

QuasiSymmetricFunctions.Monomial.**coproduct_on_basis**(*compo*)

Return the coproduct of a Monomial basis element.

Combinatorial rule: deconcatenation.

INPUT:

- compo – composition

OUTPUT:

- The coproduct applied to the Monomial quasi-symmetric function indexed by compo, expressed in the Monomial basis.

EXAMPLES:

```
sage: M=QuasiSymmetricFunctions(QQ).Monomial()
sage: M[4,2,3].coproduct()
M[] # M[4, 2, 3] + M[4] # M[2, 3] + M[4, 2] # M[3] + M[4, 2, 3] # M[]
sage: M.coproduct_on_basis(Composition([]))
M[] # M[]
```

QuasiSymmetricFunctions.Monomial.**dual**()

Return the dual basis to the Monomial basis. This is the complete basis of the non-commutative symmetric functions.

OUTPUT:

- The complete basis of the non-commutative symmetric functions.

EXAMPLES:

```
sage: M = QuasiSymmetricFunctions(QQ).M()
sage: M.dual()
Non-Commutative Symmetric Functions over the Rational Field in the Complete basis
```

QuasiSymmetricFunctions.Monomial.**lambda_of_monomial**(*I*, *n*)

Return the image of the monomial quasi-symmetric function M_I under the lambda-map λ^n, expanded in the monomial basis.

The ring of quasi-symmetric functions over the integers, $\mathrm{QSym}_{\mathbf{Z}}$ (and more generally, the ring of quasi-symmetric functions over any binomial ring) becomes a λ-ring (with the λ-structure inherited from the ring of formal power series, so that $\lambda^i(x_j)$ is x_j if $i = 1$ and 0 if $i > 1$).

The Adams operations of this λ-ring are the Frobenius endomorphisms $\mathbf{f}_n$ (see frobenius() for their definition). Using these endomorphisms, the λ-operations can be explicitly computed via the

formula

$$\exp\left(-\sum_{n=1}^{\infty}\frac{1}{n}\mathbf{f}_n(x)t^n\right)=\sum_{j=0}^{\infty}(-1)^j\lambda^j(x)t^j$$

in the ring of formal power series in a variable t over the ring of quasi-symmetric functions. In particular, every composition $I=(I_1,I_2,\cdots,I_\ell)$ satisfies

$$\exp\left(-\sum_{n=1}^{\infty}\frac{1}{n}M_{(nI_1,nI_2,\cdots,nI_\ell)}t^n\right)=\sum_{j=0}^{\infty}(-1)^j\lambda^j(M_I)t^j$$

(corrected version of Remark 2.4 in [Haz2004]).

The quasi-symmetric functions $\lambda^i(M_I)$ with n ranging over the positive integers and I ranging over the reduced Lyndon compositions (i. e., compositions which are Lyndon words and have the gcd of their entries equal to 1) form a set of free polynomial generators for QSym. See [GriRei2014], Chapter 6, for the proof, and [Haz2004] for a major part of it.

INPUT:

- `I` – composition
- `n` – nonnegative integer

OUTPUT:

The quasi-symmetric function $\lambda^n(M_I)$, expanded in the monomial basis over the ground ring of `self`.

EXAMPLES:

```
sage: M = QuasiSymmetricFunctions(CyclotomicField()).Monomial()
sage: M.lambda_of_monomial([1, 2], 2)
2*M[1, 1, 2, 2] + M[1, 1, 4] + M[1, 2, 1, 2] + M[1, 3, 2] + M[2, 2, 2]
sage: M.lambda_of_monomial([1, 1], 2)
3*M[1, 1, 1, 1] + M[1, 1, 2] + M[1, 2, 1] + M[2, 1, 1]
sage: M = QuasiSymmetricFunctions(Integers(19)).Monomial()
sage: M.lambda_of_monomial([1, 2], 3)
6*M[1, 1, 1, 2, 2, 2] + 3*M[1, 1, 1, 2, 4] + 3*M[1, 1, 1, 4, 2]
+ M[1, 1, 1, 6] + 4*M[1, 1, 2, 1, 2, 2] + 2*M[1, 1, 2, 1, 4]
+ 2*M[1, 1, 2, 2, 1, 2] + 2*M[1, 1, 2, 3, 2] + 4*M[1, 1, 3, 2, 2]
+ 2*M[1, 1, 3, 4] + M[1, 1, 4, 1, 2] + M[1, 1, 5, 2]
+ 2*M[1, 2, 1, 1, 2, 2] + M[1, 2, 1, 1, 4] + M[1, 2, 1, 2, 1, 2]
+ M[1, 2, 1, 3, 2] + 4*M[1, 2, 2, 2, 2] + M[1, 2, 2, 4] + M[1, 2, 4, 2]
+ 2*M[1, 3, 1, 2, 2] + M[1, 3, 1, 4] + M[1, 3, 2, 1, 2] + M[1, 3, 3, 2]
+ M[1, 4, 2, 2] + 3*M[2, 1, 2, 2, 2] + M[2, 1, 2, 4] + M[2, 1, 4, 2]
+ 2*M[2, 2, 1, 2, 2] + M[2, 2, 1, 4] + M[2, 2, 2, 1, 2] + M[2, 2, 3, 2]
+ 2*M[2, 3, 2, 2] + M[2, 3, 4] + M[3, 2, 2, 2]
```

The map λ^0 sends everything to 1:

```
sage: M = QuasiSymmetricFunctions(ZZ).Monomial()
sage: all( M.lambda_of_monomial(I, 0) == M.one()
....:      for I in Compositions(3) )
True
```

The map λ^1 is the identity map:

```
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: all( M.lambda_of_monomial(I, 1) == M(I)
....:      for I in Compositions(3) )
True
sage: M = QuasiSymmetricFunctions(Integers(5)).Monomial()
sage: all( M.lambda_of_monomial(I, 1) == M(I)
```

```
....:       for I in Compositions(3) )
True
sage: M = QuasiSymmetricFunctions(ZZ).Monomial()
sage: all( M.lambda_of_monomial(I, 1) == M(I)
....:       for I in Compositions(3) )
True
```

QuasiSymmetricFunctions.Monomial.**product_on_basis**(*I*, *J*)

The product on Monomial basis elements.

The product of the basis elements indexed by two compositions I and J is the sum of the basis elements indexed by compositions in the stuffle product (also called the overlapping shuffle product) of I and J.

INPUT:

- I, J – compositions

OUTPUT:

- The product of the Monomial quasi-symmetric functions indexed by I and J, expressed in the Monomial basis.

EXAMPLES:

```
sage: M = QuasiSymmetricFunctions(QQ).Monomial()
sage: c1 = Composition([2])
sage: c2 = Composition([1,3])
sage: M.product_on_basis(c1, c2)
M[1, 2, 3] + M[1, 3, 2] + M[1, 5] + M[2, 1, 3] + M[3, 3]
sage: M.product_on_basis(c1, Composition([]))
M[2]
```

class QuasiSymmetricFunctions.**Quasisymmetric_Schur**(*QSym*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The Hopf algebra of quasi-symmetric function in the Quasisymmetric Schur basis.

The basis of Quasisymmetric Schur functions is defined in [QSCHUR] and in Definition 5.1.1 of [LMvW13]. Don't mistake them for the completely unrelated quasi-Schur functions of [NCSF1]!

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: QS = QSym.QS()
sage: F = QSym.F()
sage: M = QSym.M()
sage: F(QS[1,2])
F[1, 2]
sage: M(QS[1,2])
M[1, 1, 1] + M[1, 2]
sage: s = SymmetricFunctions(QQ).s()
sage: QS(s[2,1,1])
QS[1, 1, 2] + QS[1, 2, 1] + QS[2, 1, 1]
```

QuasiSymmetricFunctions.**a_realization**()

Return the realization of the Monomial basis of the ring of quasi-symmetric functions.

OUTPUT:

- The Monomial basis of quasi-symmetric functions.

EXAMPLES:

```
sage: QuasiSymmetricFunctions(QQ).a_realization()
Quasisymmetric functions over the Rational Field in the Monomial basis
```

QuasiSymmetricFunctions.**dual**()

Return the dual Hopf algebra of the quasi-symmetric functions, which is the non-commutative symmetric functions.

OUTPUT:

- The non-commutative symmetric functions.

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: QSym.dual()
Non-Commutative Symmetric Functions over the Rational Field
```

class QuasiSymmetricFunctions.**dualImmaculate**(*QSym*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The dual immaculate basis of the quasi-symmetric functions.

This basis first appears in [BBSSZ2012].

EXAMPLES:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: dI = QSym.dI()
sage: dI([1,3,2])*dI([1])  # long time (6s on sage.math, 2013)
dI[1, 1, 3, 2] + dI[2, 3, 2]
sage: dI([1,3])*dI([1,1])
dI[1, 1, 1, 3] + dI[1, 1, 4] + dI[1, 2, 3] - dI[1, 3, 2] - dI[1, 4, 1] - dI[1, 5] + dI[2, 3,
sage: dI([3,1])*dI([2,1])  # long time (7s on sage.math, 2013)
dI[1, 1, 5] - dI[1, 4, 1, 1] - dI[1, 4, 2] - 2*dI[1, 5, 1] - dI[1, 6] - dI[2, 4, 1] - dI[2,
sage: F = QSym.F()
sage: dI(F[1,3,1])
-dI[1, 1, 1, 2] + dI[1, 1, 2, 1] - dI[1, 2, 2] + dI[1, 3, 1]
sage: F(dI(F([2,1,3])))
F[2, 1, 3]
```

QuasiSymmetricFunctions.**from_polynomial**(*f*, *check=True*)

Return the quasi-symmetric function in the Monomial basis corresponding to the quasi-symmetric polynomial `f`.

INPUT:

- `f` – a polynomial in finitely many variables over the same base ring as `self`. It is assumed that this polynomial is quasi-symmetric.
- `check` – boolean (default: `True`), checks whether the polynomial is indeed quasi-symmetric.

OUTPUT:

- quasi-symmetric function in the Monomial basis

EXAMPLES:

```
sage: P = PolynomialRing(QQ, 'x', 3)
sage: x = P.gens()
sage: f = x[0] + x[1] + x[2]
sage: QSym = QuasiSymmetricFunctions(QQ)
```

```
sage: QSym.from_polynomial(f)
M[1]
```

Beware of setting `check=False`:

```
sage: f = x[0] + 2*x[1] + x[2]
sage: QSym.from_polynomial(f, check=True)
Traceback (most recent call last):
...
ValueError: x0 + 2*x1 + x2 is not a quasi-symmetric polynomial
sage: QSym.from_polynomial(f, check=False)
M[1]
```

To expand the quasi-symmetric function in a basis other than the Monomial basis, the following shorthands are provided:

```
sage: M = QSym.Monomial()
sage: f = x[0]**2+x[1]**2+x[2]**2
sage: g = M.from_polynomial(f); g
M[2]
sage: F = QSym.Fundamental()
sage: F(g)
-F[1, 1] + F[2]
sage: F.from_polynomial(f)
-F[1, 1] + F[2]
```

5.1.131 Introduction to Quasisymmetric Functions

In this document we briefly explain the quasisymmetric function bases and related functionality in Sage. We assume the reader is familar with the package `SymmetricFunctions`.

Quasisymmetric functions, denoted $QSym$, form a subring of the power series ring in countably many variables. $QSym$ contains the symmetric functions. These functions first arose in the theory of P-partitions. The initial ideas in this field are attributed to MacMahon, Knuth, Kreweras, Glânffrwd Thomas, Stanley. In 1984, Gessel formalized the study of quasisymmetric functions and introduced the basis of fundamental quasisymmetric functions [Ges]. In 1995, Gelfand, Krob, Lascoux, Leclerc, Retakh, and Thibon showed that the ring of quasisymmetric functions is Hopf dual to the noncommutative symmetric functions [NCSF]. Many results have built on these.

One advantage of working in $QSym$ is that many interesting families of symmetric functions have explicit expansions in fundamental quasisymmetric functions such as Schur functions [Ges], Macdonald polynomials [HHL05], and plethysm of Schur functions [LW12].

For more background see Wikipedia article Quasisymmetric_function.

To begin, initialize the ring. Below we chose to use the rational numbers $\mathbf{Q}$. Other options include the integers $\mathbf{Z}$ and $\mathbf{C}$:

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: QSym
Quasisymmetric functions over the Rational Field

sage: QSym = QuasiSymmetricFunctions(CC); QSym
Quasisymmetric functions over the Complex Field with 53 bits of precision

sage: QSym = QuasiSymmetricFunctions(ZZ); QSym
Quasisymmetric functions over the Integer Ring
```

All bases of $QSym$ are indexed by compositions e.g. $[3, 1, 1, 4]$. The convention is to use capital letters for bases of $QSym$ and lowercase letters for bases of the symmetric functions Sym. Next set up names for the known bases by running `inject_shorthands()`. As with symmetric functions, you do not need to run this command and you could assign these bases other names.

```
sage: QSym = QuasiSymmetricFunctions(QQ)
sage: QSym.inject_shorthands()
Injecting M as shorthand for Quasisymmetric functions over the Rational Field in the Monomial basis
Injecting F as shorthand for Quasisymmetric functions over the Rational Field in the Fundamental basi
Injecting dI as shorthand for Quasisymmetric functions over the Rational Field in the dualImmaculate
Injecting QS as shorthand for Quasisymmetric functions over the Rational Field in the Quasisymmetric
```

Now one can start constructing quasisymmetric functions.

Note: It is best to use variables other than `M` and `F`.

```
sage: x = M[2,1] + M[1,2]
sage: x
M[1, 2] + M[2, 1]

sage: y = 3*M[1,2] + M[3]^2; y
3*M[1, 2] + 2*M[3, 3] + M[6]

sage: F[3,1,3] + 7*F[2,1]
7*F[2, 1] + F[3, 1, 3]

sage: 3*F[2,1,2] + F[3]^2
F[1, 2, 2, 1] + F[1, 2, 3] + 2*F[1, 3, 2] + F[1, 4, 1] + F[1, 5] + 3*F[2, 1, 2]
 + 2*F[2, 2, 2] + 2*F[2, 3, 1] + 2*F[2, 4] + F[3, 2, 1] + 3*F[3, 3] + 2*F[4, 2] + F[5, 1] + F[6]
```

To convert from one basis to another is easy:

```
sage: z = M[1,2,1]
sage: z
M[1, 2, 1]

sage: F(z)
-F[1, 1, 1, 1] + F[1, 2, 1]

sage: M(F(z))
M[1, 2, 1]
```

To expand in variables, one can specify a finite size alphabet $x_1, x_2, \ldots, x_m$:

```
sage: y = M[1,2,1]
sage: y.expand(4)
x0*x1^2*x2 + x0*x1^2*x3 + x0*x2^2*x3 + x1*x2^2*x3
```

The usual methods on free modules are available such as coefficients, degrees, and the support:

```
sage: z=3*M[1,2]+M[3]^2; z
3*M[1, 2] + 2*M[3, 3] + M[6]

sage: z.coefficient([1,2])
3

sage: z.degree()
6
```

```
sage: sorted(z.coefficients())
[1, 2, 3]

sage: sorted(z.monomials(), key=lambda x: x.support())
[M[1, 2], M[3, 3], M[6]]

sage: z.monomial_coefficients()
{[1, 2]: 3, [3, 3]: 2, [6]: 1}
```

As with the symmetric functions package, the quasisymmetric function `1` has several instantiations. However, the most obvious way to write `1` leads to an error (this is due to the semantics of python):

```
sage: M[[]]
M[]
sage: M.one()
M[]
sage: M(1)
M[]
sage: M[[]] == 1
True
sage: M[]
Traceback (most recent call last):
...
SyntaxError: invalid syntax
```

Working with symmetric functions

The quasisymmetric functions are a ring which contains the symmetric functions as a subring. The Monomial quasisymmetric functions are related to the monomial symmetric functions by $m_\lambda = \sum_{\mathrm{sort}(c)=\lambda} M_c$, where $\mathrm{sort}(c)$ means the partition obtained by sorting the composition c:

```
sage: SymmetricFunctions(QQ).inject_shorthands()
doctest:...: RuntimeWarning: redefining global value `e`
sage: m[2,1]
m[2, 1]
sage: M(m[2,1])
M[1, 2] + M[2, 1]
sage: M(s[2,1])
2*M[1, 1, 1] + M[1, 2] + M[2, 1]
```

There are methods to test if an expression f in the quasisymmetric functions is a symmetric function:

```
sage: f = M[1,1,2] + M[1,2,1]
sage: f.is_symmetric()
False
sage: f = M[3,1] + M[1,3]
sage: f.is_symmetric()
True
```

If f is symmetric, there are methods to convert f to an expression in the symmetric functions:

```
sage: f.to_symmetric_function()
m[3, 1]
```

The expansion of the Schur function in terms of the Fundamental quasisymmetric functions is due to [Ges]. There is one term in the expansion for each standard tableau of shape equal to the partition indexing the Schur function.

```
sage: f = F[3,2] + F[2,2,1] + F[2,3] + F[1,3,1] + F[1,2,2]
sage: f.is_symmetric()
True
sage: f.to_symmetric_function()
5*m[1, 1, 1, 1, 1] + 3*m[2, 1, 1, 1] + 2*m[2, 2, 1] + m[3, 1, 1] + m[3, 2]
sage: s(f.to_symmetric_function())
s[3, 2]
```

It is also possible to convert any symmetric function to the quasisymmetric function expansion in any known basis. The converse is not true:

```
sage: M( m[3,1,1] )
M[1, 1, 3] + M[1, 3, 1] + M[3, 1, 1]
sage: F( s[2,2,1] )
F[1, 1, 2, 1] + F[1, 2, 1, 1] + F[1, 2, 2] + F[2, 1, 2] + F[2, 2, 1]

sage: s(M[2,1])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= M[2, 1]) an element of self
```

It is possible to experiment with the quasisymmetric function expansion of other bases, but it is important that the base ring be the same for both algebras.

```
sage: R = QQ['t']
sage: Qp = SymmetricFunctions(R).hall_littlewood().Qp()
sage: QSymt = QuasiSymmetricFunctions(R)
sage: Ft = QSymt.F()
sage: Ft( Qp[2,2] )
F[1, 2, 1] + t*F[1, 3] + (t+1)*F[2, 2] + t*F[3, 1] + t^2*F[4]

sage: K = QQ['q','t'].fraction_field()
sage: Ht = SymmetricFunctions(K).macdonald().Ht()
sage: Fqt = QuasiSymmetricFunctions(Ht.base_ring()).F()
sage: Fqt(Ht[2,1])
q*t*F[1, 1, 1] + (q+t)*F[1, 2] + (q+t)*F[2, 1] + F[3]
```

The following will raise an error because the base ring of `F` is not equal to the base ring of `Ht`:

```
sage: F(Ht[2,1])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= McdHt[2, 1]) an element of self (=Quasisymmetric functions ov
```

QSym is a Hopf algebra

The product on $QSym$ is commutative and is inherited from the product by the realization within the polynomial ring:

```
sage: M[3]*M[1,1] == M[1,1]*M[3]
True
sage: M[3]*M[1,1]
M[1, 1, 3] + M[1, 3, 1] + M[1, 4] + M[3, 1, 1] + M[4, 1]
sage: F[3]*F[1,1]
F[1, 1, 3] + F[1, 2, 2] + F[1, 3, 1] + F[1, 4] + F[2, 1, 2] + F[2, 2, 1] + F[2, 3] + F[3, 1, 1] + F[
sage: M[3]*F[2]
M[1, 1, 3] + M[1, 3, 1] + M[1, 4] + M[2, 3] + M[3, 1, 1] + M[3, 2] + M[4, 1] + M[5]
```

```
sage: F[2]*M[3]
F[1, 1, 1, 2] - F[1, 2, 2] + F[2, 1, 1, 1] - F[2, 1, 2] - F[2, 2, 1] + F[5]
```

There is a coproduct on this ring as well, which in the Monomial basis acts by cutting the composition into a left half and a right half. The co-product is non-co-commutative:

```
sage: M[1,3,1].coproduct()
M[] # M[1, 3, 1] + M[1] # M[3, 1] + M[1, 3] # M[1] + M[1, 3, 1] # M[]
sage: F[1,3,1].coproduct()
F[] # F[1, 3, 1] + F[1] # F[3, 1] + F[1, 1] # F[2, 1] + F[1, 2] # F[1, 1] + F[1, 3] # F[1] + F[1, 3,
```

The Duality Pairing with Non-Commutative Symmetric Functions

These two operations endow $QSym$ with the structure of a Hopf algebra. It is the dual Hopf algebra of the non-commutative symmetric functions $NCSF$. Under this duality, the Monomial basis of $QSym$ is dual to the Complete basis of $NCSF$, and the Fundamental basis of $QSym$ is dual to the Ribbon basis of $NCSF$ (see [MR]):

```
sage: S = M.dual(); S
Non-Commutative Symmetric Functions over the Rational Field in the Complete basis
sage: M[1,3,1].duality_pairing( S[1,3,1] )
1
sage: M.duality_pairing_matrix( S, degree=4 )
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
sage: F.duality_pairing_matrix( S, degree=4 )
[1 0 0 0 0 0 0 0]
[1 1 0 0 0 0 0 0]
[1 0 1 0 0 0 0 0]
[1 1 1 1 0 0 0 0]
[1 0 0 0 1 0 0 0]
[1 1 0 0 1 1 0 0]
[1 0 1 0 1 0 1 0]
[1 1 1 1 1 1 1 1]
sage: NCSF = M.realization_of().dual()
sage: R = NCSF.Ribbon()
sage: F.duality_pairing_matrix( R, degree=4 )
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
sage: M.duality_pairing_matrix( R, degree=4 )
[ 1  0  0  0  0  0  0  0]
[-1  1  0  0  0  0  0  0]
[-1  0  1  0  0  0  0  0]
[ 1 -1 -1  1  0  0  0  0]
[-1  0  0  0  1  0  0  0]
```

```
[ 1 -1  0  0 -1  1  0  0]
[ 1  0 -1  0 -1  0  1  0]
[-1  1  1 -1  1 -1 -1  1]
```

Let H and G be elements of $QSym$ and h an element of $NCSF$. Then if we represent the duality pairing with the mathematical notation $[\cdot, \cdot]$, we have:

$$[H \cdot G, h] = [H \otimes G, \Delta(h)].$$

For example, the coefficient of `M[2,1,4,1]` in `M[1,3]*M[2,1,1]` may be computed with the duality pairing:

```
sage: I, J = Composition([1,3]), Composition([2,1,1])
sage: (M[I]*M[J]).duality_pairing(S[2,1,4,1])
1
```

And the coefficient of `S[1,3] # S[2,1,1]` in `S[2,1,4,1].coproduct()` is equal to this result:

```
sage: S[2,1,4,1].coproduct()
S[] # S[2, 1, 4, 1] + ... + S[1, 3] # S[2, 1, 1] + ... + S[4, 1] # S[2, 1]
```

The duality pairing on the tensor space is another way of getting this coefficient, but currently the method `duality_pairing()` is not defined on the tensor squared space. However, we can extend this functionality by applying a linear morphism to the terms in the coproduct, as follows:

```
sage: X = S[2,1,4,1].coproduct()
sage: def linear_morphism(x, y):
....:     return x.duality_pairing(M[1,3]) * y.duality_pairing(M[2,1,1])
sage: X.apply_multilinear_morphism(linear_morphism, codomain=ZZ)
1
```

Similarly, if H is an element of $QSym$ and g and h are elements of $NCSF$, then

$$[H, g \cdot h] = [\Delta(H), g \otimes h].$$

For example, the coefficient of `R[2,3,1]` in `R[2,1]*R[2,1]` is computed with the duality pairing by the following command:

```
sage: (R[2,1]*R[2,1]).duality_pairing(F[2,3,1])
1
sage: R[2,1]*R[2,1]
R[2, 1, 2, 1] + R[2, 3, 1]
```

This coefficient should then be equal to the coefficient of `F[2,1] # F[2,1]` in `F[2,3,1].coproduct()`:

```
sage: F[2,3,1].coproduct()
F[] # F[2, 3, 1] + ... + F[2, 1] # F[2, 1]  + ... + F[2, 3, 1] # F[]
```

This can also be computed by the duality pairing on the tensor space, as above:

```
sage: X = F[2,3,1].coproduct()
sage: def linear_morphism(x, y):
....:     return x.duality_pairing(R[2,1]) * y.duality_pairing(R[2,1])
sage: X.apply_multilinear_morphism(linear_morphism, codomain=ZZ)
1
```

The Operation Adjoint to Multiplication by a Non-Commutative Symmetric Function

Let $g \in NCSF$ and consider the linear endomorphism of $NCSF$ defined by left (respectively, right) multiplication by g. Since there is a duality between $QSym$ and $NCSF$, this linear transformation induces an operator $g^\perp$ on $QSym$ satisfying

$$[g^\perp(H), h] = [H, g \cdot h].$$

for any non-commutative symmetric function h.

This is implemented by the method `skew_by()`. Explicitly, if `H` is a quasisymmetric function and `g` a non-commutative symmetric function, then `H.skew_by(g)` and `H.skew_by(g, side='right')` are expressions that satisfy, for any non-commutative symmetric function `h`, the following identities:

```
H.skew_by(g).duality_pairing(h) == H.duality_pairing(g*h)
H.skew_by(g, side='right').duality_pairing(h) == H.duality_pairing(h*g)
```

For example, `M[J].skew_by(S[I])` is 0 unless the composition J begins with I and `M(J).skew_by(S(I), side='right')` is 0 unless the composition J ends with I:

```
sage: M[3,2,2].skew_by(S[3])
M[2, 2]
sage: M[3,2,2].skew_by(S[2])
0
sage: M[3,2,2].coproduct().apply_multilinear_morphism( lambda x,y: x.duality_pairing(S[3])*y )
M[2, 2]
sage: M[3,2,2].skew_by(S[3], side='right')
0
sage: M[3,2,2].skew_by(S[2], side='right')
M[3, 2]
```

The antipode

The antipode sends the Fundamental basis element indexed by the composition I to -1 to the size of I times the Fundamental basis element indexed by the conjugate composition to I:

```
sage: F[3,2,2].antipode()
-F[1, 2, 2, 1, 1]
sage: Composition([3,2,2]).conjugate()
[1, 2, 2, 1, 1]
sage: M[3,2,2].antipode()
-M[2, 2, 3] - M[2, 5] - M[4, 3] - M[7]
```

We demonstrate here the defining relation of the antipode:

```
sage: X = F[3,2,2].coproduct()
sage: X.apply_multilinear_morphism(lambda x,y: x*y.antipode())
0
sage: X.apply_multilinear_morphism(lambda x,y: x.antipode()*y)
0
```

REFERENCES:

5.1.132 Symmetric Functions in Non-Commuting Variables

- Introduction to Symmetric Functions in Non-Commuting Variables
- *Bases for .*
- *Dual Symmetric Functions in Non-Commuting Variables*
- *Symmetric Functions in Non-Commuting Variables*

5.1.133 Features that are imported by default in the interpreter namespace

5.1.134 Bases for $NCSym$.

AUTHORS:

- Travis Scrimshaw (08-04-2013): Initial version

class sage.combinat.ncsym.bases.**MultiplicativeNCSymBases**(*parent_with_realization*)
Bases: sage.categories.realizations.Category_realization_of_parent

Category of multiplicative bases of symmetric functions in non-commuting variables.

A multiplicative basis is one for which $\mathbf{b}_A \mathbf{b}_B = \mathbf{b}_{A|B}$ where $A|B$ is the pipe() operation on set partitions.

EXAMPLES:

```
sage: from sage.combinat.ncsym.bases import MultiplicativeNCSymBases
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: MultiplicativeNCSymBases(NCSym)
Category of multiplicative bases of symmetric functions in non-commuting variables over the Rati
```

class ElementMethods

class MultiplicativeNCSymBases.**ParentMethods**

product_on_basis(*A*, *B*)
The product on basis elements.

The product on a multiplicative basis is given by $\mathbf{b}_A \cdot \mathbf{b}_B = \mathbf{b}_{A|B}$.

The bases $\{\mathbf{e}, \mathbf{h}, \mathbf{x}, \mathbf{cp}, \mathbf{p}, \mathbf{chi}, \mathbf{rho}\}$ are all multiplicative.

INPUT:
•A, B – set partitions
OUTPUT:
•an element in the basis self
EXAMPLES:

```
sage: e = SymmetricFunctionsNonCommutingVariables(QQ).e()
sage: h = SymmetricFunctionsNonCommutingVariables(QQ).h()
sage: x = SymmetricFunctionsNonCommutingVariables(QQ).x()
sage: cp = SymmetricFunctionsNonCommutingVariables(QQ).cp()
sage: p = SymmetricFunctionsNonCommutingVariables(QQ).p()
sage: chi = SymmetricFunctionsNonCommutingVariables(QQ).chi()
sage: rho = SymmetricFunctionsNonCommutingVariables(QQ).rho()
sage: A = SetPartition([[1], [2, 3]])
sage: B = SetPartition([[1], [3], [2,4]])
sage: e.product_on_basis(A, B)
e{{1}, {2, 3}, {4}, {5, 7}, {6}}
sage: h.product_on_basis(A, B)
h{{1}, {2, 3}, {4}, {5, 7}, {6}}
sage: x.product_on_basis(A, B)
```

```
x{{1}, {2, 3}, {4}, {5, 7}, {6}}
sage: cp.product_on_basis(A, B)
cp{{1}, {2, 3}, {4}, {5, 7}, {6}}
sage: p.product_on_basis(A, B)
p{{1}, {2, 3}, {4}, {5, 7}, {6}}
sage: chi.product_on_basis(A, B)
chi{{1}, {2, 3}, {4}, {5, 7}, {6}}
sage: rho.product_on_basis(A, B)
rho{{1}, {2, 3}, {4}, {5, 7}, {6}}
sage: e.product_on_basis(A,B)==e(h(e(A))*h(e(B)))
True
sage: h.product_on_basis(A,B)==h(x(h(A))*x(h(B)))
True
sage: x.product_on_basis(A,B)==x(h(x(A))*h(x(B)))
True
sage: cp.product_on_basis(A,B)==cp(p(cp(A))*p(cp(B)))
True
sage: p.product_on_basis(A,B)==p(e(p(A))*e(p(B)))
True
```

MultiplicativeNCSymBases.**super_categories**()

Return the super categories of bases of the Hopf dual of the symmetric functions in non-commuting variables.

OUTPUT:

- a list of categories

TESTS:

```
sage: from sage.combinat.ncsym.bases import MultiplicativeNCSymBases
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: MultiplicativeNCSymBases(NCSym).super_categories()
[Category of bases of symmetric functions in non-commuting variables over the Rational Field
```

class sage.combinat.ncsym.bases.**NCSymBases**(*parent_with_realization*)

Bases: sage.categories.realizations.Category_realization_of_parent

Category of bases of symmetric functions in non-commuting variables.

EXAMPLES:

```
sage: from sage.combinat.ncsym.bases import NCSymBases
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: NCSymBases(NCSym)
Category of bases of symmetric functions in non-commuting variables over the Rational Field
```

class ElementMethods

expand(*n*, *alphabet='x'*)

Expand the symmetric function into n non-commuting variables in an alphabet, which by default is 'x'.

This computation is completed by coercing the element self into the monomial basis and computing the expansion in the alphabet there.

INPUT:

- n – the number of variables in the expansion
- alphabet – (default: 'x') the alphabet in which self is to be expanded

OUTPUT:

•an expansion of `self` into the `n` non-commuting variables specified by `alphabet`

EXAMPLES:

```
sage: h = SymmetricFunctionsNonCommutingVariables(QQ).h()
sage: h[[1,3],[2]].expand(3)
2*x0^3 + x0^2*x1 + x0^2*x2 + 2*x0*x1*x0 + x0*x1^2 + x0*x1*x2 + 2*x0*x2*x0
 + x0*x2*x1 + x0*x2^2 + x1*x0^2 + 2*x1*x0*x1 + x1*x0*x2 + x1^2*x0 + 2*x1^3
 + x1^2*x2 + x1*x2*x0 + 2*x1*x2*x1 + x1*x2^2 + x2*x0^2 + x2*x0*x1 + 2*x2*x0*x2
 + x2*x1*x0 + x2*x1^2 + 2*x2*x1*x2 + x2^2*x0 + x2^2*x1 + 2*x2^3
sage: x = SymmetricFunctionsNonCommutingVariables(QQ).x()
sage: x[[1,3],[2]].expand(3)
-x0^2*x1 - x0^2*x2 - x0*x1^2 - x0*x1*x2 - x0*x2*x1 - x0*x2^2 - x1*x0^2
 - x1*x0*x2 - x1^2*x0 - x1^2*x2 - x1*x2*x0 - x1*x2^2 - x2*x0^2 - x2*x0*x1
 - x2*x1*x0 - x2*x1^2 - x2^2*x0 - x2^2*x1
```

internal_coproduct()

Return the internal coproduct of `self`.

The internal coproduct is defined on the power sum basis as

$$\mathbf{p}_A \mapsto \mathbf{p}_A \otimes \mathbf{p}_A$$

and the map is extended linearly.

OUTPUT:

•an element of the tensor square of the basis of `self`

EXAMPLES:

```
sage: x = SymmetricFunctionsNonCommutingVariables(QQ).x()
sage: x[[1,3],[2]].internal_coproduct()
x{{1}, {2}, {3}} # x{{1, 3}, {2}} + x{{1, 3}, {2}} # x{{1}, {2}, {3}}
 + x{{1, 3}, {2}} # x{{1, 3}, {2}}
```

omega()

Return the involution ω applied to `self`.

The involution ω is defined by

$$\mathbf{e}_A \mapsto \mathbf{h}_A$$

and the result is extended linearly.

OUTPUT:

•an element in the same basis as `self`

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: m = NCSym.m()
sage: m[[1,3],[2]].omega()
-2*m{{1, 2, 3}} - m{{1, 3}, {2}}
sage: p = NCSym.p()
sage: p[[1,3],[2]].omega()
-p{{1, 3}, {2}}
sage: cp = NCSym.cp()
sage: cp[[1,3],[2]].omega()
-2*cp{{1, 2, 3}} - cp{{1, 3}, {2}}
sage: x = NCSym.x()
sage: x[[1,3],[2]].omega()
-2*x{{1}, {2}, {3}} - x{{1, 3}, {2}}
```

to_symmetric_function()

Compute the projection of an element of symmetric function in non-commuting variables to the symmetric functions.

The projection of a monomial symmetric function in non-commuting variables indexed by the set partition A is defined as

$$\mathbf{m}_A \mapsto m_{\lambda(A)} \prod_i n_i(\lambda(A))!$$

where $\lambda(A)$ is the partition associated with A by taking the sizes of the parts and $n_i(\mu)$ is the multiplicity of i in μ. For other bases this map is extended linearly.

OUTPUT:

- an element of the symmetric functions in the monomial basis

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: e = NCSym.e()
sage: h = NCSym.h()
sage: p = NCSym.p()
sage: cp = NCSym.cp()
sage: x = NCSym.x()
sage: cp[[1,3],[2]].to_symmetric_function()
m[2, 1]
sage: x[[1,3],[2]].to_symmetric_function()
-6*m[1, 1, 1] - 2*m[2, 1]
sage: e[[1,3],[2]].to_symmetric_function()
2*e[2, 1]
sage: h[[1,3],[2]].to_symmetric_function()
2*h[2, 1]
sage: p[[1,3],[2]].to_symmetric_function()
p[2, 1]
```

class NCSymBases.**ParentMethods**

from_symmetric_function(*f*)

Return the image of the symmetric function f in self.

This is performed by converting to the monomial basis and extending the method sum_of_partitions() linearly. This is a linear map from the symmetric functions to the symmetric functions in non-commuting variables that does not preserve the product or coproduct structure of the Hopf algebra.

See also:

to_symmetric_function()

INPUT:

- f – a symmetric function

OUTPUT:

- an element of self

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: Sym = SymmetricFunctions(QQ)
sage: e = NCSym.e()
sage: elem = Sym.e()
sage: elt = e.from_symmetric_function(elem[2,1,1]); elt
1/12*e{{1}, {2}, {3, 4}} + 1/12*e{{1}, {2, 3}, {4}} + 1/12*e{{1}, {2, 4}, {3}}
 + 1/12*e{{1, 2}, {3}, {4}} + 1/12*e{{1, 3}, {2}, {4}} + 1/12*e{{1, 4}, {2}, {3}}
sage: elem(elt.to_symmetric_function())
e[2, 1, 1]
sage: e.from_symmetric_function(elem[4])
1/24*e{{1, 2, 3, 4}}
```

```
sage: p = NCSym.p()
sage: pow = Sym.p()
sage: elt = p.from_symmetric_function(pow[2,1,1]); elt
1/6*p{{1}, {2}, {3, 4}} + 1/6*p{{1}, {2, 3}, {4}} + 1/6*p{{1}, {2, 4}, {3}}
 + 1/6*p{{1, 2}, {3}, {4}} + 1/6*p{{1, 3}, {2}, {4}} + 1/6*p{{1, 4}, {2}, {3}}
sage: pow(elt.to_symmetric_function())
p[2, 1, 1]
sage: p.from_symmetric_function(pow[4])
p{{1, 2, 3, 4}}
sage: h = NCSym.h()
sage: comp = Sym.complete()
sage: elt = h.from_symmetric_function(comp[2,1,1]); elt
1/12*h{{1}, {2}, {3, 4}} + 1/12*h{{1}, {2, 3}, {4}} + 1/12*h{{1}, {2, 4}, {3}}
 + 1/12*h{{1, 2}, {3}, {4}} + 1/12*h{{1, 3}, {2}, {4}} + 1/12*h{{1, 4}, {2}, {3}}
sage: comp(elt.to_symmetric_function())
h[2, 1, 1]
sage: h.from_symmetric_function(comp[4])
1/24*h{{1, 2, 3, 4}}
```

internal_coproduct()

Compute the internal coproduct of `self`.

If `internal_coproduct_on_basis()` is available, construct the internal coproduct morphism from `self` to `self` $\otimes$ `self` by extending it by linearity. Otherwise, this uses `internal_coproduct_by_coercion()`, if available.

OUTPUT:

- an element of the tensor squared of `self`

EXAMPLES:

```
sage: cp = SymmetricFunctionsNonCommutingVariables(QQ).cp()
sage: cp.internal_coproduct(cp[[1,3],[2]] - 2*cp[[1]])
-2*cp{{1}} # cp{{1}} + cp{{1, 2, 3}} # cp{{1, 3}, {2}} + cp{{1, 3}, {2}} # cp{{1, 2, 3}}
 + cp{{1, 3}, {2}} # cp{{1, 3}, {2}}
```

internal_coproduct_by_coercion(x)

Return the internal coproduct by coercing the element to the powersum basis.

INPUT:

- `x` – an element of `self`

OUTPUT:

- an element of the tensor squared of `self`

EXAMPLES:

```
sage: h = SymmetricFunctionsNonCommutingVariables(QQ).h()
sage: h[[1,3],[2]].internal_coproduct() # indirect doctest
2*h{{1}, {2}, {3}} # h{{1}, {2}, {3}} - h{{1}, {2}, {3}} # h{{1, 3}, {2}}
 - h{{1, 3}, {2}} # h{{1}, {2}, {3}} + h{{1, 3}, {2}} # h{{1, 3}, {2}}
```

internal_coproduct_on_basis(i)

The internal coproduct of the algebra on the basis (optional).

INPUT:

- `i` – the indices of an element of the basis of `self`

OUTPUT:

- an element of the tensor squared of `self`

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: m.internal_coproduct_on_basis(SetPartition([[1,2]]))
m{{1, 2}} # m{{1, 2}}
```

primitive(*A*, *i=1*)

Return the primitive associated to `A` in `self`.

See also:

`primitive()`

INPUT:

- `A` – a set partition
- `i` – a positive integer

OUTPUT:

- an element of `self`

EXAMPLES:

```
sage: e = SymmetricFunctionsNonCommutingVariables(QQ).e()
sage: elt = e.primitive(SetPartition([[1,3],[2]])); elt
e{{1, 2}, {3}} - e{{1, 3}, {2}}
sage: elt.coproduct()
e{} # e{{1, 2}, {3}} - e{} # e{{1, 3}, {2}} + e{{1, 2}, {3}} # e{} - e{{1, 3}, {2}} # e{
```

NCSymBases.**super_categories**()

Return the super categories of bases of the Hopf dual of the symmetric functions in non-commuting variables.

OUTPUT:

- a list of categories

TESTS:

```
sage: from sage.combinat.ncsym.bases import NCSymBases
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: NCSymBases(NCSym).super_categories()
[Category of bases of NCSym or NCSym^* over the Rational Field]
```

class sage.combinat.ncsym.bases.**NCSymBasis_abstract**(*R*, *basis_keys*, *element_class=None*, *category=None*, *prefix='B'*, ***kwds*)

Bases: `sage.combinat.free_module.CombinatorialFreeModule`, `sage.misc.bindable_class.BindableClass`

Abstract base class for a basis of $NCSym$ or its dual.

class sage.combinat.ncsym.bases.**NCSymDualBases**(*parent_with_realization*)

Bases: `sage.categories.realizations.Category_realization_of_parent`

Category of bases of dual symmetric functions in non-commuting variables.

EXAMPLES:

```
sage: from sage.combinat.ncsym.bases import NCSymDualBases
sage: DNCSym = SymmetricFunctionsNonCommutingVariables(QQ).dual()
sage: NCSymDualBases(DNCSym)
Category of bases of dual symmetric functions in non-commuting variables over the Rational Field
```

super_categories()

Return the super categories of bases of the Hopf dual of the symmetric functions in non-commuting variables.

OUTPUT:

- a list of categories

TESTS:

```
sage: from sage.combinat.ncsym.bases import NCSymBases
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: NCSymBases(NCSym).super_categories()
[Category of bases of NCSym or NCSym^* over the Rational Field]
```

class sage.combinat.ncsym.bases.**NCSymOrNCSymDualBases**(*parent_with_realization*)

Bases: sage.categories.realizations.Category_realization_of_parent

Base category for the category of bases of symmetric functions in non-commuting variables or its Hopf dual for the common code.

class ElementMethods

duality_pairing(*other*)

Compute the pairing between self and an element other of the dual.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: m = NCSym.m()
sage: w = m.dual_basis()
sage: elt = m[[1,3],[2]] - 3*m[[1,2],[3]]
sage: elt.duality_pairing(w[[1,3],[2]])
1
sage: elt.duality_pairing(w[[1,2],[3]])
-3
sage: elt.duality_pairing(w[[1,2]])
0
sage: e = NCSym.e()
sage: w[[1,3],[2]].duality_pairing(e[[1,3],[2]])
0
```

class NCSymOrNCSymDualBases.**ParentMethods**

counit_on_basis(*A*)

The counit is defined by sending all elements of positive degree to zero.

INPUT:

- A – a set partition

OUTPUT:

- either the 0 or the 1 of the base ring of self

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: m.counit_on_basis(SetPartition([[1,3], [2]]))
0
sage: m.counit_on_basis(SetPartition([]))
1
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w.counit_on_basis(SetPartition([[1,3], [2]]))
0
sage: w.counit_on_basis(SetPartition([]))
1
```

duality_pairing(*x*, *y*)

Compute the pairing between an element of self and an element of the dual.

Carry out this computation by converting x to the $\mathbf{m}$ basis and y to the $\mathbf{w}$ basis.

INPUT:

- x – an element of symmetric functions in non-commuting variables
- y – an element of the dual of symmetric functions in non-commuting variables

OUTPUT:

- an element of the base ring of self

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: h = NCSym.h()
sage: w = NCSym.m().dual_basis()
sage: matrix([[h(A).duality_pairing(w(B)) for A in SetPartitions(3)] for B in SetPartiti
[6 2 2 2 1]
[2 2 1 1 1]
[2 1 2 1 1]
[2 1 1 2 1]
[1 1 1 1 1]
sage: (h[[1,2],[3]] + 3*h[[1,3],[2]]).duality_pairing(2*w[[1,3],[2]] + w[[1,2,3]] + 2*w[
32
```

duality_pairing_matrix(*basis*, *degree*)

The matrix of scalar products between elements of $NCSym$ and elements of $NCSym^*$.

INPUT:

- basis – a basis of the dual Hopf algebra
- degree – a non-negative integer

OUTPUT:

- the matrix of scalar products between the basis self and the basis basis in the dual Hopf algebra of degree degree

EXAMPLES:

The matrix between the $\mathbf{m}$ basis and the $\mathbf{w}$ basis:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: m = NCSym.m()
sage: w = NCSym.dual().w()
sage: m.duality_pairing_matrix(w, 3)
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

Similarly for some of the other basis of $NCSym$ and the $\mathbf{w}$ basis:

```
sage: e = NCSym.e()
sage: e.duality_pairing_matrix(w, 3)
[0 0 0 0 1]
[0 0 1 1 1]
[0 1 0 1 1]
[0 1 1 0 1]
[1 1 1 1 1]
sage: p = NCSym.p()
sage: p.duality_pairing_matrix(w, 3)
[1 0 0 0 0]
[1 1 0 0 0]
[1 0 1 0 0]
[1 0 0 1 0]
[1 1 1 1 1]
sage: cp = NCSym.cp()
sage: cp.duality_pairing_matrix(w, 3)
[1 0 0 0 0]
[1 1 0 0 0]
```

```
[0 0 1 0 0]
[1 0 0 1 0]
[1 1 1 1 1]
sage: x = NCSym.x()
sage: w.duality_pairing_matrix(x, 3)
[ 0  0  0  0  1]
[ 1  0 -1 -1  1]
[ 1 -1  0 -1  1]
[ 1 -1 -1  0  1]
[ 2 -1 -1 -1  1]
```

A base case test:

```
sage: m.duality_pairing_matrix(w, 0)
[1]
```

one_basis()

Return the index of the basis element containing 1.

OUTPUT:

- The empty set partition

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: m.one_basis()
{}
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w.one_basis()
{}
```

NCSymOrNCSymDualBases.**super_categories**()

Return the super categories of bases of (the Hopf dual of) the symmetric functions in non-commuting variables.

OUTPUT:

- a list of categories

TESTS:

```
sage: from sage.combinat.ncsym.bases import NCSymOrNCSymDualBases
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: NCSymOrNCSymDualBases(NCSym).super_categories()
[Category of realizations of Symmetric functions in
  non-commuting variables over the Rational Field,
 Category of graded hopf algebras with basis over Rational Field,
 Join of Category of realizations of hopf algebras over Rational Field
  and Category of graded algebras over Rational Field]
```

5.1.135 Dual Symmetric Functions in Non-Commuting Variables

AUTHORS:

- Travis Scrimshaw (08-04-2013): Initial version

class sage.combinat.ncsym.dual.**SymmetricFunctionsNonCommutingVariablesDual**(*R*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The Hopf dual to the symmetric functions in non-commuting variables.

See Section 2.3 of [BZ05] for a study.

a_realization()
Return the realization of the **w** basis of `self`.

EXAMPLES:

```
sage: SymmetricFunctionsNonCommutingVariables(QQ).dual().a_realization()
Dual symmetric functions in non-commuting variables over the Rational Field in the w basis
```

dual()
Return the dual Hopf algebra of the dual symmetric functions in non-commuting variables.

EXAMPLES:

```
sage: NCSymD = SymmetricFunctionsNonCommutingVariables(QQ).dual()
sage: NCSymD.dual()
Symmetric functions in non-commuting variables over the Rational Field
```

class w(*NCSymD*)
Bases: sage.combinat.ncsym.bases.NCSymBasis_abstract

The Hopf algebra of symmetric functions in non-commuting variables in the **w** basis.

EXAMPLES:

```
sage: NCSymD = SymmetricFunctionsNonCommutingVariables(QQ).dual()
sage: w = NCSymD.w()
```

We have the embedding χ^* of Sym into $NCSym^*$ available as a coercion:

```
sage: h = SymmetricFunctions(QQ).h()
sage: w(h[2,1])
w{{1}, {2, 3}} + w{{1, 2}, {3}} + w{{1, 3}, {2}}
```

Similarly we can pull back when we are in the image of χ^*:

```
sage: elt = 3*(w[[1],[2,3]] + w[[1,2],[3]] + w[[1,3],[2]])
sage: h(elt)
3*h[2, 1]
```

class Element(*M*, *x*)
Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

An element in the **w** basis.

expand(*n*, *letter='x'*)
Expand `self` written in the **w** basis in n^2 commuting variables which satisfy the relation $x_{ij}x_{ik} = 0$ for all i, j, and k.

The expansion of an element of the **w** basis is given by equations (26) and (55) in [HNT06].

INPUT:
- `n` – an integer
- `letter` – (default: `'x'`) a string

OUTPUT:
- The symmetric function of `self` expressed in the `n*n` non-commuting variables described by `letter`.

REFERENCES:

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w[[1,3],[2]].expand(4)
x02*x11*x20 + x03*x11*x30 + x03*x22*x30 + x13*x22*x31
```

One can use a different set of variable by using the optional argument `letter`:

```
sage: w[[1,3],[2]].expand(3, letter='y')
y02*y11*y20
```

is_symmetric()

Determine if a $NCSym^*$ function, expressed in the $\mathbf{w}$ basis, is symmetric.

A function f in the $\mathbf{w}$ basis is a symmetric function if it is in the image of χ^*. That is to say we have

$$f = \sum_{\lambda} c_\lambda \prod_i m_i(\lambda)! \sum_{\lambda(A)=\lambda} \mathbf{w}_A$$

where the second sum is over all set partitions A whose shape $\lambda(A)$ is equal to λ and $m_i(\mu)$ is the multiplicity of i in the partition μ.

OUTPUT:

- `True` if $\lambda(A) = \lambda(B)$ implies the coefficients of $\mathbf{w}_A$ and $\mathbf{w}_B$ are equal, `False` otherwise

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: elt = w.sum_of_partitions([2,1,1])
sage: elt.is_symmetric()
True
sage: elt -= 3*w.sum_of_partitions([1,1])
sage: elt.is_symmetric()
True
sage: w = SymmetricFunctionsNonCommutingVariables(ZZ).dual().w()
sage: elt = w.sum_of_partitions([2,1,1]) / 2
sage: elt.is_symmetric()
False
sage: elt = w[[1,3],[2]]
sage: elt.is_symmetric()
False
sage: elt = w[[1],[2,3]] + w[[1,2],[3]] + 2*w[[1,3],[2]]
sage: elt.is_symmetric()
False
```

to_symmetric_function()

Take a function in the $\mathbf{w}$ basis, and return its symmetric realization, when possible, expressed in the homogeneous basis of symmetric functions.

OUTPUT:

- If `self` is a symmetric function, then the expansion in the homogeneous basis of the symmetric functions is returned. Otherwise an error is raised.

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: elt = w[[1],[2,3]] + w[[1,2],[3]] + w[[1,3],[2]]
sage: elt.to_symmetric_function()
h[2, 1]
sage: elt = w.sum_of_partitions([2,1,1]) / 2
sage: elt.to_symmetric_function()
1/2*h[2, 1, 1]
```

TESTS:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w(0).to_symmetric_function()
0
sage: w([]).to_symmetric_function()
h[]
sage: (2*w([])).to_symmetric_function()
2*h[]
```

SymmetricFunctionsNonCommutingVariablesDual.w.**antipode_on_basis**(*A*)

Return the antipode applied to the basis element indexed by A.

INPUT:

- A – a set partition

OUTPUT:

- an element in the basis self

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w.antipode_on_basis(SetPartition([[1],[2,3]]))
-3*w{{1}, {2}, {3}} + w{{1, 2}, {3}} + w{{1, 3}, {2}}
sage: F = w[[1,3],[5],[2,4]].coproduct()
sage: F.apply_multilinear_morphism(lambda x,y: x.antipode()*y)
0
```

SymmetricFunctionsNonCommutingVariablesDual.w.**coproduct_on_basis**(*A*)

Return the coproduct of a $\mathbf{w}$ basis element.

The coproduct on the basis element $\mathbf{w}_A$ is the sum over tensor product terms $\mathbf{w}_B \otimes \mathbf{w}_C$ where B is the restriction of A to $\{1, 2, \ldots, k\}$ and C is the restriction of A to $\{k+1, k+2, \ldots, n\}$.

INPUT:

- A – a set partition

OUTPUT:

- The coproduct applied to the $\mathbf{w}$ dual symmetric function in non-commuting variables indexed by A expressed in the $\mathbf{w}$ basis.

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w[[1], [2,3]].coproduct()
w{} # w{{1}, {2, 3}} + w{{1}} # w{{1, 2}}
 + w{{1}, {2}} # w{{1}} + w{{1}, {2, 3}} # w{}
sage: w.coproduct_on_basis(SetPartition([]))
w{} # w{}
```

SymmetricFunctionsNonCommutingVariablesDual.w.**dual_basis**()

Return the dual basis to the $\mathbf{w}$ basis.

The dual basis to the $\mathbf{w}$ basis is the monomial basis of the symmetric functions in non-commuting variables.

OUTPUT:

- the monomial basis of the symmetric functions in non-commuting variables

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w.dual_basis()
Symmetric functions in non-commuting variables over the Rational Field in the monomial ba
```

SymmetricFunctionsNonCommutingVariablesDual.w.**duality_pairing**(*x*, *y*)

Compute the pairing between an element of self and an element of the dual.

INPUT:

- x – an element of the dual of symmetric functions in non-commuting variables
- y – an element of the symmetric functions in non-commuting variables

OUTPUT:

- an element of the base ring of self

EXAMPLES:

```
sage: DNCSym = SymmetricFunctionsNonCommutingVariablesDual(QQ)
sage: w = DNCSym.w()
sage: m = w.dual_basis()
sage: matrix([[w(A).duality_pairing(m(B)) for A in SetPartitions(3)] for B in SetPartiti
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
sage: (w[[1,2],[3]] + 3*w[[1,3],[2]]).duality_pairing(2*m[[1,3],[2]] + m[[1,2,3]] + 2*m[
8
sage: h = SymmetricFunctionsNonCommutingVariables(QQ).h()
sage: matrix([[w(A).duality_pairing(h(B)) for A in SetPartitions(3)] for B in SetPartiti
[6 2 2 2 1]
[2 2 1 1 1]
[2 1 2 1 1]
[2 1 1 2 1]
[1 1 1 1 1]
sage: (2*w[[1,3],[2]] + w[[1,2,3]] + 2*w[[1,2],[3]]).duality_pairing(h[[1,2],[3]] + 3*h[
32
```

SymmetricFunctionsNonCommutingVariablesDual.w.**product_on_basis**(A, B)

The product on $\mathbf{w}$ basis elements.

The product on the $\mathbf{w}$ is the dual to the coproduct on the $\mathbf{m}$ basis. On the basis $\mathbf{w}$ it is defined as

$$\mathbf{w}_A \mathbf{w}_B = \sum_{S \subseteq [n]} \mathbf{w}_{A\uparrow_S \cup B\uparrow_{S^c}}$$

where the sum is over all possible subsets S of $[n]$ such that $|S| = |A|$ with a term indexed the union of $A\uparrow_S$ and $B\uparrow_{S^c}$. The notation $A\uparrow_S$ represents the unique set partition of the set S such that the standardization is A. This product is commutative.

INPUT:

- A, B – set partitions

OUTPUT:

- an element of the $\mathbf{w}$ basis

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: A = SetPartition([[1], [2,3]])
sage: B = SetPartition([[1, 2, 3]])
sage: w.product_on_basis(A, B)
w{{1}, {2, 3}, {4, 5, 6}} + w{{1}, {2, 3, 4}, {5, 6}}
 + w{{1}, {2, 3, 5}, {4, 6}} + w{{1}, {2, 3, 6}, {4, 5}}
 + w{{1}, {2, 4}, {3, 5, 6}} + w{{1}, {2, 4, 5}, {3, 6}}
 + w{{1}, {2, 4, 6}, {3, 5}} + w{{1}, {2, 5}, {3, 4, 6}}
 + w{{1}, {2, 5, 6}, {3, 4}} + w{{1}, {2, 6}, {3, 4, 5}}
 + w{{1, 2, 3}, {4}, {5, 6}} + w{{1, 2, 4}, {3}, {5, 6}}
 + w{{1, 2, 5}, {3}, {4, 6}} + w{{1, 2, 6}, {3}, {4, 5}}
 + w{{1, 3, 4}, {2}, {5, 6}} + w{{1, 3, 5}, {2}, {4, 6}}
 + w{{1, 3, 6}, {2}, {4, 5}} + w{{1, 4, 5}, {2}, {3, 6}}
 + w{{1, 4, 6}, {2}, {3, 5}} + w{{1, 5, 6}, {2}, {3, 4}}
sage: B = SetPartition([[1], [2]])
```

```
sage: w.product_on_basis(A, B)
3*w{{1}, {2}, {3}, {4, 5}} + 2*w{{1}, {2}, {3, 4}, {5}}
 + 2*w{{1}, {2}, {3, 5}, {4}} + w{{1}, {2, 3}, {4}, {5}}
 + w{{1}, {2, 4}, {3}, {5}} + w{{1}, {2, 5}, {3}, {4}}
sage: w.product_on_basis(A, SetPartition([]))
w{{1}, {2, 3}}
```

SymmetricFunctionsNonCommutingVariablesDual.w.**sum_of_partitions**(*la*)

Return the sum over all sets partitions whose shape is `la`, scaled by $\prod_i m_i!$ where m_i is the multiplicity of i in `la`.

INPUT:

- `la` – an integer partition

OUTPUT:

- an element of `self`

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w.sum_of_partitions([2,1,1])
2*w{{1}, {2}, {3, 4}} + 2*w{{1}, {2, 3}, {4}} + 2*w{{1}, {2, 4}, {3}}
 + 2*w{{1, 2}, {3}, {4}} + 2*w{{1, 3}, {2}, {4}} + 2*w{{1, 4}, {2}, {3}}
```

SymmetricFunctionsNonCommutingVariablesDual.w.**to_symmetric_function**()

The preimage of χ^* in the **w** basis.

EXAMPLES:

```
sage: w = SymmetricFunctionsNonCommutingVariables(QQ).dual().w()
sage: w.to_symmetric_function
Generic morphism:
  From: Dual symmetric functions in non-commuting variables over the Rational Field in t
  To:   Symmetric Functions over Rational Field in the homogeneous basis
```

5.1.136 Symmetric Functions in Non-Commuting Variables

AUTHORS:

- Travis Scrimshaw (08-04-2013): Initial version

class sage.combinat.ncsym.ncsym.**SymmetricFunctionsNonCommutingVariables**(*R*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Symmetric functions in non-commutative variables.

The ring of symmetric functions in non-commutative variables, which is not to be confused with the non-commutative symmetric functions, is the ring of all bounded-degree noncommutative power series in countably many indeterminates (i.e., elements in $R\langle\langle x_1, x_2, x_3, \ldots\rangle\rangle$ of bounded degree) which are invariant with respect to the action of the symmetric group S_∞ on the indices of the indeterminates. It can be regarded as a direct limit over all $n \to \infty$ of rings of S_n-invariant polynomials in n non-commuting variables (that is, S_n-invariant elements of $R\langle x_1, x_2, \ldots, x_n\rangle$).

This ring is implemented as a Hopf algebra whose basis elements are indexed by set paritions.

Let $A = \{A_1, A_2, \ldots, A_r\}$ be a set partition of the integers $\{1, 2, \ldots, k\}$. A monomial basis element indexed by A represents the sum of monomials $x_{i_1} x_{i_2} \cdots x_{i_k}$ where $i_c = i_d$ if and only if c and d are in the same part A_i for some i.

The k-th graded component of the ring of symmetric functions in non-commutative variables has its dimension equal to the number of set partitions of k. (If we work, instead, with finitely many – say, n – variables, then its

dimension is equal to the number of set partitions of k where the number of parts is at most n.)

Note: All set partitions are considered standard, a set partition of $[n]$ for some n, unless otherwise stated.

REFERENCES:

EXAMPLES:

We begin by first creating the ring of $NCSym$ and the bases that are analogues of the usual symmetric functions:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: m = NCSym.m()
sage: e = NCSym.e()
sage: h = NCSym.h()
sage: p = NCSym.p()
sage: m
Symmetric functions in non-commuting variables over the Rational Field in the monomial basis
```

The basis is indexed by set partitions, so we create a few elements and convert them between these bases:

```
sage: elt = m(SetPartition([[1,3],[2]])) - 2*m(SetPartition([[1],[2]])); elt
-2*m{{1}, {2}} + m{{1, 3}, {2}}
sage: e(elt)
1/2*e{{1}, {2, 3}} - 2*e{{1, 2}} + 1/2*e{{1, 2}, {3}} - 1/2*e{{1, 2, 3}} - 1/2*e{{1, 3}, {2}}
sage: h(elt)
-4*h{{1}, {2}} - 2*h{{1}, {2}, {3}} + 1/2*h{{1}, {2, 3}} + 2*h{{1, 2}}
 + 1/2*h{{1, 2}, {3}} - 1/2*h{{1, 2, 3}} + 3/2*h{{1, 3}, {2}}
sage: p(elt)
-2*p{{1}, {2}} + 2*p{{1, 2}} - p{{1, 2, 3}} + p{{1, 3}, {2}}
sage: m(p(elt))
-2*m{{1}, {2}} + m{{1, 3}, {2}}

sage: elt = p(SetPartition([[1,3],[2]])) - 4*p(SetPartition([[1],[2]])) + 2; elt
2*p{} - 4*p{{1}, {2}} + p{{1, 3}, {2}}
sage: e(elt)
2*e{} - 4*e{{1}, {2}} + e{{1}, {2}, {3}} - e{{1, 3}, {2}}
sage: m(elt)
2*m{} - 4*m{{1}, {2}} - 4*m{{1, 2}} + m{{1, 2, 3}} + m{{1, 3}, {2}}
sage: h(elt)
2*h{} - 4*h{{1}, {2}} - h{{1}, {2}, {3}} + h{{1, 3}, {2}}
sage: p(m(elt))
2*p{} - 4*p{{1}, {2}} + p{{1, 3}, {2}}
```

There is also a shorthand for creating elements. We note that we must use `p[[]]` to create the empty set partition due to python's syntax.

```
sage: eltm = m[[1,3],[2]] - 3*m[[1],[2]]; eltm
-3*m{{1}, {2}} + m{{1, 3}, {2}}
sage: elte = e[[1,3],[2]]; elte
e{{1, 3}, {2}}
sage: elth = h[[1,3],[2,4]]; elth
h{{1, 3}, {2, 4}}
sage: eltp = p[[1,3],[2,4]] + 2*p[[1]] - 4*p[[]]; eltp
-4*p{} + 2*p{{1}} + p{{1, 3}, {2, 4}}
```

There is also a natural projection to the usual symmetric functions by letting the variables commute. This projection map preserves the product and coproduct structure. We check that Theorem 2.1 of [RS06] holds:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Sm = Sym.m()
```

```
sage: Se = Sym.e()
sage: Sh = Sym.h()
sage: Sp = Sym.p()
sage: eltm.to_symmetric_function()
-6*m[1, 1] + m[2, 1]
sage: Sm(p(eltm).to_symmetric_function())
-6*m[1, 1] + m[2, 1]
sage: elte.to_symmetric_function()
2*e[2, 1]
sage: Se(h(elte).to_symmetric_function())
2*e[2, 1]
sage: elth.to_symmetric_function()
4*h[2, 2]
sage: Sh(m(elth).to_symmetric_function())
4*h[2, 2]
sage: eltp.to_symmetric_function()
-4*p[] + 2*p[1] + p[2, 2]
sage: Sp(e(eltp).to_symmetric_function())
-4*p[] + 2*p[1] + p[2, 2]
```

a_realization()

Return the realization of the powersum basis of `self`.

OUTPUT:

- The powersum basis of symmetric functions in non-commuting variables.

EXAMPLES:

```
sage: SymmetricFunctionsNonCommutingVariables(QQ).a_realization()
Symmetric functions in non-commuting variables over the Rational Field in the powersum basis
```

class coarse_powersum(*NCSym*)

Bases: sage.combinat.ncsym.bases.NCSymBasis_abstract

The Hopf algebra of symmetric functions in non-commuting variables in the $\mathbf{cp}$ basis.

This basis was defined in [BZ05] as

$$\mathbf{cp}_A = \sum_{A \leq_* B} \mathbf{m}_B,$$

where we sum over all strict coarsenings of the set partition A. An alternative description of this basis was given in [BT13] as

$$\mathbf{cp}_A = \sum_{A \subseteq B} \mathbf{m}_B,$$

where we sum over all set partitions whose arcs are a subset of the arcs of the set partition A.

Note: In [BZ05], this basis was denoted by $\mathbf{q}$. In [BT13], this basis was called the powersum basis and denoted by p. However it is a coarser basis than the usual powersum basis in the sense that it does not yield the usual powersum basis of the symmetric function under the natural map of letting the variables commute.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: cp = NCSym.cp()
sage: cp[[1,3],[2,4]]*cp[[1,2,3]]
```

```
cp{{1, 3}, {2, 4}, {5, 6, 7}}
sage: cp[[1,2],[3]].internal_coproduct()
cp{{1, 2}, {3}} # cp{{1, 2}, {3}}
sage: ps = SymmetricFunctions(NCSym.base_ring()).p()
sage: ps(cp[[1,3],[2]].to_symmetric_function())
p[2, 1] - p[3]
sage: ps(cp[[1,2],[3]].to_symmetric_function())
p[2, 1]
```

class SymmetricFunctionsNonCommutingVariables.**deformed_coarse_powersum**(*NCSym*, *q=2*)

Bases: sage.combinat.ncsym.bases.NCSymBasis_abstract

The Hopf algebra of symmetric functions in non-commuting variables in the ρ basis.

This basis was defined in [BT13] as a q-deformation of the $\mathbf{cp}$ basis:

$$\rho_A = \sum_{A \subseteq B} \frac{1}{q^{\mathrm{nst}_{B-A}^{A}}} \mathbf{m}_B,$$

where we sum over all set partitions whose arcs are a subset of the arcs of the set partition A.

INPUT:

- q – (default: 2) the parameter q

EXAMPLES:

```
sage: R = QQ['q'].fraction_field()
sage: q = R.gen()
sage: NCSym = SymmetricFunctionsNonCommutingVariables(R)
sage: rho = NCSym.rho(q)
```

We construct Example 3.1 in [BT13]:

```
sage: rnode = lambda A: sorted([a[1] for a in A.arcs()], reverse=True)
sage: dimv = lambda A: sorted([a[1]-a[0] for a in A.arcs()], reverse=True)
sage: lst = list(SetPartitions(4))
sage: S = sorted(lst, key=lambda A: (dimv(A), rnode(A)))
sage: m = NCSym.m()
sage: matrix([[m(rho[A])[B] for B in S] for A in S])
[  1   1   1   1   1   1   1   1   1   1   1   1   1   1   1]
[  0   1   0   0   1   1   0   1   0   0   1   0   0   0   0]
[  0   0   1   0   1   0   1   1   0   0   0   0   0   0   1]
[  0   0   0   1   0   1   1   1   0   0   0   1   0   0   0]
[  0   0   0   0   1   0   0   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   1   0   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   1   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   1   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0   0   1   0   0   1   1   0   0]
[  0   0   0   0   0   0   0   0   0   1   1   0   1   0   0]
[  0   0   0   0   0   0   0   0   0   0   1   0   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   1   0   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   1 1/q]
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   1]
```

q()

Return the deformation parameter q of self.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: rho = NCSym.rho(5)
sage: rho.q()
5

sage: R = QQ['q'].fraction_field()
sage: q = R.gen()
sage: NCSym = SymmetricFunctionsNonCommutingVariables(R)
sage: rho = NCSym.rho(q)
sage: rho.q() == q
True
```

SymmetricFunctionsNonCommutingVariables.**dual**()

Return the dual Hopf algebra of the symmetric functions in non-commuting variables.

EXAMPLES:

```
sage: SymmetricFunctionsNonCommutingVariables(QQ).dual()
Dual symmetric functions in non-commuting variables over the Rational Field
```

class SymmetricFunctionsNonCommutingVariables.**elementary**(*NCSym*)

Bases: sage.combinat.ncsym.bases.NCSymBasis_abstract

The Hopf algebra of symmetric functions in non-commuting variables in the elementary basis.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: e = NCSym.e()
```

class Element(*M*, *x*)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

An element in the elementary basis of $NCSym$.

omega()

Return the involution ω applied to self.

The involution ω on $NCSym$ is defined by $\omega(\mathbf{e}_A) = \mathbf{h}_A$.

OUTPUT:

- an element in the basis self

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: e = NCSym.e()
sage: h = NCSym.h()
sage: elt = e[[1,3],[2]].omega(); elt
2*e{{1}, {2}, {3}} - e{{1, 3}, {2}}
sage: elt.omega()
e{{1, 3}, {2}}
sage: h(elt)
h{{1, 3}, {2}}
```

to_symmetric_function()

The projection of self to the symmetric functions.

Take a symmetric function in non-commuting variables expressed in the $\mathbf{e}$ basis, and return the projection of expressed in the elementary basis of symmetric functions.

The map $\chi\colon NCSym \to Sym$ is given by

$$\mathbf{e}_A \mapsto e_{\lambda(A)} \prod_i \lambda(A)_i!$$

where $\lambda(A)$ is the partition associated with A by taking the sizes of the parts.

OUTPUT:

•An element of the symmetric functions in the elementary basis

EXAMPLES:

```
sage: e = SymmetricFunctionsNonCommutingVariables(QQ).e()
sage: e[[1,3],[2]].to_symmetric_function()
2*e[2, 1]
sage: e[[1],[3],[2]].to_symmetric_function()
e[1, 1, 1]
```

class SymmetricFunctionsNonCommutingVariables.**homogeneous**(*NCSym*)

Bases: sage.combinat.ncsym.bases.NCSymBasis_abstract

The Hopf algebra of symmetric functions in non-commuting variables in the homogeneous basis.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: h = NCSym.h()
sage: h[[1,3],[2,4]]*h[[1,2,3]]
h{{1, 3}, {2, 4}, {5, 6, 7}}
sage: h[[1,2]].coproduct()
h{} # h{{1, 2}} + 2*h{{1}} # h{{1}} + h{{1, 2}} # h{}
```

class **Element**(*M*, *x*)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

An element in the homogeneous basis of $NCSym$.

omega()

Return the involution ω applied to `self`.

The involution ω on $NCSym$ is defined by $\omega(\mathbf{h}_A) = \mathbf{e}_A$.

OUTPUT:

•an element in the basis `self`

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: h = NCSym.h()
sage: e = NCSym.e()
sage: elt = h[[1,3],[2]].omega(); elt
2*h{{1}, {2}, {3}} - h{{1, 3}, {2}}
sage: elt.omega()
h{{1, 3}, {2}}
sage: e(elt)
e{{1, 3}, {2}}
```

to_symmetric_function()

The projection of `self` to the symmetric functions.

Take a symmetric function in non-commuting variables expressed in the $\mathbf{h}$ basis, and return the projection of expressed in the complete basis of symmetric functions.

The map $\chi\colon NCSym \to Sym$ is given by

$$\mathbf{h}_A \mapsto h_{\lambda(A)} \prod_i \lambda(A)_i!$$

where $\lambda(A)$ is the partition associated with A by taking the sizes of the parts.

OUTPUT:

•An element of the symmetric functions in the complete basis

EXAMPLES:

```
sage: h = SymmetricFunctionsNonCommutingVariables(QQ).h()
sage: h[[1,3],[2]].to_symmetric_function()
2*h[2, 1]
sage: h[[1],[3],[2]].to_symmetric_function()
h[1, 1, 1]
```

class SymmetricFunctionsNonCommutingVariables.**monomial**(*NCSym*)

Bases: sage.combinat.ncsym.bases.NCSymBasis_abstract

The Hopf algebra of symmetric functions in non-commuting variables in the monomial basis.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: m = NCSym.m()
sage: m[[1,3],[2]]*m[[1,2]]
m{{1, 3}, {2}, {4, 5}} + m{{1, 3}, {2, 4, 5}} + m{{1, 3, 4, 5}, {2}}
sage: m[[1,3],[2]].coproduct()
m{} # m{{1, 3}, {2}} + m{{1}} # m{{1, 2}} + m{{1, 2}} # m{{1}} + m{{1,
 3}, {2}} # m{}
```

class Element(*M*, *x*)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

An element in the monomial basis of $NCSym$.

expand(*n*, *alphabet='x'*)

Expand self written in the monomial basis in n non-commuting variables.

INPUT:

•n – an integer

•alphabet – (default: 'x') a string

OUTPUT:

•The symmetric function of self expressed in the n non-commuting variables described by alphabet.

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
sage: m[[1,3],[2]].expand(4)
x0*x1*x0 + x0*x2*x0 + x0*x3*x0 + x1*x0*x1 + x1*x2*x1 + x1*x3*x1
 + x2*x0*x2 + x2*x1*x2 + x2*x3*x2 + x3*x0*x3 + x3*x1*x3 + x3*x2*x3
```

One can use a different set of variables by using the optional argument alphabet:

```
sage: m[[1],[2,3]].expand(3,alphabet='y')
y0*y1^2 + y0*y2^2 + y1*y0^2 + y1*y2^2 + y2*y0^2 + y2*y1^2
```

to_symmetric_function()

The projection of self to the symmetric functions.

Take a symmetric function in non-commuting variables expressed in the $\mathbf{m}$ basis, and return the projection of expressed in the monomial basis of symmetric functions.

The map $\chi\colon NCSym \to Sym$ is defined by

$$\mathbf{m}_A \mapsto m_{\lambda(A)} \prod_i n_i(\lambda(A))!$$

where $\lambda(A)$ is the partition associated with A by taking the sizes of the parts and $n_i(\mu)$ is the multiplicity of i in μ.

OUTPUT:

•an element of the symmetric functions in the monomial basis

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
sage: m[[1,3],[2]].to_symmetric_function()
m[2, 1]
sage: m[[1],[3],[2]].to_symmetric_function()
6*m[1, 1, 1]
```

SymmetricFunctionsNonCommutingVariables.monomial.**coproduct_on_basis**(*A*)

Return the coproduct of a monomial basis element.

INPUT:

•A – a set partition

OUTPUT:

•The coproduct applied to the monomial symmetric function in non-commuting variables indexed by A expressed in the monomial basis.

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
sage: m[[1, 3], [2]].coproduct()
m{} # m{{1, 3}, {2}} + m{{1}} # m{{1, 2}} + m{{1, 2}} # m{{1}} + m{{1, 3}, {2}} # m{}
sage: m.coproduct_on_basis(SetPartition([]))
m{} # m{}
sage: m.coproduct_on_basis(SetPartition([[1,2,3]]))
m{} # m{{1, 2, 3}} + m{{1, 2, 3}} # m{}
sage: m[[1,5],[2,4],[3,7],[6]].coproduct()
m{} # m{{1, 5}, {2, 4}, {3, 7}, {6}} + m{{1}} # m{{1, 5}, {2, 4}, {3, 6}}
 + 2*m{{1, 2}} # m{{1, 3}, {2, 5}, {4}} + m{{1, 2}} # m{{1, 4}, {2, 3}, {5}}
 + 2*m{{1, 2}, {3}} # m{{1, 3}, {2, 4}} + m{{1, 3}, {2}} # m{{1, 4}, {2, 3}}
 + 2*m{{1, 3}, {2, 4}} # m{{1, 2}, {3}} + 2*m{{1, 3}, {2, 5}, {4}} # m{{1, 2}}
 + m{{1, 4}, {2, 3}} # m{{1, 3}, {2}} + m{{1, 4}, {2, 3}, {5}} # m{{1, 2}}
 + m{{1, 5}, {2, 4}, {3, 6}} # m{{1}} + m{{1, 5}, {2, 4}, {3, 7}, {6}} # m{}
```

SymmetricFunctionsNonCommutingVariables.monomial.**dual_basis**()

Return the dual basis to the monomial basis.

OUTPUT:

•the w basis of the dual Hopf algebra

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: m.dual_basis()
Dual symmetric functions in non-commuting variables over the Rational Field in the w bas
```

SymmetricFunctionsNonCommutingVariables.monomial.**duality_pairing**(*x*, *y*)

Compute the pairing between an element of self and an element of the dual.

INPUT:

•x – an element of symmetric functions in non-commuting variables

•y – an element of the dual of symmetric functions in non-commuting variables

OUTPUT:

•an element of the base ring of self

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: m = NCSym.m()
```

```
sage: w = m.dual_basis()
sage: matrix([[m(A).duality_pairing(w(B)) for A in SetPartitions(3)] for B in SetPartiti
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
sage: (m[[1,2],[3]] + 3*m[[1,3],[2]]).duality_pairing(2*w[[1,3],[2]] + w[[1,2,3]] + 2*w[
8
```

SymmetricFunctionsNonCommutingVariables.monomial.**from_symmetric_function**(f)

Return the image of the symmetric function `f` in `self`.

This is performed by converting to the monomial basis and extending the method `sum_of_partitions()` linearly. This is a linear map from the symmetric functions to the symmetric functions in non-commuting variables that does not preserve the product or coproduct structure of the Hopf algebra.

See also:

`to_symmetric_function()`

INPUT:

- `f` – an element of the symmetric functions

OUTPUT:

- An element of the $\mathbf{m}$ basis

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: mon = SymmetricFunctions(QQ).m()
sage: elt = m.from_symmetric_function(mon[2,1,1]); elt
1/12*m{{1}, {2}, {3, 4}} + 1/12*m{{1}, {2, 3}, {4}} + 1/12*m{{1}, {2, 4}, {3}}
 + 1/12*m{{1, 2}, {3}, {4}} + 1/12*m{{1, 3}, {2}, {4}} + 1/12*m{{1, 4}, {2}, {3}}
sage: elt.to_symmetric_function()
m[2, 1, 1]
sage: e = SymmetricFunctionsNonCommutingVariables(QQ).e()
sage: elm = SymmetricFunctions(QQ).e()
sage: e(m.from_symmetric_function(elm[4]))
1/24*e{{1, 2, 3, 4}}
sage: h = SymmetricFunctionsNonCommutingVariables(QQ).h()
sage: hom = SymmetricFunctions(QQ).h()
sage: h(m.from_symmetric_function(hom[4]))
1/24*h{{1, 2, 3, 4}}
sage: p = SymmetricFunctionsNonCommutingVariables(QQ).p()
sage: pow = SymmetricFunctions(QQ).p()
sage: p(m.from_symmetric_function(pow[4]))
p{{1, 2, 3, 4}}
sage: p(m.from_symmetric_function(pow[2,1]))
1/3*p{{1}, {2, 3}} + 1/3*p{{1, 2}, {3}} + 1/3*p{{1, 3}, {2}}
sage: p([[1,2]])*p([[1]])
p{{1, 2}, {3}}
```

Check that $\chi \circ \widetilde{\chi}$ is the identity on Sym:

```
sage: all(m.from_symmetric_function(pow(la)).to_symmetric_function() == pow(la)
....:     for la in Partitions(4))
True
```

SymmetricFunctionsNonCommutingVariables.monomial.**internal_coproduct_on_basis**(A)

Return the internal coproduct of a monomial basis element.

The internal coproduct is defined by

$$\Delta^{\odot}(\mathbf{m}_A) = \sum_{B \wedge C = A} \mathbf{m}_B \otimes \mathbf{m}_C$$

where we sum over all pairs of set partitions B and C whose infimum is A.

INPUT:

•A – a set partition

OUTPUT:

•an element of the tensor square of the $\mathbf{m}$ basis

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
sage: m.internal_coproduct_on_basis(SetPartition([[1,3],[2]]))
m{{1, 2, 3}} # m{{1, 3}, {2}} + m{{1, 3}, {2}} # m{{1, 2, 3}} + m{{1, 3}, {2}} # m{{1, 3
```

SymmetricFunctionsNonCommutingVariables.monomial.**product_on_basis**(*A*, *B*)

The product on monomial basis elements.

The product of the basis elements indexed by two set partitions A and B is the sum of the basis elements indexed by set partitions C such that $C \wedge ([n]|[k]) = A|B$ where $n = |A|$ and $k = |B|$. Here $A \wedge B$ is the infimum of A and B and $A|B$ is the `SetPartition.pipe()` operation. Equivalently we can describe all C as matchings between the partitions of A and B where if $a \in A$ is matched with $b \in B$, we take $a \cup b$ instead of a and b in C.

INPUT:

•A, B – set partitions

OUTPUT:

•an element of the $\mathbf{m}$ basis

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
sage: A = SetPartition([[1], [2,3]])
sage: B = SetPartition([[1], [3], [2,4]])
sage: m.product_on_basis(A, B)
m{{1}, {2, 3}, {4}, {5, 7}, {6}} + m{{1}, {2, 3, 4}, {5, 7}, {6}}
 + m{{1}, {2, 3, 5, 7}, {4}, {6}} + m{{1}, {2, 3, 6}, {4}, {5, 7}}
 + m{{1, 4}, {2, 3}, {5, 7}, {6}} + m{{1, 4}, {2, 3, 5, 7}, {6}}
 + m{{1, 4}, {2, 3, 6}, {5, 7}} + m{{1, 5, 7}, {2, 3}, {4}, {6}}
 + m{{1, 5, 7}, {2, 3, 4}, {6}} + m{{1, 5, 7}, {2, 3, 6}, {4}}
 + m{{1, 6}, {2, 3}, {4}, {5, 7}} + m{{1, 6}, {2, 3, 4}, {5, 7}}
 + m{{1, 6}, {2, 3, 5, 7}, {4}}
sage: B = SetPartition([[1], [2]])
sage: m.product_on_basis(A, B)
m{{1}, {2, 3}, {4}, {5}} + m{{1}, {2, 3, 4}, {5}}
 + m{{1}, {2, 3, 5}, {4}} + m{{1, 4}, {2, 3}, {5}} + m{{1, 4}, {2, 3, 5}}
 + m{{1, 5}, {2, 3}, {4}} + m{{1, 5}, {2, 3, 4}}
sage: m.product_on_basis(A, SetPartition([]))
m{{1}, {2, 3}}
```

TESTS:

We check that we get all of the correct set partitions:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).monomial()
sage: A = SetPartition([[1], [2,3]])
sage: B = SetPartition([[1], [2]])
sage: S = SetPartition([[1,2,3], [4,5]])
sage: AB = SetPartition([[1], [2,3], [4], [5]])
sage: L = sorted(filter(lambda x: S.inf(x) == AB, SetPartitions(5)), key=str)
```

```
sage: map(list, L) == map(list, sorted(m.product_on_basis(A, B).support(), key=str))
True
```

SymmetricFunctionsNonCommutingVariables.monomial.**sum_of_partitions**(*la*)

Return the sum over all set partitions whose shape is `la` with a fixed coefficient C defined below.

Fix a partition λ, we define $\lambda! := \prod_i \lambda_i!$ and $\lambda^! := \prod_i m_i!$. Recall that $|\lambda| = \sum_i \lambda_i$ and m_i is the number of parts of length i of λ. Thus we defined the coefficient as

$$C := \frac{\lambda!\lambda^!}{|\lambda|!}.$$

Hence we can define a lift $\widetilde{\chi}$ from Sym to $NCSym$ by

$$m_\lambda \mapsto C \sum_A \mathbf{m}_A$$

where the sum is over all set partitions whose shape is λ.

INPUT:

- `la` – an integer partition

OUTPUT:

- an element of the $\mathbf{m}$ basis

EXAMPLES:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: m.sum_of_partitions(Partition([2,1,1]))
1/12*m{{1}, {2}, {3, 4}} + 1/12*m{{1}, {2, 3}, {4}} + 1/12*m{{1}, {2, 4}, {3}}
 + 1/12*m{{1, 2}, {3}, {4}} + 1/12*m{{1, 3}, {2}, {4}} + 1/12*m{{1, 4}, {2}, {3}}
```

TESTS:

Check that $\chi \circ \widetilde{\chi}$ is the identity on Sym:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: mon = SymmetricFunctions(QQ).monomial()
sage: all(m.from_symmetric_function(mon[la]).to_symmetric_function() == mon[la]
....:     for i in range(6) for la in Partitions(i))
True
```

class SymmetricFunctionsNonCommutingVariables.**powersum**(*NCSym*)

Bases: sage.combinat.ncsym.bases.NCSymBasis_abstract

The Hopf algebra of symmetric functions in non-commuting variables in the powersum basis.

The powersum basis is given by

$$\mathbf{p}_A = \sum_{A \leq B} \mathbf{m}_B,$$

where we sum over all coarsenings of the set partition A. If we allow our variables to commute, then $\mathbf{p}_A$ goes to the usual powersum symmetric function p_λ whose (integer) partition λ is the shape of A.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: p = NCSym.p()

sage: x = p.an_element()**2; x
4*p{} + 8*p{{1}} + 4*p{{1}, {2}} + 6*p{{1}, {2, 3}}
 + 12*p{{1, 2}} + 6*p{{1, 2}, {3}} + 9*p{{1, 2}, {3, 4}}
sage: x.to_symmetric_function()
4*p[] + 8*p[1] + 4*p[1, 1] + 12*p[2] + 12*p[2, 1] + 9*p[2, 2]
```

class Element(M, x)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

An element in the powersum basis of $NCSym$.

to_symmetric_function()

The projection of self to the symmetric functions.

Take a symmetric function in non-commuting variables expressed in the $\mathbf{p}$ basis, and return the projection of expressed in the powersum basis of symmetric functions.

The map $\chi\colon NCSym \to Sym$ is given by

$$\mathbf{p}_A \mapsto p_{\lambda(A)}$$

where $\lambda(A)$ is the partition associated with A by taking the sizes of the parts.

OUTPUT:

•an element of symmetric functions in the power sum basis

EXAMPLES:

```
sage: p = SymmetricFunctionsNonCommutingVariables(QQ).p()
sage: p[[1,3],[2]].to_symmetric_function()
p[2, 1]
sage: p[[1],[3],[2]].to_symmetric_function()
p[1, 1, 1]
```

SymmetricFunctionsNonCommutingVariables.powersum.**antipode_on_basis**(A)

Return the result of the antipode applied to a powersum basis element.

Let A be a set partition. The antipode given in [LM2011] is

$$S(\mathbf{p}_A) = \sum_{\gamma} (-1)^{\ell(\gamma)} \mathbf{p}_{\gamma[A]}$$

where we sum over all ordered set partitions (i.e. set compositions) of $[\ell(A)]$ and

$$\gamma[A] = A^{\downarrow}_{\gamma_1} | \cdots | A^{\downarrow}_{\gamma_{\ell(A)}}$$

is the action of γ on A defined in SetPartition.ordered_set_partition_action().

INPUT:

•A – a set partition

OUTPUT:

•an element in the basis self

EXAMPLES:

```
sage: p = SymmetricFunctionsNonCommutingVariables(QQ).powersum()
sage: p.antipode_on_basis(SetPartition([[1], [2,3]]))
p{{1, 2}, {3}}
sage: p.antipode_on_basis(SetPartition([]))
p{}
sage: F = p[[1,3],[5],[2,4]].coproduct()
sage: F.apply_multilinear_morphism(lambda x,y: x.antipode()*y)
0
```

SymmetricFunctionsNonCommutingVariables.powersum.**coproduct_on_basis**(A)

Return the coproduct of a monomial basis element.

INPUT:

•A – a set partition

OUTPUT:

•The coproduct applied to the monomial symmetric function in non-commuting variables indexed by A expressed in the monomial basis.

EXAMPLES:

```
sage: p = SymmetricFunctionsNonCommutingVariables(QQ).powersum()
sage: p[[1, 3], [2]].coproduct()
p{} # p{{1, 3}, {2}} + p{{1}} # p{{1, 2}} + p{{1, 2}} # p{{1}} + p{{1, 3}, {2}} # p{}
sage: p.coproduct_on_basis(SetPartition([[1]]))
p{} # p{{1}} + p{{1}} # p{}
sage: p.coproduct_on_basis(SetPartition([]))
p{} # p{}
```

SymmetricFunctionsNonCommutingVariables.powersum.**internal_coproduct_on_basis**(*A*)

Return the internal coproduct of a powersum basis element.

The internal coproduct is defined by

$$\Delta^{\odot}(\mathbf{p}_A) = \mathbf{p}_A \otimes \mathbf{p}_A$$

INPUT:

•A – a set partition

OUTPUT:

•an element of the tensor square of self

EXAMPLES:

```
sage: p = SymmetricFunctionsNonCommutingVariables(QQ).powersum()
sage: p.internal_coproduct_on_basis(SetPartition([[1,3],[2]]))
p{{1, 3}, {2}} # p{{1, 3}, {2}}
```

SymmetricFunctionsNonCommutingVariables.powersum.**primitive**(*A*, *i=1*)

Return the primitive associated to A in self.

Fix some $i \in S$. Let A be an atomic set partition of S, then the primitive $p(A)$ given in [LM2011] is

$$p(A) = \sum_{\gamma} (-1)^{\ell(\gamma)-1} \mathbf{p}_{\gamma[A]}$$

where we sum over all ordered set partitions of $[\ell(A)]$ such that $i \in \gamma_1$ and $\gamma[A]$ is the action of γ on A defined in SetPartition.ordered_set_partition_action(). If A is not atomic, then $p(A) = 0$.

See also:

SetPartition.is_atomic()

INPUT:

•A – a set partition

•i – (default: 1) index in the base set for A specifying which set of primitives this belongs to

OUTPUT:

•an element in the basis self

EXAMPLES:

```
sage: p = SymmetricFunctionsNonCommutingVariables(QQ).powersum()
sage: elt = p.primitive(SetPartition([[1,3], [2]])); elt
-p{{1, 2}, {3}} + p{{1, 3}, {2}}
sage: elt.coproduct()
-p{} # p{{1, 2}, {3}} + p{} # p{{1, 3}, {2}} - p{{1, 2}, {3}} # p{} + p{{1, 3}, {2}} # p
sage: p.primitive(SetPartition([[1], [2,3]]))
0
sage: p.primitive(SetPartition([]))
p{}
```

SymmetricFunctionsNonCommutingVariables.**q**()
Old name for the **cp**-basis. Deprecated in trac ticket #18371.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: NCSym.q()
doctest:...: DeprecationWarning: q is deprecated, use instead cp or coarse_powersum.
See http://trac.sagemath.org/18371 for details.
Symmetric functions in non-commuting variables over the Rational Field in the coarse_powersu
```

class SymmetricFunctionsNonCommutingVariables.**supercharacter**(*NCSym*, *q=2*)
Bases: sage.combinat.ncsym.bases.NCSymBasis_abstract

The Hopf algebra of symmetric functions in non-commuting variables in the supercharacter χ basis.

This basis was defined in [BT13] as a q-deformation of the supercharacter basis.

$$\chi_A = \sum_B \chi_A(B)\mathbf{m}_B,$$

where we sum over all set partitions A and $\chi_A(B)$ is the evaluation of the supercharacter χ_A on the superclass μ_B.

Note: The supercharacters considered in [BT13] are coarser than those considered by Aguiar et. al.

INPUT:

- q – (default: 2) the parameter q

EXAMPLES:

```
sage: R = QQ['q'].fraction_field()
sage: q = R.gen()
sage: NCSym = SymmetricFunctionsNonCommutingVariables(R)
sage: chi = NCSym.chi(q)
sage: chi[[1,3],[2]]*chi[[1,2]]
chi{{1, 3}, {2}, {4, 5}}
sage: chi[[1,3],[2]].coproduct()
chi{} # chi{{1, 3}, {2}} + (2*q-2)*chi{{1}} # chi{{1}, {2}} +
 (3*q-2)*chi{{1}} # chi{{1, 2}} + (2*q-2)*chi{{1}, {2}} # chi{{1}} +
 (3*q-2)*chi{{1, 2}} # chi{{1}} + chi{{1, 3}, {2}} # chi{}
sage: chi2 = NCSym.chi()
sage: chi(chi2[[1,2],[3]])
((-q+2)/q)*chi{{1}, {2}, {3}} + 2/q*chi{{1, 2}, {3}}
sage: chi2
Symmetric functions in non-commuting variables over the Fraction Field
 of Univariate Polynomial Ring in q over Rational Field in the
 supercharacter basis with parameter q=2
```

q()
Return the deformation parameter q of self.

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: chi = NCSym.chi(5)
sage: chi.q()
5

sage: R = QQ['q'].fraction_field()
sage: q = R.gen()
```

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(R)
sage: chi = NCSym.chi(q)
sage: chi.q() == q
True
```

class SymmetricFunctionsNonCommutingVariables.**x_basis**(*NCSym*)

Bases: sage.combinat.ncsym.bases.NCSymBasis_abstract

The Hopf algebra of symmetric functions in non-commuting variables in the $\mathbf{x}$ basis.

This basis is defined in [BHRZ06] by the formula:

$$\mathbf{x}_A = \sum_{B \leq A} \mu(B, A) \mathbf{p}_B$$

and has the following properties:

$$\mathbf{x}_A \mathbf{x}_B = \mathbf{x}_{A|B}, \qquad \Delta^{\odot}(\mathbf{x}_C) = \sum_{A \vee B = C} \mathbf{x}_A \otimes \mathbf{x}_B.$$

EXAMPLES:

```
sage: NCSym = SymmetricFunctionsNonCommutingVariables(QQ)
sage: x = NCSym.x()
sage: x[[1,3],[2,4]]*x[[1,2,3]]
x{{1, 3}, {2, 4}, {5, 6, 7}}
sage: x[[1,2],[3]].internal_coproduct()
x{{1}, {2}, {3}} # x{{1, 2}, {3}} + x{{1, 2}, {3}} # x{{1}, {2}, {3}} +
 x{{1, 2}, {3}} # x{{1, 2}, {3}}
```

sage.combinat.ncsym.ncsym.**matchings**(*A*, *B*)

Iterate through all matchings of the sets A and B.

EXAMPLES:

```
sage: from sage.combinat.ncsym.ncsym import matchings
sage: list(matchings([1, 2, 3], [-1, -2]))
[[[1], [2], [3], [-1], [-2]],
 [[1], [2], [3, -1], [-2]],
 [[1], [2], [3, -2], [-1]],
 [[1], [2, -1], [3], [-2]],
 [[1], [2, -1], [3, -2]],
 [[1], [2, -2], [3], [-1]],
 [[1], [2, -2], [3, -1]],
 [[1, -1], [2], [3], [-2]],
 [[1, -1], [2], [3, -2]],
 [[1, -1], [2, -2], [3]],
 [[1, -2], [2], [3], [-1]],
 [[1, -2], [2], [3, -1]],
 [[1, -2], [2, -1], [3]]]
```

sage.combinat.ncsym.ncsym.**nesting**(*la*, *nu*)

Return the nesting number of `la` inside of `nu`.

If we consider a set partition A as a set of arcs $i - j$ where i and j are in the same part of A. Define

$$\mathrm{nst}_{\lambda}^{\nu} = \#\{i < j < k < l \mid i - l \in \nu, j - k \in \lambda\},$$

and this corresponds to the number of arcs of λ strictly contained inside of ν.

EXAMPLES:

```
sage: from sage.combinat.ncsym.ncsym import nesting
sage: nu = SetPartition([[1,4], [2], [3]])
sage: mu = SetPartition([[1,4], [2,3]])
sage: nesting(set(mu).difference(nu), nu)
1
```

```
sage: lst = list(SetPartitions(4))
sage: d = {}
sage: for i, nu in enumerate(lst):
....:     for mu in nu.coarsenings():
....:         if set(nu.arcs()).issubset(mu.arcs()):
....:             d[i, lst.index(mu)] = nesting(set(mu).difference(nu), nu)
sage: matrix(d)
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

5.1.137 Necklaces

The algorithm used in this file comes from

- Sawada, Joe. "A fast algorithm to generate necklaces with fixed content", Source Theoretical Computer Science archive Volume 301 , Issue 1-3 (May 2003)

`sage.combinat.necklace.`**`Necklaces`**(*content*)

Return the combinatorial class of necklaces with evaluation `content`.

A necklace is a list of integers that such that the list is the smallest lexicographic representative of all the cyclic shifts of the list.

See also:

`LyndonWords`

INPUT:

- `content` – a list of non-negative integers

EXAMPLES:

```
sage: Necklaces([2,1,1])
Necklaces with evaluation [2, 1, 1]
sage: Necklaces([2,1,1]).cardinality()
3
sage: Necklaces([2,1,1]).first()
[1, 1, 2, 3]
sage: Necklaces([2,1,1]).last()
[1, 2, 1, 3]
```

```
sage: Necklaces([2,1,1]).list()
[[1, 1, 2, 3], [1, 1, 3, 2], [1, 2, 1, 3]]
sage: Necklaces([0,2,1,1]).list()
[[2, 2, 3, 4], [2, 2, 4, 3], [2, 3, 2, 4]]
sage: Necklaces([2,0,1,1]).list()
[[1, 1, 3, 4], [1, 1, 4, 3], [1, 3, 1, 4]]
```

class sage.combinat.necklace.**Necklaces_evaluation**(*content*)

Bases: sage.combinat.combinat.CombinatorialClass

Necklaces with a fixed evaluation (content).

INPUT:

- content – a list of non-negative integers

cardinality()

Return the number of integer necklaces with the evaluation content.

The formula for the number of necklaces of content α a composition of n is:

$$\sum_{d|gcd(\alpha)} \phi(d) \binom{n/d}{\alpha_1/d, \ldots, \alpha_\ell/d},$$

where $\phi(d)$ is the Euler ϕ function.

EXAMPLES:

```
sage: Necklaces([]).cardinality()
0
sage: Necklaces([2,2]).cardinality()
2
sage: Necklaces([2,3,2]).cardinality()
30
sage: Necklaces([0,3,2]).cardinality()
2
```

Check to make sure that the count matches up with the number of necklace words generated.

```
sage: comps = [[],[2,2],[3,2,7],[4,2],[0,4,2],[2,0,4]]+Compositions(4).list()
sage: ns = [ Necklaces(comp) for comp in comps]
sage: all( [ n.cardinality() == len(n.list()) for n in ns] )
True
```

content()

Return the content (or evaluation) of the necklaces.

TESTS:

```
sage: N = Necklaces([2,2,2])
sage: N.content()
[2, 2, 2]
```

e()

Deprecated in trac ticket #17436. Use content() instead.

TESTS:

```
sage: N = Necklaces([2,2,2])
sage: N.e
doctest:...: DeprecationWarning: e attribute is deprecated. Use the content method instead
```

```
See http://trac.sagemath.org/17436 for details.
[2, 2, 2]
```

5.1.138 Non-Decreasing Parking Functions

A *non-decreasing parking function* of size n is a non-decreasing function f from $\{1, \ldots, n\}$ to itself such that for all i, one has $f(i) \leq i$.

The number of non-decreasing parking functions of size n is the n-th Catalan number.

The set of non-decreasing parking functions of size n is in bijection with the set of Dyck words of size n.

AUTHORS:

- Florent Hivert (2009-04)
- Christian Stump (2012-11) added pretty printing

class sage.combinat.non_decreasing_parking_function.**NonDecreasingParkingFunction**(*lst*)

Bases: sage.combinat.combinat.CombinatorialObject

A *non decreasing parking function* of size n is a non-decreasing function f from $\{1, \ldots, n\}$ to itself such that for all i, one has $f(i) \leq i$.

EXAMPLES:

```
sage: NonDecreasingParkingFunction([])
[]
sage: NonDecreasingParkingFunction([1])
[1]
sage: NonDecreasingParkingFunction([2])
Traceback (most recent call last):
...
ValueError: [2] is not a non-decreasing parking function
sage: NonDecreasingParkingFunction([1,2])
[1, 2]
sage: NonDecreasingParkingFunction([1,1,2])
[1, 1, 2]
sage: NonDecreasingParkingFunction([1,1,4])
Traceback (most recent call last):
...
ValueError: [1, 1, 4] is not a non-decreasing parking function
```

classmethod from_dyck_word(*dw*)

Bijection from Dyck words. It is the inverse of the bijection to_dyck_word(). You can find there the mathematical definition.

EXAMPLES:

```
sage: NonDecreasingParkingFunction.from_dyck_word([])
[]
sage: NonDecreasingParkingFunction.from_dyck_word([1,0])
[1]
sage: NonDecreasingParkingFunction.from_dyck_word([1,1,0,0])
[1, 1]
sage: NonDecreasingParkingFunction.from_dyck_word([1,0,1,0])
[1, 2]
sage: NonDecreasingParkingFunction.from_dyck_word([1,0,1,1,0,1,0,0,1,0])
[1, 2, 2, 3, 5]
```

TESTS:

```
sage: ndpf=NonDecreasingParkingFunctions(5);
sage: list(ndpf) == [NonDecreasingParkingFunction.from_dyck_word(pf.to_dyck_word()) for pf i
True
```

to_dyck_word()

Implements the bijection to Dyck words, which is defined as follows. Take a non decreasing parking function, say [1,1,2,4,5,5], and draw its graph:

```
          ___
         |  . 5
        _|  . 5
    ___|  . . 4
 _|  . . . . 2
|  . . . . . 1
|  . . . . . 1
```

The corresponding Dyck word [1,1,0,1,0,0,1,0,1,1,0,0] is then read off from the sequence of horizontal and vertical steps. The converse bijection is from_dyck_word().

EXAMPLES:

```
sage: NonDecreasingParkingFunction([1,1,2,4,5,5]).to_dyck_word()
[1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0]
sage: NonDecreasingParkingFunction([]).to_dyck_word()
[]
sage: NonDecreasingParkingFunction([1,1,1]).to_dyck_word()
[1, 1, 1, 0, 0, 0]
sage: NonDecreasingParkingFunction([1,2,3]).to_dyck_word()
[1, 0, 1, 0, 1, 0]
sage: NonDecreasingParkingFunction([1,1,3,3,4,6,6]).to_dyck_word()
[1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0]
```

TESTS:

```
sage: ndpf=NonDecreasingParkingFunctions(5);
sage: list(ndpf) == [pf.to_dyck_word().to_non_decreasing_parking_function() for pf in ndpf]
True
```

sage.combinat.non_decreasing_parking_function.**NonDecreasingParkingFunctions**(*n=None*)

Returns the combinatorial class of Non-Decreasing Parking Functions.

A *non-decreasing parking function* of size n is a non-decreasing function f from $\{1, \ldots, n\}$ to itself such that for all i, one has $f(i) \leq i$.

EXAMPLES:

Here are all the-non decreasing parking functions of size 5:

```
sage: NonDecreasingParkingFunctions(3).list()
[[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 2, 2], [1, 2, 3]]
```

If no size is specified, then NonDecreasingParkingFunctions returns the combinatorial class of all non-decreasing parking functions.

```
sage: PF = NonDecreasingParkingFunctions(); PF
Non-decreasing parking functions
sage: [] in PF
True
sage: [1] in PF
True
sage: [2] in PF
```

```
False
sage: [1,1,3] in PF
True
sage: [1,1,4] in PF
False
```

If the size n is specified, then NonDecreasingParkingFunctions returns combinatorial class of all non-decreasing parking functions of size n.

```
sage: PF = NonDecreasingParkingFunctions(0)
sage: PF.list()
[[]]
sage: PF = NonDecreasingParkingFunctions(1)
sage: PF.list()
[[1]]
sage: PF = NonDecreasingParkingFunctions(3)
sage: PF.list()
[[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 2, 2], [1, 2, 3]]

sage: PF3 = NonDecreasingParkingFunctions(3); PF3
Non-decreasing parking functions of size 3
sage: [] in PF3
False
sage: [1] in PF3
False
sage: [1,1,3] in PF3
True
sage: [1,1,4] in PF3
False
```

TESTS:

```
sage: PF = NonDecreasingParkingFunctions(5)
sage: len(PF.list()) == PF.cardinality()
True
sage: NonDecreasingParkingFunctions("foo")
Traceback (most recent call last):
...
TypeError: unable to convert 'foo' to an integer
```

class sage.combinat.non_decreasing_parking_function.**NonDecreasingParkingFunctions_all**

Bases: sage.combinat.combinat.InfiniteAbstractCombinatorialClass

TESTS:

```
sage: DW = NonDecreasingParkingFunctions()
sage: DW == loads(dumps(DW))
True
```

class sage.combinat.non_decreasing_parking_function.**NonDecreasingParkingFunctions_n**(n)

Bases: sage.combinat.combinat.CombinatorialClass

The combinatorial class of non-decreasing parking functions of size n.

A *non-decreasing parking function* of size n is a non-decreasing function f from $\{1, \ldots, n\}$ to itself such that for all i, one has $f(i) \leq i$.

The number of non-decreasing parking functions of size n is the n-th Catalan number.

EXAMPLES:

```
sage: PF = NonDecreasingParkingFunctions(3)
sage: PF.list()
[[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 2, 2], [1, 2, 3]]
sage: PF = NonDecreasingParkingFunctions(4)
sage: PF.list()
[[1, 1, 1, 1], [1, 1, 1, 2], [1, 1, 1, 3], [1, 1, 1, 4], [1, 1, 2, 2], [1, 1, 2, 3], [1, 1, 2, 4
sage: [ NonDecreasingParkingFunctions(i).cardinality() for i in range(10)]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

Warning: The precise order in which the parking function are generated or listed is not fixed, and may change in the future.

AUTHORS:

•Florent Hivert

cardinality()

Returns the number of non-decreasing parking functions of size n. This number is the n-th Catalan number.

EXAMPLES:

```
sage: PF = NonDecreasingParkingFunctions(0)
sage: PF.cardinality()
1
sage: PF = NonDecreasingParkingFunctions(1)
sage: PF.cardinality()
1
sage: PF = NonDecreasingParkingFunctions(3)
sage: PF.cardinality()
5
sage: PF = NonDecreasingParkingFunctions(5)
sage: PF.cardinality()
42
```

sage.combinat.non_decreasing_parking_function.**is_a**(*x*, *n=None*)

Check whether a list is a non-decreasing parking function. If a size n is specified, checks if a list is a non-decreasing parking function of size n.

TESTS:

```
sage: from sage.combinat.non_decreasing_parking_function import is_a
sage: is_a([1,1,2])
True
sage: is_a([1,1,4])
False
sage: is_a([1,1,3], 3)
True
```

5.1.139 Ordered Rooted Trees

AUTHORS:

- Florent Hivert (2010-2011): initial revision
- Frederic Chapoton (2010): contributed some methods

class sage.combinat.ordered_tree.**LabelledOrderedTree**(*parent*, *children*, *label=None*, *check=True*)

Bases: sage.combinat.abstract_tree.AbstractLabelledClonableTree, sage.combinat.ordered_tree.OrderedTree

Labelled ordered trees.

A labelled ordered tree is an ordered tree with a label attached at each node.

INPUT:

- children – a list or tuple or more generally any iterable of trees or object convertible to trees
- label – any Sage object (default: None)

EXAMPLES:

```
sage: x = LabelledOrderedTree([], label = 3); x
3[]
sage: LabelledOrderedTree([x, x, x], label = 2)
2[3[], 3[], 3[]]
sage: LabelledOrderedTree((x, x, x), label = 2)
2[3[], 3[], 3[]]
sage: LabelledOrderedTree([[],[[], []]], label = 3)
3[None[], None[None[], None[]]]
```

left_right_symmetry()

Return the symmetric tree of self.

The symmetric tree $s(T)$ of a labelled ordered tree T is defined as follows: If T is a labelled ordered tree with children $C_1, C_2, \ldots, C_k$ (listed from left to right), then the symmetric tree $s(T)$ of T is a labelled ordered tree with children $s(C_k), s(C_{k-1}), \ldots, s(C_1)$ (from left to right), and with the same root label as T.

Note: If you have a subclass of LabelledOrderedTree() which also inherits from another subclass of OrderedTree() which does not come with a labelling, then (depending on the method resolution order) it might happen that this method gets overridden by an implementation from that other subclass, and thus forgets about the labels. In this case you need to manually override this method on your subclass.

EXAMPLES:

```
sage: L2 = LabelledOrderedTree([], label=2)
sage: L3 = LabelledOrderedTree([], label=3)
sage: T23 = LabelledOrderedTree([L2, L3], label=4)
sage: T23.left_right_symmetry()
4[3[], 2[]]
sage: T223 = LabelledOrderedTree([L2, T23], label=17)
sage: T223.left_right_symmetry()
17[4[3[], 2[]], 2[]]
sage: T223.left_right_symmetry().left_right_symmetry() == T223
True
```

sort_key()

Return a tuple of nonnegative integers encoding the labelled tree self.

The first entry of the tuple is a pair consisting of the number of children of the root and the label of the root. Then the rest of the tuple is the concatenation of the tuples associated to these children (we view the children of a tree as trees themselves) from left to right.

This tuple characterizes the labelled tree uniquely, and can be used to sort the labelled ordered trees provided that the labels belong to a type which is totally ordered.

Warning: This method overrides `OrderedTree.sort_key()` and returns a result different from what the latter would return, as it wants to encode the whole labelled tree including its labelling rather than just the unlabelled tree. Therefore, be careful with using this method on subclasses of `LabelledOrderedTree`; under some circumstances they could inherit it from another superclass instead of from `OrderedTree`, which would cause the method to forget the labelling. See the docstring of `OrderedTree.sort_key()`.

EXAMPLES:

```
sage: L2 = LabelledOrderedTree([], label=2)
sage: L3 = LabelledOrderedTree([], label=3)
sage: T23 = LabelledOrderedTree([L2, L3], label=4)
sage: T23.sort_key()
((2, 4), (0, 2), (0, 3))
sage: T32 = LabelledOrderedTree([L3, L2], label=5)
sage: T32.sort_key()
((2, 5), (0, 3), (0, 2))
sage: T23322 = LabelledOrderedTree([T23, T32, L2], label=14)
sage: T23322.sort_key()
((3, 14), (2, 4), (0, 2), (0, 3), (2, 5), (0, 3), (0, 2), (0, 2))
```

class sage.combinat.ordered_tree.**LabelledOrderedTrees**(*category=None*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

This is a parent stub to serve as a factory class for trees with various label constraints.

EXAMPLES:

```
sage: LOT = LabelledOrderedTrees(); LOT
Labelled ordered trees
sage: x = LOT([], label = 3); x
3[]
sage: x.parent() is LOT
True
sage: y = LOT([x, x, x], label = 2); y
2[3[], 3[], 3[]]
sage: y.parent() is LOT
True
```

Element

alias of `LabelledOrderedTree`

cardinality()

Return the cardinality of `self`.

EXAMPLE:

```
sage: LabelledOrderedTrees().cardinality()
+Infinity
```

labelled_trees()

Return the set of labelled trees associated to `self`.

This is precisely `self`, because `self` already is the set of labelled ordered trees.

EXAMPLES:

```
sage: LabelledOrderedTrees().labelled_trees()
Labelled ordered trees
sage: LOT = LabelledOrderedTrees()
```

```
sage: x = LOT([], label = 3)
sage: y = LOT([x, x, x], label = 2)
sage: y.canonical_labelling()
1[2[], 3[], 4[]]
```

unlabelled_trees()

Return the set of unlabelled trees associated to `self`.

This is the set of ordered trees, since `self` is the set of labelled ordered trees.

EXAMPLES:

```
sage: LabelledOrderedTrees().unlabelled_trees()
Ordered trees
```

class sage.combinat.ordered_tree.**OrderedTree**(*parent=None*, *children=[]*, *check=True*)

Bases: sage.combinat.abstract_tree.AbstractClonableTree, sage.structure.list_clone.ClonableList

The class of (ordered rooted) trees.

An ordered tree is constructed from a node, called the root, on which one has grafted a possibly empty list of trees. There is a total order on the children of a node which is given by the order of the elements in the list. Note that there is no empty ordered tree (so the smallest ordered tree consists of just one node).

INPUT:

One can create a tree from any list (or more generally iterable) of trees or objects convertible to a tree. Alternatively a string is also accepted. The syntax is the same as for printing: children are grouped by square brackets.

EXAMPLES:

```
sage: x = OrderedTree([])
sage: x1 = OrderedTree([x,x])
sage: x2 = OrderedTree([[],[]])
sage: x1 == x2
True
sage: tt1 = OrderedTree([x,x1,x2])
sage: tt2 = OrderedTree([[], [[], []], x2])
sage: tt1 == tt2
True

sage: OrderedTree([]) == OrderedTree()
True
```

TESTS:

```
sage: x1.__hash__() == x2.__hash__()
True
sage: tt1.__hash__() == tt2.__hash__()
True
```

Trees are usually immutable. However they inherit from `sage.structure.list_clone.ClonableList`, so that they can be modified using the clone protocol. Let us now see what this means.

Trying to modify a non-mutable tree raises an error:

```
sage: tt1[1] = tt2
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

Here is the correct way to do it:

```
sage: with tt2.clone() as tt2:
....:     tt2[1] = tt1
sage: tt2
[[], [[], [[], []], [[], []]], [[], []]]
```

It is also possible to append a child to a tree:

```
sage: with tt2.clone() as tt3:
....:     tt3.append(OrderedTree([]))
sage: tt3
[[], [[], [[], []], [[], []]], [[], []], []]
```

Or to insert a child in a tree:

```
sage: with tt2.clone() as tt3:
....:     tt3.insert(2, OrderedTree([]))
sage: tt3
[[], [[], [[], []], [[], []]], [], [[], []]]
```

We check that `tt1` is not modified and that everything is correct with respect to equality:

```
sage: tt1
[[], [[], []], [[], []]]
sage: tt1 == tt2
False
sage: tt1.__hash__() == tt2.__hash__()
False
```

TESTS:

```
sage: tt1bis = OrderedTree(tt1)
sage: with tt1.clone() as tt1:
....:     tt1[1] = tt1bis
sage: tt1
[[], [[], [[], []], [[], []]], [[], []]]
sage: tt1 == tt2
True
sage: tt1.__hash__() == tt2.__hash__()
True
sage: len(tt1)
3
sage: tt1[2]
[[], []]
sage: tt1[3]
Traceback (most recent call last):
...
IndexError: list index out of range
sage: tt1[1:2]
[[[], [[], []], [[], []]]]
```

Various tests involving construction, equality and hashing:

```
sage: OrderedTree() == OrderedTree()
True
sage: t1 = OrderedTree([[],[[]]])
sage: t2 = OrderedTree([[],[[]]])
sage: t1 == t2
```

```
True
sage: t2 = OrderedTree(t1)
sage: t1 == t2
True
sage: t1 = OrderedTree([[],[[]]])
sage: t2 = OrderedTree([[[]],[]])
sage: t1 == t2
False

sage: t1 = OrderedTree([[],[[]]])
sage: t2 = OrderedTree([[],[[]]])
sage: t1.__hash__() == t2.__hash__()
True
sage: t2 = OrderedTree([[[]],[]])
sage: t1.__hash__() == t2.__hash__()
False
sage: OrderedTree().__hash__() == OrderedTree([]).__hash__()
True
sage: tt1 = OrderedTree([t1,t2,t1])
sage: tt2 = OrderedTree([t1, [[[]],[]], t1])
sage: tt1.__hash__() == tt2.__hash__()
True
```

Check that the hash value is correctly updated after modification:

```
sage: with tt2.clone() as tt2:
....:     tt2[1,1] = tt1
sage: tt1.__hash__() == tt2.__hash__()
False
```

dendrog_cmp(*other*)

Return -1 if `self` is smaller than `other` in the dendrographical order; return 0 if they are equal; return 1 if `other` is smaller.

The dendrographical order is a total order on the set of unlabelled ordered rooted trees; it is defined recursively as follows: An ordered rooted tree T with children $T_1, T_2, \ldots, T_a$ is smaller than an ordered rooted tree S with children $S_1, S_2, \ldots, S_b$ if either $a < b$ or ($a = b$ and there exists a $1 \leq i \leq a$ such that $T_1 = S_1, T_2 = S_2, \ldots, T_{i-1} = S_{i-1}$ and $T_i < S_i$).

INPUT:

- `other` – an ordered rooted tree

OUTPUT:

- -1, if `smaller < other` with respect to the dendrographical order.
- 0, if `smaller == other` (as unlabelled ordered rooted trees).
- 1, if `smaller > other` with respect to the dendrographical order.

Note: It is possible to provide labelled trees to this method; however, their labels are ignored.

EXAMPLES:

```
sage: OT = OrderedTree
sage: ta = OT([])
sage: tb = OT([[], [], [[], []]])
sage: tc = OT([[], [[], []], []])
sage: td = OT([[[], []], [], []])
sage: te = OT([[], []])
```

```
sage: tf = OT([[], [], []])
sage: tg = OT([[[], []], [[], []]])
sage: l = [ta, tb, tc, td, te, tf, tg]
sage: [l[i].dendrog_cmp(l[j]) for i in range(7) for j in range(7)]
[0, -1, -1, -1, -1, -1, -1,
 1, 0, -1, -1, 1, 1, 1,
 1, 1, 0, -1, 1, 1, 1,
 1, 1, 1, 0, 1, 1, 1,
 1, -1, -1, -1, 0, -1, -1,
 1, -1, -1, -1, 1, 0, 1,
 1, -1, -1, -1, 1, -1, 0]
```

dendrog_normalize(*inplace=False*)

Return the normalized tree of the *unlabelled* ordered rooted tree `self` with respect to the dendrographical order.

INPUT:

- `inplace` – (default `False`) boolean; if `True`, then `self` is modified and nothing returned; otherwise the normalized tree is returned

The normalized tree of an unlabelled ordered rooted tree t with respect to the dendrographical order is an unlabelled ordered rooted tree defined recursively as follows: We first replace all children of t by their normalized trees (with respect to the dendrographical order); then, we reorder these children in weakly increasing order with respect to the dendrographical order (`dendrog_cmp()`).

This can be viewed as an alternative to `normalize()` for the case of unlabelled ordered rooted trees.

EXAMPLES:

```
sage: OT = OrderedTree
sage: ta = OT([[],[[]]])
sage: tb = OT([[[]],[]])
sage: ta.dendrog_normalize() == tb.dendrog_normalize()
True
sage: ta == tb
False
sage: ta.dendrog_normalize()
[[], [[]]]
```

An example with inplace normalization:

```
sage: OT = OrderedTree
sage: ta = OT([[],[[]]])
sage: tb = OT([[[]],[]])
sage: ta.dendrog_normalize(inplace=True); ta
[[], [[]]]
sage: tb.dendrog_normalize(inplace=True); tb
[[], [[]]]
```

is_empty()

Return if `self` is the empty tree.

For ordered trees, this always returns `False`.

Note: this is different from `bool(t)` which returns whether `t` has some child or not.

EXAMPLES:

```
sage: t = OrderedTrees(4)([[],[[]]])
sage: t.is_empty()
False
sage: bool(t)
True
```

left_right_symmetry()

Return the symmetric tree of `self`.

The symmetric tree $s(T)$ of an ordered tree T is defined as follows: If T is an ordered tree with children $C_1, C_2, \ldots, C_k$ (listed from left to right), then the symmetric tree $s(T)$ of T is the ordered tree with children $s(C_k), s(C_{k-1}), \ldots, s(C_1)$ (from left to right).

EXAMPLES:

```
sage: T = OrderedTree([[],[[]]])
sage: T.left_right_symmetry()
[[[]], []]
sage: T = OrderedTree([[], [[], []], [[], [[]]]])
sage: T.left_right_symmetry()
[[[[]], []], [[], []], []]
```

normalize(*inplace=False*)

Return the normalized tree of `self`.

INPUT:

- `inplace` – boolean, (default `False`) if `True`, then `self` is modified and nothing returned. Otherwise the normalized tree is returned.

The normalization of an ordered tree t is an ordered tree s which has the property that t and s are isomorphic as *unordered* rooted trees, and that if two ordered trees t and t' are isomorphic as *unordered* rooted trees, then the normalizations of t and t' are identical. In other words, normalization is a map from the set of ordered trees to itself which picks a representative from every equivalence class with respect to the relation of "being isomorphic as unordered trees", and maps every ordered tree to the representative chosen from its class.

This map proceeds recursively by first normalizing every subtree, and then sorting the subtrees according to the value of the `sort_key()` method.

See also `dendrog_normalize()` for an alternative that works for unlabelled trees.

Consider the quotient map π that sends a planar rooted tree to the associated unordered rooted tree. Normalization is the composite $s \circ \pi$, where s is a section of π.

EXAMPLES:

```
sage: OT = OrderedTree
sage: ta = OT([[],[[]]])
sage: tb = OT([[[]],[]])
sage: ta.normalize() == tb.normalize()
True
sage: ta == tb
False
```

An example with inplace normalization:

```
sage: OT = OrderedTree
sage: ta = OT([[],[[]]])
sage: tb = OT([[[]],[]])
sage: ta.normalize(inplace=True); ta
[[], [[]]]
```

```
sage: tb.normalize(inplace=True); tb
[[], [[]]]
```

sort_key()

Return a tuple of nonnegative integers encoding the ordered tree `self`.

The first entry of the tuple is the number of children of the root. Then the rest of the tuple is the concatenation of the tuples associated to these children (we view the children of a tree as trees themselves) from left to right.

This tuple characterizes the tree uniquely, and can be used to sort the ordered trees.

Note: By default, this method does not encode any extra structure that `self` might have – e.g., if you were to define a class `EdgeColoredOrderedTree` which implements edge-colored trees and which inherits from `OrderedTree`, then the `sort_key()` method it would inherit would forget about the colors of the edges (and thus would not characterize edge-colored trees uniquely). If you want to preserve extra data, you need to override this method or use a new method. For instance, on the `LabelledOrderedTree` subclass, this method is overridden by a slightly different method, which encodes not only the numbers of children of the nodes of `self`, but also their labels. Be careful with using overridden methods, however: If you have (say) a class `BalancedTree` which inherits from `OrderedTree` and which encodes balanced trees, and if you have another class `BalancedLabelledOrderedTree` which inherits both from `BalancedOrderedTree` and from `LabelledOrderedTree`, then (depending on the MRO) the default `sort_key()` method on `BalancedLabelledOrderedTree` (unless manually overridden) will be taken either from `BalancedTree` or from `LabelledOrderedTree`, and in the former case will ignore the labelling!

EXAMPLES:

```
sage: RT = OrderedTree
sage: RT([[],[[]]]).sort_key()
(2, 0, 1, 0)
sage: RT([[[]],[]]).sort_key()
(2, 1, 0, 0)
```

to_binary_tree_left_branch()

Return a binary tree of size $n-1$ (where n is the size of t, and where t is `self`) obtained from t by the following recursive rule:

- if x is the left brother of y in t, then x becomes the left child of y;
- if x is the last child of y in t, then x becomes the right child of y,

and removing the root of t.

EXAMPLES:

```
sage: T = OrderedTree([[],[]])
sage: T.to_binary_tree_left_branch()
[[., .], .]
sage: T = OrderedTree([[], [[], []], [[], [[]]]])
sage: T.to_binary_tree_left_branch()
[[[., .], [[., .], .]], [[., .], [., .]]]
```

TESTS:

```
sage: T = OrderedTree([[],[]])
sage: T == T.to_binary_tree_left_branch().to_ordered_tree_left_branch()
True
sage: T = OrderedTree([[], [[], []], [[], [[]]]])
```

```
sage: T == T.to_binary_tree_left_branch().to_ordered_tree_left_branch()
True
```

to_binary_tree_right_branch()

Return a binary tree of size $n-1$ (where n is the size of t, and where t is `self`) obtained from t by the following recursive rule:

- if x is the right brother of y in t, then'x' becomes the right child of y;
- if x is the first child of y in t, then x becomes the left child of y,

and removing the root of t.

EXAMPLES:

```
sage: T = OrderedTree([[],[]])
sage: T.to_binary_tree_right_branch()
[., [., .]]
sage: T = OrderedTree([[], [[], []], [[], [[]]]])
sage: T.to_binary_tree_right_branch()
[., [[., [., .]], [[., [[., .], .]], .]]]
```

TESTS:

```
sage: T = OrderedTree([[],[]])
sage: T == T.to_binary_tree_right_branch().to_ordered_tree_right_branch()
True
sage: T = OrderedTree([[], [[], []], [[], [[]]]])
sage: T == T.to_binary_tree_right_branch().to_ordered_tree_right_branch()
True
```

to_dyck_word()

Return the Dyck path corresponding to `self` where the maximal height of the Dyck path is the depth of `self`.

EXAMPLES:

```
sage: T = OrderedTree([[],[]])
sage: T.to_dyck_word()
[1, 0, 1, 0]
sage: T = OrderedTree([[],[[]]])
sage: T.to_dyck_word()
[1, 0, 1, 1, 0, 0]
sage: T = OrderedTree([[], [[], []], [[], [[]]]])
sage: T.to_dyck_word()
[1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0]
```

to_poset(*root_to_leaf=False*)

Return the poset obtained by interpreting the tree as a Hasse diagram. The default orientation is from leaves to root but you can pass `root_to_leaf=True` to obtain the inverse orientation.

INPUT:

- `root_to_leaf` – boolean, true if the poset orientation should be from root to leaves. It is false by default.

EXAMPLES:

```
sage: t = OrderedTree([])
sage: t.to_poset()
Finite poset containing 1 elements
sage: p = OrderedTree([[[]],[],[]]).to_poset()
```

```
sage: p.height(), p.width()
(3, 3)
```

If the tree is labelled, we use its labelling to label the poset. Otherwise, we use the poset canonical labelling:

```
sage: t = OrderedTree([[[]],[],[]]).canonical_labelling().to_poset()
sage: t.height(), t.width()
(3, 3)
```

to_undirected_graph()

Return the undirected graph obtained from the tree nodes and edges.

EXAMPLES:

```
sage: t = OrderedTree([])
sage: t.to_undirected_graph()
Graph on 1 vertex
sage: t = OrderedTree([[[]],[],[]])
sage: t.to_undirected_graph()
Graph on 5 vertices
```

If the tree is labelled, we use its labelling to label the graph. Otherwise, we use the graph canonical labelling which means that two different trees can have the same graph.

EXAMPLES:

```
sage: t = OrderedTree([[[]],[],[]])
sage: t.canonical_labelling().to_undirected_graph()
Graph on 5 vertices
sage: t.canonical_labelling().to_undirected_graph() == t.to_undirected_graph()
False
sage: OrderedTree([[],[]]).to_undirected_graph() == OrderedTree([[[]]]).to_undirected_graph(
True
sage: OrderedTree([[],[],[]]).to_undirected_graph() == OrderedTree([[[[]]]]).to_undirected_g
False
```

class sage.combinat.ordered_tree.**OrderedTrees**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Factory for ordered trees

INPUT:

- size – (optional) an integer

OUTPUT:

- the set of all ordered trees (of the given size if specified)

EXAMPLES:

```
sage: OrderedTrees()
Ordered trees

sage: OrderedTrees(2)
Ordered trees of size 2
```

Note: this is a factory class whose constructor returns instances of subclasses.

Note: the fact that OrderedTrees is a class instead of a simple callable is an implementation detail. It could be changed in the future and one should not rely on it.

---

leaf()

Return a leaf tree with `self` as parent

EXAMPLES:

```
sage: OrderedTrees().leaf()
[]
```

TEST:

```
sage: (OrderedTrees().leaf() is
....:     sage.combinat.ordered_tree.OrderedTrees_all().leaf())
True
```

class sage.combinat.ordered_tree.**OrderedTrees_all**

Bases: sage.sets.disjoint_union_enumerated_sets.DisjointUnionEnumeratedSets, sage.combinat.ordered_tree.OrderedTrees

The set of all ordered trees.

EXAMPLES:

```
sage: OT = OrderedTrees(); OT
Ordered trees
sage: OT.cardinality()
+Infinity
```

Element

alias of OrderedTree

labelled_trees()

Return the set of labelled trees associated to `self`

EXAMPLES:

```
sage: OrderedTrees().labelled_trees()
Labelled ordered trees
```

unlabelled_trees()

Return the set of unlabelled trees associated to `self`

EXAMPLES:

```
sage: OrderedTrees().unlabelled_trees()
Ordered trees
```

class sage.combinat.ordered_tree.**OrderedTrees_size**(*size*)

Bases: sage.combinat.ordered_tree.OrderedTrees

The enumerated sets of binary trees of a given size

EXAMPLES:

```
sage: S = OrderedTrees(3); S
Ordered trees of size 3
sage: S.cardinality()
2
sage: S.list()
[[[], []], [[[]]]]
```

cardinality()
The cardinality of `self`

This is a Catalan number.

TESTS:

```
sage: OrderedTrees(0).cardinality()
0
sage: OrderedTrees(1).cardinality()
1
sage: OrderedTrees(6).cardinality()
42
```

element_class()
The class of the element of `self`

EXAMPLES:

```
sage: from sage.combinat.ordered_tree import OrderedTrees_size, OrderedTrees_all
sage: S = OrderedTrees_size(3)
sage: S.element_class is OrderedTrees().element_class
True
sage: S.first().__class__ == OrderedTrees_all().first().__class__
True
```

random_element()
Return a random `OrderedTree` with uniform probability.

This method generates a random `DyckWord` and then uses a bijection between Dyck words and ordered trees.

EXAMPLES:

```
sage: OrderedTrees(5).random_element() # random
[[[], []], []]
sage: OrderedTrees(0).random_element()
Traceback (most recent call last):
...
EmptySetError: There are no ordered trees of size 0
sage: OrderedTrees(1).random_element()
[]
```

TESTS:

```
sage: all([OrderedTrees(10).random_element() in OrderedTrees(10) for i in range(20)])
True
```

5.1.140 Output functions

These are the output functions for latexing partitions and tableaux.

AUTHORS:

- Mike Hansen (?): initial version
- Andrew Mathas (2013-02-14): Added support for displaying conventions and lines, and tableaux of skew partition, composition, and skew/composition/partition/tableaux tuple shape.

`sage.combinat.output.`**tex_from_array**(*array*, *with_lines=True*)
Return a latex string for a two dimensional array of partition, composition or skew composition shape

INPUT:

- `array` – a list of list
- **`with_lines` – a boolean (default: `True`)** Whether to draw a line to separate the entries in the array.

Empty rows are allowed; however, such rows should be given as `[None]` rather than `[]`.

The array is drawn using either the English or French convention following `Tableaux.global_options`().

See also:

`tex_from_array_tuple()`

EXAMPLES:

```
sage: from sage.combinat.output import tex_from_array
sage: print tex_from_array([[1,2,3],[4,5]])
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\cline{1-3}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-3}
\lr{4}&\lr{5}\\\cline{1-2}
\end{array}$}
}
sage: print tex_from_array([[1,2,3],[4,5]], with_lines=False)
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\\
\lr{1}&\lr{2}&\lr{3}\\
\lr{4}&\lr{5}\\
\end{array}$}
}
sage: print tex_from_array([[1,2,3],[4,5,6,7],[8]])
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\cline{1-3}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-4}
\lr{4}&\lr{5}&\lr{6}&\lr{7}\\\cline{1-4}
\lr{8}\\\cline{1-1}
\end{array}$}
}
sage: print tex_from_array([[1,2,3],[4,5,6,7],[8]], with_lines=False)
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\\
\lr{1}&\lr{2}&\lr{3}\\
\lr{4}&\lr{5}&\lr{6}&\lr{7}\\
\lr{8}\\
\end{array}$}
}
sage: print tex_from_array([[None,None,3],[None,5,6,7],[8]])
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\cline{3-3}
&&\lr{3}\\\cline{2-4}
&\lr{5}&\lr{6}&\lr{7}\\\cline{1-4}
\lr{8}\\\cline{1-1}
\end{array}$}
}
sage: print tex_from_array([[None,None,3],[None,5,6,7],[None,8]])
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\cline{3-3}
&&\lr{3}\\\cline{2-4}
&\lr{5}&\lr{6}&\lr{7}\\\cline{2-4}
&\lr{8}\\\cline{2-2}
```

```
\end{array}$}
}
sage: print tex_from_array([[None,None,3],[None,5,6,7],[8]], with_lines=False)
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\\
&&\lr{3}\\
&\lr{5}&\lr{6}&\lr{7}\\
\lr{8}\\
\end{array}$}
}
sage: print tex_from_array([[None,None,3],[None,5,6,7],[None,8]], with_lines=False)
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\\
&&\lr{3}\\
&\lr{5}&\lr{6}&\lr{7}\\
&\lr{8}\\
\end{array}$}
}
sage: Tableaux.global_options(convention="french")
sage: print tex_from_array([[1,2,3],[4,5]])
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\cline{1-2}
\lr{4}&\lr{5}\\\cline{1-3}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-3}
\end{array}$}
}
sage: print tex_from_array([[1,2,3],[4,5]], with_lines=False)
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\\
\lr{4}&\lr{5}\\
\lr{1}&\lr{2}&\lr{3}\\
\end{array}$}
}
sage: print tex_from_array([[1,2,3],[4,5,6,7],[8]])
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\cline{1-1}
\lr{8}\\\cline{1-4}
\lr{4}&\lr{5}&\lr{6}&\lr{7}\\\cline{1-4}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-3}
\end{array}$}
}
sage: print tex_from_array([[1,2,3],[4,5,6,7],[8]], with_lines=False)
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\\
\lr{8}\\
\lr{4}&\lr{5}&\lr{6}&\lr{7}\\
\lr{1}&\lr{2}&\lr{3}\\
\end{array}$}
}
sage: print tex_from_array([[None,None,3],[None,5,6,7],[8]])
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\cline{1-1}
\lr{8}\\\cline{1-4}
&\lr{5}&\lr{6}&\lr{7}\\\cline{2-4}
&&\lr{3}\\\cline{3-3}
\end{array}$}
}
sage: print tex_from_array([[None,None,3],[None,5,6,7],[None,8]])
```

```
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\cline{2-2}
&\lr{8}\\\cline{2-4}
&\lr{5}&\lr{6}&\lr{7}\\\cline{2-4}
&&\lr{3}\\\cline{3-3}
\end{array}$}
}
sage: print tex_from_array([[None,None,3],[None,5,6,7],[8]], with_lines=False)
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\\
\lr{8}\\
&\lr{5}&\lr{6}&\lr{7}\\
&&\lr{3}\\
\end{array}$}
}
sage: print tex_from_array([[None,None,3],[None,5,6,7],[None,8]], with_lines=False)
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{4}c}\\
&\lr{8}\\
&\lr{5}&\lr{6}&\lr{7}\\
&&\lr{3}\\
\end{array}$}
}
sage: Tableaux.global_options.reset()
```

sage.combinat.output.**tex_from_array_tuple**(*a_tuple*, *with_lines=True*)

Return a latex string for a tuple of two dimensional array of partition, composition or skew composition shape.

INPUT:

- `a_tuple` – a tuple of lists of lists
- `with_lines` – a boolean (default: `True`) Whether to draw lines to separate the entries in the components of `a_tuple`.

See also:

`tex_from_array()` for the description of each array

EXAMPLES:

```
sage: from sage.combinat.output import tex_from_array_tuple
sage: print tex_from_array_tuple([[[1,2,3],[4,5]],[],[[None,6,7],[None,8],[9]]])
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\cline{1-3}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-3}
\lr{4}&\lr{5}\\\cline{1-2}
\end{array}$},\emptyset,\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\cline{2-3}
&\lr{6}&\lr{7}\\\cline{2-3}
&\lr{8}\\\cline{1-2}
\lr{9}\\\cline{1-1}
\end{array}$}
}
sage: print tex_from_array_tuple([[[1,2,3],[4,5]],[],[[None,6,7],[None,8],[9]]], with_lines=Fals
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\\
\lr{1}&\lr{2}&\lr{3}\\
\lr{4}&\lr{5}\\
\end{array}$},\emptyset,\raisebox{-.6ex}{$\begin{array}[b]{*{3}c}\\
&\lr{6}&\lr{7}\\
```

```
&\lr{8}\\
\lr{9}\\
\end{array}$}
}
sage: Tableaux.global_options(convention="french")
sage: print tex_from_array_tuple([[[1,2,3],[4,5]],[],[[None,6,7],[None,8],[9]]])
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\cline{1-2}
\lr{4}&\lr{5}\\\cline{1-3}
\lr{1}&\lr{2}&\lr{3}\\\cline{1-3}
\end{array}$},\emptyset,\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\cline{1-1}
\lr{9}\\\cline{1-2}
&\lr{8}\\\cline{2-3}
&\lr{6}&\lr{7}\\\cline{2-3}
\end{array}$}
}
sage: print tex_from_array_tuple([[[1,2,3],[4,5]],[],[[None,6,7],[None,8],[9]]], with_lines=Fals
{\def\lr#1{\multicolumn{1}{@{\hspace{.6ex}}c@{\hspace{.6ex}}}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\\
\lr{4}&\lr{5}\\
\lr{1}&\lr{2}&\lr{3}\\
\end{array}$},\emptyset,\raisebox{-.6ex}{$\begin{array}[t]{*{3}c}\\
\lr{9}\\
&\lr{8}\\
&\lr{6}&\lr{7}\\
\end{array}$}
}
```

sage.combinat.output.**tex_from_skew_array**(*array*, *with_lines=False*, *align='b'*)

This function creates latex code for a "skew composition" `array`. That is, for a two dimensional array in which each row can begin with an arbitrary number `None`'s and the remaining entries could, in principe, be anything but probably should be strings or integers of similar width. A row consisting completely of `None`'s is allowed.

INPUT:

- `array` – The array
- `with_lines` – (Default: `False`) If `True` lines are drawn, if `False` they are not
- `align` – (Default: `'b'`) Determines the alignment on the latex array environments

EXAMPLES:

```
sage: array=[[None, 2,3,4],[None,None],[5,6,7,8]]
sage: print sage.combinat.output.tex_from_skew_array(array)
\raisebox{-.6ex}{$\begin{array}[b]{*{4}c}\\
&\lr{2}&\lr{3}&\lr{4}\\
&\\
\lr{5}&\lr{6}&\lr{7}&\lr{8}\\
\end{array}$}
```

5.1.141 Parking Functions

INFORMALLY (reference [Beck]):

Imagine a one-way cul-de-sac with n parking spots. We will give the first parking spot the number 1, the next one number 2, etc., down to the last one, number n. Initially they are all free, but there are n cars approaching the street, and they would all like to park there. To make life interesting, every car has a parking preference, and we record the

preferences in a sequence; For example, if $n = 3$, the sequence $(2, 1, 1)$ means that the first car would like to park at spot number 2, the second car prefers parking spot number 1, and the last car would also like to part at number 1. The street is very narrow, so there is no way to back up. Now each car enters the street and approaches its preferred parking spot; if it is free, it parks there, and if not, it moves down the street to the first available spot. We call a sequence a parking function (of length n) if all cars end up finding a parking spot. For example, the sequence $(2, 1, 1)$ is a parking sequence (of length 3), whereas the sequence $(2, 3, 2)$ is not.

FORMALLY:

A parking function of size n is a sequence $(a_1, \ldots, a_n)$ of positive integers such that if $b_1 \leq b_2 \leq \cdots \leq b_n$ is the increasing rearrangement of $a_1, \ldots, a_n$, then $b_i \leq i$.

A parking function of size n is a pair (L, D) of two sequences L and D where L is a permutation and D is an area sequence of a Dyck path of size n such that $D[i] \geq 0$, $D[i+1] \leq D[i] + 1$ and if $D[i+1] = D[i] + 1$ then $L[i+1] > L[i]$.

The number of parking functions of size n is equal to the number of rooted forests on n vertices and is equal to $(n+1)^{n-1}$.

REFERENCES:

AUTHORS:

- used non-decreasing_parking_functions code by Florent Hivert (2009 - 04)
- Dorota Mazur (2012 - 09)

`sage.combinat.parking_functions.`**`ParkingFunction`**(*pf=None*, *labelling=None*, *area_sequence=None*, *labelled_dyck_word=None*)

Return the combinatorial class of Parking Functions.

A *parking function* of size n is a sequence $(a_1, \ldots, a_n)$ of positive integers such that if $b_1 \leq b_2 \leq \cdots \leq b_n$ is the increasing rearrangement of $a_1, \ldots, a_n$, then $b_i \leq i$.

A *parking function* of size n is a pair (L, D) of two sequences L and D where L is a permutation and D is an area sequence of a Dyck Path of size n such that $D[i] \geq 0$, $D[i+1] \leq D[i]+1$ and if $D[i+1] = D[i]+1$ then $L[i+1] > L[i]$.

The number of parking functions of size n is equal to the number of rooted forests on n vertices and is equal to $(n+1)^{n-1}$.

INPUT:

- `pf` – (default: None) a list whose increasing rearrangement satisfies $b_i \leq i$
- `labelling` – (default: None) a labelling of the Dyck path
- `area_sequence` – (default: None) an area sequence of a Dyck path
- `labelled_dyck_word` – (default: None) a Dyck word with 1's replaced by labelling

OUTPUT:

- A parking function

EXAMPLES:

```
sage: ParkingFunction([])
[]
sage: ParkingFunction([1])
[1]
sage: ParkingFunction([2])
Traceback (most recent call last):
...
```

```
ValueError: [2] is not a parking function.
sage: ParkingFunction([1,2])
[1, 2]
sage: ParkingFunction([1,1,2])
[1, 1, 2]
sage: ParkingFunction([1,4,1])
Traceback (most recent call last):
...
ValueError: [1, 4, 1] is not a parking function.
sage: ParkingFunction(labelling=[3,1,2], area_sequence=[0,0,1])
[2, 2, 1]
sage: ParkingFunction([2,2,1]).to_labelled_dyck_word()
[3, 0, 1, 2, 0, 0]
sage: ParkingFunction(labelled_dyck_word = [3,0,1,2,0,0])
[2, 2, 1]
sage: ParkingFunction(labelling=[3,1,2], area_sequence=[0,1,1])
Traceback (most recent call last):
...
ValueError: [3, 1, 2] is not a valid labeling of area sequence [0, 1, 1]
```

class sage.combinat.parking_functions.**ParkingFunction_class**(*lst*)

Bases: sage.combinat.combinat.CombinatorialObject

TESTS:

```
sage: ParkingFunction([1, 1, 2, 2, 5, 6])
[1, 1, 2, 2, 5, 6]
```

area()

Return the area of the labelled Dyck path corresponding to the parking function.

INPUT:

- self – parking function word

OUTPUT:

- the sum of squares under and over the main diagonal the Dyck Path, corresponding to the parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.area()
6

sage: ParkingFunction([3,1,1,4]).area()
1
sage: ParkingFunction([4,1,1,1]).area()
3
sage: ParkingFunction([2,1,4,1]).area()
2
```

cars_permutation()

Return the sequence of cars that take parking spots 1 through n and corresponding to the parking function.

For example, cars_permutation(PF) = [2, 4, 5, 6, 3, 1, 7] means that car 2 takes spots 1, car 4 takes spot 2, ..., car 1 takes spot 6 and car 7 takes spot 7.

INPUT:

- self – parking function word

OUTPUT:

- the permutation of cars corresponding to the parking function and which is the same size as parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.cars_permutation()
[2, 4, 5, 6, 3, 1, 7]

sage: ParkingFunction([3,1,1,4]).cars_permutation()
[2, 3, 1, 4]
sage: ParkingFunction([4,1,1,1]).cars_permutation()
[2, 3, 4, 1]
sage: ParkingFunction([2,1,4,1]).cars_permutation()
[2, 1, 4, 3]
```

characteristic_quasisymmetric_function(*q=None*, *R=Fraction Field of Multivariate Polynomial Ring in q, t over Rational Field*)

Return the characteristic quasisymmetric function of `self`.

The characteristic function of the Parking Function is the sum over all permutation labellings of the Dyck path $q^{dinv(PF)}F_{ides(PF)}$ where $ides(PF)$ (`ides_composition()`) is the descent composition of diagonal reading word of the parking function.

INPUT:

- `q` – (default: `q = R('q')`) a parameter for the generating function power
- `R` – (default: `R = QQ['q','t'].fraction_field()`) the base ring to do the calculations over

OUTPUT:

- an element of the quasisymmetric functions over the ring `R`

EXAMPLES:

```
sage: R = QQ['q','t'].fraction_field()
sage: (q,t) = R.gens()
sage: cqf = sum(t**PF.area()*PF.characteristic_quasisymmetric_function() for PF in ParkingFu
(q^3+q^2*t+q*t^2+t^3+q*t)*F[1, 1, 1] + (q^2+q*t+t^2+q+t)*F[1, 2] + (q^2+q*t+t^2+q+t)*F[2, 1]
sage: s = SymmetricFunctions(R).s()
sage: s(cqf.to_symmetric_function())
(q^3+q^2*t+q*t^2+t^3+q*t)*s[1, 1, 1] + (q^2+q*t+t^2+q+t)*s[2, 1] + s[3]
sage: s(cqf.to_symmetric_function()).nabla(power = -1)
s[1, 1, 1]

sage: p = ParkingFunction([3, 1, 2])
sage: p.characteristic_quasisymmetric_function()
q*F[2, 1]
sage: pf = ParkingFunction([1,2,7,2,1,2,3,2,1])
sage: pf.characteristic_quasisymmetric_function()
q^2*F[1, 1, 1, 2, 1, 3]
```

diagonal_composition()

Return the composition of the labelled Dyck path corresponding to the parking function.

For example, `touch_composition(PF) = [4, 3]` means that the first touch is four diagonal units from the starting point, and the second is three units further (see [GXZ] p. 2).

INPUT:

- self – parking function word

OUTPUT:

- the length between the corresponding touch points which of the labelled Dyck path that corresponds to the parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.touch_composition()
[4, 3]

sage: ParkingFunction([3,1,1,4]).touch_composition()
[2, 1, 1]
sage: ParkingFunction([4,1,1,1]).touch_composition()
[3, 1]
sage: ParkingFunction([2,1,4,1]).touch_composition()
[3, 1]
```

diagonal_reading_word()

Return a diagonal word of the labelled Dyck path corresponding to parking function (see [Hag08] p. 75).

INPUT:

- self – parking function word

OUTPUT:

- returns a word, read diagonally from NE to SW of the pretty print of the labelled Dyck path that corresponds to self and the same size as self

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.diagonal_reading_word()
[5, 1, 7, 4, 6, 3, 2]

sage: ParkingFunction([1, 1, 1]).diagonal_reading_word()
[3, 2, 1]
sage: ParkingFunction([1, 2, 3]).diagonal_reading_word()
[3, 2, 1]
sage: ParkingFunction([1, 1, 3, 4]).diagonal_reading_word()
[2, 4, 3, 1]

sage: ParkingFunction([1, 1, 1]).diagonal_word()
[3, 2, 1]
sage: ParkingFunction([1, 2, 3]).diagonal_word()
[3, 2, 1]
sage: ParkingFunction([1, 4, 3, 1]).diagonal_word()
[4, 2, 3, 1]
```

diagonal_word()

Return a diagonal word of the labelled Dyck path corresponding to parking function (see [Hag08] p. 75).

INPUT:

- self – parking function word

OUTPUT:

- returns a word, read diagonally from NE to SW of the pretty print of the labelled Dyck path that corresponds to self and the same size as self

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.diagonal_reading_word()
[5, 1, 7, 4, 6, 3, 2]

sage: ParkingFunction([1, 1, 1]).diagonal_reading_word()
[3, 2, 1]
sage: ParkingFunction([1, 2, 3]).diagonal_reading_word()
[3, 2, 1]
sage: ParkingFunction([1, 1, 3, 4]).diagonal_reading_word()
[2, 4, 3, 1]

sage: ParkingFunction([1, 1, 1]).diagonal_word()
[3, 2, 1]
sage: ParkingFunction([1, 2, 3]).diagonal_word()
[3, 2, 1]
sage: ParkingFunction([1, 4, 3, 1]).diagonal_word()
[4, 2, 3, 1]
```

dinv()

Return the number of inversions of a labelled Dyck path corresponding to the parking function (see [Hag08] p. 74).

Same as the cardinality of `dinversion_pairs()`.

INPUT:

- `self` – parking function word

OUTPUT:

- the number of dinversion pairs

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.dinv()
6

sage: ParkingFunction([3,1,1,4]).dinv()
3
sage: ParkingFunction([4,1,1,1]).dinv()
1
sage: ParkingFunction([2,1,4,1]).dinv()
2
```

dinversion_pairs()

Return the descent inversion pairs of a labelled Dyck path corresponding to the parking function.

INPUT:

- `self` – parking function word

OUTPUT:

- the primary and secondary diversion pairs

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.dinversion_pairs()
[(0, 4), (1, 5), (2, 5), (1, 4), (2, 4), (3, 6)]
```

```
sage: ParkingFunction([3,1,1,4]).dinversion_pairs()
[(0, 3), (2, 3), (1, 2)]
sage: ParkingFunction([4,1,1,1]).dinversion_pairs()
[(1, 3)]
sage: ParkingFunction([2,1,4,1]).dinversion_pairs()
[(0, 3), (1, 3)]
```

ides()

Return the descents() sequence of the inverse of the diagonal_reading_word() of self.

For example, ides(PF) = [1, 2, 3, 5] means that descents are at the 2nd, 3rd, 4th and 6th positions in the inverse of the diagonal_reading_word() of the parking function (see [GXZ] p. 2).

INPUT:

- self – parking function word

OUTPUT:

- the descents sequence of the inverse of the diagonal_reading_word() of the parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.ides()
[1, 2, 3, 5]

sage: ParkingFunction([3,1,1,4]).ides()
[1]
sage: ParkingFunction([4,1,1,1]).ides()
[1, 2]
sage: ParkingFunction([4,3,1,1]).ides()
[2]
```

ides_composition()

Return the descents_composition() of the inverse of the diagonal_reading_word() of corresponding parking function.

For example, ides_composition(PF) = [4, 2, 1] means that the descents of the inverse of the permutation diagonal_reading_word() of the parking function with word PF are at the 4th and 6th positions.

INPUT:

- self – parking function word

OUTPUT:

- the descents composition of the inverse of the diagonal_reading_word() of the parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.ides_composition()
[2, 1, 1, 2, 1]

sage: ParkingFunction([3,1,1,4]).ides_composition()
[2, 2]
sage: ParkingFunction([4,1,1,1]).ides_composition()
[2, 1, 1]
sage: ParkingFunction([4,3,1,1]).ides_composition()
[3, 1]
```

jump()

Return the sum of the differences between the parked and preferred parking spots.

See [Shin] p. 18.

INPUT:

- `self` – a parking function word

OUTPUT:

- the sum of the differences between the parked and preferred parking spots

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.jump()
6

sage: ParkingFunction([3,1,1,4]).jump()
1
sage: ParkingFunction([4,1,1,1]).jump()
3
sage: ParkingFunction([2,1,4,1]).jump()
2
```

jump_list()

Return the displacements of cars that corresponds to the parking function.

For example, `jump_list(PF) = [0, 0, 0, 0, 1, 3, 2]` means that car 1 through 4 parked in their preferred spots, car 5 had to park one spot farther (jumped or was displaced by one spot), car 6 had to jump 3 spots, and car 7 had to jump two spots.

INPUT:

- `self` – parking function word

OUTPUT:

- the displacements sequence of parked cars which corresponds to the parking function and which is the same size as parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.jump_list()
[0, 0, 0, 0, 1, 3, 2]

sage: ParkingFunction([3,1,1,4]).jump_list()
[0, 0, 1, 0]
sage: ParkingFunction([4,1,1,1]).jump_list()
[0, 0, 1, 2]
sage: ParkingFunction([2,1,4,1]).jump_list()
[0, 0, 0, 2]
```

luck()

Return the number of cars that parked in their preferred parking spots (see [Shin] p. 33).

INPUT:

- `self` – parking function word

OUTPUT:

•the number of cars that parked in their preferred parking spots

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.luck()
4

sage: ParkingFunction([3,1,1,4]).luck()
3
sage: ParkingFunction([4,1,1,1]).luck()
2
sage: ParkingFunction([2,1,4,1]).luck()
3
```

lucky_cars()

Return the cars that can park in their preferred spots. For example, `lucky_cars(PF) = [1, 2, 7]` means that cars 1, 2 and 7 parked in their preferred spots and all the other cars did not.

INPUT:

•`self` – parking function word

OUTPUT:

•the cars that can park in their preferred spots

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.lucky_cars()
[1, 2, 3, 4]

sage: ParkingFunction([3,1,1,4]).lucky_cars()
[1, 2, 4]
sage: ParkingFunction([4,1,1,1]).lucky_cars()
[1, 2]
sage: ParkingFunction([2,1,4,1]).lucky_cars()
[1, 2, 3]
```

parking_permutation()

Return the sequence of parking spots that are taken by cars 1 through n and corresponding to the parking function.

For example, `parking_permutation(PF) = [6, 1, 5, 2, 3, 4, 7]` means that spot 6 is taken by car 1, spot 1 by car 2, spot 5 by car 3, spot 2 is taken by car 4, spot 3 is taken by car 5, spot 4 is taken by car 6 and spot 7 is taken by car 7.

INPUT:

•`self` – parking function word

OUTPUT:

•the permutation of parking spots that corresponds to the parking function and which is the same size as parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.parking_permutation()
[6, 1, 5, 2, 3, 4, 7]
```

```
sage: ParkingFunction([3,1,1,4]).parking_permutation()
[3, 1, 2, 4]
sage: ParkingFunction([4,1,1,1]).parking_permutation()
[4, 1, 2, 3]
sage: ParkingFunction([2,1,4,1]).parking_permutation()
[2, 1, 4, 3]
```

pretty_print (*underpath=True*)

Displays a parking function as a lattice path consisting of a Dyck path and a labelling with the labels displayed along the edges of the Dyck path.

INPUT:

- `underpath` – if the length of the parking function is less than or equal to 9 then display the labels under the path if `underpath` is True otherwise display them to the right of the path (default: `True`)

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.pretty_print()
          ___
         _|1x
        |7x  .
   _____|3 . .
  |5x x  . . .
 _|4x  . . . .
|6x  . . . . .
|2 . . . . . .

sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.pretty_print(underpath = false)
          ___
         _| x  1
        | x  . 7
   _____|  . . 3
  | x x  . . . 5
 _| x  . . . . 4
| x  . . . . . 6
|  . . . . . . 2

sage: ParkingFunction([3, 1, 1, 4]).pretty_print()
       _
     _|4
 ___|1 .
|3x  . .
|2 . . .

sage: ParkingFunction([1,1,1]).pretty_print()
 _____
|3x x
|2x  .
|1 . .

sage: ParkingFunction([4,1,1,1]).pretty_print()
       _
 _____|1
|4x x  .
|3x  . .
|2 . . .
```

```
sage: ParkingFunction([2,1,4,1]).pretty_print()
       _
   ___|3
 _|1x  .
|4x  . .
|2 . . .

sage: ParkingFunction([2,1,4,1]).pretty_print(underpath = false)
       _
   ___|  3
 _| x  . 1
| x  . . 4
|  . . . 2

sage: pf = ParkingFunction([1,2,3,7,3,2,1,2,3,2,1])
sage: pf.pretty_print()
              _________
     ________| x x x x   4
    | x x x x x x x x  .  9
    | x x x x x x x  . .  5
   _| x x x x x  . . .  3
  | x x x x x  . . . . 10
  | x x x x  . . . . .  8
  | x x x  . . . . . .  6
 _| x x  . . . . . . .  2
| x x  . . . . . . . . 11
| x  . . . . . . . . .  7
|  . . . . . . . . . .  1
```

primary_dinversion_pairs()

Return the primary descent inversion pairs of a labelled Dyck path corresponding to the parking function.

INPUT:

- self – parking function word

OUTPUT:

- the pairs (i, j) such that $i < j$, and i^{th} area = j^{th} area, and i^{th} label < j^{th} label

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.primary_dinversion_pairs()
[(0, 4), (1, 5), (2, 5)]

sage: ParkingFunction([3,1,1,4]).primary_dinversion_pairs()
[(0, 3), (2, 3)]
sage: ParkingFunction([4,1,1,1]).primary_dinversion_pairs()
[]
sage: ParkingFunction([2,1,4,1]).primary_dinversion_pairs()
[(0, 3)]
```

secondary_dinversion_pairs()

Return the secondary descent inversion pairs of a labelled Dyck path corresponding to the parking function.

INPUT:

- self – parking function word

OUTPUT:

•the pairs (i, j) such that $i < j$, and i^{th} area = j^{th} area +1, and i^{th} label > j^{th} label

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.secondary_dinversion_pairs()
[(1, 4), (2, 4), (3, 6)]

sage: ParkingFunction([3,1,1,4]).secondary_dinversion_pairs()
[(1, 2)]
sage: ParkingFunction([4,1,1,1]).secondary_dinversion_pairs()
[(1, 3)]
sage: ParkingFunction([2,1,4,1]).secondary_dinversion_pairs()
[(1, 3)]
```

to_NonDecreasingParkingFunction()

Return the non-decreasing parking function which underlies the parking function.

INPUT:

•`self` – parking function word

OUTPUT:

•a sorted parking function

See also:

`NonDecreasingParkingFunction()`

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.to_NonDecreasingParkingFunction()
[1, 1, 2, 2, 5, 5, 6]

sage: ParkingFunction([3,1,1,4]).to_NonDecreasingParkingFunction()
[1, 1, 3, 4]
sage: ParkingFunction([4,1,1,1]).to_NonDecreasingParkingFunction()
[1, 1, 1, 4]
sage: ParkingFunction([2,1,4,1]).to_NonDecreasingParkingFunction()
[1, 1, 2, 4]
sage: ParkingFunction([4,1,2,1]).to_NonDecreasingParkingFunction()
[1, 1, 2, 4]
```

to_area_sequence()

Return the area sequence of the support Dyck path of the parking function.

INPUT:

•`self` – parking function word

OUTPUT:

•the area sequence of the Dyck path

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.to_area_sequence()
[0, 1, 1, 2, 0, 1, 1]
```

```
sage: ParkingFunction([3,1,1,4]).to_area_sequence()
[0, 1, 0, 0]
sage: ParkingFunction([4,1,1,1]).to_area_sequence()
[0, 1, 2, 0]
sage: ParkingFunction([2,1,4,1]).to_area_sequence()
[0, 1, 1, 0]
```

to_dyck_word()

Return the support Dyck word of the parking function.

INPUT:

- self – parking function word

OUTPUT:

- the Dyck word of the corresponding parking function

See also:

DyckWord()

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.to_dyck_word()
[1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0]

sage: ParkingFunction([3,1,1,4]).to_dyck_word()
[1, 1, 0, 0, 1, 0, 1, 0]
sage: ParkingFunction([4,1,1,1]).to_dyck_word()
[1, 1, 1, 0, 0, 0, 1, 0]
sage: ParkingFunction([2,1,4,1]).to_dyck_word()
[1, 1, 0, 1, 0, 0, 1, 0]
```

to_labelled_dyck_word()

Return the labelled Dyck word corresponding to the parking function.

This is a representation of the parking function as a list where the entries of 1 in the Dyck word are replaced with the corresponding label.

INPUT:

- self – parking function word

OUTPUT:

- the labelled Dyck word of the corresponding parking function which is twice the size of parking function word

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.to_labelled_dyck_word()
[2, 6, 0, 4, 5, 0, 0, 0, 3, 7, 0, 1, 0, 0]

sage: ParkingFunction([3,1,1,4]).to_labelled_dyck_word()
[2, 3, 0, 0, 1, 0, 4, 0]
sage: ParkingFunction([4,1,1,1]).to_labelled_dyck_word()
[2, 3, 4, 0, 0, 0, 1, 0]
sage: ParkingFunction([2,1,4,1]).to_labelled_dyck_word()
[2, 4, 0, 1, 0, 0, 3, 0]
```

to_labelling_area_sequence_pair()

Return a pair consisting of a labelling and an area sequence of a Dyck path which corresponds to the given parking function.

INPUT:

- `self` – the parking function word

OUTPUT:

- returns a pair `(L, D)` where `L` is a labelling and `D` is the area sequence of the underlying Dyck path

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.to_labelling_area_sequence_pair()
([2, 6, 4, 5, 3, 7, 1], [0, 1, 1, 2, 0, 1, 1])

sage: ParkingFunction([1, 1, 1]).to_labelling_area_sequence_pair()
([1, 2, 3], [0, 1, 2])
sage: ParkingFunction([1, 2, 3]).to_labelling_area_sequence_pair()
([1, 2, 3], [0, 0, 0])
sage: ParkingFunction([1, 1, 2]).to_labelling_area_sequence_pair()
([1, 2, 3], [0, 1, 1])
sage: ParkingFunction([1, 1, 3, 1]).to_labelling_area_sequence_pair()
([1, 2, 4, 3], [0, 1, 2, 1])
```

to_labelling_dyck_word_pair()

Return the pair `(L, D)` where `L` is a labelling and `D` is the Dyck word of the parking function.

INPUT:

- `self` – parking function word

OUTPUT:

- the pair `(L, D)`, where `L` is the labelling and `D` is the Dyck word of the parking function

See also:

`DyckWord()`

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.to_labelling_dyck_word_pair()
([2, 6, 4, 5, 3, 7, 1], [1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0])

sage: ParkingFunction([3,1,1,4]).to_labelling_dyck_word_pair()
([2, 3, 1, 4], [1, 1, 0, 0, 1, 0, 1, 0])
sage: ParkingFunction([4,1,1,1]).to_labelling_dyck_word_pair()
([2, 3, 4, 1], [1, 1, 1, 0, 0, 0, 1, 0])
sage: ParkingFunction([2,1,4,1]).to_labelling_dyck_word_pair()
([2, 4, 1, 3], [1, 1, 0, 1, 0, 0, 1, 0])
```

to_labelling_permutation()

Return the labelling of the support Dyck path of the parking function.

INPUT:

- `self` – parking function word

OUTPUT:

- the labelling of the Dyck path

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.to_labelling_permutation()
[2, 6, 4, 5, 3, 7, 1]

sage: ParkingFunction([3,1,1,4]).to_labelling_permutation()
[2, 3, 1, 4]
sage: ParkingFunction([4,1,1,1]).to_labelling_permutation()
[2, 3, 4, 1]
sage: ParkingFunction([2,1,4,1]).to_labelling_permutation()
[2, 4, 1, 3]
```

touch_composition()

Return the composition of the labelled Dyck path corresponding to the parking function.

For example, `touch_composition(PF) = [4, 3]` means that the first touch is four diagonal units from the starting point, and the second is three units further (see [GXZ] p. 2).

INPUT:

- `self` – parking function word

OUTPUT:

- the length between the corresponding touch points which of the labelled Dyck path that corresponds to the parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.touch_composition()
[4, 3]

sage: ParkingFunction([3,1,1,4]).touch_composition()
[2, 1, 1]
sage: ParkingFunction([4,1,1,1]).touch_composition()
[3, 1]
sage: ParkingFunction([2,1,4,1]).touch_composition()
[3, 1]
```

touch_points()

Return the sequence of touch points which corresponds to the labelled Dyck path after initial step.

For example, `touch_points(PF) = [4, 7]` means that after the initial step, the path touches the main diagonal at points $(4, 4)$ and $(7, 7)$.

INPUT:

- `self` – parking function word

OUTPUT:

- the sequence of touch points after the initial step of the labelled Dyck path that corresponds to the parking function

EXAMPLES:

```
sage: PF = ParkingFunction([6, 1, 5, 2, 2, 1, 5])
sage: PF.touch_points()
[4, 7]

sage: ParkingFunction([3,1,1,4]).touch_points()
[2, 3, 4]
```

```
sage: ParkingFunction([4,1,1,1]).touch_points()
[3, 4]
sage: ParkingFunction([2,1,4,1]).touch_points()
[3, 4]
```

sage.combinat.parking_functions.**ParkingFunctions**(*n=None*)

Return the combinatorial class of Parking Functions.

A *parking function* of size n is a sequence $(a_1, \ldots, a_n)$ of positive integers such that if $b_1 \leq b_2 \leq \cdots \leq b_n$ is the increasing rearrangement of $a_1, \ldots, a_n$, then $b_i \leq i$.

A *parking function* of size n is a pair (L, D) of two sequences L and D where L is a permutation and D is an area sequence of a Dyck Path of size n such that $D[i] \geq 0$, $D[i+1] \leq D[i]+1$ and if $D[i+1] = D[i]+1$ then $L[i+1] > L[i]$.

The number of parking functions of size n is equal to the number of rooted forests on n vertices and is equal to $(n+1)^{n-1}$.

EXAMPLES:

Here are all parking functions of size 3:

```
sage: from sage.combinat.parking_functions import ParkingFunctions
sage: ParkingFunctions(3).list()
[[1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [1, 1, 3], [1, 3, 1], [3, 1, 1],
 [1, 2, 2], [2, 1, 2], [2, 2, 1], [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1],
 [3, 1, 2], [3, 2, 1]]
```

If no size is specified, then ParkingFunctions returns the combinatorial class of all parking functions.

```
sage: PF = ParkingFunctions(); PF
Parking functions
sage: [] in PF
True
sage: [1] in PF
True
sage: [2] in PF
False
sage: [1,3,1] in PF
True
sage: [1,4,1] in PF
False
```

If the size n is specified, then ParkingFunctions returns the combinatorial class of all parking functions of size n.

```
sage: PF = ParkingFunctions(0)
sage: PF.list()
[[]]
sage: PF = ParkingFunctions(1)
sage: PF.list()
[[1]]
sage: PF = ParkingFunctions(3)
sage: PF.list()
[[1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [1, 1, 3],
 [1, 3, 1], [3, 1, 1], [1, 2, 2], [2, 1, 2], [2, 2, 1],
 [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

sage: PF3 = ParkingFunctions(3); PF3
Parking functions of size 3
sage: [] in PF3
```

```
False
sage: [1] in PF3
False
sage: [1,3,1] in PF3
True
sage: [1,4,1] in PF3
False
```

TESTS:

```
sage: PF = ParkingFunctions(5)
sage: len(PF.list()) == PF.cardinality()
True
```

class sage.combinat.parking_functions.**ParkingFunctions_all**

Bases: sage.combinat.combinat.InfiniteAbstractCombinatorialClass

TESTS:

```
sage: from sage.combinat.parking_functions import ParkingFunctions
sage: DW = ParkingFunctions()
sage: DW == loads(dumps(DW))
True
```

class sage.combinat.parking_functions.**ParkingFunctions_n**(*n*)

Bases: sage.combinat.combinat.CombinatorialClass

The combinatorial class of parking functions of size n.

A *parking function* of size n is a sequence $(a_1, \ldots, a_n)$ of positive integers such that if $b_1 \leq b_2 \leq \cdots \leq b_n$ is the increasing rearrangement of $a_1, \ldots, a_n$, then $b_i \leq i$.

A *parking function* of size n is a pair (L, D) of two sequences L and D where L is a permutation and D is an area sequence of a Dyck Path of size n such that $D[i] \geq 0$, $D[i+1] \leq D[i]+1$ and if $D[i+1] = D[i]+1$ then $L[i+1] > L[i]$.

The number of parking functions of size n is equal to the number of rooted forests on n vertices and is equal to $(n+1)^{n-1}$.

EXAMPLES:

```
sage: PF = ParkingFunctions(3)
sage: PF.list()
[[1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [1, 1, 3],
[1, 3, 1], [3, 1, 1], [1, 2, 2], [2, 1, 2], [2, 2, 1],
[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

sage: [ParkingFunctions(i).cardinality() for i in range(6)]
[1, 1, 3, 16, 125, 1296]
```

Warning: The precise order in which the parking function are generated or listed is not fixed, and may change in the future.

cardinality()

Return the number of parking functions of size n.

The cardinality is equal to $(n+1)^{n-1}$.

EXAMPLES:

```
sage: [ParkingFunctions(i).cardinality() for i in range(6)]
[1, 1, 3, 16, 125, 1296]
```

random_element()

Return a random parking function of size n.

The algorithm uses a circular parking space with $n+1$ spots. Then all n cars can park and there remains one empty spot. Spots are then renumbered so that the empty spot is 0.

The probability distribution is uniform on the set of $(n+1)^{n-1}$ parking functions of size n.

EXAMPLES:

```
sage: pf = ParkingFunctions(8)
sage: a = pf.random_element(); a  # random
[5, 7, 2, 4, 2, 5, 1, 3]
sage: a in pf
True
```

sage.combinat.parking_functions.**from_labelled_dyck_word**(*LDW*)

Return the parking function corresponding to the labelled Dyck word.

INPUT:

- LDW – labelled Dyck word

OUTPUT:

- the parking function corresponding to the labelled Dyck word that is half the size of LDW

EXAMPLES:

```
sage: from sage.combinat.parking_functions import from_labelled_dyck_word
sage: LDW = [2, 6, 0, 4, 5, 0, 0, 0, 3, 7, 0, 1, 0, 0]
sage: from_labelled_dyck_word(LDW)
[6, 1, 5, 2, 2, 1, 5]

sage: from_labelled_dyck_word([2, 3, 0, 0, 1, 0, 4, 0])
[3, 1, 1, 4]
sage: from_labelled_dyck_word([2, 3, 4, 0, 0, 0, 1, 0])
[4, 1, 1, 1]
sage: from_labelled_dyck_word([2, 4, 0, 1, 0, 0, 3, 0])
[2, 1, 4, 1]
```

sage.combinat.parking_functions.**from_labelling_and_area_sequence**(*L*, *D*)

Return the parking function corresponding to the labelling area sequence pair.

INPUT:

- L – a labelling permutation
- D – an area sequence for a Dyck word

OUTPUT:

- the parking function corresponding the labelling permutation L and D an area sequence of the corresponding Dyck path

EXAMPLES:

```
sage: from sage.combinat.parking_functions import from_labelling_and_area_sequence
sage: from_labelling_and_area_sequence([2, 6, 4, 5, 3, 7, 1], [0, 1, 1, 2, 0, 1, 1])
[6, 1, 5, 2, 2, 1, 5]
```

```
sage: from_labelling_and_area_sequence([1, 2, 3], [0, 1, 2])
[1, 1, 1]
sage: from_labelling_and_area_sequence([1, 2, 3], [0, 0, 0])
[1, 2, 3]
sage: from_labelling_and_area_sequence([1, 2, 3], [0, 1, 1])
[1, 1, 2]
sage: from_labelling_and_area_sequence([1, 2, 4, 3], [0, 1, 2, 1])
[1, 1, 3, 1]
```

sage.combinat.parking_functions.**is_a**(*x*, *n=None*)

Check whether a list is a parking function.

If a size n is specified, checks if a list is a parking function of size n.

TESTS:

```
sage: from sage.combinat.parking_functions import is_a
sage: is_a([1,1,2])
True
sage: is_a([1,2,1])
True
sage: is_a([1,1,4])
False
sage: is_a([3,1,1], 3)
True
```

5.1.142 Integer partitions

A partition p of a nonnegative integer n is a non-increasing list of positive integers (the *parts* of the partition) with total sum n.

A partition can be depicted by a diagram made of rows of cells, where the number of cells in the i^{th} row starting from the top is the i^{th} part of the partition.

The coordinate system related to a partition applies from the top to the bottom and from left to right. So, the corners of the partition $[5, 3, 1]$ are $[[0, 4], [1, 2], [2, 0]]$.

For display options, see `Partitions.global_options`.

Note:

- Boxes is a synonym for cells. All methods will use 'cell' and 'cells' instead of 'box' and 'boxes'.
- Partitions are 0 based with coordinates in the form of (row-index, column-index).
- If given coordinates of the form `(r, c)`, then use Python's *-operator.
- Throughout this documentation, for a partition λ we will denote its conjugate partition by λ'. For more on conjugate partitions, see `Partition.conjugate()`.
- The comparisons on partitions use lexicographic order.

Note: We use the convention that lexicographic ordering is read from left-to-right. That is to say $[1, 3, 7]$ is smaller than $[2, 3, 4]$.

AUTHORS:

- Mike Hansen (2007): initial version

- Dan Drake (2009-03-28): deprecate RestrictedPartitions and implement Partitions_parts_in
- Travis Scrimshaw (2012-01-12): Implemented latex function to Partition_class
- Travis Scrimshaw (2012-05-09): Fixed Partitions(-1).list() infinite recursion loop by saying Partitions_n is the empty set.
- Travis Scrimshaw (2012-05-11): Fixed bug in inner where if the length was longer than the length of the inner partition, it would include 0's.
- Andrew Mathas (2012-06-01): Removed depreciated functions and added compatibility with the PartitionTuple classes. See trac ticket #13072
- Travis Scrimshaw (2012-10-12): Added global options. Made `Partition_class` to the element `Partition`. `Partitions*` are now all in the category framework except `PartitionsRestricted` (which will eventually be removed). Cleaned up documentation.

EXAMPLES:

There are 5 partitions of the integer 4:

```
sage: Partitions(4).cardinality()
5
sage: Partitions(4).list()
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

We can use the method `.first()` to get the 'first' partition of a number:

```
sage: Partitions(4).first()
[4]
```

Using the method `.next()`, we can calculate the 'next' partition. When we are at the last partition, `None` will be returned:

```
sage: Partitions(4).next([4])
[3, 1]
sage: Partitions(4).next([1,1,1,1]) is None
True
```

We can use `iter` to get an object which iterates over the partitions one by one to save memory. Note that when we do something like `for part in Partitions(4)` this iterator is used in the background:

```
sage: g = iter(Partitions(4))
sage: next(g)
[4]
sage: next(g)
[3, 1]
sage: next(g)
[2, 2]
sage: for p in Partitions(4): print p
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
```

We can add constraints to the type of partitions we want. For example, to get all of the partitions of 4 of length 2, we'd do the following:

```
sage: Partitions(4, length=2).list()
[[3, 1], [2, 2]]
```

Here is the list of partitions of length at least 2 and the list of ones with length at most 2:

```
sage: Partitions(4, min_length=2).list()
[[3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
sage: Partitions(4, max_length=2).list()
[[4], [3, 1], [2, 2]]
```

The options `min_part` and `max_part` can be used to set constraints on the sizes of all parts. Using `max_part`, we can select partitions having only 'small' entries. The following is the list of the partitions of 4 with parts at most 2:

```
sage: Partitions(4, max_part=2).list()
[[2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

The `min_part` options is complementary to `max_part` and selects partitions having only 'large' parts. Here is the list of all partitions of 4 with each part at least 2:

```
sage: Partitions(4, min_part=2).list()
[[4], [2, 2]]
```

The options `inner` and `outer` can be used to set part-by-part constraints. This is the list of partitions of 4 with `[3, 1, 1]` as an outer bound (that is, partitions of 4 contained in the partition `[3, 1, 1]`):

```
sage: Partitions(4, outer=[3,1,1]).list()
[[3, 1], [2, 1, 1]]
```

`outer` sets `max_length` to the length of its argument. Moreover, the parts of `outer` may be infinite to clear constraints on specific parts. Here is the list of the partitions of 4 of length at most 3 such that the second and third part are 1 when they exist:

```
sage: Partitions(4, outer=[oo,1,1]).list()
[[4], [3, 1], [2, 1, 1]]
```

Finally, here are the partitions of 4 with `[1,1,1]` as an inner bound (i. e., the partitions of 4 containing the partition `[1,1,1]`). Note that `inner` sets `min_length` to the length of its argument:

```
sage: Partitions(4, inner=[1,1,1]).list()
[[2, 1, 1], [1, 1, 1, 1]]
```

The options `min_slope` and `max_slope` can be used to set constraints on the slope, that is on the difference `p[i+1]-p[i]` of two consecutive parts. Here is the list of the strictly decreasing partitions of 4:

```
sage: Partitions(4, max_slope=-1).list()
[[4], [3, 1]]
```

The constraints can be combined together in all reasonable ways. Here are all the partitions of 11 of length between 2 and 4 such that the difference between two consecutive parts is between -3 and -1:

```
sage: Partitions(11,min_slope=-3,max_slope=-1,min_length=2,max_length=4).list()
[[7, 4], [6, 5], [6, 4, 1], [6, 3, 2], [5, 4, 2], [5, 3, 2, 1]]
```

Partition objects can also be created individually with `Partition`:

```
sage: Partition([2,1])
[2, 1]
```

Once we have a partition object, then there are a variety of methods that we can use. For example, we can get the conjugate of a partition. Geometrically, the conjugate of a partition is the reflection of that partition through its main diagonal. Of course, this operation is an involution:

```
sage: Partition([4,1]).conjugate()
[2, 1, 1, 1]
sage: Partition([4,1]).conjugate().conjugate()
[4, 1]
```

If we create a partition with extra zeros at the end, they will be dropped:

```
sage: Partition([4,1,0,0])
[4, 1]
sage: Partition([0])
[]
sage: Partition([0,0])
[]
```

The idea of a partition being followed by infinitely many parts of size 0 is consistent with the `get_part` method:

```
sage: p = Partition([5, 2])
sage: p.get_part(0)
5
sage: p.get_part(10)
0
```

We can go back and forth between the standard and the exponential notations of a partition. The exponential notation can be padded with extra zeros:

```
sage: Partition([6,4,4,2,1]).to_exp()
[1, 1, 0, 2, 0, 1]
sage: Partition(exp=[1,1,0,2,0,1])
[6, 4, 4, 2, 1]
sage: Partition([6,4,4,2,1]).to_exp(5)
[1, 1, 0, 2, 0, 1]
sage: Partition([6,4,4,2,1]).to_exp(7)
[1, 1, 0, 2, 0, 1, 0]
sage: Partition([6,4,4,2,1]).to_exp(10)
[1, 1, 0, 2, 0, 1, 0, 0, 0, 0]
```

We can get the (zero-based!) coordinates of the corners of a partition:

```
sage: Partition([4,3,1]).corners()
[(0, 3), (1, 2), (2, 0)]
```

We can compute the core and quotient of a partition and build the partition back up from them:

```
sage: Partition([6,3,2,2]).core(3)
[2, 1, 1]
sage: Partition([7,7,5,3,3,3,1]).quotient(3)
([2], [1], [2, 2, 2])
sage: p = Partition([11,5,5,3,2,2,2])
sage: p.core(3)
[]
sage: p.quotient(3)
([2, 1], [4], [1, 1, 1])
sage: Partition(core=[],quotient=([2, 1], [4], [1, 1, 1]))
[11, 5, 5, 3, 2, 2, 2]
```

We can compute the $0-1$ sequence and go back and forth:

```
sage: Partitions().from_zero_one([1, 1, 1, 1, 0, 1, 0])
[5, 4]
sage: all(Partitions().from_zero_one(mu.zero_one_sequence())
....:     == mu for n in range(5) for mu in Partitions(n))
True
```

We can compute the Frobenius coordinates and go back and forth:

```
sage: Partition([7,3,1]).frobenius_coordinates()
([6, 1], [2, 0])
sage: Partition(frobenius_coordinates=([6,1],[2,0]))
[7, 3, 1]
sage: all(mu == Partition(frobenius_coordinates=mu.frobenius_coordinates())
....:     for n in range(30) for mu in Partitions(n))
True
```

We use the lexicographic ordering:

```
sage: pl = Partition([4,1,1])
sage: ql = Partitions()([3,3])
sage: pl > ql
True
sage: PL = Partitions()
sage: pl = PL([4,1,1])
sage: ql = PL([3,3])
sage: pl > ql
True
```

class sage.combinat.partition.**OrderedPartitions**(n, k)

Bases: sage.combinat.partition.Partitions

The class of ordered partitions of n. If k is specified, then this contains only the ordered partitions of length k.

An *ordered partition* of a nonnegative integer n means a list of positive integers whose sum is n. This is the same as a composition of n.

Note: It is recommended that you use `Compositions()` instead as `OrderedPartitions()` wraps GAP.

EXAMPLES:

```
sage: OrderedPartitions(3)
Ordered partitions of 3
sage: OrderedPartitions(3).list()
[[3], [2, 1], [1, 2], [1, 1, 1]]
sage: OrderedPartitions(3,2)
Ordered partitions of 3 of length 2
sage: OrderedPartitions(3,2).list()
[[2, 1], [1, 2]]

sage: OrderedPartitions(10,k=2).list()
[[9, 1], [8, 2], [7, 3], [6, 4], [5, 5], [4, 6], [3, 7], [2, 8], [1, 9]]
sage: OrderedPartitions(4).list()
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: OrderedPartitions(3).cardinality()
4
sage: OrderedPartitions(3,2).cardinality()
2
sage: OrderedPartitions(10,2).cardinality()
9
sage: OrderedPartitions(15).cardinality()
16384
```

list()

Return a list of partitions in `self`.

EXAMPLES:

```
sage: OrderedPartitions(3).list()
[[3], [2, 1], [1, 2], [1, 1, 1]]
sage: OrderedPartitions(3,2).list()
[[2, 1], [1, 2]]
```

class sage.combinat.partition.**Partition**(*parent*, *mu*)

Bases: sage.combinat.combinat.CombinatorialElement

A partition p of a nonnegative integer n is a non-increasing list of positive integers (the *parts* of the partition) with total sum n.

A partition is often represented as a diagram consisting of **cells**, or **boxes**, placed in rows on top of each other such that the number of cells in the i^{th} row, reading from top to bottom, is the i^{th} part of the partition. The rows are left-justified (and become shorter and shorter the farther down one goes). This diagram is called the **Young diagram** of the partition, or more precisely its Young diagram in English notation. (French and Russian notations are variations on this representation.)

The coordinate system related to a partition applies from the top to the bottom and from left to right. So, the corners of the partition `[5, 3, 1]` are `[[0,4], [1,2], [2,0]]`.

For display options, see `Partitions.global_options()`.

Note: Partitions are 0 based with coordinates in the form of (row-index, column-index). For example consider the partition `mu=Partition([4,3,2,2])`, the first part is `mu[0]` (which is 4), the second is `mu[1]`, and so on, and the upper-left cell in English convention is `(0, 0)`.

A partition can be specified in one of the following ways:

- a list (the default)
- using exponential notation
- by Frobenius coordinates
- specifying its $0-1$ sequence
- specifying the core and the quotient

See the examples below.

EXAMPLES:

Creating partitions though parents:

```
sage: mu = Partitions(8)([3,2,1,1,1]); mu
[3, 2, 1, 1, 1]
sage: nu = Partition([3,2,1,1,1]); nu
[3, 2, 1, 1, 1]
```

```
sage: mu == nu
True
sage: mu is nu
False
sage: mu in Partitions()
True
sage: mu.parent()
Partitions of the integer 8
sage: mu.size()
8
sage: mu.category()
Category of elements of Partitions of the integer 8
sage: nu.parent()
Partitions
sage: nu.category()
Category of elements of Partitions
sage: mu[0]
3
sage: mu[1]
2
sage: mu[2]
1
sage: mu.pp()
***
**
*
*
*
sage: mu.removable_cells()
[(0, 2), (1, 1), (4, 0)]
sage: mu.down_list()
[[2, 2, 1, 1, 1], [3, 1, 1, 1, 1], [3, 2, 1, 1]]
sage: mu.addable_cells()
[(0, 3), (1, 2), (2, 1), (5, 0)]
sage: mu.up_list()
[[4, 2, 1, 1, 1], [3, 3, 1, 1, 1], [3, 2, 2, 1, 1], [3, 2, 1, 1, 1, 1]]
sage: mu.conjugate()
[5, 2, 1]
sage: mu.dominates(nu)
True
sage: nu.dominates(mu)
True
```

Creating partitions using `Partition`:

```
sage: Partition([3,2,1])
[3, 2, 1]
sage: Partition(exp=[2,1,1])
[3, 2, 1, 1]
sage: Partition(core=[2,1], quotient=[[2,1],[3],[1,1,1]])
[11, 5, 5, 3, 2, 2, 2]
sage: Partition(frobenius_coordinates=([3,2],[4,0]))
[4, 4, 1, 1, 1]
sage: Partitions().from_zero_one([1, 1, 1, 1, 0, 1, 0])
[5, 4]
sage: [2,1] in Partitions()
True
sage: [2,1,0] in Partitions()
True
```

```
sage: Partition([1,2,3])
Traceback (most recent call last):
...
ValueError: [1, 2, 3] is not an element of Partitions
```

Sage ignores trailing zeros at the end of partitions:

```
sage: Partition([3,2,1,0])
[3, 2, 1]
sage: Partitions()([3,2,1,0])
[3, 2, 1]
sage: Partitions(6)([3,2,1,0])
[3, 2, 1]
```

TESTS:

Check that only trailing zeros are stripped:

```
sage: TestSuite( Partition([]) ).run()
sage: TestSuite( Partition([4,3,2,2,2,1]) ).run()
sage: Partition([3,2,2,2,1,0,0,0])
[3, 2, 2, 2, 1]
sage: Partition([3,0,2,2,2,1,0])
Traceback (most recent call last):
...
ValueError: [3, 0, 2, 2, 2, 1, 0] is not an element of Partitions
sage: Partition([0,7,3])
Traceback (most recent call last):
...
ValueError: [0, 7, 3] is not an element of Partitions
```

add_cell(*i*, *j=None*)

Return a partition corresponding to `self` with a cell added in row `i`. (This does not change `self`.)

EXAMPLES:

```
sage: Partition([3, 2, 1, 1]).add_cell(0)
[4, 2, 1, 1]
sage: cell = [4, 0]; Partition([3, 2, 1, 1]).add_cell(*cell)
[3, 2, 1, 1, 1]
```

add_horizontal_border_strip(*k*)

Return a list of all the partitions that can be obtained by adding a horizontal border strip of length `k` to `self`.

EXAMPLES:

```
sage: Partition([]).add_horizontal_border_strip(0)
[[]]
sage: Partition([]).add_horizontal_border_strip(2)
[[2]]
sage: Partition([2,2]).add_horizontal_border_strip(2)
[[2, 2, 2], [3, 2, 1], [4, 2]]
sage: Partition([3,2,2]).add_horizontal_border_strip(2)
[[3, 2, 2, 2], [3, 3, 2, 1], [4, 2, 2, 1], [4, 3, 2], [5, 2, 2]]
```

Todo

Reimplement like `remove_horizontal_border_strip` using `IntegerListsLex`

---

add_vertical_border_strip(*k*)

Return a list of all the partitions that can be obtained by adding a vertical border strip of length `k` to `self`.

EXAMPLES:

```
sage: Partition([]).add_vertical_border_strip(0)
[[]]
sage: Partition([]).add_vertical_border_strip(2)
[[1, 1]]
sage: Partition([2,2]).add_vertical_border_strip(2)
[[3, 3], [3, 2, 1], [2, 2, 1, 1]]
sage: Partition([3,2,2]).add_vertical_border_strip(2)
[[4, 3, 2], [4, 2, 2, 1], [3, 3, 3], [3, 3, 2, 1], [3, 2, 2, 1, 1]]
```

addable_cells()

Return a list of the outside corners of the partition `self`.

An outside corner (also called a cocorner) of a partition λ is a cell on $\mathbf{Z}^2$ which does not belong to the Young diagram of λ but can be added to this Young diagram to still form a straight-shape Young diagram.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

EXAMPLES:

```
sage: Partition([2,2,1]).outside_corners()
[(0, 2), (2, 1), (3, 0)]
sage: Partition([2,2]).outside_corners()
[(0, 2), (2, 0)]
sage: Partition([6,3,3,1,1,1]).outside_corners()
[(0, 6), (1, 3), (3, 1), (6, 0)]
sage: Partition([]).outside_corners()
[(0, 0)]
```

addable_cells_residue(*i*, *l*)

Return a list of the outside corners of the partition `self` having `l`-residue `i`.

An outside corner (also called a cocorner) of a partition λ is a cell on $\mathbf{Z}^2$ which does not belong to the Young diagram of λ but can be added to this Young diagram to still form a straight-shape Young diagram. See `residue()` for the definition of the `l`-residue.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

EXAMPLES:

```
sage: Partition([3,2,1]).outside_corners_residue(0, 3)
[(0, 3), (3, 0)]
sage: Partition([3,2,1]).outside_corners_residue(1, 3)
[(1, 2)]
sage: Partition([3,2,1]).outside_corners_residue(2, 3)
[(2, 1)]
```

arm_cells(*i*, *j*)

Return the list of the cells of the arm of cell (i, j) in `self`.

The arm of cell $c = (i, j)$ is the boxes that appear to the right of c.

The cell coordinates are zero-based, i. e., the northwesternmost cell is $(0, 0)$.

INPUT:

- `i, j` – two integers

OUTPUT:

A list of pairs of integers

EXAMPLES:

```
sage: Partition([4,4,3,1]).arm_cells(1,1)
[(1, 2), (1, 3)]

sage: Partition([]).arm_cells(0,0)
Traceback (most recent call last):
...
ValueError: The cell is not in the diagram
```

arm_length(*i*, *j*)

Return the length of the arm of cell (i, j) in `self`.

The arm of cell (i, j) is the cells that appear to the right of cell (i, j).

The cell coordinates are zero-based, i. e., the northwesternmost cell is $(0, 0)$.

INPUT:

- `i, j` – two integers

OUTPUT:

An integer or a `ValueError`

EXAMPLES:

```
sage: p = Partition([2,2,1])
sage: p.arm_length(0, 0)
1
sage: p.arm_length(0, 1)
0
sage: p.arm_length(2, 0)
0
sage: Partition([3,3]).arm_length(0, 0)
2
sage: Partition([3,3]).arm_length(*[0,0])
2
```

arm_lengths(*flat=False*)

Return a tableau of shape `self` where each cell is filled with its arm length. The optional boolean parameter `flat` provides the option of returning a flat list.

EXAMPLES:

```
sage: Partition([2,2,1]).arm_lengths()
[[1, 0], [1, 0], [0]]
sage: Partition([2,2,1]).arm_lengths(flat=True)
[1, 0, 1, 0, 0]
sage: Partition([3,3]).arm_lengths()
[[2, 1, 0], [2, 1, 0]]
sage: Partition([3,3]).arm_lengths(flat=True)
[2, 1, 0, 2, 1, 0]
```

arms_legs_coeff(*i*, *j*)

This is a statistic on a cell $c = (i, j)$ in the diagram of partition p given by

$$\frac{1 - q^a \cdot t^{\ell+1}}{1 - q^{a+1} \cdot t^{\ell}}$$

where a is the arm length of c and ℓ is the leg length of c.

The coordinates `i` and `j` of the cell are understood to be 0-based, so that `(0, 0)` is the northwesternmost cell (in English notation).

EXAMPLES:

```
sage: Partition([3,2,1]).arms_legs_coeff(1,1)
(-t + 1)/(-q + 1)
sage: Partition([3,2,1]).arms_legs_coeff(0,0)
(-q^2*t^3 + 1)/(-q^3*t^2 + 1)
sage: Partition([3,2,1]).arms_legs_coeff(*[0,0])
(-q^2*t^3 + 1)/(-q^3*t^2 + 1)
```

atom()

Return a list of the standard tableaux of size `self.size()` whose atom is equal to `self`.

EXAMPLES:

```
sage: Partition([2,1]).atom()
[[[1, 2], [3]]]
sage: Partition([3,2,1]).atom()
[[[1, 2, 3, 6], [4, 5]], [[1, 2, 3], [4, 5], [6]]]
```

attacking_pairs()

Return a list of the attacking pairs of the Young diagram of `self`.

A pair of cells (c, d) of a Young diagram (in English notation) is said to be attacking if one of the following conditions holds:

1. c and d lie in the same row with c strictly to the west of d.

2. c is in the row immediately to the south of d, and c lies strictly east of d.

This particular method returns each pair (c, d) as a tuple, where each of c and d is given as a tuple (i, j) with i and j zero-based (so $i = 0$ means that the cell lies in the topmost row).

EXAMPLES:

```
sage: p = Partition([3, 2])
sage: p.attacking_pairs()
[((0, 0), (0, 1)),
 ((0, 0), (0, 2)),
 ((0, 1), (0, 2)),
 ((1, 0), (1, 1)),
 ((1, 1), (0, 0))]
sage: Partition([]).attacking_pairs()
[]
```

aut()

Return a factor for the number of permutations with cycle type `self`.

This method returns $1^{j_1} j_1! \cdots n^{j_n} j_n!$ where j_k is the number of parts in `self` equal to k.

The number of permutations having `self` as a cycle type is given by

$$\frac{n!}{1^{j_1} j_1! \cdots n^{j_n} j_n!}$$

(where n is the size of `self`).

EXAMPLES:

```
sage: Partition([2,1]).aut()
2
```

beta_numbers(*length=None*)

Return the set of beta numbers corresponding to `self`.

The optional argument `length` specifies the length of the beta set (which must be at least the length of `self`).

For more on beta numbers, see `frobenius_coordinates()`.

EXAMPLES:

```
sage: Partition([4,3,2]).beta_numbers()
[6, 4, 2]
sage: Partition([4,3,2]).beta_numbers(5)
[8, 6, 4, 1, 0]
sage: Partition([]).beta_numbers()
[]
sage: Partition([]).beta_numbers(3)
[2, 1, 0]
sage: Partition([6,4,1,1]).beta_numbers()
[9, 6, 2, 1]
sage: Partition([6,4,1,1]).beta_numbers(6)
[11, 8, 4, 3, 1, 0]
sage: Partition([1,1,1]).beta_numbers()
[3, 2, 1]
sage: Partition([1,1,1]).beta_numbers(4)
[4, 3, 2, 0]
```

cell_poset(*orientation='SE'*)

Return the Young diagram of `self` as a poset. The optional keyword variable `orientation` determines the order relation of the poset.

The poset always uses the set of cells of the Young diagram of `self` as its ground set. The order relation of the poset depends on the `orientation` variable (which defaults to `"SE"`). Concretely, `orientation` has to be specified to one of the strings `"NW"`, `"NE"`, `"SW"`, and `"SE"`, standing for "northwest", "northeast", "southwest" and "southeast", respectively. If `orientation` is `"SE"`, then the order relation of the poset is such that a cell u is greater or equal to a cell v in the poset if and only if u lies weakly southeast of v (this means that u can be reached from v by a sequence of south and east steps; the sequence is allowed to consist of south steps only, or of east steps only, or even be empty). Similarly the order relation is defined for the other three orientations. The Young diagram is supposed to be drawn in English notation.

The elements of the poset are the cells of the Young diagram of `self`, written as tuples of zero-based coordinates (so that $(3, 7)$ stands for the 8-th cell of the 4-th row, etc.).

EXAMPLES:

```
sage: p = Partition([3,3,1])
sage: Q = p.cell_poset(); Q
Finite poset containing 7 elements
sage: sorted(Q)
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0)]
sage: sorted(Q.maximal_elements())
[(1, 2), (2, 0)]
sage: Q.minimal_elements()
[(0, 0)]
sage: sorted(Q.upper_covers((1, 0)))
[(1, 1), (2, 0)]
sage: Q.upper_covers((1, 1))
```

```
[(1, 2)]

sage: P = p.cell_poset(orientation="NW"); P
Finite poset containing 7 elements
sage: sorted(P)
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0)]
sage: sorted(P.minimal_elements())
[(1, 2), (2, 0)]
sage: P.maximal_elements()
[(0, 0)]
sage: P.upper_covers((2, 0))
[(1, 0)]
sage: sorted(P.upper_covers((1, 2)))
[(0, 2), (1, 1)]
sage: sorted(P.upper_covers((1, 1)))
[(0, 1), (1, 0)]
sage: sorted([len(P.upper_covers(v)) for v in P])
[0, 1, 1, 1, 1, 2, 2]

sage: R = p.cell_poset(orientation="NE"); R
Finite poset containing 7 elements
sage: sorted(R)
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0)]
sage: R.maximal_elements()
[(0, 2)]
sage: R.minimal_elements()
[(2, 0)]
sage: sorted([len(R.upper_covers(v)) for v in R])
[0, 1, 1, 1, 1, 2, 2]
sage: R.is_isomorphic(P)
False
sage: R.is_isomorphic(P.dual())
False
```

Linear extensions of `p.cell_poset()` are in 1-to-1 correspondence with standard Young tableaux of shape p:

```
sage: all( len(p.cell_poset().linear_extensions())
....:      == len(p.standard_tableaux())
....:      for n in range(8) for p in Partitions(n) )
True
```

This is not the case for northeast orientation:

```
sage: q = Partition([3, 1])
sage: q.cell_poset(orientation="NE").is_chain()
True
```

TESTS:

We check that the posets are really what they should be for size up to 7:

```
sage: def check_NW(n):
....:     for p in Partitions(n):
....:         P = p.cell_poset(orientation="NW")
....:         for c in p.cells():
....:             for d in p.cells():
....:                 if P.le(c, d) != (c[0] >= d[0]
....:                                   and c[1] >= d[1]):
....:                     return False
```

```
....:     return True
sage: all( check_NW(n) for n in range(8) )
True

sage: def check_NE(n):
....:     for p in Partitions(n):
....:         P = p.cell_poset(orientation="NE")
....:         for c in p.cells():
....:             for d in p.cells():
....:                 if P.le(c, d) != (c[0] >= d[0]
....:                                   and c[1] <= d[1]):
....:                     return False
....:     return True
sage: all( check_NE(n) for n in range(8) )
True

sage: def test_duality(n, ori1, ori2):
....:     for p in Partitions(n):
....:         P = p.cell_poset(orientation=ori1)
....:         Q = p.cell_poset(orientation=ori2)
....:         for c in p.cells():
....:             for d in p.cells():
....:                 if P.lt(c, d) != Q.lt(d, c):
....:                     return False
....:     return True
sage: all( test_duality(n, "NW", "SE") for n in range(8) )
True
sage: all( test_duality(n, "NE", "SW") for n in range(8) )
True
sage: all( test_duality(n, "NE", "SE") for n in range(4) )
False
```

cells()

Return the coordinates of the cells of `self`.

EXAMPLES:

```
sage: Partition([2,2]).cells()
[(0, 0), (0, 1), (1, 0), (1, 1)]
sage: Partition([3,2]).cells()
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1)]
```

centralizer_size(*t=0*, *q=0*)

Return the size of the centralizer of any permutation of cycle type `self`.

If m_i is the multiplicity of i as a part of p, this is given by

$$\prod_i m_i! i^{m_i}.$$

Including the optional parameters t and q gives the $q-t$ analog which is the former product times

$$\prod_{i=1}^{\text{length}(p)} \frac{1-q^{p_i}}{1-t^{p_i}}.$$

See [Ker].

EXAMPLES:

```
sage: Partition([2,2,1]).centralizer_size()
8
sage: Partition([2,2,2]).centralizer_size()
48
sage: Partition([2,2,1]).centralizer_size(q=2, t=3)
9/16
sage: Partition([]).centralizer_size()
1
sage: Partition([]).centralizer_size(q=2, t=4)
1
```

character_polynomial()

Return the character polynomial associated to the partition `self`.

The character polynomial q_μ associated to a partition μ is defined by

$$q_\mu(x_1, x_2, \ldots, x_k) = \downarrow \sum_{\alpha \vdash k} \frac{\chi^\mu_\alpha}{1^{a_1} 2^{a_2} \cdots k^{a_k} a_1! a_2! \cdots a_k!} \prod_{i=1}^{k} (i x_i - 1)^{a_i}$$

where k is the size of μ, and a_i is the multiplicity of i in α.

It is computed in the following manner:

1. Expand the Schur function s_μ in the power-sum basis,
2. Replace each p_i with $ix_i - 1$,
3. Apply the umbral operator $\downarrow$ to the resulting polynomial.

EXAMPLES:

```
sage: Partition([1]).character_polynomial()
x - 1
sage: Partition([1,1]).character_polynomial()
1/2*x0^2 - 3/2*x0 - x1 + 1
sage: Partition([2,1]).character_polynomial()
1/3*x0^3 - 2*x0^2 + 8/3*x0 - x2
```

components()

Return a list containing the shape of `self`.

This method exists only for compatibility with `PartitionTuples`.

EXAMPLES:

```
sage: Partition([3,2]).components()
[[3, 2]]
```

conjugacy_class_size()

Return the size of the conjugacy class of the symmetric group indexed by `self`.

EXAMPLES:

```
sage: Partition([2,2,2]).conjugacy_class_size()
15
sage: Partition([2,2,1]).conjugacy_class_size()
15
sage: Partition([2,1,1]).conjugacy_class_size()
6
```

REFERENCES:

conjugate()

Return the conjugate partition of the partition `self`. This is also called the associated partition or the transpose in the literature.

EXAMPLES:

```
sage: Partition([2,2]).conjugate()
[2, 2]
sage: Partition([6,3,1]).conjugate()
[3, 2, 2, 1, 1, 1]
```

The conjugate partition is obtained by transposing the Ferrers diagram of the partition (see `ferrers_diagram()`):

```
sage: print Partition([6,3,1]).ferrers_diagram()
******
***
*
sage: print Partition([6,3,1]).conjugate().ferrers_diagram()
***
**
**
*
*
*
```

contains(x)

Return `True` if `x` is a partition whose Ferrers diagram is contained in the Ferrers diagram of `self`.

EXAMPLES:

```
sage: p = Partition([3,2,1])
sage: p.contains([2,1])
True
sage: all(p.contains(mu) for mu in Partitions(3))
True
sage: all(p.contains(mu) for mu in Partitions(4))
False
```

content(*r, c, multicharge=[0]*)

Return the content of the cell at row r and column c.

The content of a cell is $c - r$.

For consistency with partition tuples there is also an optional `multicharge` argument which is an offset to the usual content. By setting the `multicharge` equal to the 0-element of the ring $\mathbf{Z}/e\mathbf{Z}$, the corresponding e-residue will be returned. This is the content modulo e.

The content (and residue) do not strictly depend on the partition, however, this method is included because it is often useful in the context of partitions.

EXAMPLES:

```
sage: Partition([2,1]).content(1,0)
-1
sage: p = Partition([3,2])
sage: sum([p.content(*c) for c in p.cells()])
2
```

and now we return the 3-residue of a cell:

```
sage: Partition([2,1]).content(1,0, multicharge=[IntegerModRing(3)(0)])
2
```

core(*length*)

Return the `length`-core of the partition – in the literature the core is commonly referred to as the k-core, p-core, r-core, ...

The r-core of a partition λ can be obtained by repeatedly removing rim hooks of size r from (the Young diagram of) λ until this is no longer possible. The remaining partition is the core.

EXAMPLES:

```
sage: Partition([6,3,2,2]).core(3)
[2, 1, 1]
sage: Partition([]).core(3)
[]
sage: Partition([8,7,7,4,1,1,1,1,1]).core(3)
[2, 1, 1]
```

TESTS:

```
sage: Partition([3,3,3,2,1]).core(3)
[]
sage: Partition([10,8,7,7]).core(4)
[]
sage: Partition([21,15,15,9,6,6,6,3,3]).core(3)
[]
```

corners()

Return a list of the corners of the partition `self`.

A corner of a partition λ is a cell of the Young diagram of λ which can be removed from the Young diagram while still leaving a straight shape behind.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

EXAMPLES:

```
sage: Partition([3,2,1]).corners()
[(0, 2), (1, 1), (2, 0)]
sage: Partition([3,3,1]).corners()
[(1, 2), (2, 0)]
sage: Partition([]).corners()
[]
```

corners_residue(*i*, *l*)

Return a list of the corners of the partition `self` having `l`-residue `i`.

A corner of a partition λ is a cell of the Young diagram of λ which can be removed from the Young diagram while still leaving a straight shape behind. See `residue()` for the definition of the `l`-residue.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

EXAMPLES:

```
sage: Partition([3,2,1]).corners_residue(0, 3)
[(1, 1)]
sage: Partition([3,2,1]).corners_residue(1, 3)
[(2, 0)]
```

```
sage: Partition([3,2,1]).corners_residue(2, 3)
[(0, 2)]
```

crank()

Return the Dyson crank of `self`.

The Dyson crank of a partition λ is defined as follows: If λ contains at least one 1, then the crank is $\mu(\lambda) - \omega(\lambda)$, where $\omega(\lambda)$ is the number of 1'sin'λ, and $\mu(\lambda)$ is the number of parts of λ larger than $\omega(\lambda)$. If λ contains no 1, then the crank is simply the largest part of λ.

REFERENCES:

EXAMPLES:

```
sage: Partition([]).crank()
0
sage: Partition([3,2,2]).crank()
3
sage: Partition([5,4,2,1,1]).crank()
0
sage: Partition([1,1,1]).crank()
-3
sage: Partition([6,4,4,3]).crank()
6
sage: Partition([6,3,3,1,1]).crank()
1
sage: Partition([6]).crank()
6
sage: Partition([5,1]).crank()
0
sage: Partition([4,2]).crank()
4
sage: Partition([4,1,1]).crank()
-1
sage: Partition([3,3]).crank()
3
sage: Partition([3,2,1]).crank()
1
sage: Partition([3,1,1,1]).crank()
-3
sage: Partition([2,2,2]).crank()
2
sage: Partition([2,2,1,1]).crank()
-2
sage: Partition([2,1,1,1,1]).crank()
-4
sage: Partition([1,1,1,1,1,1]).crank()
-6
```

dimension(*smaller=*[ ], *k=1*)

Return the number of paths from the `smaller` partition to the partition `self`, where each step consists of adding a k-ribbon while keeping a partition.

Note that a 1-ribbon is just a single cell, so this counts paths in the Young graph when $k = 1$.

Note also that the default case ($k = 1$ and `smaller = []`) gives the dimension of the irreducible representation of the symmetric group corresponding to `self`.

INPUT:

- `smaller` – a partition (default: an empty list `[]`)
- k – a positive integer (default: 1)

OUTPUT:

The number of such paths

EXAMPLES:

Looks at the number of ways of getting from `[5,4]` to the empty partition, removing one cell at a time:

```
sage: mu = Partition([5,4])
sage: mu.dimension()
42
```

Same, but removing one 3-ribbon at a time. Note that the 3-core of `mu` is empty:

```
sage: mu.dimension(k=3)
3
```

The 2-core of `mu` is not the empty partition:

```
sage: mu.dimension(k=2)
0
```

Indeed, the 2-core of `mu` is `[1]`:

```
sage: mu.dimension(Partition([1]),k=2)
2
```

TESTS:

Checks that the sum of squares of dimensions of characters of the symmetric group is the order of the group:

```
sage: all(sum(mu.dimension()^2 for mu in Partitions(i))==factorial(i) for i in range(10))
True
```

A check coming from the theory of k-differentiable posets:

```
sage: k=2; core = Partition([2,1])
sage: all(sum(mu.dimension(core,k=2)^2
....:         for mu in Partitions(3+i*2) if mu.core(2) == core)
....:     == 2^i*factorial(i) for i in range(10))
True
```

Checks that the dimension satisfies the obvious recursion relation:

```
sage: test = lambda larger, smaller: larger.dimension(smaller) == sum(mu.dimension(smaller)
sage: all(test(larger,smaller) for l in xrange(1,10) for s in xrange(0,10)
....:     for larger in Partitions(l) for smaller in Partitions(s) if smaller != larger)
True
```

ALGORITHM:

Depending on the parameters given, different simplifications occur. When $k = 1$ and `smaller` is empty, this function uses the hook formula. When $k = 1$ and `smaller` is not empty, it uses a formula from [ORV].

When $k \neq 1$, we first check that both `self` and `smaller` have the same k-core, then use the k-quotients and the same algorithm on each of the k-quotients.

REFERENCES:

AUTHORS:

- Paul-Olivier Dehaye (2011-06-07)

dominated_partitions(*rows=None*)

Return a list of the partitions dominated by n. If `rows` is specified, then it only returns the ones whose number of rows is at most `rows`.

EXAMPLES:

```
sage: Partition([3,2,1]).dominated_partitions()
[[3, 2, 1], [3, 1, 1, 1], [2, 2, 2], [2, 2, 1, 1], [2, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1]]
sage: Partition([3,2,1]).dominated_partitions(rows=3)
[[3, 2, 1], [2, 2, 2]]
```

dominates(*p2*)

Return `True` if `self` dominates the partition `p2`. Otherwise it returns `False`.

EXAMPLES:

```
sage: p = Partition([3,2])
sage: p.dominates([3,1])
True
sage: p.dominates([2,2])
True
sage: p.dominates([2,1,1])
True
sage: p.dominates([3,3])
False
sage: p.dominates([4])
False
sage: Partition([4]).dominates(p)
False
sage: Partition([]).dominates([1])
False
sage: Partition([]).dominates([])
True
sage: Partition([1]).dominates([])
True
```

down()

Return a generator for partitions that can be obtained from `self` by removing a cell.

EXAMPLES:

```
sage: [p for p in Partition([2,1,1]).down()]
[[1, 1, 1], [2, 1]]
sage: [p for p in Partition([3,2]).down()]
[[2, 2], [3, 1]]
sage: [p for p in Partition([3,2,1]).down()]
[[2, 2, 1], [3, 1, 1], [3, 2]]
```

TESTS:

We check that trac ticket #11435 is fixed:

```
sage: Partition([]).down_list() #indirect doctest
[]
```

down_list()

Return a list of the partitions that can be obtained from `self` by removing a cell.

EXAMPLES:

```
sage: Partition([2,1,1]).down_list()
[[1, 1, 1], [2, 1]]
sage: Partition([3,2]).down_list()
[[2, 2], [3, 1]]
sage: Partition([3,2,1]).down_list()
[[2, 2, 1], [3, 1, 1], [3, 2]]
sage: Partition([]).down_list()  # checks trac #11435
[]
```

dual_equivalence_graph(*directed=False*, *coloring=None*)

Return the dual equivalence graph of `self`.

Two permutations p and q in the symmetric group S_n differ by an *i-elementary dual equivalence (or dual Knuth) relation* (where i is an integer with $1 < i < n$) when the following two conditions are satisfied:

- In the one-line notation of the permutation p, the letter i does not appear inbetween $i-1$ and $i+1$.
- The permutation q is obtained from p by switching two of the three letters $i-1, i, i+1$ (in its one-line notation) – namely, the leftmost and the rightmost one in order of their appearance in p.

Notice that this is equivalent to the statement that the permutations p^{-1} and q^{-1} differ by an elementary Knuth equivalence at positions $i-1, i, i+1$.

Two standard Young tableaux of shape λ differ by an i-elementary dual equivalence relation (of color i), if their reading words differ by an i-elementary dual equivalence relation.

The *dual equivalence graph* of the partition λ is the edge-colored graph whose vertices are the standard Young tableaux of shape λ, and whose edges colored by i are given by the i-elementary dual equivalences.

INPUT:

- `directed` – (default: `False`) whether to have the dual equivalence graph be directed (where we have a directed edge $S \to T$ if i appears to the left of $i+1$ in the reading word of T; otherwise we have the directed edge $T \to S$)
- `coloring` – (optional) a function which sends each integer $i > 1$ to a color (as a string, e.g., `'red'` or `'black'`) to be used when visually representing the resulting graph using dot2tex; the default choice is `2 -> 'red', 3 -> 'blue', 4 -> 'green', 5 -> 'purple', 6 -> 'brown', 7 -> 'orange', 8 -> 'yellow', anything greater than 8 -> 'black'`.

REFERENCES:

EXAMPLES:

```
sage: P = Partition([3,1,1])
sage: G = P.dual_equivalence_graph()
sage: sorted(G.edges())
[([[1, 2, 3], [4], [5]], [[1, 2, 4], [3], [5]], 3),
 ([[1, 2, 4], [3], [5]], [[1, 2, 5], [3], [4]], 4),
 ([[1, 2, 4], [3], [5]], [[1, 3, 4], [2], [5]], 2),
 ([[1, 2, 5], [3], [4]], [[1, 3, 5], [2], [4]], 2),
 ([[1, 3, 4], [2], [5]], [[1, 3, 5], [2], [4]], 4),
 ([[1, 3, 5], [2], [4]], [[1, 4, 5], [2], [3]], 3)]
sage: G = P.dual_equivalence_graph(directed=True)
sage: sorted(G.edges())
[([[1, 2, 4], [3], [5]], [[1, 2, 3], [4], [5]], 3),
 ([[1, 2, 5], [3], [4]], [[1, 2, 4], [3], [5]], 4),
 ([[1, 3, 4], [2], [5]], [[1, 2, 4], [3], [5]], 2),
 ([[1, 3, 5], [2], [4]], [[1, 2, 5], [3], [4]], 2),
```

```
 ([[1, 3, 5], [2], [4]], [[1, 3, 4], [2], [5]], 4),
 ([[1, 4, 5], [2], [3]], [[1, 3, 5], [2], [4]], 3)]
```

TESTS:

```
sage: G = Partition([1]).dual_equivalence_graph()
sage: G.vertices()
[[[1]]]
sage: G = Partition([]).dual_equivalence_graph()
sage: G.vertices()
[[]]

sage: P = Partition([3,1,1])
sage: G = P.dual_equivalence_graph(coloring=lambda x: 'red')
sage: G2 = P.dual_equivalence_graph(coloring={2: 'black', 3: 'blue', 4: 'cyan', 5: 'grey'})
sage: G is G2
False
sage: G == G2
True
```

evaluation()

Return the evaluation of `self`.

The **commutative evaluation**, often shortened to **evaluation**, of a word (we think of a partition as a word in $\{1, 2, 3, \ldots\}$) is its image in the free commutative monoid. In other words, this counts how many occurrences there are of each letter.

This is also is known as **Parikh vector** and **abelianization** and has the same output as `to_exp()`.

EXAMPLES:

```
sage: Partition([4,3,1,1]).evaluation()
[2, 0, 1, 1]
```

ferrers_diagram()

Return the Ferrers diagram of `self`.

EXAMPLES:

```
sage: mu=Partition([5,5,2,1])
sage: Partitions.global_options(diagram_str='*', convention="english")
sage: print mu.ferrers_diagram()
*****
*****
**
*
sage: Partitions.global_options(diagram_str='#')
sage: print mu.ferrers_diagram()
#####
#####
##
#
sage: Partitions.global_options(convention="french")
sage: print mu.ferrers_diagram()
#
##
#####
#####
sage: print Partition([]).ferrers_diagram()
-
```

```
sage: Partitions.global_options(diagram_str='-')
sage: print Partition([]).ferrers_diagram()
(/)
sage: Partitions.global_options.reset()
```

frobenius_coordinates()

Return a pair of sequences of Frobenius coordinates aka beta numbers of the partition.

These are two strictly decreasing sequences of nonnegative integers of the same length.

EXAMPLES:

```
sage: Partition([]).frobenius_coordinates()
([], [])
sage: Partition([1]).frobenius_coordinates()
([0], [0])
sage: Partition([3,3,3]).frobenius_coordinates()
([2, 1, 0], [2, 1, 0])
sage: Partition([9,1,1,1,1,1,1]).frobenius_coordinates()
([8], [6])
```

frobenius_rank()

Return the Frobenius rank of the partition `self`.

The Frobenius rank of a partition $\lambda = (\lambda_1, \lambda_2, \lambda_3, \cdots)$ is defined to be the largest i such that $\lambda_i \geq i$. In other words, it is the number of cells on the main diagonal of λ. In yet other words, it is the size of the largest square fitting into the Young diagram of λ.

EXAMPLES:

```
sage: Partition([]).frobenius_rank()
0
sage: Partition([1]).frobenius_rank()
1
sage: Partition([3,3,3]).frobenius_rank()
3
sage: Partition([9,1,1,1,1,1]).frobenius_rank()
1
sage: Partition([2,1,1,1,1,1]).frobenius_rank()
1
sage: Partition([2,2,1,1,1,1]).frobenius_rank()
2
sage: Partition([3,2]).frobenius_rank()
2
sage: Partition([3,2,2]).frobenius_rank()
2
sage: Partition([8,4,4,4,4]).frobenius_rank()
4
sage: Partition([8,4,1]).frobenius_rank()
2
sage: Partition([3,3,1]).frobenius_rank()
2
```

from_kbounded_to_grassmannian(k)

Maps a k-bounded partition to a Grassmannian element in the affine Weyl group of type $A_k^{(1)}$.

For details, see the documentation of the method `from_kbounded_to_reduced_word()` .

EXAMPLES:

```
sage: p=Partition([2,1,1])
sage: p.from_kbounded_to_grassmannian(2)
[-1  1  1]
[-2  2  1]
[-2  1  2]
sage: p=Partition([])
sage: p.from_kbounded_to_grassmannian(2)
[1 0 0]
[0 1 0]
[0 0 1]
```

from_kbounded_to_reduced_word(*k*)

Maps a k-bounded partition to a reduced word for an element in the affine permutation group.

This uses the fact that there is a bijection between k-bounded partitions and $(k+1)$-cores and an action of the affine nilCoxeter algebra of type $A_k^{(1)}$ on $(k+1)$-cores as described in [LM2006].

REFERENCES:

EXAMPLES:

```
sage: p=Partition([2,1,1])
sage: p.from_kbounded_to_reduced_word(2)
[2, 1, 2, 0]
sage: p=Partition([3,1])
sage: p.from_kbounded_to_reduced_word(3)
[3, 2, 1, 0]
sage: p.from_kbounded_to_reduced_word(2)
Traceback (most recent call last):
...
ValueError: the partition must be 2-bounded
sage: p=Partition([])
sage: p.from_kbounded_to_reduced_word(2)
[]
```

garnir_tableau(**cell*)

Return the Garnir tableau of shape `self` corresponding to the cell `cell`. If `cell` $= (a, c)$ then $(a+1, c)$ must belong to the diagram of `self`.

The Garnir tableaux play an important role in integral and non-semisimple representation theory because they determine the "straightening" rules for the Specht modules over an arbitrary ring.

The Garnir tableaux are the "first" non-standard tableaux which arise when you act by simple transpositions. If (a, c) is a cell in the Young diagram of a partition, which is not at the bottom of its column, then the corresponding Garnir tableau has the integers $1, 2, \ldots, n$ entered in order from left to right along the rows of the diagram up to the cell $(a, c-1)$, then along the cells $(a+1, 1)$ to $(a+1, c)$, then (a, c) until the end of row a and then continuing from left to right in the remaining positions. The examples below probably make this clearer!

Note: The function also sets `g._garnir_cell`, where `g` is the resulting Garnir tableau, equal to `cell` which is used by some other functions.

EXAMPLES:

```
sage: g=Partition([5,3,3,2]).garnir_tableau((0,2)); g.pp()
  1  2  6  7  8
  3  4  5
  9 10 11
 12 13
```

```
sage: g.is_row_strict(); g.is_column_strict()
True
False

sage: Partition([5,3,3,2]).garnir_tableau(0,2).pp()
  1  2  6  7  8
  3  4  5
  9 10 11
 12 13
sage: Partition([5,3,3,2]).garnir_tableau(2,1).pp()
  1  2  3  4  5
  6  7  8
  9 12 13
 10 11
sage: Partition([5,3,3,2]).garnir_tableau(2,2).pp()
Traceback (most recent call last):
...
ValueError: (row+1, col) must be inside the diagram
```

See also:

- top_garnir_tableau()

generalized_pochhammer_symbol(*a*, *alpha*)

Return the generalized Pochhammer symbol $(a)^{(\alpha)}_{self}$. This is the product over all cells (i,j) in self of $a-(i-1)/\alpha+j-1$.

EXAMPLES:

```
sage: Partition([2,2]).generalized_pochhammer_symbol(2,1)
12
```

get_part(*i*, *default=0*)

Return the i^{th} part of self, or default if it does not exist.

EXAMPLES:

```
sage: p = Partition([2,1])
sage: p.get_part(0), p.get_part(1), p.get_part(2)
(2, 1, 0)
sage: p.get_part(10,-1)
-1
sage: Partition([]).get_part(0)
0
```

hook_length(*i*, *j*)

Return the length of the hook of cell (i,j) in self.

The (length of the) hook of cell (i,j) of a partition λ is

$$\lambda_i + \lambda'_j - i - j + 1$$

where λ' is the conjugate partition. In English convention, the hook length is the number of cells horizontally to the right and vertically below the cell (i,j) (including that cell).

EXAMPLES:

```
sage: p = Partition([2,2,1])
sage: p.hook_length(0, 0)
4
```

```
sage: p.hook_length(0, 1)
2
sage: p.hook_length(2, 0)
1
sage: Partition([3,3]).hook_length(0, 0)
4
sage: cell = [0,0]; Partition([3,3]).hook_length(*cell)
4
```

hook_lengths()

Return a tableau of shape `self` with the cells filled in with the hook lengths.

In each cell, put the sum of one plus the number of cells horizontally to the right and vertically below the cell (the hook length).

For example, consider the partition `[3,2,1]` of 6 with Ferrers diagram:

```
# # #
# #
#
```

When we fill in the cells with the hook lengths, we obtain:

```
5 3 1
3 1
1
```

EXAMPLES:

```
sage: Partition([2,2,1]).hook_lengths()
[[4, 2], [3, 1], [1]]
sage: Partition([3,3]).hook_lengths()
[[4, 3, 2], [3, 2, 1]]
sage: Partition([3,2,1]).hook_lengths()
[[5, 3, 1], [3, 1], [1]]
sage: Partition([2,2]).hook_lengths()
[[3, 2], [2, 1]]
sage: Partition([5]).hook_lengths()
[[5, 4, 3, 2, 1]]
```

REFERENCES:

- http://mathworld.wolfram.com/HookLengthFormula.html

hook_polynomial(q, t)

Return the two-variable hook polynomial.

EXAMPLES:

```
sage: R.<q,t> = PolynomialRing(QQ)
sage: a = Partition([2,2]).hook_polynomial(q,t)
sage: a == (1 - t)*(1 - q*t)*(1 - t^2)*(1 - q*t^2)
True
sage: a = Partition([3,2,1]).hook_polynomial(q,t)
sage: a == (1 - t)^3*(1 - q*t^2)^2*(1 - q^2*t^3)
True
```

hook_product(a)

Return the Jack hook-product.

EXAMPLES:

```
sage: Partition([3,2,1]).hook_product(x)
(2*x + 3)*(x + 2)^2
sage: Partition([2,2]).hook_product(x)
2*(x + 2)*(x + 1)
```

hooks()

Return a sorted list of the hook lengths in `self`.

EXAMPLES:

```
sage: Partition([3,2,1]).hooks()
[5, 3, 3, 1, 1, 1]
```

initial_tableau()

Return the `standard tableau` which has the numbers $1, 2, \ldots, n$ where n is the `size()` of `self` entered in order from left to right along the rows of each component, where the components are ordered from left to right.

EXAMPLES:

```
sage: Partition([3,2,2]).initial_tableau()
[[1, 2, 3], [4, 5], [6, 7]]
```

inside_corners()

Return a list of the corners of the partition `self`.

A corner of a partition λ is a cell of the Young diagram of λ which can be removed from the Young diagram while still leaving a straight shape behind.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

EXAMPLES:

```
sage: Partition([3,2,1]).corners()
[(0, 2), (1, 1), (2, 0)]
sage: Partition([3,3,1]).corners()
[(1, 2), (2, 0)]
sage: Partition([]).corners()
[]
```

inside_corners_residue(i, l)

Return a list of the corners of the partition `self` having `l`-residue `i`.

A corner of a partition λ is a cell of the Young diagram of λ which can be removed from the Young diagram while still leaving a straight shape behind. See `residue()` for the definition of the `l`-residue.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

EXAMPLES:

```
sage: Partition([3,2,1]).corners_residue(0, 3)
[(1, 1)]
sage: Partition([3,2,1]).corners_residue(1, 3)
[(2, 0)]
sage: Partition([3,2,1]).corners_residue(2, 3)
[(0, 2)]
```

is_core(k)

Tests whether the partition is a k-core or not. Visuallly, this can be checked by trying to remove border

strips of size k from `self`. If this is not possible, then `self` is a k-core.

A partition is said to be a *'k'-core* if it has no hooks of length k. Equivalently, a partition is said to be a k-core if it is its own k-core (where the latter is defined as in `core()`).

EXAMPLES:

```
sage: p = Partition([12,8,5,5,2,2,1])
sage: p.is_core(4)
False
sage: p.is_core(5)
True
sage: p.is_core(0)
True
```

is_empty()

Return `True` if `self` is the empty partition.

EXAMPLES:

```
sage: Partition([]).is_empty()
True
sage: Partition([2,1,1]).is_empty()
False
```

jacobi_trudi()

Return the Jacobi-Trudi matrix of `self` thought of as a skew partition. See `SkewPartition.jacobi_trudi()`.

EXAMPLES:

```
sage: part = Partition([3,2,1])
sage: jt = part.jacobi_trudi(); jt
[h[3] h[1]    0]
[h[4] h[2]  h[]]
[h[5] h[3] h[1]]
sage: s = SymmetricFunctions(QQ).schur()
sage: h = SymmetricFunctions(QQ).homogeneous()
sage: h( s(part) )
h[3, 2, 1] - h[3, 3] - h[4, 1, 1] + h[5, 1]
sage: jt.det()
h[3, 2, 1] - h[3, 3] - h[4, 1, 1] + h[5, 1]
```

k_atom(k)

Return a list of the standard tableaux of size `self.size()` whose `k`-atom is equal to `self`.

EXAMPLES:

```
sage: p = Partition([3,2,1])
sage: p.k_atom(1)
[]
sage: p.k_atom(3)
[[[1, 1, 1], [2, 2], [3]],
 [[1, 1, 1, 2], [2], [3]],
 [[1, 1, 1, 3], [2, 2]],
 [[1, 1, 1, 2, 3], [2]]]
sage: Partition([3,2,1]).k_atom(4)
[[[1, 1, 1], [2, 2], [3]], [[1, 1, 1, 3], [2, 2]]]
```

TESTS:

```
sage: Partition([1]).k_atom(1)
[[[1]]]
sage: Partition([1]).k_atom(2)
[[[1]]]
sage: Partition([]).k_atom(1)
[[]]
```

k_boundary(*k*)

Return the skew partition formed by removing the cells of the k-interior, see k_interior().

EXAMPLES:

```
sage: p = Partition([3,2,1])
sage: p.k_boundary(2)
[3, 2, 1] / [2, 1]
sage: p.k_boundary(3)
[3, 2, 1] / [1]

sage: p = Partition([12,8,5,5,2,2,1])
sage: p.k_boundary(4)
[12, 8, 5, 5, 2, 2, 1] / [8, 5, 2, 2]
```

k_conjugate(*k*)

Return the k-conjugate of self.

The k-conjugate is the partition that is given by the columns of the k-skew diagram of the partition.

We can also define the k-conjugate in the following way. Let P denote the bijection from $(k+1)$-cores to k-bounded partitions. The k-conjugate of a $(k+1)$-core λ is

$$\lambda^{(k)} = P^{-1}\left((P(\lambda))'\right).$$

EXAMPLES:

```
sage: p = Partition([4,3,2,2,1,1])
sage: p.k_conjugate(4)
[3, 2, 2, 1, 1, 1, 1, 1, 1]
```

k_interior(*k*)

Return the partition consisting of the cells of self whose hook lengths are greater than k.

EXAMPLES:

```
sage: p = Partition([3,2,1])
sage: p.hook_lengths()
[[5, 3, 1], [3, 1], [1]]
sage: p.k_interior(2)
[2, 1]
sage: p.k_interior(3)
[1]

sage: p = Partition([])
sage: p.k_interior(3)
[]
```

k_irreducible(*k*)

Return the partition with all $r \times (k+1-r)$ rectangles removed.

If self is a k-bounded partition, then this method will return the partition where all rectangles of dimension $r \times (k+1-r)$ for $1 \leq r \leq k$ have been deleted.

If `self` is not a k-bounded partition then the method will raise an error.

INPUT:

- `k` – a non-negative integer

OUTPUT:

- a partition

EXAMPLES:

```
sage: Partition([3,2,2,1,1,1]).k_irreducible(4)
[3, 2, 2, 1, 1, 1]
sage: Partition([3,2,2,1,1,1]).k_irreducible(3)
[]
sage: Partition([3,3,3,2,2,2,2,2,1,1,1,1]).k_irreducible(3)
[2, 1]
```

k_skew(k)

Return the k-skew partition.

The k-skew diagram of a k-bounded partition is the skew diagram denoted $\lambda/^k$ satisfying the conditions:

1. row i of $\lambda/^k$ has length λ_i,
2. no cell in $\lambda/^k$ has hook-length exceeding k,
3. every square above the diagram of $\lambda/^k$ has hook length exceeding k.

REFERENCES:

EXAMPLES:

```
sage: p = Partition([4,3,2,2,1,1])
sage: p.k_skew(4)
[9, 5, 3, 2, 1, 1] / [5, 2, 1]
```

k_split(k)

Return the `k`-split of `self`.

EXAMPLES:

```
sage: Partition([4,3,2,1]).k_split(3)
[]
sage: Partition([4,3,2,1]).k_split(4)
[[4], [3, 2], [1]]
sage: Partition([4,3,2,1]).k_split(5)
[[4, 3], [2, 1]]
sage: Partition([4,3,2,1]).k_split(6)
[[4, 3, 2], [1]]
sage: Partition([4,3,2,1]).k_split(7)
[[4, 3, 2, 1]]
sage: Partition([4,3,2,1]).k_split(8)
[[4, 3, 2, 1]]
```

larger_lex(*rhs*)

Return `True` if `self` is larger than `rhs` in lexicographic order. Otherwise return `False`.

EXAMPLES:

```
sage: p = Partition([3,2])
sage: p.larger_lex([3,1])
True
sage: p.larger_lex([1,4])
```

```
True
sage: p.larger_lex([3,2,1])
False
sage: p.larger_lex([3])
True
sage: p.larger_lex([5])
False
sage: p.larger_lex([3,1,1,1,1,1,1,1])
True
```

leg_cells (i, j)

Return the list of the cells of the leg of cell (i, j) in `self`.

The leg of cell $c = (i, j)$ is defined to be the cells below c (in English convention).

The cell coordinates are zero-based, i. e., the northwesternmost cell is $(0, 0)$.

INPUT:

- `i, j` – two integers

OUTPUT:

A list of pairs of integers

EXAMPLES:

```
sage: Partition([4,4,3,1]).leg_cells(1,1)
[(2, 1)]
sage: Partition([4,4,3,1]).leg_cells(0,1)
[(1, 1), (2, 1)]

sage: Partition([]).leg_cells(0,0)
Traceback (most recent call last):
...
ValueError: The cell is not in the diagram
```

leg_length (i, j)

Return the length of the leg of cell (i, j) in `self`.

The leg of cell $c = (i, j)$ is defined to be the cells below c (in English convention).

The cell coordinates are zero-based, i. e., the northwesternmost cell is $(0, 0)$.

INPUT:

- `i, j` – two integers

OUTPUT:

An integer or a `ValueError`

EXAMPLES:

```
sage: p = Partition([2,2,1])
sage: p.leg_length(0, 0)
2
sage: p.leg_length(0,1)
1
sage: p.leg_length(2,0)
0
sage: Partition([3,3]).leg_length(0, 0)
1
```

```
sage: cell = [0,0]; Partition([3,3]).leg_length(*cell)
1
```

leg_lengths(*flat=False*)

Return a tableau of shape `self` with each cell filled in with its leg length. The optional boolean parameter `flat` provides the option of returning a flat list.

EXAMPLES:

```
sage: Partition([2,2,1]).leg_lengths()
[[2, 1], [1, 0], [0]]
sage: Partition([2,2,1]).leg_lengths(flat=True)
[2, 1, 1, 0, 0]
sage: Partition([3,3]).leg_lengths()
[[1, 1, 1], [0, 0, 0]]
sage: Partition([3,3]).leg_lengths(flat=True)
[1, 1, 1, 0, 0, 0]
```

length()

Return the number of parts in `self`.

EXAMPLES:

```
sage: Partition([3,2]).length()
2
sage: Partition([2,2,1]).length()
3
sage: Partition([]).length()
0
```

level()

Return the level of `self`, which is always 1.

This method exists only for compatibility with `PartitionTuples`.

EXAMPLE:

```
sage: Partition([4,3,2]).level()
1
```

lower_hook(*i*, *j*, *alpha*)

Return the lower hook length of the cell (i, j) in `self`. When `alpha = 1`, this is just the normal hook length.

The lower hook length of a cell (i, j) in a partition κ is defined by

$$h^\kappa_*(i, j) = \kappa'_j - i + 1 + \alpha(\kappa_i - j).$$

EXAMPLES:

```
sage: p = Partition([2,1])
sage: p.lower_hook(0,0,1)
3
sage: p.hook_length(0,0)
3
sage: [ p.lower_hook(i,j,x) for i,j in p.cells() ]
[x + 2, 1, 1]
```

lower_hook_lengths(*alpha*)

Return a tableau of shape `self` with the cells filled in with the lower hook lengths. When `alpha = 1`, these are just the normal hook lengths.

The lower hook length of a cell (i, j) in a partition κ is defined by

$$h_*^\kappa(i,j) = \kappa'_j - i + 1 + \alpha(\kappa_i - j).$$

EXAMPLES:

```
sage: Partition([3,2,1]).lower_hook_lengths(x)
[[2*x + 3, x + 2, 1], [x + 2, 1], [1]]
sage: Partition([3,2,1]).lower_hook_lengths(1)
[[5, 3, 1], [3, 1], [1]]
sage: Partition([3,2,1]).hook_lengths()
[[5, 3, 1], [3, 1], [1]]
```

next()

Return the partition that lexicographically follows `self`. If `self` is the last partition, then return `False`.

EXAMPLES:

```
sage: next(Partition([4]))
[3, 1]
sage: next(Partition([1,1,1,1]))
False
```

outer_rim()

Return the outer rim of `self`.

The outer rim of a partition λ is defined as the cells which do not belong to λ and which are adjacent to cells in λ.

EXAMPLES:

The outer rim of the partition $[4, 1]$ consists of the cells marked with `#` below:

```
****#
*####
##
```

```
sage: Partition([4,1]).outer_rim()
[(2, 0), (2, 1), (1, 1), (1, 2), (1, 3), (1, 4), (0, 4)]

sage: Partition([2,2,1]).outer_rim()
[(3, 0), (3, 1), (2, 1), (2, 2), (1, 2), (0, 2)]
sage: Partition([2,2]).outer_rim()
[(2, 0), (2, 1), (2, 2), (1, 2), (0, 2)]
sage: Partition([6,3,3,1,1]).outer_rim()
[(5, 0), (5, 1), (4, 1), (3, 1), (3, 2), (3, 3), (2, 3), (1, 3), (1, 4), (1, 5), (1, 6), (0,
sage: Partition([]).outer_rim()
[(0, 0)]
```

outline(*variable=x*)

Return the outline of the partition `self`.

This is a piecewise linear function, normalized so that the area under the partition `[1]` is 2.

INPUT:

- variable – a variable (default: `'x'` in the symbolic ring)

EXAMPLES:

```
sage: [Partition([5,4]).outline()(x=i) for i in range(-10,11)]
[10, 9, 8, 7, 6, 5, 6, 5, 6, 5, 4, 3, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
sage: Partition([]).outline()
abs(x)

sage: Partition([1]).outline()
abs(x + 1) + abs(x - 1) - abs(x)

sage: y=sage.symbolic.ring.var("y")
sage: Partition([6,5,1]).outline(variable=y)
abs(y + 6) - abs(y + 5) + abs(y + 4) - abs(y + 3) + abs(y - 1) - abs(y - 2) + abs(y - 3)
```

TESTS:

```
sage: integrate(Partition([1]).outline()-abs(x),(x,-10,10))
2
```

outside_corners()

Return a list of the outside corners of the partition `self`.

An outside corner (also called a cocorner) of a partition λ is a cell on $\mathbf{Z}^2$ which does not belong to the Young diagram of λ but can be added to this Young diagram to still form a straight-shape Young diagram.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

EXAMPLES:

```
sage: Partition([2,2,1]).outside_corners()
[(0, 2), (2, 1), (3, 0)]
sage: Partition([2,2]).outside_corners()
[(0, 2), (2, 0)]
sage: Partition([6,3,3,1,1,1]).outside_corners()
[(0, 6), (1, 3), (3, 1), (6, 0)]
sage: Partition([]).outside_corners()
[(0, 0)]
```

outside_corners_residue(i, l)

Return a list of the outside corners of the partition `self` having `l`-residue `i`.

An outside corner (also called a cocorner) of a partition λ is a cell on $\mathbf{Z}^2$ which does not belong to the Young diagram of λ but can be added to this Young diagram to still form a straight-shape Young diagram. See `residue()` for the definition of the `l`-residue.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

EXAMPLES:

```
sage: Partition([3,2,1]).outside_corners_residue(0, 3)
[(0, 3), (3, 0)]
sage: Partition([3,2,1]).outside_corners_residue(1, 3)
[(1, 2)]
sage: Partition([3,2,1]).outside_corners_residue(2, 3)
[(2, 1)]
```

plancherel_measure()

Return the probability of `self` under the Plancherel probability measure on partitions of the same size.

This probability distribution comes from the uniform distribution on permutations via the Robinson-Schensted correspondence.

See Wikipedia article Plancherel_measure and `Partitions_n.random_element_plancherel()`.

EXAMPLES:

```
sage: Partition([]).plancherel_measure()
1
sage: Partition([1]).plancherel_measure()
1
sage: Partition([2]).plancherel_measure()
1/2
sage: [mu.plancherel_measure() for mu in Partitions(3)]
[1/6, 2/3, 1/6]
sage: Partition([5,4]).plancherel_measure()
7/1440
```

TESTS:

```
sage: all(sum(mu.plancherel_measure() for mu in Partitions(n))==1 for n in range(10))
True
```

power(k)

Return the cycle type of the k-th power of any permutation with cycle type `self` (thus describes the powermap of symmetric groups).

Equivalent to GAP's `PowerPartition`.

EXAMPLES:

```
sage: p = Partition([5,3])
sage: p.power(1)
[5, 3]
sage: p.power(2)
[5, 3]
sage: p.power(3)
[5, 1, 1, 1]
sage: p.power(4)
[5, 3]
```

Now let us compare this to the power map on S_8:

```
sage: G = SymmetricGroup(8)
sage: g = G([(1,2,3,4,5),(6,7,8)])
sage: g
(1,2,3,4,5)(6,7,8)
sage: g^2
(1,3,5,2,4)(6,8,7)
sage: g^3
(1,4,2,5,3)
sage: g^4
(1,5,4,3,2)(6,7,8)

sage: Partition([3,2,1]).power(3)
[2, 1, 1, 1, 1]
```

pp()

Prints the Ferrers diagram.

See `ferrers_diagram()` for more on the Ferrers diagram.

EXAMPLES:

```
sage: Partition([5,5,2,1]).pp()
*****
*****
```

```
**
*
sage: Partitions.global_options(convention='French')
sage: Partition([5,5,2,1]).pp()
*
**
*****
*****
sage: Partitions.global_options.reset()
```

quotient(*length*)

Return the quotient of the partition – in the literature the quotient is commonly referred to as the k-quotient, p-quotient, r-quotient, ...

The r-quotient of a partition λ is a list of r partitions (labelled from 0 to $r-1$), constructed in the following way. Label each cell in the Young diagram of λ with its content modulo r. Let R_i be the set of rows ending in a cell labelled i, and C_i be the set of columns ending in a cell labelled i. Then the j-th component of the quotient of λ is the partition defined by intersecting R_j with C_{j+1}. (See Theorem 2.7.37 in [JamesKerber].)

REFERENCES:

EXAMPLES:

```
sage: Partition([7,7,5,3,3,3,1]).quotient(3)
([2], [1], [2, 2, 2])
```

TESTS:

```
sage: Partition([8,7,7,4,1,1,1,1,1]).quotient(3)
([2, 1], [2, 2], [2])
sage: Partition([10,8,7,7]).quotient(4)
([2], [3], [2], [1])
sage: Partition([6,3,3]).quotient(3)
([1], [1], [2])
sage: Partition([3,3,3,2,1]).quotient(3)
([1], [1, 1], [1])
sage: Partition([6,6,6,3,3,3]).quotient(3)
([2, 1], [2, 1], [2, 1])
sage: Partition([21,15,15,9,6,6,6,3,3]).quotient(3)
([5, 2, 1], [5, 2, 1], [7, 3, 2])
sage: Partition([21,15,15,9,6,6,3,3]).quotient(3)
([5, 2], [5, 2, 1], [7, 3, 1])
sage: Partition([14,12,11,10,10,10,10,9,6,4,3,3,2,1]).quotient(5)
([3, 3], [2, 2, 1], [], [3, 3, 3], [1])

sage: all(p == Partition(core=p.core(k), quotient=p.quotient(k))
....:     for i in range(10) for p in Partitions(i)
....:     for k in range(1,6))
True
```

reading_tableau()

Return the RSK recording tableau of the reading word of the (standard) tableau T labeled down (in English convention) each column to the shape of `self`.

For an example of the tableau T, consider the partition $\lambda = (3, 2, 1)$, then we have:

```
1 4 6
2 5
3
```

For more, see RSK().

EXAMPLES:

```
sage: Partition([3,2,1]).reading_tableau()
[[1, 3, 6], [2, 5], [4]]
```

removable_cells()

Return a list of the corners of the partition self.

A corner of a partition λ is a cell of the Young diagram of λ which can be removed from the Young diagram while still leaving a straight shape behind.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

EXAMPLES:

```
sage: Partition([3,2,1]).corners()
[(0, 2), (1, 1), (2, 0)]
sage: Partition([3,3,1]).corners()
[(1, 2), (2, 0)]
sage: Partition([]).corners()
[]
```

removable_cells_residue(i, l)

Return a list of the corners of the partition self having l-residue i.

A corner of a partition λ is a cell of the Young diagram of λ which can be removed from the Young diagram while still leaving a straight shape behind. See residue() for the definition of the l-residue.

The entries of the list returned are pairs of the form (i, j), where i and j are the coordinates of the respective corner. The coordinates are counted from 0.

EXAMPLES:

```
sage: Partition([3,2,1]).corners_residue(0, 3)
[(1, 1)]
sage: Partition([3,2,1]).corners_residue(1, 3)
[(2, 0)]
sage: Partition([3,2,1]).corners_residue(2, 3)
[(0, 2)]
```

remove_cell(i, *j=None*)

Return the partition obtained by removing a cell at the end of row i of self.

EXAMPLES:

```
sage: Partition([2,2]).remove_cell(1)
[2, 1]
sage: Partition([2,2,1]).remove_cell(2)
[2, 2]
sage: #Partition([2,2]).remove_cell(0)

sage: Partition([2,2]).remove_cell(1,1)
[2, 1]
sage: #Partition([2,2]).remove_cell(1,0)
```

remove_horizontal_border_strip(k)

Return the partitions obtained from self by removing an horizontal border strip of length k.

EXAMPLES:

```
sage: Partition([5,3,1]).remove_horizontal_border_strip(0).list()
[[5, 3, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(1).list()
[[5, 3], [5, 2, 1], [4, 3, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(2).list()
[[5, 2], [5, 1, 1], [4, 3], [4, 2, 1], [3, 3, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(3).list()
[[5, 1], [4, 2], [4, 1, 1], [3, 3], [3, 2, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(4).list()
[[4, 1], [3, 2], [3, 1, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(5).list()
[[3, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(6).list()
[]
```

The result is returned as an instance of `Partitions_with_constraints`:

```
sage: Partition([5,3,1]).remove_horizontal_border_strip(5)
The subpartitions of [5, 3, 1] obtained by removing an horizontal border strip of length 5
```

TESTS:

```
sage: Partition([3,2,2]).remove_horizontal_border_strip(2).list()
[[3, 2], [2, 2, 1]]
sage: Partition([3,2,2]).remove_horizontal_border_strip(2).first().parent()
The subpartitions of [3, 2, 2] obtained by removing an horizontal border strip of length 2
sage: Partition([]).remove_horizontal_border_strip(0).list()
[[]]
sage: Partition([]).remove_horizontal_border_strip(6).list()
[]
```

residue(r, c, l)

Return the `l`-residue of the cell at row `r` and column `c`.

The ℓ-residue of a cell is $c - r$ modulo ℓ.

This does not strictly depend upon the partition, however, this method is included because it is often useful in the context of partitions.

EXAMPLES:

```
sage: Partition([2,1]).residue(1, 0, 3)
2
```

rim()

Return the rim of `self`.

The rim of a partition λ is defined as the cells which belong to λ and which are adjacent to cells not in λ.

EXAMPLES:

The rim of the partition $[5, 5, 2, 1]$ consists of the cells marked with # below:

```
****#
*####
##
#
```

```
sage: Partition([5,5,2,1]).rim()
[(3, 0), (2, 0), (2, 1), (1, 1), (1, 2), (1, 3), (1, 4), (0, 4)]
```

```
sage: Partition([2,2,1]).rim()
```

```
[(2, 0), (1, 0), (1, 1), (0, 1)]
sage: Partition([2,2]).rim()
[(1, 0), (1, 1), (0, 1)]
sage: Partition([6,3,3,1,1]).rim()
[(4, 0), (3, 0), (2, 0), (2, 1), (2, 2), (1, 2), (0, 2), (0, 3), (0, 4), (0, 5)]
sage: Partition([]).rim()
[]
```

sign()

Return the sign of any permutation with cycle type `self`.

This function corresponds to a homomorphism from the symmetric group S_n into the cyclic group of order 2, whose kernel is exactly the alternating group A_n. Partitions of sign 1 are called even partitions while partitions of sign -1 are called odd.

EXAMPLES:

```
sage: Partition([5,3]).sign()
1
sage: Partition([5,2]).sign()
-1
```

Zolotarev's lemma states that the Legendre symbol $\left(\frac{a}{p}\right)$ for an integer $a \pmod p$ (p a prime number), can be computed as sign(p_a), where sign denotes the sign of a permutation and p_a the permutation of the residue classes $\pmod p$ induced by modular multiplication by a, provided p does not divide a.

We verify this in some examples.

```
sage: F = GF(11)
sage: a = F.multiplicative_generator();a
2
sage: plist = [int(a*F(x)) for x in range(1,11)]; plist
[2, 4, 6, 8, 10, 1, 3, 5, 7, 9]
```

This corresponds to the permutation (1, 2, 4, 8, 5, 10, 9, 7, 3, 6) (acting the set $\{1, 2, ..., 10\}$) and to the partition [10].

```
sage: p = PermutationGroupElement('(1, 2, 4, 8, 5, 10, 9, 7, 3, 6)')
sage: p.sign()
-1
sage: Partition([10]).sign()
-1
sage: kronecker_symbol(11,2)
-1
```

Now replace 2 by 3:

```
sage: plist = [int(F(3*x)) for x in range(1,11)]; plist
[3, 6, 9, 1, 4, 7, 10, 2, 5, 8]
sage: range(1,11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: p = PermutationGroupElement('(3,4,8,7,9)')
sage: p.sign()
1
sage: kronecker_symbol(3,11)
1
sage: Partition([5,1,1,1,1,1]).sign()
1
```

In both cases, Zolotarev holds.

REFERENCES:

Wikipedia article Zolotarev's_lemma

size()

Return the size of `self`.

EXAMPLES:

```
sage: Partition([2,2]).size()
4
sage: Partition([3,2,1]).size()
6
```

standard_tableaux()

Return the `standard tableaux` of this shape.

EXAMPLE:

```
sage: Partition([3,2,2,1]).standard_tableaux()
Standard tableaux of shape [3, 2, 2, 1]
```

suter_diagonal_slide(*n*, *exp=1*)

Return the image of `self` in Y_n under Suter's diagonal slide σ_n, where the notations used are those defined in [Sut2002].

The set Y_n is defined as the set of all partitions λ such that the hook length of the $(0,0)$-cell (i.e. the northwestern most cell in English notation) of λ is less than n, including the empty partition.

The map σ_n sends a partition (with non-zero entries) $(\lambda_1, \lambda_2, \ldots, \lambda_m) \in Y_n$ to the partition $(\lambda_2+1, \lambda_3+1, \ldots, \lambda_m+1, \underbrace{1, 1, \ldots, 1}_{n-m-\lambda_1 \text{ ones}})$. In other words, it pads the partition with trailing zeroes until it has length $n-\lambda_1$, then removes its first part, and finally adds 1 to each part.

By Theorem 2.1 of [Sut2002], the dihedral group D_n with $2n$ elements acts on Y_n by letting the primitive rotation act as σ_n and the reflection act as conjugation of partitions (`conjugate()`). This action is faithful if $n \geq 3$.

INPUT:

- `n` – nonnegative integer
- `exp` – (default: 1) how many times σ_n should be applied

OUTPUT:

The result of applying Suter's diagonal slide σ_n to `self`, assuming that `self` lies in Y_n. If the optional argument `exp` is set, then the slide σ_n is applied not just once, but `exp` times (note that `exp` is allowed to be negative, since the slide has finite order).

EXAMPLES:

```
sage: Partition([5,4,1]).suter_diagonal_slide(8)
[5, 2]
sage: Partition([5,4,1]).suter_diagonal_slide(9)
[5, 2, 1]
sage: Partition([]).suter_diagonal_slide(7)
[1, 1, 1, 1, 1, 1]
sage: Partition([]).suter_diagonal_slide(1)
[]
sage: Partition([]).suter_diagonal_slide(7, exp=-1)
[6]
sage: Partition([]).suter_diagonal_slide(1, exp=-1)
[]
```

```
sage: P7 = Partitions(7)
sage: all( p == p.suter_diagonal_slide(9, exp=-1).suter_diagonal_slide(9)
....:      for p in P7 )
True
sage: all( p == p.suter_diagonal_slide(9, exp=3)
....:            .suter_diagonal_slide(9, exp=3)
....:            .suter_diagonal_slide(9, exp=3)
....:      for p in P7 )
True
sage: all( p == p.suter_diagonal_slide(9, exp=6)
....:            .suter_diagonal_slide(9, exp=6)
....:            .suter_diagonal_slide(9, exp=6)
....:      for p in P7 )
True
sage: all( p == p.suter_diagonal_slide(9, exp=-1)
....:            .suter_diagonal_slide(9, exp=1)
....:      for p in P7 )
True
```

Check of the assertion in [Sut2002] that $\sigma_n(\sigma_n(\lambda')') = \lambda$:

```
sage: all( p.suter_diagonal_slide(8).conjugate()
....:      == p.conjugate().suter_diagonal_slide(8, exp=-1)
....:      for p in P7 )
True
```

Check of Claim 1 in [Sut2002]:

```
sage: P5 = Partitions(5)
sage: all( all( (p.suter_diagonal_slide(6) in q.suter_diagonal_slide(6).down())
....:           or (q.suter_diagonal_slide(6) in p.suter_diagonal_slide(6).down())
....:           for p in q.down() )
....:      for q in P5 )
True
```

TESTS:

Check for `exp = 0`:

```
sage: P = Partitions(4)
sage: all(p == p.suter_diagonal_slide(7, 0) for p in P)
True
```

Check for invalid input:

```
sage: p = Partition([2,1])
sage: p.hook_length(0, 0)
3
sage: p.suter_diagonal_slide(2)
Traceback (most recent call last):
...
ValueError: the hook length must be less than n
```

REFERENCES:

t_completion(t)

Return the `t`-completion of the partition `self`.

If $\lambda = (\lambda_1, \lambda_2, \lambda_3, \ldots)$ is a partition and t is an integer greater or equal to $|\lambda| + \lambda_1$, then the *t-completion of λ* is defined as the partition $(t - |\lambda|, \lambda_1, \lambda_2, \lambda_3, \ldots)$ of t. This partition is denoted by $\lambda[t]$ in [BOR09], by $\lambda_{[t]}$ in [BdVO12], and by $\lambda(t)$ in [CO10].

REFERENCES:

EXAMPLES:

```
sage: Partition([]).t_completion(0)
[]
sage: Partition([]).t_completion(1)
[1]
sage: Partition([]).t_completion(2)
[2]
sage: Partition([]).t_completion(3)
[3]
sage: Partition([2, 1]).t_completion(5)
[2, 2, 1]
sage: Partition([2, 1]).t_completion(6)
[3, 2, 1]
sage: Partition([4, 2, 2, 1]).t_completion(13)
[4, 4, 2, 2, 1]
sage: Partition([4, 2, 2, 1]).t_completion(19)
[10, 4, 2, 2, 1]
sage: Partition([4, 2, 2, 1]).t_completion(10)
Traceback (most recent call last):
...
ValueError: 10-completion is not defined
sage: Partition([4, 2, 2, 1]).t_completion(5)
Traceback (most recent call last):
...
ValueError: 5-completion is not defined
```

to_core(k)

Maps the k-bounded partition `self` to its corresponding $k+1$-core.

See also `k_skew()`.

EXAMPLES:

```
sage: p = Partition([4,3,2,2,1,1])
sage: c = p.to_core(4); c
[9, 5, 3, 2, 1, 1]
sage: type(c)
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
sage: c.to_bounded_partition() == p
True
```

to_dyck_word(*n=None*)

Return the `n`-Dyck word whose corresponding partition is `self` (or, if `n` is not specified, the n-Dyck word with smallest n to satisfy this property).

If w is an n-Dyck word (that is, a Dyck word with n open symbols and n close symbols), then the Dyck path corresponding to w can be regarded as a lattice path in the northeastern half of an $n \times n$-square. The region to the northeast of this Dyck path can be regarded as a partition. It is called the partition corresponding to the Dyck word w. (See `to_partition()`.)

For every partition λ and every nonnegative integer n, there exists at most one n-Dyck word w such that the partition corresponding to w is λ (in fact, such w exists if and only if $\lambda_i + i \leq n$ for every i, where λ is written in the form $(\lambda_1, \lambda_2, \ldots, \lambda_k)$ with $\lambda_k > 0$). This method computes this w for a given λ and n. If n is not specified, this method computes the w for the smallest possible n for which such an w exists. (The minimality of n means that the partition demarcated by the Dyck path touches the diagonal.)

EXAMPLES:

```
sage: Partition([2,2]).to_dyck_word()
[1, 1, 0, 0, 1, 1, 0, 0]
sage: Partition([2,2]).to_dyck_word(4)
[1, 1, 0, 0, 1, 1, 0, 0]
sage: Partition([2,2]).to_dyck_word(5)
[1, 1, 1, 0, 0, 1, 1, 0, 0, 0]
sage: Partition([6,3,1]).to_dyck_word()
[1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0]
sage: Partition([]).to_dyck_word()
[]
sage: Partition([]).to_dyck_word(3)
[1, 1, 1, 0, 0, 0]
```

The partition corresponding to `self.dyck_word()` is `self` indeed:

```
sage: all( p.to_dyck_word().to_partition() == p
....:      for p in Partitions(5) )
True
```

to_exp(*k=0*)

Return a list of the multiplicities of the parts of a partition. Use the optional parameter `k` to get a return list of length at least `k`.

EXAMPLES:

```
sage: Partition([3,2,2,1]).to_exp()
[1, 2, 1]
sage: Partition([3,2,2,1]).to_exp(5)
[1, 2, 1, 0, 0]
```

TESTS:

```
sage: [parent(x) for x in Partition([3,2,2,1]).to_exp(5)]
[Integer Ring, Integer Ring, Integer Ring, Integer Ring, Integer Ring]
```

to_exp_dict()

Return a dictionary containing the multiplicities of the parts of `self`.

EXAMPLES:

```
sage: p = Partition([4,2,2,1])
sage: d = p.to_exp_dict()
sage: d[4]
1
sage: d[2]
2
sage: d[1]
1
sage: 5 in d
False
```

to_list()

Return `self` as a list.

EXAMPLES:

```
sage: p = Partition([2,1]).to_list(); p
[2, 1]
sage: type(p)
<type 'list'>
```

TESTS:

```
sage: p = Partition([2,1])
sage: pl = p.to_list()
sage: pl[0] = 0; p
[2, 1]
```

top_garnir_tableau(*e*, *cell*)

Return the most dominant *standard* tableau which dominates the corresponding Garnir tableau and has the same `e`-residue.

The Garnir tableau play an important role in integral and non-semisimple representation theory because they determine the "straightening" rules for the Specht modules. The *top Garnir tableaux* arise in the graded representation theory of the symmetric groups and higher level Hecke algebras. They were introduced in [KMR].

If the Garnir node is `cell=(r,c)` and m and M are the entries in the cells `(r,c)` and `(r+1,c)`, respectively, in the initial tableau then the top `e`-Garnir tableau is obtained by inserting the numbers $m, m+1, \ldots, M$ in order from left to right first in the cells in row `r+1` which are not in the `e`-Garnir belt, then in the cell in rows `r` and `r+1` which are in the Garnir belt and then, finally, in the remaining cells in row `r` which are not in the Garnir belt. All other entries in the tableau remain unchanged.

If `e = 0`, or if there are no `e`-bricks in either row `r` or `r+1`, then the top Garnir tableau is the corresponding Garnir tableau.

EXAMPLES:

```
sage: Partition([5,4,3,2]).top_garnir_tableau(2,(0,2)).pp()
   1  2  4  5  8
   3  6  7  9
  10 11 12
  13 14
sage: Partition([5,4,3,2]).top_garnir_tableau(3,(0,2)).pp()
   1  2  3  4  5
   6  7  8  9
  10 11 12
  13 14
sage: Partition([5,4,3,2]).top_garnir_tableau(4,(0,2)).pp()
   1  2  6  7  8
   3  4  5  9
  10 11 12
  13 14
sage: Partition([5,4,3,2]).top_garnir_tableau(0,(0,2)).pp()
   1  2  6  7  8
   3  4  5  9
  10 11 12
  13 14
```

TESTS:

```
sage: Partition([5,4,3,2]).top_garnir_tableau(0,(3,2)).pp()
Traceback (most recent call last):
...
ValueError: (4,2)=(row+1,col) must be inside the diagram
```

REFERENCE:

- [KMR]

up()

Returns a generator for partitions that can be obtained from `self` by adding a cell.

EXAMPLES:

```
sage: [p for p in Partition([2,1,1]).up()]
[[3, 1, 1], [2, 2, 1], [2, 1, 1, 1]]
sage: [p for p in Partition([3,2]).up()]
[[4, 2], [3, 3], [3, 2, 1]]
sage: [p for p in Partition([]).up()]
[[1]]
```

up_list()

Return a list of the partitions that can be formed from `self` by adding a cell.

EXAMPLES:

```
sage: Partition([2,1,1]).up_list()
[[3, 1, 1], [2, 2, 1], [2, 1, 1, 1]]
sage: Partition([3,2]).up_list()
[[4, 2], [3, 3], [3, 2, 1]]
sage: Partition([]).up_list()
[[1]]
```

upper_hook(*i*, *j*, *alpha*)

Return the upper hook length of the cell (i, j) in `self`. When `alpha = 1`, this is just the normal hook length.

The upper hook length of a cell (i, j) in a partition κ is defined by

$$h^*_\kappa(i,j) = \kappa'_j - i + \alpha(\kappa_i - j + 1).$$

EXAMPLES:

```
sage: p = Partition([2,1])
sage: p.upper_hook(0,0,1)
3
sage: p.hook_length(0,0)
3
sage: [ p.upper_hook(i,j,x) for i,j in p.cells() ]
[2*x + 1, x, x]
```

upper_hook_lengths(*alpha*)

Return a tableau of shape `self` with the cells filled in with the upper hook lengths. When `alpha = 1`, these are just the normal hook lengths.

The upper hook length of a cell (i, j) in a partition κ is defined by

$$h^*_\kappa(i,j) = \kappa'_j - i + \alpha(\kappa_i - j + 1).$$

EXAMPLES:

```
sage: Partition([3,2,1]).upper_hook_lengths(x)
[[3*x + 2, 2*x + 1, x], [2*x + 1, x], [x]]
sage: Partition([3,2,1]).upper_hook_lengths(1)
[[5, 3, 1], [3, 1], [1]]
sage: Partition([3,2,1]).hook_lengths()
[[5, 3, 1], [3, 1], [1]]
```

weighted_size()

Return the weighted size of `self`.

The weighted size of a partition λ is

$$\sum_i i \cdot \lambda_i,$$

where $\lambda = (\lambda_0, \lambda_1, \lambda_2, \cdots)$.

This also the sum of the leg length of every cell in λ, or

$$\sum_i \binom{\lambda'_i}{2}$$

where λ' is the conjugate partition of λ.

EXAMPLES:

```
sage: Partition([2,2]).weighted_size()
2
sage: Partition([3,3,3]).weighted_size()
9
sage: Partition([5,2]).weighted_size()
2
sage: Partition([]).weighted_size()
0
```

young_subgroup()

Return the corresponding Young, or parabolic, subgroup of the symmetric group.

The Young subgroup of a partition $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_\ell)$ of n is the group:

$$S_{\lambda_1} \times S_{\lambda_2} \times \cdots \times S_{\lambda_\ell}$$

embedded into S_n in the standard way (i.e., the S_{λ_i} factor acts on the numbers from $\lambda_1+\lambda_2+\cdots+\lambda_{i-1}+1$ to $\lambda_1 + \lambda_2 + \cdots + \lambda_i$).

EXAMPLES:

```
sage: Partition([4,2]).young_subgroup()
Permutation Group with generators [(), (5,6), (3,4), (2,3), (1,2)]
```

young_subgroup_generators()

Return an indexing set for the generators of the corresponding Young subgroup. Here the generators correspond to the simple adjacent transpositions $s_i = (i\ i+1)$.

EXAMPLES:

```
sage: Partition([4,2]).young_subgroup_generators()
[1, 2, 3, 5]
sage: Partition([1,1,1]).young_subgroup_generators()
[]
sage: Partition([2,2]).young_subgroup_generators()
[1, 3]
```

zero_one_sequence()

Compute the finite $0-1$ sequence of the partition.

The full $0-1$ sequence is the sequence (infinite in both directions) indicating the steps taken when following the outer rim of the diagram of the partition. We use the convention that in English convention, a 1 corresponds to an East step, and a 0 corresponds to a North step.

Note that every full $0-1$ sequence starts with infinitely many 0's and ends with infinitely many 1's.

One place where these arise is in the affine symmetric group where one takes an affine permutation w and every i such that $w(i) \leq 0$ corresponds to a 1 and $w(i) > 0$ corresponds to a 0. See pages 24-25 of [LLMMSZ13] for connections to affine Grassmannian elements (note there they use the French convention for their partitions).

These are also known as **path sequences**, **Maya diagrams**, **plus-minus diagrams**, **Comet code** [Sta-EC2], among others.

OUTPUT:

The finite $0-1$ sequence is obtained from the full $0-1$ sequence by omitting all heading 0's and trailing 1's. The output sequence is finite, starts with a 1 and ends with a 0 (unless it is empty, for the empty partition). Its length is the sum of the first part of the partition with the length of the partition.

REFERENCES:

EXAMPLES:

```
sage: Partition([5,4]).zero_one_sequence()
[1, 1, 1, 1, 0, 1, 0]
sage: Partition([]).zero_one_sequence()
[]
sage: Partition([2]).zero_one_sequence()
[1, 1, 0]
```

TESTS:

```
sage: all(Partitions().from_zero_one(mu.zero_one_sequence()) == mu for n in range(10) for mu
True
```

class sage.combinat.partition.**Partitions**(*is_infinite=False*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

`Partitions(n, **kwargs)` returns the combinatorial class of integer partitions of n subject to the constraints given by the keywords.

Valid keywords are: `starting`, `ending`, `min_part`, `max_part`, `max_length`, `min_length`, `length`, `max_slope`, `min_slope`, `inner`, `outer`, `parts_in` and `regular`. They have the following meanings:

- `starting=p` specifies that the partitions should all be less than or equal to p in lex order. This argument cannot be combined with any other (see trac ticket #15467).
- `ending=p` specifies that the partitions should all be greater than or equal to p in lex order. This argument cannot be combined with any other (see trac ticket #15467).
- `length=k` specifies that the partitions have exactly k parts.
- `min_length=k` specifies that the partitions have at least k parts.
- `min_part=k` specifies that all parts of the partitions are at least k.
- `inner=p` specifies that the partitions must contain the partition p.
- `outer=p` specifies that the partitions be contained inside the partition p.
- `min_slope=k` specifies that the partitions have slope at least k; the slope at position i is the difference between the $(i+1)$-th part and the i-th part.
- `parts_in=S` specifies that the partitions have parts in the set S, which can be any sequence of pairwise distinct positive integers. This argument cannot be combined with any other (see trac ticket #15467).
- `regular=ell` specifies that the partitions are ℓ-regular, and can only be combined with the `max_length` or `max_part`, but not both, keywords if n is not specified

The `max_*` versions, along with `inner` and `ending`, work analogously.

Right now, the `parts_in`, `starting`, `ending`, and `regular` keyword arguments are mutually exclusive, both of each other and of other keyword arguments. If you specify, say, `parts_in`, all other keyword arguments will be ignored; `starting`, `ending`, and `regular` work the same way.

EXAMPLES:

If no arguments are passed, then the combinatorial class of all integer partitions is returned:

```
sage: Partitions()
Partitions
sage: [2,1] in Partitions()
True
```

If an integer n is passed, then the combinatorial class of integer partitions of n is returned:

```
sage: Partitions(3)
Partitions of the integer 3
sage: Partitions(3).list()
[[3], [2, 1], [1, 1, 1]]
```

If `starting=p` is passed, then the combinatorial class of partitions greater than or equal to p in lexicographic order is returned:

```
sage: Partitions(3, starting=[2,1])
Partitions of the integer 3 starting with [2, 1]
sage: Partitions(3, starting=[2,1]).list()
[[2, 1], [1, 1, 1]]
```

If `ending=p` is passed, then the combinatorial class of partitions at most p in lexicographic order is returned:

```
sage: Partitions(3, ending=[2,1])
Partitions of the integer 3 ending with [2, 1]
sage: Partitions(3, ending=[2,1]).list()
[[3], [2, 1]]
```

Using `max_slope=-1` yields partitions into distinct parts – each part differs from the next by at least 1. Use a different `max_slope` to get parts that differ by, say, 2:

```
sage: Partitions(7, max_slope=-1).list()
[[7], [6, 1], [5, 2], [4, 3], [4, 2, 1]]
sage: Partitions(15, max_slope=-1).cardinality()
27
```

The number of partitions of n into odd parts equals the number of partitions into distinct parts. Let's test that for n from 10 to 20:

```
sage: test = lambda n: Partitions(n, max_slope=-1).cardinality() == Partitions(n, parts_in=[1,3.
sage: all(test(n) for n in [10..20])
True
```

The number of partitions of n into distinct parts that differ by at least 2 equals the number of partitions into parts that equal 1 or 4 modulo 5; this is one of the Rogers-Ramanujan identities:

```
sage: test = lambda n: Partitions(n, max_slope=-2).cardinality() == Partitions(n, parts_in=([1,6
sage: all(test(n) for n in [10..20])
True
```

Here are some more examples illustrating `min_part`, `max_part`, and `length`:

```
sage: Partitions(5,min_part=2)
Partitions of the integer 5 satisfying constraints min_part=2
sage: Partitions(5,min_part=2).list()
[[5], [3, 2]]
```

```
sage: Partitions(3,max_length=2).list()
[[3], [2, 1]]

sage: Partitions(10, min_part=2, length=3).list()
[[6, 2, 2], [5, 3, 2], [4, 4, 2], [4, 3, 3]]
```

Some examples using the `regular` keyword:

```
sage: Partitions(regular=4)
4-Regular Partitions
sage: Partitions(regular=4, max_length=3)
4-Regular Partitions with max length 3
sage: Partitions(regular=4, max_part=3)
4-Regular 3-Bounded Partitions
sage: Partitions(3, regular=4)
4-Regular Partitions of the integer 3
```

Here are some further examples using various constraints:

```
sage: [x for x in Partitions(4)]
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
sage: [x for x in Partitions(4, length=2)]
[[3, 1], [2, 2]]
sage: [x for x in Partitions(4, min_length=2)]
[[3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
sage: [x for x in Partitions(4, max_length=2)]
[[4], [3, 1], [2, 2]]
sage: [x for x in Partitions(4, min_length=2, max_length=2)]
[[3, 1], [2, 2]]
sage: [x for x in Partitions(4, max_part=2)]
[[2, 2], [2, 1, 1], [1, 1, 1, 1]]
sage: [x for x in Partitions(4, min_part=2)]
[[4], [2, 2]]
sage: [x for x in Partitions(4, outer=[3,1,1])]
[[3, 1], [2, 1, 1]]
sage: [x for x in Partitions(4, outer=[infinity, 1, 1])]
[[4], [3, 1], [2, 1, 1]]
sage: [x for x in Partitions(4, inner=[1,1,1])]
[[2, 1, 1], [1, 1, 1, 1]]
sage: [x for x in Partitions(4, max_slope=-1)]
[[4], [3, 1]]
sage: [x for x in Partitions(4, min_slope=-1)]
[[4], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
sage: [x for x in Partitions(11, max_slope=-1, min_slope=-3, min_length=2, max_length=4)]
[[7, 4], [6, 5], [6, 4, 1], [6, 3, 2], [5, 4, 2], [5, 3, 2, 1]]
sage: [x for x in Partitions(11, max_slope=-1, min_slope=-3, min_length=2, max_length=4, outer=[
[[6, 5], [6, 4, 1], [6, 3, 2], [5, 4, 2]]
```

Note that if you specify `min_part=0`, then it will treat the minimum part as being 1 (see trac ticket #13605):

```
sage: [x for x in Partitions(4, length=3, min_part=0)]
[[2, 1, 1]]
sage: [x for x in Partitions(4, min_length=3, min_part=0)]
[[2, 1, 1], [1, 1, 1, 1]]
```

Except for very special cases, counting is done by brute force iteration through all the partitions. However the iteration itself has a reasonable complexity (see `IntegerListsLex`), which allows for manipulating large partitions:

```
sage: Partitions(1000, max_length=1).list()
[[1000]]
```

In particular, getting the first element is also constant time:

```
sage: Partitions(30, max_part=29).first()
[29, 1]
```

TESTS:

```
sage: TestSuite(Partitions(0)).run()
sage: TestSuite(Partitions(5)).run()
sage: TestSuite(Partitions(5, min_part=2)).run()

sage: repr( Partitions(5, min_part=2) )
'Partitions of the integer 5 satisfying constraints min_part=2'

sage: P = Partitions(5, min_part=2)
sage: P.first().parent()
Partitions...
sage: [2,1] in P
False
sage: [2,2,1] in P
False
sage: [3,2] in P
True

sage: Partitions(5, inner=[2,1], min_length=3).list()
[[3, 1, 1], [2, 2, 1], [2, 1, 1, 1]]
sage: Partitions(5, inner=Partition([2,2]), min_length=3).list()
[[2, 2, 1]]
sage: Partitions(7, inner=(2, 2), min_length=3).list()
[[4, 2, 1], [3, 3, 1], [3, 2, 2], [3, 2, 1, 1], [2, 2, 2, 1], [2, 2, 1, 1, 1]]
sage: Partitions(5, inner=[2,0,0,0,0,0]).list()
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1]]
sage: Partitions(6, length=2, max_slope=-1).list()
[[5, 1], [4, 2]]

sage: Partitions(length=2, max_slope=-1).list()
Traceback (most recent call last):
...
ValueError: the size must be specified with any keyword argument

sage: Partitions(max_part = 3)
3-Bounded Partitions
```

Check that trac ticket #14145 has been fixed:

```
sage: 1 in Partitions()
False
```

Check trac ticket #15467:

```
sage: Partitions(5,parts_in=[1,2,3,4], length=4)
Traceback (most recent call last):
...
ValueError: The parameters 'parts_in', 'starting' and 'ending' cannot be combined with anything
sage: Partitions(5,starting=[3,2], length=2)
Traceback (most recent call last):
...
```

```
ValueError: The parameters 'parts_in', 'starting' and 'ending' cannot be combined with anything
sage: Partitions(5,ending=[3,2], length=2)
Traceback (most recent call last):
...
ValueError: The parameters 'parts_in', 'starting' and 'ending' cannot be combined with anything
sage: Partitions(NN, length=2)
Traceback (most recent call last):
...
ValueError: the size must be specified with any keyword argument
sage: Partitions(('la','la','laaaa'), max_part=8)
Traceback (most recent call last):
...
ValueError: n must be an integer or be equal to one of None, NN, NonNegativeIntegers()
```

Check that calling `Partitions` with `outer=a` no longer mutates `a` (trac ticket #16234):

```
sage: a = [4,3,2,1,1,1,1]
sage: for p in Partitions(8, outer=a, min_slope=-1):
....:     print p
[3, 3, 2]
[3, 2, 2, 1]
[3, 2, 1, 1, 1]
[2, 2, 2, 1, 1]
[2, 2, 1, 1, 1, 1]
[2, 1, 1, 1, 1, 1, 1]
sage: a
[4, 3, 2, 1, 1, 1, 1]
```

Check that `inner` and `outer` indeed accept a partition as argument (trac ticket #18423):

```
sage: P = Partitions(5, inner=Partition([2,1]), outer=Partition([3,2])); P
Partitions of the integer 5 satisfying constraints inner=[2, 1], outer=[3, 2]
sage: P.list()
[[3, 2]]
```

Element
alias of `Partition`

global_options(**get_value*, ***set_value*)
Sets and displays the global options for elements of the partition, skew partition, and partition tuple classes. If no parameters are set, then the function returns a copy of the options dictionary.

The `options` to partitions can be accessed as the method `Partitions.global_options` of `Partitions` and related parent classes.

OPTIONS:

- `convention` – (default: `English`) Sets the convention used for displaying tableaux and partitions
  - `English` – use the English convention
  - `French` – use the French convention
- `diagram_str` – (default: `*`) The character used for the cells when printing Ferrers diagrams
- `display` – (default: `list`) Specifies how partitions should be printed
  - `array` – alias for `diagram`
  - `compact` – alias for `compact_low`
  - `compact_high` – compact form of `exp_high`

– `compact_low` – compact form of `exp_low`

– `diagram` – as a Ferrers diagram

– `exp` – alias for `exp_low`

– `exp_high` – in exponential form (highest first)

– `exp_low` – in exponential form (lowest first)

– `ferrers_diagram` – alias for `diagram`

– `list` – displayed as a list

– `young_diagram` – alias for `diagram`

- `latex` – (default: `young_diagram`) Specifies how partitions should be latexed

  – `array` – alias for `diagram`

  – `diagram` – latex as a Ferrers diagram

  – `exp` – alias for `exp_low`

  – `exp_high` – latex as a list in exponential notation (highest first)

  – `exp_low` – as a list latex in exponential notation (lowest first)

  – `ferrers_diagram` – alias for `diagram`

  – `list` – latex as a list

  – `young_diagram` – latex as a Young diagram

- `latex_diagram_str` – (default: `\ast`) The character used for the cells when latexing Ferrers diagrams

- `notation` – alternative name for `convention`

EXAMPLES:

```
sage: P = Partition([4,2,2,1])
sage: P
[4, 2, 2, 1]
sage: Partitions.global_options(display="exp")
sage: P
1, 2^2, 4
sage: Partitions.global_options(display="exp_high")
sage: P
4, 2^2, 1
```

It is also possible to use user defined functions for the `display` and `latex` options:

```
sage: Partitions.global_options(display=lambda mu: '<%s>' % ','.join('%s'%m for m in mu._lis
<4,2,2,1>
sage: Partitions.global_options(latex=lambda mu: '\\Diagram{%s}' % ','.join('%s'%m for m in
\Diagram{4,2,2,1}
sage: Partitions.global_options(display="diagram", diagram_str="#")
sage: P
####
##
##
#
sage: Partitions.global_options(diagram_str="*", convention="french")
sage: print P.ferrers_diagram()
*
```

```
**
**
****
```

Changing the `convention` for partitions also changes the `convention` option for tableaux and vice versa:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T.pp()
  4  5
  1  2  3
sage: Tableaux.global_options(convention="english")
sage: print P.ferrers_diagram()
****
**
**
*
sage: T.pp()
  1  2  3
  4  5
sage: Partitions.global_options.reset()
```

See `GlobalOptions` for more features of these options.

subset(**args*, ***kwargs*)

Return `self` if no arguments are given, otherwise raises a `ValueError`.

EXAMPLES:

```
sage: P = Partitions(5, starting=[3,1]); P
Partitions of the integer 5 starting with [3, 1]
sage: P.subset()
Partitions of the integer 5 starting with [3, 1]
sage: P.subset(ending=[3,1])
Traceback (most recent call last):
...
ValueError: Invalid combination of arguments
```

class sage.combinat.partition.**PartitionsGreatestEQ**(*n*, *k*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.combinat.integer_lists.invlex.IntegerListsLex

The class of all (unordered) "restricted" partitions of the integer n having its greatest part equal to the integer k.

EXAMPLES:

```
sage: PartitionsGreatestEQ(10,2)
Partitions of 10 having greatest part equal to 2
sage: PartitionsGreatestEQ(10,2).list()
[[2, 2, 2, 2, 2],
 [2, 2, 2, 2, 1, 1],
 [2, 2, 2, 1, 1, 1, 1],
 [2, 2, 1, 1, 1, 1, 1, 1],
 [2, 1, 1, 1, 1, 1, 1, 1, 1]]

sage: [4,3,2,1] in PartitionsGreatestEQ(10,2)
False
sage: [2,2,2,2,2] in PartitionsGreatestEQ(10,2)
True
sage: [1]*10 in PartitionsGreatestEQ(10,2)
False
```

```
sage: PartitionsGreatestEQ(10,2).first().parent()
Partitions...
```

Element
alias of Partition

global_options(**get_value*, ***set_value*)
Sets and displays the global options for elements of the partition, skew partition, and partition tuple classes. If no parameters are set, then the function returns a copy of the options dictionary.

The options to partitions can be accessed as the method Partitions.global_options of Partitions and related parent classes.

OPTIONS:

- convention – (default: English) Sets the convention used for displaying tableaux and partitions
  - English – use the English convention
  - French – use the French convention
- diagram_str – (default: *) The character used for the cells when printing Ferrers diagrams
- display – (default: list) Specifies how partitions should be printed
  - array – alias for diagram
  - compact – alias for compact_low
  - compact_high – compact form of exp_high
  - compact_low – compact form of exp_low
  - diagram – as a Ferrers diagram
  - exp – alias for exp_low
  - exp_high – in exponential form (highest first)
  - exp_low – in exponential form (lowest first)
  - ferrers_diagram – alias for diagram
  - list – displayed as a list
  - young_diagram – alias for diagram
- latex – (default: young_diagram) Specifies how partitions should be latexed
  - array – alias for diagram
  - diagram – latex as a Ferrers diagram
  - exp – alias for exp_low
  - exp_high – latex as a list in exponential notation (highest first)
  - exp_low – as a list latex in exponential notation (lowest first)
  - ferrers_diagram – alias for diagram
  - list – latex as a list
  - young_diagram – latex as a Young diagram
- latex_diagram_str – (default: \ast) The character used for the cells when latexing Ferrers diagrams

- notation – alternative name for convention

EXAMPLES:

```
sage: P = Partition([4,2,2,1])
sage: P
[4, 2, 2, 1]
sage: Partitions.global_options(display="exp")
sage: P
1, 2^2, 4
sage: Partitions.global_options(display="exp_high")
sage: P
4, 2^2, 1
```

It is also possible to use user defined functions for the display and latex options:

```
sage: Partitions.global_options(display=lambda mu: '<%s>' % ','.join('%s'%m for m in mu._lis
<4,2,2,1>
sage: Partitions.global_options(latex=lambda mu: '\\Diagram{%s}' % ','.join('%s'%m for m in
\Diagram{4,2,2,1}
sage: Partitions.global_options(display="diagram", diagram_str="#")
sage: P
####
##
##
#
sage: Partitions.global_options(diagram_str="*", convention="french")
sage: print P.ferrers_diagram()
*
**
**
****
```

Changing the convention for partitions also changes the convention option for tableaux and vice versa:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T.pp()
  4  5
  1  2  3
sage: Tableaux.global_options(convention="english")
sage: print P.ferrers_diagram()
****
**
**
*
sage: T.pp()
  1  2  3
  4  5
sage: Partitions.global_options.reset()
```

See GlobalOptions for more features of these options.

class sage.combinat.partition.**PartitionsGreatestLE**(n, k)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.combinat.integer_lists.invlex.IntegerListsLex

The class of all (unordered) "restricted" partitions of the integer n having parts less than or equal to the integer k.

EXAMPLES:

```
sage: PartitionsGreatestLE(10,2)
Partitions of 10 having parts less than or equal to 2
sage: PartitionsGreatestLE(10,2).list()
[[2, 2, 2, 2, 2],
 [2, 2, 2, 2, 1, 1],
 [2, 2, 2, 1, 1, 1, 1],
 [2, 2, 1, 1, 1, 1, 1, 1],
 [2, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

sage: [4,3,2,1] in PartitionsGreatestLE(10,2)
False
sage: [2,2,2,2,2] in PartitionsGreatestLE(10,2)
True
sage: PartitionsGreatestLE(10,2).first().parent()
Partitions...
```

Element
alias of Partition

global_options(**get_value*, ***set_value*)
Sets and displays the global options for elements of the partition, skew partition, and partition tuple classes. If no parameters are set, then the function returns a copy of the options dictionary.

The options to partitions can be accessed as the method Partitions.global_options of Partitions and related parent classes.

OPTIONS:

- convention – (default: English) Sets the convention used for displaying tableaux and partitions
  - English – use the English convention
  - French – use the French convention
- diagram_str – (default: *) The character used for the cells when printing Ferrers diagrams
- display – (default: list) Specifies how partitions should be printed
  - array – alias for diagram
  - compact – alias for compact_low
  - compact_high – compact form of exp_high
  - compact_low – compact form of exp_low
  - diagram – as a Ferrers diagram
  - exp – alias for exp_low
  - exp_high – in exponential form (highest first)
  - exp_low – in exponential form (lowest first)
  - ferrers_diagram – alias for diagram
  - list – displayed as a list
  - young_diagram – alias for diagram
- latex – (default: young_diagram) Specifies how partitions should be latexed
  - array – alias for diagram
  - diagram – latex as a Ferrers diagram

– `exp` – alias for `exp_low`

– `exp_high` – latex as a list in exponential notation (highest first)

– `exp_low` – as a list latex in exponential notation (lowest first)

– `ferrers_diagram` – alias for `diagram`

– `list` – latex as a list

– `young_diagram` – latex as a Young diagram

- `latex_diagram_str` – (default: `\ast`) The character used for the cells when latexing Ferrers diagrams
- `notation` – alternative name for `convention`

EXAMPLES:

```
sage: P = Partition([4,2,2,1])
sage: P
[4, 2, 2, 1]
sage: Partitions.global_options(display="exp")
sage: P
1, 2^2, 4
sage: Partitions.global_options(display="exp_high")
sage: P
4, 2^2, 1
```

It is also possible to use user defined functions for the `display` and `latex` options:

```
sage: Partitions.global_options(display=lambda mu: '<%s>' % ','.join('%s'%m for m in mu._lis
<4,2,2,1>
sage: Partitions.global_options(latex=lambda mu: '\\Diagram{%s}' % ','.join('%s'%m for m in
\Diagram{4,2,2,1}
sage: Partitions.global_options(display="diagram", diagram_str="#")
sage: P
####
##
##
#
sage: Partitions.global_options(diagram_str="*", convention="french")
sage: print P.ferrers_diagram()
*
**
**
****
```

Changing the `convention` for partitions also changes the `convention` option for tableaux and vice versa:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T.pp()
  4  5
  1  2  3
sage: Tableaux.global_options(convention="english")
sage: print P.ferrers_diagram()
****
**
**
*
sage: T.pp()
  1  2  3
```

```
  4  5
sage: Partitions.global_options.reset()
```

See `GlobalOptions` for more features of these options.

class sage.combinat.partition.**PartitionsInBox**(h, w)
Bases: sage.combinat.partition.Partitions

All partitions which fit in an $h \times w$ box.

EXAMPLES:

```
sage: PartitionsInBox(2,2)
Integer partitions which fit in a 2 x 2 box
sage: PartitionsInBox(2,2).list()
[[], [1], [1, 1], [2], [2, 1], [2, 2]]
```

list()
Return a list of all the partitions inside a box of height h and width w.

EXAMPLES:

```
sage: PartitionsInBox(2,2).list()
[[], [1], [1, 1], [2], [2, 1], [2, 2]]
sage: PartitionsInBox(2,3).list()
[[], [1], [1, 1], [2], [2, 1], [2, 2], [3], [3, 1], [3, 2], [3, 3]]
```

TESTS:

Check trac ticket #10890:

```
sage: type(PartitionsInBox(0,0)[0])
<class 'sage.combinat.partition.PartitionsInBox_with_category.element_class'>
```

class sage.combinat.partition.**Partitions_all**
Bases: sage.combinat.partition.Partitions

Class of all partitions.

TESTS:

```
sage: TestSuite( sage.combinat.partition.Partitions_all() ).run()
```

from_beta_numbers(*beta*)
Return a partition corresponding to a sequence of beta numbers.

A sequence of beta numbers is a strictly increasing sequence $0 \leq b_1 < \cdots < b_k$ of non-negative integers. The corresponding partition $\mu = (\mu_k, \ldots, \mu_1)$ is given by $\mu_i = [1, i) \setminus \{b_1, \ldots, b_i\}$. This gives a bijection from the set of partitions with at most k non-zero parts to the set of strictly increasing sequences of non-negative integers of length k.

EXAMPLES:

```
sage: Partitions().from_beta_numbers([0,1,2,4,5,8])
[3, 1, 1]
sage: Partitions().from_beta_numbers([0,2,3,6])
[3, 1, 1]
```

from_core_and_quotient(*core*, *quotient*)
Returns a partition from its core and quotient.

Algorithm from mupad-combinat.

EXAMPLES:

```
sage: Partitions().from_core_and_quotient([2,1], [[2,1],[3],[1,1,1]])
[11, 5, 5, 3, 2, 2, 2]
```

TESTS:

```
sage: Partitions().from_core_and_quotient([2,1], [[2,1],[2,3,1],[1,1,1]])
Traceback (most recent call last):
...
ValueError: the quotient [[2, 1], [2, 3, 1], [1, 1, 1]] must be a tuple of partitions
```

We check that trac ticket #11412 is actually fixed:

```
sage: test = lambda x, k: x == Partition(core=x.core(k),
...                                      quotient=x.quotient(k))
sage: all(test(mu,k) for k in range(1,5)
...       for n in range(10) for mu in Partitions(n))
True
sage: test2 = lambda core, mus: (
...       Partition(core=core, quotient=mus).core(mus.level()) == core
...       and
...       Partition(core=core, quotient=mus).quotient(mus.level()) == mus)
sage: all(test2(core,mus)  # long time (5s on sage.math, 2011)
...       for k in range(1,10)
...       for n_core in range(10-k)
...       for core in Partitions(n_core)
...       if core.core(k) == core
...       for n_mus in range(10-k)
...       for mus in PartitionTuples(k,n_mus))
True
```

from_exp(*exp*)

Returns a partition from its list of multiplicities.

EXAMPLES:

```
sage: Partitions().from_exp([2,2,1])
[3, 2, 2, 1, 1]
```

from_frobenius_coordinates(*frobenius_coordinates*)

Returns a partition from a pair of sequences of Frobenius coordinates.

EXAMPLES:

```
sage: Partitions().from_frobenius_coordinates(([],[]))
[]
sage: Partitions().from_frobenius_coordinates(([0],[0]))
[1]
sage: Partitions().from_frobenius_coordinates(([1],[1]))
[2, 1]
sage: Partitions().from_frobenius_coordinates(([6,3,2],[4,1,0]))
[7, 5, 5, 1, 1]
```

from_zero_one(*seq*)

Return a partition from its $0-1$ sequence.

The full $0-1$ sequence is the sequence (infinite in both directions) indicating the steps taken when following the outer rim of the diagram of the partition. We use the convention that in English convention, a 1 corresponds to an East step, and a 0 corresponds to a North step.

Note that every full $0-1$ sequence starts with infinitely many 0's and ends with infinitely many 1's.

See also:

`Partition.zero_one_sequence()`

INPUT:

The input should be a finite sequence of 0's and 1's. The heading 0's and trailing 1's will be discarded.

EXAMPLES:

```
sage: Partitions().from_zero_one([])
[]
sage: Partitions().from_zero_one([1,0])
[1]
sage: Partitions().from_zero_one([1, 1, 1, 1, 0, 1, 0])
[5, 4]
```

Heading 0's and trailing 1's are correctly handled:

```
sage: Partitions().from_zero_one([0,0,1,1,1,1,0,1,0,1,1,1])
[5, 4]
```

TESTS:

```
sage: all(Partitions().from_zero_one(mu.zero_one_sequence()) == mu for n in range(10) for mu
True
```

subset(*size=None*, ***kwargs*)

Returns the subset of partitions of a given size and additional keyword arguments.

EXAMPLES:

```
sage: P = Partitions()
sage: P.subset(4)
Partitions of the integer 4
```

class sage.combinat.partition.**Partitions_all_bounded**(*k*)

Bases: `sage.combinat.partition.Partitions`

TESTS:

```
sage: TestSuite( sage.combinat.partition.Partitions_all_bounded(3) ).run() # long time
```

class sage.combinat.partition.**Partitions_constraints**(**args*, ***kwds*)

Bases: `sage.combinat.integer_lists.invlex.IntegerListsLex`

For unpickling old constrained `Partitions_constraints` objects created with sage <= 3.4.1. See `Partitions`.

class sage.combinat.partition.**Partitions_ending**(*n*, *ending_partition*)

Bases: `sage.combinat.partition.Partitions`

All partitions with a given ending.

first()

Return the first partition in `self`.

EXAMPLES:

```
sage: Partitions(4, ending=[1,1,1,1]).first()
[4]
```

next(*part*)

Return the next partition after `part` in `self`.

EXAMPLES:

```
sage: Partitions(4, ending=[1,1,1,1]).next(Partition([4]))
[3, 1]
sage: Partitions(4, ending=[1,1,1,1]).next(Partition([1,1,1,1])) is None
True
```

class sage.combinat.partition.**Partitions_n**(*n*)

Bases: sage.combinat.partition.Partitions

Partitions of the integer n.

TESTS:

```
sage: TestSuite( sage.combinat.partition.Partitions_n(0) ).run()
sage: TestSuite( sage.combinat.partition.Partitions_n(0) ).run()
```

cardinality(*algorithm='flint'*)

Return the number of partitions of the specified size.

INPUT:

- `algorithm` - (default: `'flint'`)
  - `'flint'` – use FLINT (currently the fastest)
  - `'bober'` – Use Jonathan Bober's implementation (*very* fast)
  - `'gap'` – use GAP (VERY *slow*)
  - `'pari'` – use PARI. Speed seems the same as GAP until n is in the thousands, in which case PARI is faster.

It is possible to associate with every partition of the integer n a conjugacy class of permutations in the symmetric group on n points and vice versa. Therefore the number of partitions p_n is the number of conjugacy classes of the symmetric group on n points.

EXAMPLES:

```
sage: v = Partitions(5).list(); v
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]
sage: len(v)
7
sage: Partitions(5).cardinality(algorithm='gap')
7
sage: Partitions(5).cardinality(algorithm='pari')
7
sage: Partitions(5).cardinality(algorithm='bober')
7
sage: number_of_partitions(5, algorithm='flint')
7
```

The input must be a nonnegative integer or a `ValueError` is raised.

```
sage: Partitions(10).cardinality()
42
sage: Partitions(3).cardinality()
3
sage: Partitions(10).cardinality()
42
sage: Partitions(3).cardinality(algorithm='pari')
3
sage: Partitions(10).cardinality(algorithm='pari')
42
```

```
sage: Partitions(40).cardinality()
37338
sage: Partitions(100).cardinality()
190569292
```

A generating function for p_n is given by the reciprocal of Euler's function:

$$\sum_{n=0}^{\infty} p_n x^n = \prod_{k=1}^{\infty} \frac{1}{1-x^k}.$$

We use Sage to verify that the first several coefficients do indeed agree:

```
sage: q = PowerSeriesRing(QQ, 'q', default_prec=9).gen()
sage: prod([(1-q^k)^(-1) for k in range(1,9)])  ## partial product of
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + 7*q^5 + 11*q^6 + 15*q^7 + 22*q^8 + O(q^9)
sage: [Partitions(k).cardinality() for k in range(2,10)]
[2, 3, 5, 7, 11, 15, 22, 30]
```

Another consistency test for n up to 500:

```
sage: len([n for n in [1..500] if Partitions(n).cardinality() != Partitions(n).cardinality(a
0
```

REFERENCES:

- Wikipedia article Partition_(number_theory)

first()

Returns the lexicographically first partition of a positive integer n. This is the partition [n].

EXAMPLES:

```
sage: Partitions(4).first()
[4]
```

last()

Return the lexicographically last partition of the positive integer n. This is the all-ones partition.

EXAMPLES:

```
sage: Partitions(4).last()
[1, 1, 1, 1]
```

next(*p*)

Return the lexicographically next partition after the partition p.

EXAMPLES:

```
sage: Partitions(4).next([4])
[3, 1]
sage: Partitions(4).next([1,1,1,1]) is None
True
```

random_element(*measure='uniform'*)

Return a random partitions of n for the specified measure.

INPUT:

- measure – 'uniform' or 'Plancherel' (default: 'uniform')

See also:

- random_element_uniform()
- random_element_plancherel()

EXAMPLES:

```
sage: Partitions(5).random_element() # random
[2, 1, 1, 1]
sage: Partitions(5).random_element(measure='Plancherel') # random
[2, 1, 1, 1]
```

random_element_plancherel()

Return a random partition of n (for the Plancherel measure).

This probability distribution comes from the uniform distribution on permutations via the Robinson-Schensted correspondence.

See Wikipedia article Plancherel_measure and Partition.plancherel_measure().

EXAMPLES:

```
sage: Partitions(5).random_element_plancherel()    # random
[2, 1, 1, 1]
sage: Partitions(20).random_element_plancherel()   # random
[9, 3, 3, 2, 2, 1]
```

TESTS:

```
sage: all(Part.random_element_plancherel() in Part
...       for Part in map(Partitions, range(10)))
True
```

Check that trac ticket #18752 is fixed:

```
sage: P = Partitions(5)
sage: la = P.random_element_plancherel()
sage: la.parent() is P
True
```

ALGORITHM:

- insert by Robinson-Schensted a uniform random permutations of n and returns the shape of the resulting tableau. The complexity is $O(n \ln(n))$ which is likely optimal. However, the implementation could be optimized.

AUTHOR:

- Florent Hivert (2009-11-23)

random_element_uniform()

Return a random partition of n with uniform probability.

EXAMPLES:

```
sage: Partitions(5).random_element_uniform()  # random
[2, 1, 1, 1]
sage: Partitions(20).random_element_uniform() # random
[9, 3, 3, 2, 2, 1]
```

TESTS:

```
sage: all(Part.random_element_uniform() in Part
....:     for Part in map(Partitions, range(10)))
True
```

Check that trac ticket #18752 is fixed:

```
sage: P = Partitions(5)
sage: la = P.random_element_uniform()
sage: la.parent() is P
True
```

ALGORITHM:

- It is a python Implementation of RANDPAR, see [nw]. The complexity is unknown, there may be better algorithms.

Todo

Check in Knuth AOCP4.

- There is also certainly a lot of room for optimizations, see comments in the code.

REFERENCES:

AUTHOR:

- Florent Hivert (2009-11-23)

subset(***kwargs*)

Return a subset of `self` with the additional optional arguments.

EXAMPLES:

```
sage: P = Partitions(5); P
Partitions of the integer 5
sage: P.subset(starting=[3,1])
Partitions of the integer 5 starting with [3, 1]
```

class sage.combinat.partition.**Partitions_nk**(*n*, *k*)

Bases: sage.combinat.partition.Partitions

Partitions of the integer n of length equal to k.

TESTS:

```
sage: TestSuite( sage.combinat.partition.Partitions_nk(0,0) ).run()
sage: TestSuite( sage.combinat.partition.Partitions_nk(0,0) ).run()
```

cardinality(*algorithm='hybrid'*)

Return the number of partitions of the specified size with the specified length.

INPUT:

- `algorithm` – (default: `'hybrid'`) the algorithm to compute the cardinality and can be one of the following:
  - `'hybrid'` - use a hybrid algorithm which uses heuristics to reduce the complexity
  - `'gap'` - use GAP

EXAMPLES:

```
sage: v = Partitions(5, length=2).list(); v
[[4, 1], [3, 2]]
sage: len(v)
2
sage: Partitions(5, length=2).cardinality()
2
```

More generally, the number of partitions of n of length 2 is $\left\lfloor \frac{n}{2} \right\rfloor$:

```
sage: all( Partitions(n, length=2).cardinality()
....:      == n // 2 for n in range(10) )
True
```

The number of partitions of n of length 1 is 1 for n positive:

```
sage: all( Partitions(n, length=1).cardinality() == 1
....:      for n in range(1, 10) )
True
```

Further examples:

```
sage: Partitions(5, length=3).cardinality()
2
sage: Partitions(6, length=3).cardinality()
3
sage: Partitions(8, length=4).cardinality()
5
sage: Partitions(8, length=5).cardinality()
3
sage: Partitions(15, length=6).cardinality()
26
sage: Partitions(0, length=0).cardinality()
1
sage: Partitions(0, length=1).cardinality()
0
sage: Partitions(1, length=0).cardinality()
0
sage: Partitions(1, length=4).cardinality()
0
```

TESTS:

We check the hybrid approach gives the same results as GAP:

```
sage: N = [0, 1, 2, 3, 5, 10, 20, 500, 850]
sage: K = [0, 1, 2, 3, 5, 10, 11, 20, 21, 250, 499, 500]
sage: all(Partitions(n,length=k).cardinality() == Partitions(n,length=k).cardinality('gap')
....:     for n in N for k in K)
True
sage: P = Partitions(4562, length=2800)
sage: P.cardinality() == P.cardinality('gap')
True
```

subset (***kwargs*)

Return a subset of `self` with the additional optional arguments.

EXAMPLES:

```
sage: P = Partitions(5, length=2); P
Partitions of the integer 5 of length 2
sage: P.subset(max_part=3)
Partitions of the integer 5 satisfying constraints length=2, max_part=3
```

class sage.combinat.partition.**Partitions_parts_in**(*n*, *parts*)

Bases: sage.combinat.partition.Partitions

Partitions of n with parts in a given set S.

This is invoked indirectly when calling `Partitions(n, parts_in=parts)`, where `parts` is a list of

pairwise distinct integers.

TESTS:

```
sage: TestSuite( sage.combinat.partition.Partitions_parts_in(6, parts=[2,1]) ).run()
```

cardinality()

Return the number of partitions with parts in `self`. Wraps GAP's `NrRestrictedPartitions`.

EXAMPLES:

```
sage: Partitions(15, parts_in=[2,3,7]).cardinality()
5
```

If you can use all parts 1 through n, we'd better get $p(n)$:

```
sage: Partitions(20, parts_in=[1..20]).cardinality() == Partitions(20).cardinality()
True
```

TESTS:

Let's check the consistency of GAP's function and our own algorithm that actually generates the partitions:

```
sage: ps = Partitions(15, parts_in=[1,2,3])
sage: ps.cardinality() == len(ps.list())
True
sage: ps = Partitions(15, parts_in=[])
sage: ps.cardinality() == len(ps.list())
True
sage: ps = Partitions(3000, parts_in=[50,100,500,1000])
sage: ps.cardinality() == len(ps.list())
True
sage: ps = Partitions(10, parts_in=[3,6,9])
sage: ps.cardinality() == len(ps.list())
True
sage: ps = Partitions(0, parts_in=[1,2])
sage: ps.cardinality() == len(ps.list())
True
```

first()

Return the lexicographically first partition of a positive integer n with the specified parts, or `None` if no such partition exists.

EXAMPLES:

```
sage: Partitions(9, parts_in=[3,4]).first()
[3, 3, 3]
sage: Partitions(6, parts_in=[1..6]).first()
[6]
sage: Partitions(30, parts_in=[4,7,8,10,11]).first()
[11, 11, 8]
```

last()

Return the lexicographically last partition of the positive integer n with the specified parts, or `None` if no such partition exists.

EXAMPLES:

```
sage: Partitions(15, parts_in=[2,3]).last()
[3, 2, 2, 2, 2, 2, 2]
sage: Partitions(30, parts_in=[4,7,8,10,11]).last()
[7, 7, 4, 4, 4, 4]
sage: Partitions(10, parts_in=[3,6]).last() is None
```

```
True
sage: Partitions(50, parts_in=[11,12,13]).last()
[13, 13, 12, 12]
sage: Partitions(30, parts_in=[4,7,8,10,11]).last()
[7, 7, 4, 4, 4, 4]
```

TESTS:

```
sage: Partitions(6, parts_in=[1..6]).last()
[1, 1, 1, 1, 1, 1]
sage: Partitions(0, parts_in=[]).last()
[]
sage: Partitions(50, parts_in=[11,12]).last() is None
True
```

class sage.combinat.partition.**Partitions_starting**(*n*, *starting_partition*)

Bases: sage.combinat.partition.Partitions

All partitions with a given start.

first()

Return the first partition in self.

EXAMPLES:

```
sage: Partitions(3, starting=[2,1]).first()
[2, 1]
```

next(*part*)

Return the next partition after part in self.

EXAMPLES:

```
sage: Partitions(3, starting=[2,1]).next(Partition([2,1]))
[1, 1, 1]
```

class sage.combinat.partition.**Partitions_with_constraints**(**args*, ***kwds*)

Bases: sage.combinat.integer_lists.invlex.IntegerListsLex

Partitions which satisfy a set of constraints.

EXAMPLES:

```
sage: P = Partitions(6, inner=[1,1], max_slope=-1)
sage: list(P)
[[5, 1], [4, 2], [3, 2, 1]]
```

TESTS:

```
sage: P = Partitions(6, min_part=2, max_slope=-1)
sage: TestSuite(P).run()
```

Test that trac ticket #15525 is fixed:

```
sage: loads(dumps(P)) == P
True
```

Element

alias of Partition

global_options(**get_value*, ***set_value*)

Sets and displays the global options for elements of the partition, skew partition, and partition tuple classes.

If no parameters are set, then the function returns a copy of the options dictionary.

The `options` to partitions can be accessed as the method `Partitions.global_options` of `Partitions` and related parent classes.

OPTIONS:

- `convention` – (default: `English`) Sets the convention used for displaying tableaux and partitions
  - `English` – use the English convention
  - `French` – use the French convention
- `diagram_str` – (default: `*`) The character used for the cells when printing Ferrers diagrams
- `display` – (default: `list`) Specifies how partitions should be printed
  - `array` – alias for `diagram`
  - `compact` – alias for `compact_low`
  - `compact_high` – compact form of `exp_high`
  - `compact_low` – compact form of `exp_low`
  - `diagram` – as a Ferrers diagram
  - `exp` – alias for `exp_low`
  - `exp_high` – in exponential form (highest first)
  - `exp_low` – in exponential form (lowest first)
  - `ferrers_diagram` – alias for `diagram`
  - `list` – displayed as a list
  - `young_diagram` – alias for `diagram`
- `latex` – (default: `young_diagram`) Specifies how partitions should be latexed
  - `array` – alias for `diagram`
  - `diagram` – latex as a Ferrers diagram
  - `exp` – alias for `exp_low`
  - `exp_high` – latex as a list in exponential notation (highest first)
  - `exp_low` – as a list latex in exponential notation (lowest first)
  - `ferrers_diagram` – alias for `diagram`
  - `list` – latex as a list
  - `young_diagram` – latex as a Young diagram
- `latex_diagram_str` – (default: `\ast`) The character used for the cells when latexing Ferrers diagrams
- `notation` – alternative name for `convention`

EXAMPLES:

```
sage: P = Partition([4,2,2,1])
sage: P
[4, 2, 2, 1]
sage: Partitions.global_options(display="exp")
sage: P
1, 2^2, 4
```

```
sage: Partitions.global_options(display="exp_high")
sage: P
4, 2^2, 1
```

It is also possible to use user defined functions for the `display` and `latex` options:

```
sage: Partitions.global_options(display=lambda mu: '<%s>' % ','.join('%s'%m for m in mu._lis
<4,2,2,1>
sage: Partitions.global_options(latex=lambda mu: '\\Diagram{%s}' % ','.join('%s'%m for m in
\Diagram{4,2,2,1}
sage: Partitions.global_options(display="diagram", diagram_str="#")
sage: P
####
##
##
#
sage: Partitions.global_options(diagram_str="*", convention="french")
sage: print P.ferrers_diagram()
*
**
**
****
```

Changing the `convention` for partitions also changes the `convention` option for tableaux and vice versa:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T.pp()
  4  5
  1  2  3
sage: Tableaux.global_options(convention="english")
sage: print P.ferrers_diagram()
****
**
**
*
sage: T.pp()
  1  2  3
  4  5
sage: Partitions.global_options.reset()
```

See `GlobalOptions` for more features of these options.

class sage.combinat.partition.**RegularPartitions**(*ell*, *is_infinite=False*)

Bases: sage.combinat.partition.Partitions

Base class for ℓ-regular partitions.

Let ℓ be a positive integer. A partition λ is *ℓ-regular* if $m_i < \ell$ for all i, where m_i is the multiplicity of i in λ.

Note: This is conjugate to the notion of *ℓ-restricted* partitions, where the difference between any two parts is at most ℓ.

INPUT:

- `ell` – the integer ℓ
- `is_infinite` – boolean; if the subset of ℓ-regular partitions is infinite

ell()
Return the value ℓ.

EXAMPLES:

```
sage: P = Partitions(regular=2)
sage: P.ell()
2
```

class sage.combinat.partition.**RegularPartitions_all**(*ell*)
Bases: sage.combinat.partition.RegularPartitions

The class of all ℓ-regular partitions.

INPUT:

- ell – the integer ℓ

See also:

RegularPartitions

class sage.combinat.partition.**RegularPartitions_bounded**(*ell*, *k*)
Bases: sage.combinat.partition.RegularPartitions

The class of ℓ-regular k-bounded partitions.

INPUT:

- ell – the integer ℓ
- k – integer; the value k

See also:

RegularPartitions

class sage.combinat.partition.**RegularPartitions_n**(*n*, *ell*)
Bases: sage.combinat.partition.RegularPartitions, sage.combinat.partition.Partitions_n

The class of ℓ-regular partitions of n.

INPUT:

- n – the integer n to partition
- ell – the integer ℓ

See also:

RegularPartitions

cardinality()
Return the cardinality of self.

EXAMPLES:

```
sage: P = Partitions(5, regular=3)
sage: P.cardinality()
5
sage: P = Partitions(5, regular=6)
sage: P.cardinality()
7
sage: P.cardinality() == Partitions(5).cardinality()
True
```

class sage.combinat.partition.**RegularPartitions_truncated**(*ell*, *max_len*)
Bases: sage.combinat.partition.RegularPartitions

The class of ℓ-regular partitions with max length k.

INPUT:

- ell – the integer ℓ
- max_len – integer; the maximum length

See also:

RegularPartitions

max_length()
Return the maximum length of the partitions of self.

EXAMPLES:

```
sage: P = Partitions(regular=4, max_length=3)
sage: P.max_length()
3
```

sage.combinat.partition.**RestrictedPartitions**(*n*, *S*, *k=None*)
This function has been deprecated and will be removed in a future version of Sage; use Partitions with the parts_in keyword. Note, however, that the current implementation of Partitions does not allow the parts_in keyword to be combined with keywords such as max_length; see trac ticket #13072 and trac ticket #12278 for more details. This class should not be removed until this problem has been fixed.

Original docstring follows.

A restricted partition is, like an ordinary partition, an unordered sum $n = p_1 + p_2 + \ldots + p_k$ of positive integers and is represented by the list $p = [p_1, p_2, \ldots, p_k]$, in nonincreasing order. The difference is that here the p_i must be elements from the set S, while for ordinary partitions they may be elements from $[1..n]$.

Returns the list of all restricted partitions of the positive integer n into sums with k summands with the summands of the partition coming from the set S. If k is not given all restricted partitions for all k are returned.

Wraps GAP's RestrictedPartitions.

EXAMPLES:

```
sage: from sage.combinat.partition import RestrictedPartitions
sage: RestrictedPartitions(5,[3,2,1])
doctest:...: DeprecationWarning: RestrictedPartitions is deprecated; use Partitions with the par
See http://trac.sagemath.org/13072 for details.
doctest:...: DeprecationWarning: RestrictedPartitions_nsk is deprecated; use Partitions with the
See http://trac.sagemath.org/13072 for details.
Partitions of 5 restricted to the values [1, 2, 3]
sage: RestrictedPartitions(5,[3,2,1]).list()
[[3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]
sage: RestrictedPartitions(5,[3,2,1],4)
Partitions of 5 restricted to the values [1, 2, 3] of length 4
sage: RestrictedPartitions(5,[3,2,1],4).list()
[[2, 1, 1, 1]]
```

class sage.combinat.partition.**RestrictedPartitions_nsk**(*n*, *S*, *k=None*)
Bases: sage.combinat.combinat.CombinatorialClass

We are deprecating RestrictedPartitions(), so this class should be deprecated too. See trac ticket #13072.

Element
alias of Partition

cardinality()
Returns the size of self.

Wraps GAP's NrRestrictedPartitions.

EXAMPLES:

```
sage: from sage.combinat.partition import RestrictedPartitions
sage: RestrictedPartitions(8,[1,3,5,7]).cardinality()
doctest:...: DeprecationWarning: RestrictedPartitions is deprecated; use Partitions with the
See http://trac.sagemath.org/13072 for details.
6
sage: RestrictedPartitions(8,[1,3,5,7],2).cardinality()
2
```

global_options(**get_value*, ***set_value*)
Sets and displays the global options for elements of the partition, skew partition, and partition tuple classes. If no parameters are set, then the function returns a copy of the options dictionary.

The options to partitions can be accessed as the method Partitions.global_options of Partitions and related parent classes.

OPTIONS:

- convention – (default: English) Sets the convention used for displaying tableaux and partitions
  - English – use the English convention
  - French – use the French convention
- diagram_str – (default: *) The character used for the cells when printing Ferrers diagrams
- display – (default: list) Specifies how partitions should be printed
  - array – alias for diagram
  - compact – alias for compact_low
  - compact_high – compact form of exp_high
  - compact_low – compact form of exp_low
  - diagram – as a Ferrers diagram
  - exp – alias for exp_low
  - exp_high – in exponential form (highest first)
  - exp_low – in exponential form (lowest first)
  - ferrers_diagram – alias for diagram
  - list – displayed as a list
  - young_diagram – alias for diagram
- latex – (default: young_diagram) Specifies how partitions should be latexed
  - array – alias for diagram
  - diagram – latex as a Ferrers diagram
  - exp – alias for exp_low
  - exp_high – latex as a list in exponential notation (highest first)

– `exp_low` – as a list latex in exponential notation (lowest first)

– `ferrers_diagram` – alias for `diagram`

– `list` – latex as a list

– `young_diagram` – latex as a Young diagram

- `latex_diagram_str` – (default: `\ast`) The character used for the cells when latexing Ferrers diagrams
- `notation` – alternative name for `convention`

EXAMPLES:

```
sage: P = Partition([4,2,2,1])
sage: P
[4, 2, 2, 1]
sage: Partitions.global_options(display="exp")
sage: P
1, 2^2, 4
sage: Partitions.global_options(display="exp_high")
sage: P
4, 2^2, 1
```

It is also possible to use user defined functions for the `display` and `latex` options:

```
sage: Partitions.global_options(display=lambda mu: '<%s>' % ','.join('%s'%m for m in mu._lis
<4,2,2,1>
sage: Partitions.global_options(latex=lambda mu: '\\Diagram{%s}' % ','.join('%s'%m for m in
\Diagram{4,2,2,1}
sage: Partitions.global_options(display="diagram", diagram_str="#")
sage: P
####
##
##
#
sage: Partitions.global_options(diagram_str="*", convention="french")
sage: print P.ferrers_diagram()
*
**
**
****
```

Changing the `convention` for partitions also changes the `convention` option for tableaux and vice versa:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T.pp()
  4  5
  1  2  3
sage: Tableaux.global_options(convention="english")
sage: print P.ferrers_diagram()
****
**
**
*
sage: T.pp()
  1  2  3
  4  5
sage: Partitions.global_options.reset()
```

See GlobalOptions for more features of these options.

list()

Returns the list of all restricted partitions of the positive integer n into sums with k summands with the summands of the partition coming from the set S. If k is not given all restricted partitions for all k are returned.

Wraps GAP's RestrictedPartitions.

EXAMPLES:

```
sage: from sage.combinat.partition import RestrictedPartitions
sage: RestrictedPartitions(8,[1,3,5,7]).list()
doctest:...: DeprecationWarning: RestrictedPartitions is deprecated; use Partitions with the
See http://trac.sagemath.org/13072 for details.
[[7, 1], [5, 3], [5, 1, 1, 1], [3, 3, 1, 1], [3, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1]]
sage: RestrictedPartitions(8,[1,3,5,7],2).list()
[[7, 1], [5, 3]]
```

sage.combinat.partition.**number_of_partitions**(*n*, *algorithm='default'*)

Returns the number of partitions of n with, optionally, at most k parts.

The options of number_of_partitions() are being deprecated trac ticket #13072 in favour of Partitions_n.cardinality() so that number_of_partitions() can become a stripped down version of the fastest algorithm available (currently this is using FLINT).

INPUT:

- n – an integer
- algorithm – (default: 'default') [Will be deprecated except in Partition().cardinality()]
  - 'default' – If k is not None, then use Gap (very slow). If k is None, use FLINT.
  - 'flint' – use FLINT
  - 'bober' – use Jonathan Bober's implementation

EXAMPLES:

```
sage: v = Partitions(5).list(); v
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]
sage: len(v)
7
sage: number_of_partitions(5, algorithm='bober')
7
```

The input must be a nonnegative integer or a ValueError is raised.

```
sage: number_of_partitions(-5)
Traceback (most recent call last):
...
ValueError: n (=-5) must be a nonnegative integer

sage: number_of_partitions(10)
42
sage: number_of_partitions(3)
3
sage: number_of_partitions(10)
42
sage: number_of_partitions(40)
37338
sage: number_of_partitions(100)
190569292
```

```
sage: number_of_partitions(100000)
27493510569775696512677516320986352688173429315980054758203125984302147328114964173055050741660
```

A generating function for the number of partitions p_n is given by the reciprocal of Euler's function:

$$\sum_{n=0}^{\infty} p_n x^n = \prod_{k=1}^{\infty} \left(\frac{1}{1-x^k} \right).$$

We use Sage to verify that the first several coefficients do instead agree:

```
sage: q = PowerSeriesRing(QQ, 'q', default_prec=9).gen()
sage: prod([(1-q^k)^(-1) for k in range(1,9)])  ## partial product of
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + 7*q^5 + 11*q^6 + 15*q^7 + 22*q^8 + O(q^9)
sage: [number_of_partitions(k) for k in range(2,10)]
[2, 3, 5, 7, 11, 15, 22, 30]
```

REFERENCES:

- Wikipedia article Partition_(number_theory)

TESTS:

```
sage: n = 500 + randint(0,500)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1500 + randint(0,1500)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 100000000 + randint(0,100000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0  # long time (4s on sage.math, 2011)
True
```

sage.combinat.partition.**number_of_partitions_length**(*n*, *k*, *algorithm='hybrid'*)

Return the number of partitions of n with length k.

This is a wrapper for GAP's `NrPartitions` function.

EXAMPLES:

```
sage: from sage.combinat.partition import number_of_partitions_length
sage: number_of_partitions_length(5, 2)
2
```

```
sage: number_of_partitions_length(10, 2)
5
sage: number_of_partitions_length(10, 4)
9
sage: number_of_partitions_length(10, 0)
0
sage: number_of_partitions_length(10, 1)
1
sage: number_of_partitions_length(0, 0)
1
sage: number_of_partitions_length(0, 1)
0
```

5.1.143 Partition/Diagram Algebras

class sage.combinat.partition_algebra.**PartitionAlgebraElement_ak**(*M*, *x*)
Bases: sage.combinat.partition_algebra.PartitionAlgebraElement_generic

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebraElement_bk**(*M*, *x*)
Bases: sage.combinat.partition_algebra.PartitionAlgebraElement_generic

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebraElement_generic**(*M*, *x*)
Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebraElement_pk**(*M*, *x*)
Bases: sage.combinat.partition_algebra.PartitionAlgebraElement_generic

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebraElement_prk**(*M*, *x*)
Bases: sage.combinat.partition_algebra.PartitionAlgebraElement_generic

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebraElement_rk**(*M*, *x*)
Bases: sage.combinat.partition_algebra.PartitionAlgebraElement_generic

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebraElement_sk**(*M*, *x*)
Bases: sage.combinat.partition_algebra.PartitionAlgebraElement_generic

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebraElement_tk**(*M*, *x*)
Bases: sage.combinat.partition_algebra.PartitionAlgebraElement_generic

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_ak**(*R*, *k*, *n*, *name=None*)

Bases: sage.combinat.partition_algebra.PartitionAlgebra_generic

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_ak(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_bk**(*R*, *k*, *n*, *name=None*)

Bases: sage.combinat.partition_algebra.PartitionAlgebra_generic

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_bk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_generic**(*R*, *cclass*, *n*, *k*, *name=None*, *prefix=None*)

Bases: sage.combinat.combinatorial_algebra.CombinatorialAlgebra

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: s = PartitionAlgebra_sk(QQ, 3, 1)
sage: s == loads(dumps(s))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_pk**(*R*, *k*, *n*, *name=None*)

Bases: sage.combinat.partition_algebra.PartitionAlgebra_generic

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_pk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_prk**(*R*, *k*, *n*, *name=None*)

Bases: sage.combinat.partition_algebra.PartitionAlgebra_generic

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_prk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_rk**(*R*, *k*, *n*, *name=None*)
Bases: sage.combinat.partition_algebra.PartitionAlgebra_generic

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_rk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_sk**(*R*, *k*, *n*, *name=None*)
Bases: sage.combinat.partition_algebra.PartitionAlgebra_generic

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_sk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_tk**(*R*, *k*, *n*, *name=None*)
Bases: sage.combinat.partition_algebra.PartitionAlgebra_generic

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_tk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

sage.combinat.partition_algebra.**SetPartitionsAk**(*k*)
Returns the combinatorial class of set partitions of type A_k.

EXAMPLES:

```
sage: A3 = SetPartitionsAk(3); A3
Set partitions of {1, ..., 3, -1, ..., -3}

sage: A3.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: A3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
sage: A3.random_element()  #random
{{1, 3, -3, -1}, {2, -2}}

sage: A3.cardinality()
203

sage: A2p5 = SetPartitionsAk(2.5); A2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block
sage: A2p5.cardinality()
52

sage: A2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: A2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
sage: A2p5.random_element() #random
{{-1}, {-2}, {3, -3}, {1, 2}}
```

class sage.combinat.partition_algebra.**SetPartitionsAk_k**(*k*)
Bases: sage.combinat.set_partition.SetPartitions_set

TESTS:

```
sage: A3 = SetPartitionsAk(3); A3
Set partitions of {1, ..., 3, -1, ..., -3}
sage: A3 == loads(dumps(A3))
True
```

Element
alias of SetPartitionsXkElement

class sage.combinat.partition_algebra.**SetPartitionsAkhalf_k**(*k*)
Bases: sage.combinat.set_partition.SetPartitions_set

TESTS:

```
sage: A2p5 = SetPartitionsAk(2.5); A2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block
sage: A2p5 == loads(dumps(A2p5))
True
```

Element
alias of SetPartitionsXkElement

sage.combinat.partition_algebra.**SetPartitionsBk**(*k*)
Returns the combinatorial class of set partitions of type B_k. These are the set partitions where every block has size 2.

EXAMPLES:

```
sage: B3 = SetPartitionsBk(3); B3
Set partitions of {1, ..., 3, -1, ..., -3} with block size 2

sage: B3.first() #random
{{2, -2}, {1, -3}, {3, -1}}
sage: B3.last() #random
{{1, 2}, {3, -2}, {-3, -1}}
sage: B3.random_element() #random
{{2, -1}, {1, -3}, {3, -2}}

sage: B3.cardinality()
15

sage: B2p5 = SetPartitionsBk(2.5); B2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and with block size 2

sage: B2p5.first() #random
{{2, -1}, {3, -3}, {1, -2}}
sage: B2p5.last() #random
{{1, 2}, {3, -3}, {-1, -2}}
sage: B2p5.random_element() #random
{{2, -2}, {3, -3}, {1, -1}}

sage: B2p5.cardinality()
3
```

class sage.combinat.partition_algebra.**SetPartitionsBk_k**(*k*)
Bases: sage.combinat.partition_algebra.SetPartitionsAk_k

TESTS:

```
sage: A3 = SetPartitionsAk(3); A3
Set partitions of {1, ..., 3, -1, ..., -3}
```

```
sage: A3 == loads(dumps(A3))
True
```

cardinality()

Returns the number of set partitions in B_k where k is an integer. This is given by (2k)!! = (2k-1)*(2k-3)*...*5*3*1.

EXAMPLES:

```
sage: SetPartitionsBk(3).cardinality()
15
sage: SetPartitionsBk(2).cardinality()
3
sage: SetPartitionsBk(1).cardinality()
1
sage: SetPartitionsBk(4).cardinality()
105
sage: SetPartitionsBk(5).cardinality()
945
```

class sage.combinat.partition_algebra.**SetPartitionsBkhalf_k**(*k*)

Bases: sage.combinat.partition_algebra.SetPartitionsAkhalf_k

TESTS:

```
sage: A2p5 = SetPartitionsAk(2.5); A2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block
sage: A2p5 == loads(dumps(A2p5))
True
```

cardinality()

TESTS:

```
sage: B3p5 = SetPartitionsBk(3.5)
sage: B3p5.cardinality()
15
```

sage.combinat.partition_algebra.**SetPartitionsIk**(*k*)

Returns the combinatorial class of set partitions of type I_k. These are set partitions with a propagating number of less than k. Note that the identity set partition {{1, -1}, ..., {k, -k}} is not in I_k.

EXAMPLES:

```
sage: I3 = SetPartitionsIk(3); I3
Set partitions of {1, ..., 3, -1, ..., -3} with propagating number < 3
sage: I3.cardinality()
197

sage: I3.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: I3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
sage: I3.random_element() #random
{{-1}, {-3, -2}, {2, 3}, {1}}

sage: I2p5 = SetPartitionsIk(2.5); I2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and propagating numbe
sage: I2p5.cardinality()
50
```

```
sage: I2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: I2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
sage: I2p5.random_element() #random
{{-1}, {-2}, {1, 3, -3}, {2}}
```

class sage.combinat.partition_algebra.**SetPartitionsIk_k**(k)

Bases: sage.combinat.partition_algebra.SetPartitionsAk_k

TESTS:

```
sage: A3 = SetPartitionsAk(3); A3
Set partitions of {1, ..., 3, -1, ..., -3}
sage: A3 == loads(dumps(A3))
True
```

cardinality()

TESTS:

```
sage: SetPartitionsIk(2).cardinality()
13
```

class sage.combinat.partition_algebra.**SetPartitionsIkhalf_k**(k)

Bases: sage.combinat.partition_algebra.SetPartitionsAkhalf_k

TESTS:

```
sage: A2p5 = SetPartitionsAk(2.5); A2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block
sage: A2p5 == loads(dumps(A2p5))
True
```

cardinality()

TESTS:

```
sage: SetPartitionsIk(1.5).cardinality()
4
sage: SetPartitionsIk(2.5).cardinality()
50
sage: SetPartitionsIk(3.5).cardinality()
871
```

sage.combinat.partition_algebra.**SetPartitionsPRk**(k)

class sage.combinat.partition_algebra.**SetPartitionsPRk_k**(k)

Bases: sage.combinat.partition_algebra.SetPartitionsRk_k

TESTS:

```
sage: PR3 = SetPartitionsPRk(3); PR3
Set partitions of {1, ..., 3, -1, ..., -3} with at most 1 positive and negative entry in each bl
sage: PR3 == loads(dumps(PR3))
True
```

cardinality()

TESTS:

```
sage: SetPartitionsPRk(2).cardinality()
6
sage: SetPartitionsPRk(3).cardinality()
20
```

```
sage: SetPartitionsPRk(4).cardinality()
70
sage: SetPartitionsPRk(5).cardinality()
252
```

class sage.combinat.partition_algebra.**SetPartitionsPRkhalf_k**(k)

Bases: sage.combinat.partition_algebra.SetPartitionsRkhalf_k

TESTS:

```
sage: A2p5 = SetPartitionsAk(2.5); A2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block
sage: A2p5 == loads(dumps(A2p5))
True
```

cardinality()

TESTS:

```
sage: SetPartitionsPRk(2.5).cardinality()
6
sage: SetPartitionsPRk(3.5).cardinality()
20
sage: SetPartitionsPRk(4.5).cardinality()
70
```

sage.combinat.partition_algebra.**SetPartitionsPk**(k)

Returns the combinatorial class of set partitions of type P_k. These are the planar set partitions.

EXAMPLES:

```
sage: P3 = SetPartitionsPk(3); P3
Set partitions of {1, ..., 3, -1, ..., -3} that are planar
sage: P3.cardinality()
132

sage: P3.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: P3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
sage: P3.random_element() #random
{{1, 2, -1}, {-3}, {3, -2}}

sage: P2p5 = SetPartitionsPk(2.5); P2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and that are planar
sage: P2p5.cardinality()
42

sage: P2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: P2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
sage: P2p5.random_element() #random
{{1, 2, 3, -3}, {-1, -2}}
```

class sage.combinat.partition_algebra.**SetPartitionsPk_k**(k)

Bases: sage.combinat.partition_algebra.SetPartitionsAk_k

TESTS:

```
sage: A3 = SetPartitionsAk(3); A3
Set partitions of {1, ..., 3, -1, ..., -3}
sage: A3 == loads(dumps(A3))
True
```

cardinality()

TESTS:

```
sage: SetPartitionsPk(2).cardinality()
14
sage: SetPartitionsPk(3).cardinality()
132
sage: SetPartitionsPk(4).cardinality()
1430
```

class sage.combinat.partition_algebra.**SetPartitionsPkhalf_k**(k)

Bases: sage.combinat.partition_algebra.SetPartitionsAkhalf_k

TESTS:

```
sage: A2p5 = SetPartitionsAk(2.5); A2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block
sage: A2p5 == loads(dumps(A2p5))
True
```

cardinality()

TESTS:

```
sage: SetPartitionsPk(2.5).cardinality()
42
sage: SetPartitionsPk(1.5).cardinality()
5
```

sage.combinat.partition_algebra.**SetPartitionsRk**(k)

class sage.combinat.partition_algebra.**SetPartitionsRk_k**(k)

Bases: sage.combinat.partition_algebra.SetPartitionsAk_k

TESTS:

```
sage: R3 = SetPartitionsRk(3); R3
Set partitions of {1, ..., 3, -1, ..., -3} with at most 1 positive and negative entry in each bl
sage: R3 == loads(dumps(R3))
True
```

cardinality()

TESTS:

```
sage: SetPartitionsRk(2).cardinality()
7
sage: SetPartitionsRk(3).cardinality()
34
sage: SetPartitionsRk(4).cardinality()
209
sage: SetPartitionsRk(5).cardinality()
1546
```

class sage.combinat.partition_algebra.**SetPartitionsRkhalf_k**(k)

Bases: sage.combinat.partition_algebra.SetPartitionsAkhalf_k

TESTS:

```
sage: A2p5 = SetPartitionsAk(2.5); A2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block
sage: A2p5 == loads(dumps(A2p5))
True
```

cardinality()

TESTS:

```
sage: SetPartitionsRk(2.5).cardinality()
7
sage: SetPartitionsRk(3.5).cardinality()
34
sage: SetPartitionsRk(4.5).cardinality()
209
```

sage.combinat.partition_algebra.**SetPartitionsSk**(*k*)

Returns the combinatorial class of set partitions of type S_k. There is a bijection between these set partitions and the permutations of 1, ..., k.

EXAMPLES:

```
sage: S3 = SetPartitionsSk(3); S3
Set partitions of {1, ..., 3, -1, ..., -3} with propagating number 3
sage: S3.cardinality()
6

sage: S3.list()  #random
[{{2, -2}, {3, -3}, {1, -1}},
 {{1, -1}, {2, -3}, {3, -2}},
 {{2, -1}, {3, -3}, {1, -2}},
 {{1, -2}, {2, -3}, {3, -1}},
 {{1, -3}, {2, -1}, {3, -2}},
 {{1, -3}, {2, -2}, {3, -1}}]
sage: S3.first() #random
{{2, -2}, {3, -3}, {1, -1}}
sage: S3.last() #random
{{1, -3}, {2, -2}, {3, -1}}
sage: S3.random_element() #random
{{1, -3}, {2, -1}, {3, -2}}

sage: S3p5 = SetPartitionsSk(3.5); S3p5
Set partitions of {1, ..., 4, -1, ..., -4} with 4 and -4 in the same block and propagating numbe
sage: S3p5.cardinality()
6

sage: S3p5.list() #random
[{{2, -2}, {3, -3}, {1, -1}, {4, -4}},
 {{2, -3}, {1, -1}, {4, -4}, {3, -2}},
 {{2, -1}, {3, -3}, {1, -2}, {4, -4}},
 {{2, -3}, {1, -2}, {4, -4}, {3, -1}},
 {{1, -3}, {2, -1}, {4, -4}, {3, -2}},
 {{1, -3}, {2, -2}, {4, -4}, {3, -1}}]
sage: S3p5.first() #random
{{2, -2}, {3, -3}, {1, -1}, {4, -4}}
sage: S3p5.last() #random
{{1, -3}, {2, -2}, {4, -4}, {3, -1}}
sage: S3p5.random_element() #random
{{1, -3}, {2, -2}, {4, -4}, {3, -1}}
```

class sage.combinat.partition_algebra.**SetPartitionsSk_k**(*k*)
Bases: sage.combinat.partition_algebra.SetPartitionsAk_k

TESTS:

```
sage: A3 = SetPartitionsAk(3); A3
Set partitions of {1, ..., 3, -1, ..., -3}
sage: A3 == loads(dumps(A3))
True
```

cardinality()
Returns k!.

TESTS:

```
sage: SetPartitionsSk(2).cardinality()
2
sage: SetPartitionsSk(3).cardinality()
6
sage: SetPartitionsSk(4).cardinality()
24
sage: SetPartitionsSk(5).cardinality()
120
```

class sage.combinat.partition_algebra.**SetPartitionsSkhalf_k**(*k*)
Bases: sage.combinat.partition_algebra.SetPartitionsAkhalf_k

TESTS:

```
sage: A2p5 = SetPartitionsAk(2.5); A2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block
sage: A2p5 == loads(dumps(A2p5))
True
```

cardinality()
TESTS:

```
sage: SetPartitionsSk(2.5).cardinality()
2
sage: SetPartitionsSk(3.5).cardinality()
6
sage: SetPartitionsSk(4.5).cardinality()
24

sage: ks = [2.5, 3.5, 4.5, 5.5]
sage: sks = [SetPartitionsSk(k) for k in ks]
sage: all([ sk.cardinality() == len(sk.list()) for sk in sks])
True
```

sage.combinat.partition_algebra.**SetPartitionsTk**(*k*)
Returns the combinatorial class of set partitions of type T_k. These are planar set partitions where every block is of size 2.

EXAMPLES:

```
sage: T3 = SetPartitionsTk(3); T3
Set partitions of {1, ..., 3, -1, ..., -3} with block size 2 and that are planar
sage: T3.cardinality()
5

sage: T3.first() #random
{{1, -3}, {2, 3}, {-1, -2}}
```

```
sage: T3.last() #random
{{1, 2}, {3, -1}, {-3, -2}}
sage: T3.random_element() #random
{{1, -3}, {2, 3}, {-1, -2}}

sage: T2p5 = SetPartitionsTk(2.5); T2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and with block size 2
sage: T2p5.cardinality()
2

sage: T2p5.first() #random
{{2, -2}, {3, -3}, {1, -1}}
sage: T2p5.last() #random
{{1, 2}, {3, -3}, {-1, -2}}
```

class sage.combinat.partition_algebra.**SetPartitionsTk_k**(*k*)
Bases: sage.combinat.partition_algebra.SetPartitionsBk_k

TESTS:

```
sage: A3 = SetPartitionsAk(3); A3
Set partitions of {1, ..., 3, -1, ..., -3}
sage: A3 == loads(dumps(A3))
True
```

cardinality()
TESTS:

```
sage: SetPartitionsTk(2).cardinality()
2
sage: SetPartitionsTk(3).cardinality()
5
sage: SetPartitionsTk(4).cardinality()
14
sage: SetPartitionsTk(5).cardinality()
42
```

class sage.combinat.partition_algebra.**SetPartitionsTkhalf_k**(*k*)
Bases: sage.combinat.partition_algebra.SetPartitionsBkhalf_k

TESTS:

```
sage: A2p5 = SetPartitionsAk(2.5); A2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block
sage: A2p5 == loads(dumps(A2p5))
True
```

cardinality()
TESTS:

```
sage: SetPartitionsTk(2.5).cardinality()
2
sage: SetPartitionsTk(3.5).cardinality()
5
sage: SetPartitionsTk(4.5).cardinality()
14
```

class sage.combinat.partition_algebra.**SetPartitionsXkElement**(*parent*, *s*)
Bases: sage.combinat.set_partition.SetPartition

An element for the classes of SetPartitionXk where X is some letter.

check()

Check to make sure this is a set partition.

EXAMPLES:

```
sage: A2p5 = SetPartitionsAk(2.5)
sage: x = A2p5.first(); x # random
{{1, 2, 3, -1, -3, -2}}
sage: x.check()
sage: y = A2p5.next(x); y
{{-3, -2, -1, 2, 3}, {1}}
sage: y.check()
```

sage.combinat.partition_algebra.**create_set_partition_function**(*letter*, *k*)

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import create_set_partition_function
sage: create_set_partition_function('A', 3)
Set partitions of {1, ..., 3, -1, ..., -3}
```

sage.combinat.partition_algebra.**identity**(*k*)

Returns the identity set partition 1, -1, ..., k, -k

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: pa.identity(2)
{{2, -2}, {1, -1}}
```

sage.combinat.partition_algebra.**is_planar**(*sp*)

Returns True if the diagram corresponding to the set partition is planar; otherwise, it returns False.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: pa.is_planar( pa.to_set_partition([[1,-2],[2,-1]]))
False
sage: pa.is_planar( pa.to_set_partition([[1,-1],[2,-2]]))
True
```

sage.combinat.partition_algebra.**pair_to_graph**(*sp1*, *sp2*)

Return a graph consisting of the disjoint union of the graphs of set partitions sp1 and sp2 along with edges joining the bottom row (negative numbers) of sp1 to the top row (positive numbers) of sp2.

The vertices of the graph sp1 appear in the result as pairs (k, 1), whereas the vertices of the graph sp2 appear as pairs (k, 2).

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: sp1 = pa.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = pa.to_set_partition([[1,-2],[2,-1]])
sage: g = pa.pair_to_graph( sp1, sp2 ); g
Graph on 8 vertices

sage: g.vertices() #random
[(1, 2), (-1, 1), (-2, 2), (-1, 2), (-2, 1), (2, 1), (2, 2), (1, 1)]
sage: g.edges() #random
[((1, 2), (-1, 1), None),
 ((1, 2), (-2, 2), None),
```

```
 ((-1, 1), (2, 1), None),
 ((-1, 2), (2, 2), None),
 ((-2, 1), (1, 1), None),
 ((-2, 1), (2, 2), None)]
```

Another example which used to be wrong until trac ticket #15958:

```
sage: sp3 = pa.to_set_partition([[1, -1], [2], [-2]])
sage: sp4 = pa.to_set_partition([[1], [-1], [2], [-2]])
sage: g = pa.pair_to_graph( sp3, sp4 ); g
Graph on 8 vertices

sage: g.vertices()
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
sage: g.edges()
[((-2, 1), (2, 2), None), ((-1, 1), (1, 1), None),
 ((-1, 1), (1, 2), None)]
```

sage.combinat.partition_algebra.**propagating_number**(*sp*)

Returns the propagating number of the set partition sp. The propagating number is the number of blocks with both a positive and negative number.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: sp1 = pa.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = pa.to_set_partition([[1,2],[-2,-1]])
sage: pa.propagating_number(sp1)
2
sage: pa.propagating_number(sp2)
0
```

sage.combinat.partition_algebra.**set_partition_composition**(*sp1*, *sp2*)

Returns a tuple consisting of the composition of the set partitions sp1 and sp2 and the number of components removed from the middle rows of the graph.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: sp1 = pa.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = pa.to_set_partition([[1,-2],[2,-1]])
sage: pa.set_partition_composition(sp1, sp2) == (pa.identity(2), 0)
True
```

sage.combinat.partition_algebra.**to_graph**(*sp*)

Returns a graph representing the set partition sp.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: g = pa.to_graph( pa.to_set_partition([[1,-2],[2,-1]])); g
Graph on 4 vertices

sage: g.vertices() #random
[1, 2, -2, -1]
sage: g.edges() #random
[(1, -2, None), (2, -1, None)]
```

sage.combinat.partition_algebra.**to_set_partition**(*l*, *k=None*)

Coverts a list of a list of numbers to a set partitions. Each list of numbers in the outer list specifies the numbers

contained in one of the blocks in the set partition.

If k is specified, then the set partition will be a set partition of 1, ..., k, -1, ..., -k. Otherwise, k will default to the minimum number needed to contain all of the specified numbers.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: pa.to_set_partition([[1,-1],[2,-2]]) == pa.identity(2)
True
```

5.1.144 Partition tuples

A `PartitionTuple` is a tuple of partitions. That is, an ordered k-tuple of partitions $\mu = (\mu^{(1)}, \mu^{(2)}, ..., \mu^{(k)})$. If

$$n = |\mu| = |\mu^{(1)}| + |\mu^{(2)}| + \cdots + |\mu^{(k)}|$$

then we say that μ is a k-partition of n.

In representation theory partition tuples arise as the natural indexing set for the ordinary irreducible representations of:

- the wreath products of cyclic groups with symmetric groups,
- the Ariki-Koike algebras, or the cyclotomic Hecke algebras of the complex reflection groups of type $G(r, 1, n)$,
- the degenerate cyclotomic Hecke algebras of type $G(r, 1, n)$.

When these algebras are not semisimple, partition tuples index an important class of modules for the algebras, which are generalisations of the Specht modules of the symmetric groups.

Tuples of partitions also index the standard basis of the higher level combinatorial Fock spaces. As a consequence, the combinatorics of partition tuples encapsulates the canonical bases of crystal graphs for the irreducible integrable highest weight modules of the (quantized) affine special linear groups and the (quantized) affine general linear groups. By the categorification theorems of Ariki, Varagnolo-Vasserot, Stroppel-Webster and others, in characteristic zero the degenerate and non-degenerate cyclotomic Hecke algebras, via their Khovanov-Lauda-Rouquier grading, categorify the canonical bases of the quantum affine special and general linear groups.

Partitions are naturally in bijection with 1-tuples of partitions. Most of the combinatorial operations defined on partitions extend to partition tuples in a meaningful way. For example, the semisimple branching rules for the Specht modules are described by adding and removing cells from partition tuples and the modular branching rules correspond to adding and removing good and cogood nodes, which is the underlying combinatorics for the associated crystal graphs.

A `PartitionTuple` belongs to `PartitionTuples` and its derived classes. `PartitionTuples` is the parent class for all partitions tuples. Four different classes of tuples of partitions are currently supported:

- `PartitionTuples(level=k,size=n)` are k-tuple of partitions of n.
- `PartitionTuples(level=k)` are k-tuple of partitions.
- `PartitionTuples(size=n)` are tuples of partitions of n.
- `PartitionTuples()` are tuples of partitions.

Note: As with `Partitions`, in sage the cells, or nodes, of partition tuples are 0-based. For example, the (lexicographically) first cell in any non-empty partition tuple is $[0, 0, 0]$.

EXAMPLES:

```
sage: PartitionTuple([[2,2],[1,1],[2]]).cells()
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 1, 0), (2, 0, 0), (2, 0, 1)]
```

Note: Many `PartitionTuple` methods take the individual coordinates (k, r, c) as their arguments, here k is the component, r is the row index and c is the column index. If your coordinates are in the form `(k,r,c)` then use Python's *-operator.

EXAMPLES:

```
sage: mu=PartitionTuple([[1,1],[2],[2,1]])
sage: [ mu.arm_length(*c) for c in mu.cells()]
[0, 0, 1, 0, 1, 0, 0]
```

Warning: In sage, if `mu` is a partition tuple then `mu[k]` most naturally refers to the k-th component of `mu`, so we use the convention of the (k, r, c)-th cell in a partition tuple refers to the cell in component k, row r, and column c. In the literature, the cells of a partition tuple are usually written in the form (r, c, k), where r is the row index, c is the column index, and k is the component index.

REFERENCES:

AUTHORS:

- Andrew Mathas (2012-06-01): Initial classes.

EXAMPLES:

First is a finite enumerated set and the remaining classes are infinite enumerated sets:

```
sage: PartitionTuples().an_element()
([1, 1, 1, 1], [2, 1, 1], [3, 1], [4])
sage: PartitionTuples(4).an_element()
([], [1], [2], [3])
sage: PartitionTuples(size=5).an_element()
([1], [1], [1], [1], [1])
sage: PartitionTuples(4,5).an_element()
([1], [], [], [4])
sage: PartitionTuples(3,2)[:]
[([2], [], []),
 ([1, 1], [], []),
 ([1], [1], []),
 ([1], [], [1]),
 ([], [2], []),
 ([], [1, 1], []),
 ([], [1], [1]),
 ([], [], [2]),
 ([], [], [1, 1])]
sage: PartitionTuples(2,3).list()
[([3], []),
 ([2, 1], []),
 ([1, 1, 1], []),
 ([2], [1]),
 ([1, 1], [1]),
 ([1], [2]),
 ([1], [1, 1]),
 ([], [3]),
 ([], [2, 1]),
 ([], [1, 1, 1])]
```

One tuples of partitions are naturally in bijection with partitions and, as far as possible, partition tuples attempts to identify one tuples with partitions:

```
sage: Partition([4,3]) == PartitionTuple([[4,3]])
True
sage: Partition([4,3]) == PartitionTuple([4,3])
True
sage: PartitionTuple([4,3])
[4, 3]
sage: Partition([4,3]) in PartitionTuples()
True
```

Partition tuples come equipped with many of the corresponding methods for partitions. For example, it is possible to add and remove cells, to conjugate partition tuples, to work with their diagrams, compare partition tuples in dominance and so:

```
sage: PartitionTuple([[4,1],[],[2,2,1],[3]]).pp()
   ****   -   **   ***
   *          **
              *
sage: PartitionTuple([[4,1],[],[2,2,1],[3]]).conjugate()
([1, 1, 1], [3, 2], [], [2, 1, 1, 1])
sage: PartitionTuple([[4,1],[],[2,2,1],[3]]).conjugate().pp()
   *   ***   -   **
   *   **        *
   *             *
                 *
sage: lam=PartitionTuples(3)([[3,2],[],[1,1,1,1]]); lam
([3, 2], [], [1, 1, 1, 1])
sage: lam.level()
3
sage: lam.size()
9
sage: lam.category()
Category of elements of Partition tuples of level 3
sage: lam.parent()
Partition tuples of level 3
sage: lam[0]
[3, 2]
sage: lam[1]
[]
sage: lam[2]
[1, 1, 1, 1]
sage: lam.pp()
   ***   -   *
   **        *
             *
             *
sage: lam.removable_cells()
[(0, 0, 2), (0, 1, 1), (2, 3, 0)]
sage: lam.down_list()
[([2, 2], [], [1, 1, 1, 1]),
 ([3, 1], [], [1, 1, 1, 1]),
 ([3, 2], [], [1, 1, 1])]
sage: lam.addable_cells()
[(0, 0, 3), (0, 1, 2), (0, 2, 0), (1, 0, 0), (2, 0, 1), (2, 4, 0)]
sage: lam.up_list()
```

```
[([4, 2], [], [1, 1, 1, 1]),
 ([3, 3], [], [1, 1, 1, 1]),
 ([3, 2, 1], [], [1, 1, 1, 1]),
 ([3, 2], [1], [1, 1, 1, 1]),
 ([3, 2], [], [2, 1, 1, 1]),
 ([3, 2], [], [1, 1, 1, 1, 1])]
sage: lam.conjugate()
([4], [], [2, 2, 1])
sage: lam.dominates( PartitionTuple([[3],[1],[2,2,1]]) )
False
sage: lam.dominates( PartitionTuple([[3],[2],[1,1,1]]))
True
```

Every partition tuple behaves every much like a tuple of partitions:

```
sage: mu=PartitionTuple([[4,1],[],[2,2,1],[3]])
sage: [ nu for nu in mu ]
[[4, 1], [], [2, 2, 1], [3]]
sage: Set([ type(nu) for nu in mu ])
{<class 'sage.combinat.partition.Partitions_all_with_category.element_class'>}
sage: mu[2][2]
1
sage: mu[3]
[3]
sage: mu.components()
[[4, 1], [], [2, 2, 1], [3]]
sage: mu.components() == [ nu for nu in mu ]
True
sage: mu[0]
[4, 1]
sage: mu[1]
[]
sage: mu[2]
[2, 2, 1]
sage: mu[2][0]
2
sage: mu[2][1]
2
sage: mu.level()
4
sage: len(mu)
4
sage: mu.cells()
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 0, 3), (0, 1, 0), (2, 0, 0), (2, 0, 1), (2, 1, 0), (2, 1, 1),
sage: mu.addable_cells()
[(0, 0, 4), (0, 1, 1), (0, 2, 0), (1, 0, 0), (2, 0, 2), (2, 2, 1), (2, 3, 0), (3, 0, 3), (3, 1, 0)]
sage: mu.removable_cells()
[(0, 0, 3), (0, 1, 0), (2, 1, 1), (2, 2, 0), (3, 0, 2)]
```

Attached to a partition tuple is the corresponding Young, or parabolic, subgroup:

```
sage: mu.young_subgroup()
Permutation Group with generators [(), (12,13), (11,12), (8,9), (6,7), (3,4), (2,3), (1,2)]
sage: mu.young_subgroup_generators()
[1, 2, 3, 6, 8, 11, 12]
```

class sage.combinat.partition_tuple.**PartitionTuple**(*parent*, *mu*)

Bases: sage.combinat.combinat.CombinatorialElement

A tuple of Partition.

A tuple of partition comes equipped with many of methods available to partitions. The level of the PartitionTuple is the length of the tuple.

This is an ordered k-tuple of partitions $\mu = (\mu^{(1)}, \mu^{(2)}, ..., \mu^{(k)})$. If

$$n = |\mu| = |\mu^{(1)}| + |\mu^{(2)}| + \cdots + |\mu^{(k)}|$$

then μ is a k-partition of n.

In representation theory PartitionTuples arise as the natural indexing set for the ordinary irreducible representations of:

- the wreath products of cyclic groups with symmetric groups
- the Ariki-Koike algebras, or the cyclotomic Hecke algebras of the complex reflection groups of type $G(r, 1, n)$
- the degenerate cyclotomic Hecke algebras of type $G(r, 1, n)$

When these algebras are not semisimple, partition tuples index an important class of modules for the algebras which are generalisations of the Specht modules of the symmetric groups.

Tuples of partitions also index the standard basis of the higher level combinatorial Fock spaces. As a consequence, the combinatorics of partition tuples encapsulates the canonical bases of crystal graphs for the irreducible integrable highest weight modules of the (quantized) affine special linear groups and the (quantized) affine general linear groups. By the categorification theorems of Ariki, Varagnolo-Vasserot, Stroppel-Webster and others, in characteristic zero the degenerate and non-degenerate cyclotomic Hecke algebras, via their Khovanov-Lauda-Rouquier grading, categorify the canonical bases of the quantum affine special and general linear groups.

Partitions are naturally in bijection with 1-tuples of partitions. Most of the combinatorial operations defined on partitions extend to PartitionTuples in a meaningful way. For example, the semisimple branching rules for the Specht modules are described by adding and removing cells from partition tuples and the modular branching rules correspond to adding and removing good and cogood nodes, which is the underlying combinatorics for the associated crystal graphs.

Warning: In the literature, the cells of a partition tuple are usually written in the form (r, c, k), where r is the row index, c is the column index, and k is the component index. In sage, if mu is a partition tuple then mu[k] most naturally refers to the k-th component of mu, so we use the convention of the (k, r, c)-th cell in a partition tuple refers to the cell in component k, row r, and column c.

INPUT:

Anything which can reasonably be interpreted as a tuple of partitions. That is, a list or tuple of partitions or valid input to Partition.

EXAMPLES:

```
sage: mu=PartitionTuple( [[3,2],[2,1],[],[1,1,1,1]] ); mu
([3, 2], [2, 1], [], [1, 1, 1, 1])
sage: nu=PartitionTuple( ([3,2],[2,1],[],[1,1,1,1]) ); nu
([3, 2], [2, 1], [], [1, 1, 1, 1])
sage: mu == nu
True
sage: mu is nu
False
sage: mu in PartitionTuples()
True
sage: mu.parent()
```

```
Partition tuples

sage: lam=PartitionTuples(3)([[3,2],[],[1,1,1,1]]); lam
([3, 2], [], [1, 1, 1, 1])
sage: lam.level()
3
sage: lam.size()
9
sage: lam.category()
Category of elements of Partition tuples of level 3
sage: lam.parent()
Partition tuples of level 3
sage: lam[0]
[3, 2]
sage: lam[1]
[]
sage: lam[2]
[1, 1, 1, 1]
sage: lam.pp()
   ***   -   *
   **        *
             *
             *
sage: lam.removable_cells()
[(0, 0, 2), (0, 1, 1), (2, 3, 0)]
sage: lam.down_list()
[([2, 2], [], [1, 1, 1, 1]), ([3, 1], [], [1, 1, 1, 1]), ([3, 2], [], [1, 1, 1])]
sage: lam.addable_cells()
[(0, 0, 3), (0, 1, 2), (0, 2, 0), (1, 0, 0), (2, 0, 1), (2, 4, 0)]
sage: lam.up_list()
[([4, 2], [], [1, 1, 1, 1]), ([3, 3], [], [1, 1, 1, 1]), ([3, 2, 1], [], [1, 1, 1, 1]), ([3, 2],
sage: lam.conjugate()
([4], [], [2, 2, 1])
sage: lam.dominates( PartitionTuple([[3],[1],[2,2,1]]) )
False
sage: lam.dominates( PartitionTuple([[3],[2],[1,1,1]]))
True
```

TESTS:

```
sage: TestSuite( PartitionTuple([4,3,2]) ).run()
sage: TestSuite( PartitionTuple([[4,3,2],[],[],[3,2,1]]) ).run()
```

See also:

- PartitionTuples
- Partitions

Element
alias of Partition

add_cell(k, r, c)
Return the partition tuple obtained by adding a cell in row r, column c, and component k.

This does not change self.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[4,3],[2,1,1]]).add_cell(0,0,1)
([2, 1], [4, 3], [2, 1, 1])
```

addable_cells()

Return a list of the removable cells of this partition tuple.

All indices are of the form `(k, r, c)`, where `r` is the row-index, `c` is the column index and `k` is the component.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[2],[2,1]]).addable_cells()
[(0, 0, 1), (0, 2, 0), (1, 0, 2), (1, 1, 0), (2, 0, 2), (2, 1, 1), (2, 2, 0)]
sage: PartitionTuple([[1,1],[4,3],[2,1,1]]).addable_cells()
[(0, 0, 1), (0, 2, 0), (1, 0, 4), (1, 1, 3), (1, 2, 0), (2, 0, 2), (2, 1, 1), (2, 3, 0)]
```

arm_length(*k*, *r*, *c*)

Return the length of the arm of cell `(k, r, c)` in `self`.

INPUT:

- `k` – The component
- `r` – The row
- `c` – The cell

OUTPUT:

- The arm length as an integer

The arm of cell `(k, r, c)` is the number of cells in the `k`-th component which are to the right of the cell in row `r` and column `c`.

EXAMPLES:

```
sage: PartitionTuple([[],[2,1],[2,2,1],[3]]).arm_length(2,0,0)
1
sage: PartitionTuple([[],[2,1],[2,2,1],[3]]).arm_length(2,0,1)
0
sage: PartitionTuple([[],[2,1],[2,2,1],[3]]).arm_length(2,2,0)
0
```

cells()

Return the coordinates of the cells of `self`. Coordinates are given as (component index, row index, column index) and are 0 based.

EXAMPLES:

```
sage: PartitionTuple([[2,1],[1],[1,1,1]]).cells()
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (2, 0, 0), (2, 1, 0), (2, 2, 0)]
```

components()

Return a list containing the shape of this partition.

This function exists in order to give a uniform way of iterating over the "components" of partition tuples of level 1 (partitions) and for higher levels.

EXAMPLE:

```
sage: for t in PartitionTuple([[2,1],[3,2],[3]]).components():  print '%s\n' % t.ferrers_dia
...
**
*
```

```
***
**

***

sage: for t in PartitionTuple([3,2]).components(): print '%s\n' % t.ferrers_diagram()
...
***
**
```

conjugate()

Return the conjugate partition tuple of `self`.

The conjugate partition tuple is obtained by reversing the order of the components and then swapping the rows and columns in each component.

EXAMPLES:

```
sage: PartitionTuple([[2,1],[1],[1,1,1]]).conjugate()
([3], [1], [2, 1])
```

contains(*mu*)

Returns `True` if this partition tuple contains μ.

If $\lambda = (\lambda^{(1)}, \ldots, \lambda^{(l)})$ and $\mu = (\mu^{(1)}, \ldots, \mu^{(m)})$ are two partition tuples then λ contains μ if $m \leq l$ and $\mu_r^{(i)} \leq \lambda_r^{(i)}$ for $1 \leq i \leq m$ and $r \geq 0$.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[2],[2,1]]).contains( PartitionTuple([[1,1],[2],[2,1]]) )
True
```

content(*k*, *r*, *c*, *multicharge*)

Returns the content of the cell.

Let $m_k =$ `multicharge[k]`, then the content of a cell is $m_k + c - r$.

If the `multicharge` is a list of integers then it simply offsets the values of the contents in each component. On the other hand, if the `multicharge` belongs to $\mathbf{Z}/e\mathbf{Z}$ then the corresponding e-residue is returned (that is, the content mod e).

As with the content method for partitions, the content of a cell does not technically depend on the partition tuple, but this method is included because it is often useful.

EXAMPLES:

```
sage: PartitionTuple([[2,1],[2],[1,1,1]]).content(0,1,0, [0,0,0])
-1
sage: PartitionTuple([[2,1],[2],[1,1,1]]).content(0,1,0, [1,0,0])
0
sage: PartitionTuple([[2,1],[2],[1,1,1]]).content(2,1,0, [0,0,0])
-1
```

and now we return the 3-residue of a cell:

```
sage: multicharge = [IntegerModRing(3)(c) for c in [0,0,0]]
sage: PartitionTuple([[2,1],[2],[1,1,1]]).content(0,1,0, multicharge)
2
```

corners()

Returns a list of the removable cells of this partition tuple.

All indice are of the form `(k, r, c)`, where `r` is the row-index, `c` is the column index and `k` is the component.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[2],[2,1]]).removable_cells()
[(0, 1, 0), (1, 0, 1), (2, 0, 1), (2, 1, 0)]
sage: PartitionTuple([[1,1],[4,3],[2,1,1]]).removable_cells()
[(0, 1, 0), (1, 0, 3), (1, 1, 2), (2, 0, 1), (2, 2, 0)]
```

diagram()

Return a string for the Ferrers diagram of `self`.

EXAMPLES:

```
sage: print PartitionTuple([[2,1],[3,2],[1,1,1]]).diagram()
   **   ***   *
   *    **    *
             *
sage: print PartitionTuple([[3,2],[2,1],[],[1,1,1,1]]).diagram()
   ***   **   -   *
   **    *        *
                  *
                  *
sage: PartitionTuples.global_options(convention="french")
sage: print PartitionTuple([[3,2],[2,1],[],[1,1,1,1]]).diagram()
                  *
                  *
   **    *        *
   ***   **   -   *
sage: PartitionTuples.global_options.reset()
```

dominates(*mu*)

Return `True` if the PartitionTuple dominates or equals μ and `False` otherwise.

Given partition tuples $\mu = (\mu^{(1)}, ..., \mu^{(m)})$ and $\nu = (\nu^{(1)}, ..., \nu^{(n)})$ then μ dominates ν if

$$\sum_{k=1}^{l-1} |\mu^{(k)}| + \sum_{r \geq 1} \mu_r^{(l)} \geq \sum_{k=1}^{l-1} |\nu^{(k)}| + \sum_{r \geq 1} \nu_r^{(l)}$$

EXAMPLES:

```
sage: mu=PartitionTuple([[1,1],[2],[2,1]])
sage: nu=PartitionTuple([[1,1],[1,1],[2,1]])
sage: mu.dominates(mu)
True
sage: mu.dominates(nu)
True
sage: nu.dominates(mu)
False
sage: tau=PartitionTuple([[],[2,1],[]])
sage: tau.dominates([[2,1],[],[]])
False
sage: tau.dominates([[],[],[2,1]])
True
```

down()

Generator (iterator) for the partition tuples that are obtained from `self` by removing a cell.

EXAMPLES:

```
sage: [mu for mu in PartitionTuple([[],[3,1],[1,1]]).down()]
[([], [2, 1], [1, 1]), ([], [3], [1, 1]), ([], [3, 1], [1])]
sage: [mu for mu in PartitionTuple([[],[],[]]).down()]
[]
```

down_list()

Return a list of the partition tuples that can be formed from `self` by removing a cell.

EXAMPLES:

```
sage: PartitionTuple([[],[3,1],[1,1]]).down_list()
[([], [2, 1], [1, 1]), ([], [3], [1, 1]), ([], [3, 1], [1])]
sage: PartitionTuple([[],[],[]]).down_list()
[]
```

ferrers_diagram()

Return a string for the Ferrers diagram of `self`.

EXAMPLES:

```
sage: print PartitionTuple([[2,1],[3,2],[1,1,1]]).diagram()
   **   ***   *
   *    **    *
             *
sage: print PartitionTuple([[3,2],[2,1],[],[1,1,1,1]]).diagram()
   ***   **   -   *
   **    *        *
                  *
                  *
sage: PartitionTuples.global_options(convention="french")
sage: print PartitionTuple([[3,2],[2,1],[],[1,1,1,1]]).diagram()
                  *
                  *
   **    *        *
   ***   **   -   *
sage: PartitionTuples.global_options.reset()
```

garnir_tableau(**cell*)

Return the Garnir tableau of shape `self` corresponding to the cell `cell`.

If `cell` $= (k, a, c)$ then $(k, a+1, c)$ must belong to the diagram of the `PartitionTuple`. If this is not the case then we return `False`.

Note: The function also sets `g._garnir_cell` equal to `cell` which is used by some other functions.

The Garnir tableaux play an important role in integral and non-semisimple representation theory because they determine the "straightening" rules for the Specht modules over an arbitrary ring.

The Garnir tableau are the "first" non-standard tableaux which arise when you act by simple transpositions. If (k, a, c) is a cell in the Young diagram of a partition, which is not at the bottom of its column, then the corresponding Garnir tableau has the integers $1, 2, \ldots, n$ entered in order from left to right along the rows of the diagram up to the cell $(k, a, c-1)$, then along the cells $(k, a+1, 1)$ to $(k, a+1, c)$, then (k, a, c) until the end of row a and then continuing from left to right in the remaining positions. The examples below probably make this clearer!

EXAMPLES:

```
sage: PartitionTuple([[5,3],[2,2],[4,3]]).garnir_tableau((0,0,2)).pp()
     1  2  6  7  8     9 10    13 14 15 16
     3  4  5          11 12    17 18 19
```

```
sage: PartitionTuple([[5,3,3],[2,2],[4,3]]).garnir_tableau((0,0,2)).pp()
     1  2  6  7  8    12 13    16 17 18 19
     3  4  5          14 15    20 21 22
     9 10 11
sage: PartitionTuple([[5,3,3],[2,2],[4,3]]).garnir_tableau((0,1,2)).pp()
     1  2  3  4  5    12 13    16 17 18 19
     6  7 11          14 15    20 21 22
     8  9 10
sage: PartitionTuple([[5,3,3],[2,2],[4,3]]).garnir_tableau((1,0,0)).pp()
     1  2  3  4  5    13 14    16 17 18 19
     6  7  8          12 15    20 21 22
     9 10 11
sage: PartitionTuple([[5,3,3],[2,2],[4,3]]).garnir_tableau((1,0,1)).pp()
     1  2  3  4  5    12 15    16 17 18 19
     6  7  8          13 14    20 21 22
     9 10 11
sage: PartitionTuple([[5,3,3],[2,2],[4,3]]).garnir_tableau((2,0,1)).pp()
     1  2  3  4  5    12 13    16 19 20 21
     6  7  8          14 15    17 18 22
     9 10 11
sage: PartitionTuple([[5,3,3],[2,2],[4,3]]).garnir_tableau((2,1,1)).pp()
Traceback (most recent call last):
...
ValueError: (comp, row+1, col) must be inside the diagram
```

See also:

- top_garnir_tableau()

hook_length(*k*, *r*, *c*)

Return the length of the hook of cell `(k, r, c)` in the partition.

The hook of cell `(k, r, c)` is defined as the cells to the right or below (in the English convention). If your coordinates are in the form `(k,r,c)`, use Python's *-operator.

EXAMPLES:

```
sage: mu=PartitionTuple([[1,1],[2],[2,1]])
sage: [ mu.hook_length(*c) for c in mu.cells()]
[2, 1, 2, 1, 3, 1, 1]
```

initial_tableau()

Return the StandardTableauTuple which has the numbers $1, 2, \ldots, n$, where n is the size() of self, entered in order from left to right along the rows of each component, where the components are ordered from left to right.

EXAMPLE:

```
sage: PartitionTuple([ [2,1],[3,2] ]).initial_tableau()
([[1, 2], [3]], [[4, 5, 6], [7, 8]])
```

leg_length(*k*, *r*, *c*)

Return the length of the leg of cell `(k, r, c)` in self.

INPUT:

- k – The component
- r – The row

- c – The cell

OUTPUT:

- The leg length as an integer

The leg of cell (k, r, c) is the number of cells in the k-th component which are below the node in row r and column c.

EXAMPLES:

```
sage: PartitionTuple([[],[2,1],[2,2,1],[3]]).leg_length(2,0,0)
2
sage: PartitionTuple([[],[2,1],[2,2,1],[3]]).leg_length(2,0,1)
1
sage: PartitionTuple([[],[2,1],[2,2,1],[3]]).leg_length(2,2,0)
0
```

level()

Return the level of this partition tuple.

The level is the length of the tuple.

EXAMPLES:

```
sage: PartitionTuple([[2,1,1,0],[2,1]]).level()
2
sage: PartitionTuple([[],[],[2,1,1]]).level()
3
```

outside_corners()

Return a list of the removable cells of this partition tuple.

All indices are of the form (k, r, c), where r is the row-index, c is the column index and k is the component.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[2],[2,1]]).addable_cells()
[(0, 0, 1), (0, 2, 0), (1, 0, 2), (1, 1, 0), (2, 0, 2), (2, 1, 1), (2, 2, 0)]
sage: PartitionTuple([[1,1],[4,3],[2,1,1]]).addable_cells()
[(0, 0, 1), (0, 2, 0), (1, 0, 4), (1, 1, 3), (1, 2, 0), (2, 0, 2), (2, 1, 1), (2, 3, 0)]
```

pp()

Pretty prints this partition tuple. See diagram().

EXAMPLES:

```
sage: PartitionTuple([[5,5,2,1],[3,2]]).pp()
*****   ***
*****   **
**
*
```

removable_cells()

Returns a list of the removable cells of this partition tuple.

All indice are of the form (k, r, c), where r is the row-index, c is the column index and k is the component.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[2],[2,1]]).removable_cells()
[(0, 1, 0), (1, 0, 1), (2, 0, 1), (2, 1, 0)]
```

```
sage: PartitionTuple([[1,1],[4,3],[2,1,1]]).removable_cells()
[(0, 1, 0), (1, 0, 3), (1, 1, 2), (2, 0, 1), (2, 2, 0)]
```

remove_cell(*k*, *r*, *c*)

Return the partition tuple obtained by removing a cell in row `r`, column `c`, and component `k`.

This does not change `self`.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[4,3],[2,1,1]]).remove_cell(0,1,0)
([1], [4, 3], [2, 1, 1])
```

size()

Return the size of a partition tuple.

EXAMPLES:

```
sage: PartitionTuple([[2,1],[],[2,2]]).size()
7
sage: PartitionTuple([[],[],[1],[3,2,1]]).size()
7
```

standard_tableaux()

Return the standard tableau tuples of this shape.

EXAMPLE:

```
sage: PartitionTuple([[],[3,2,2,1],[2,2,1],[3]]).standard_tableaux()
Standard tableau tuples of shape ([], [3, 2, 2, 1], [2, 2, 1], [3])
```

to_exp(*k=0*)

Return a tuple of the multiplicities of the parts of a partition.

Use the optional parameter `k` to get a return list of length at least `k`.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[2],[2,1]]).to_exp()
([2], [0, 1], [1, 1])
sage: PartitionTuple([[1,1],[2,2,2,2],[2,1]]).to_exp()
([2], [0, 4], [1, 1])
```

to_list()

Return `self` as a list of lists.

EXAMPLES:

```
sage: PartitionTuple([[1,1],[4,3],[2,1,1]]).to_list()
[[1, 1], [4, 3], [2, 1, 1]]
```

TESTS:

```
sage: all(mu==PartitionTuple(mu.to_list()) for mu in PartitionTuples(4,4))
True
```

top_garnir_tableau(*e*, *cell*)

Return the most dominant *standard* tableau which dominates the corresponding Garnir tableau and has the same residue that has shape `self` and is determined by `e` and `cell`.

The Garnir tableau play an important role in integral and non-semisimple representation theory because they determine the "straightening" rules for the Specht modules over an arbitrary ring. The *top Garnir*

tableaux arise in the graded representation theory of the symmetric groups and higher level Hecke algebras. They were introduced in [KMR].

If the Garnir node is `cell=(k,r,c)` and m and M are the entries in the cells `(k,r,c)` and `(k,r+1,c)`, respectively, in the initial tableau then the top `e`-Garnir tableau is obtained by inserting the numbers $m, m+1, \ldots, M$ in order from left to right first in the cells in row `r+1` which are not in the `e`-Garnir belt, then in the cell in rows `r` and `r+1` which are in the Garnir belt and then, finally, in the remaining cells in row `r` which are not in the Garnir belt. All other entries in the tableau remain unchanged.

If `e = 0`, or if there are no `e`-bricks in either row `r` or `r+1`, then the top Garnir tableau is the corresponding Garnir tableau.

EXAMPLES:

```
sage: PartitionTuple([[3,3,2],[5,4,3,2]]).top_garnir_tableau(2,(1,0,2)).pp()
    1  2  3     9 10 12 13 16
    4  5  6    11 14 15 17
    7  8       18 19 20
               21 22
sage: PartitionTuple([[3,3,2],[5,4,3,2]]).top_garnir_tableau(2,(1,0,1)).pp()
    1  2  3     9 10 11 12 13
    4  5  6    14 15 16 17
    7  8       18 19 20
               21 22
sage: PartitionTuple([[3,3,2],[5,4,3,2]]).top_garnir_tableau(3,(1,0,1)).pp()
    1  2  3     9 12 13 14 15
    4  5  6    10 11 16 17
    7  8       18 19 20
               21 22

sage: PartitionTuple([[3,3,2],[5,4,3,2]]).top_garnir_tableau(3,(3,0,1)).pp()
Traceback (most recent call last):
...
ValueError: (comp, row+1, col) must be inside the diagram
```

See also:

- `garnir_tableau()`

REFERENCE:

up()

Generator (iterator) for the partition tuples that are obtained from `self` by adding a cell.

EXAMPLES:

```
sage: [mu for mu in PartitionTuple([[],[3,1],[1,1]]).up()]
[([1], [3, 1], [1, 1]), ([], [4, 1], [1, 1]), ([], [3, 2], [1, 1]), ([], [3, 1, 1], [1, 1]),
sage: [mu for mu in PartitionTuple([[],[],[],[]]).up()]
[([1], [], [], []), ([], [1], [], []), ([], [], [1], []), ([], [], [], [1])]
```

up_list()

Return a list of the partition tuples that can be formed from `self` by adding a cell.

EXAMPLES:

```
sage: PartitionTuple([[],[3,1],[1,1]]).up_list()
[([1], [3, 1], [1, 1]), ([], [4, 1], [1, 1]), ([], [3, 2], [1, 1]), ([], [3, 1, 1], [1, 1]),
sage: PartitionTuple([[],[],[],[]]).up_list()
[([1], [], [], []), ([], [1], [], []), ([], [], [1], []), ([], [], [], [1])]
```

young_subgroup()
Return the corresponding Young, or parabolic, subgroup of the symmetric group.

EXAMPLE:

```
sage: PartitionTuple([[2,1],[4,2],[1]]).young_subgroup()
Permutation Group with generators [(), (8,9), (6,7), (5,6), (4,5), (1,2)]
```

young_subgroup_generators()
Return an indexing set for the generators of the corresponding Young subgroup.

EXAMPLE:

```
sage: PartitionTuple([[2,1],[4,2],[1]]).young_subgroup_generators()
[1, 4, 5, 6, 8]
```

class sage.combinat.partition_tuple.**PartitionTuples**
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

List of all partition tuples.

For more information about partition tuples, see PartitionTuple.

This is a factory class which returns the appropriate parent based on the values of level and size.

INPUT:

- level – The length of the tuple
- size – The total number of cells

TESTS:

```
sage: TestSuite( PartitionTuples() ).run()
sage: TestSuite( PartitionTuples(level=4) ).run()
sage: TestSuite( PartitionTuples(size=6) ).run()
sage: TestSuite( PartitionTuples(level=4, size=5) ).run()

sage: [ [2,1],[],[3] ] in PartitionTuples()
True
sage: ( [2,1],[],[3] ) in PartitionTuples()
True
sage: ( [] ) in PartitionTuples()
True
```

Check that trac ticket #14145 has been fixed:

```
sage: 1 in PartitionTuples()
False
```

Element
alias of PartitionTuple

global_options(**get_value*, ***set_value*)
Sets and displays the global options for elements of the partition, skew partition, and partition tuple classes. If no parameters are set, then the function returns a copy of the options dictionary.

The options to partitions can be accessed as the method Partitions.global_options of Partitions and related parent classes.

OPTIONS:

- convention – (default: English) Sets the convention used for displaying tableaux and partitions

– `English` – use the English convention

– `French` – use the French convention

- `diagram_str` – (default: `*`) The character used for the cells when printing Ferrers diagrams
- `display` – (default: `list`) Specifies how partitions should be printed
  - `array` – alias for `diagram`
  - `compact` – alias for `compact_low`
  - `compact_high` – compact form of `exp_high`
  - `compact_low` – compact form of `exp_low`
  - `diagram` – as a Ferrers diagram
  - `exp` – alias for `exp_low`
  - `exp_high` – in exponential form (highest first)
  - `exp_low` – in exponential form (lowest first)
  - `ferrers_diagram` – alias for `diagram`
  - `list` – displayed as a list
  - `young_diagram` – alias for `diagram`
- `latex` – (default: `young_diagram`) Specifies how partitions should be latexed
  - `array` – alias for `diagram`
  - `diagram` – latex as a Ferrers diagram
  - `exp` – alias for `exp_low`
  - `exp_high` – latex as a list in exponential notation (highest first)
  - `exp_low` – as a list latex in exponential notation (lowest first)
  - `ferrers_diagram` – alias for `diagram`
  - `list` – latex as a list
  - `young_diagram` – latex as a Young diagram
- `latex_diagram_str` – (default: `\ast`) The character used for the cells when latexing Ferrers diagrams
- `notation` – alternative name for `convention`

EXAMPLES:

```
sage: P = Partition([4,2,2,1])
sage: P
[4, 2, 2, 1]
sage: Partitions.global_options(display="exp")
sage: P
1, 2^2, 4
sage: Partitions.global_options(display="exp_high")
sage: P
4, 2^2, 1
```

It is also possible to use user defined functions for the `display` and `latex` options:

```
sage: Partitions.global_options(display=lambda mu: '<%s>' % ','.join('%s'%m for m in mu._lis
<4,2,2,1>
sage: Partitions.global_options(latex=lambda mu: '\\Diagram{%s}' % ','.join('%s'%m for m in
\Diagram{4,2,2,1}
sage: Partitions.global_options(display="diagram", diagram_str="#")
sage: P
####
##
##
#
sage: Partitions.global_options(diagram_str="*", convention="french")
sage: print P.ferrers_diagram()
*
**
**
****
```

Changing the `convention` for partitions also changes the `convention` option for tableaux and vice versa:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T.pp()
  4  5
  1  2  3
sage: Tableaux.global_options(convention="english")
sage: print P.ferrers_diagram()
****
**
**
*
sage: T.pp()
  1  2  3
  4  5
sage: Partitions.global_options.reset()
```

See `GlobalOptions` for more features of these options.

level()

Return the level or `None` if it is not defined.

EXAMPLES:

```
sage: PartitionTuples().level() is None
True
sage: PartitionTuples(7).level()
7
```

size()

Return the size or `None` if it is not defined.

EXAMPLES:

```
sage: PartitionTuples().size() is None
True
sage: PartitionTuples(size=7).size()
7
```

class sage.combinat.partition_tuple.**PartitionTuples_all**

Bases: sage.combinat.partition_tuple.PartitionTuples

Class of partition tuples of a arbitrary level and arbitrary sum.

class sage.combinat.partition_tuple.**PartitionTuples_level**(*level*)
Bases: sage.combinat.partition_tuple.PartitionTuples

Class of partition tuples of a fixed level, but summing to an arbitrary integer.

class sage.combinat.partition_tuple.**PartitionTuples_level_size**(*level*, *size*)
Bases: sage.combinat.partition_tuple.PartitionTuples

Class of partition tuples with a fixed level and a fixed size.

cardinality()
Returns the number of `level`-tuples of partitions of size `n`.

Wraps a pari function call.

EXAMPLES:

```
sage: PartitionTuples(2,3).cardinality()
10
sage: PartitionTuples(2,8).cardinality()
185
```

TESTS:

The following calls used to fail (trac ticket #11476):

```
sage: PartitionTuples(17,2).cardinality()
170
sage: PartitionTuples(2,17).cardinality()
8470
sage: PartitionTuples(100,13).cardinality()
1103200201478868 00
sage: PartitionTuples(13,90).cardinality()
915064737412001861523528436 11
```

These answers were checked against Gap4 (the last of which takes an awful long time for gap to compute).

class sage.combinat.partition_tuple.**PartitionTuples_size**(*size*)
Bases: sage.combinat.partition_tuple.PartitionTuples

Class of partition tuples of a fixed size, but arbitrary level.

5.1.145 Number of partitions of an integer

AUTHOR:

- William Stein (2007-07-28): initial version
- Jonathan Bober (2007-07-28): wrote the program `partitions_c.cc` that does all the actual heavy lifting.

sage.combinat.partitions.**ZS1_iterator**(*n*)
A fast iterator for the partitions of `n` (in the decreasing lexicographic order) which returns lists and not objects of type `Partition`.

This is an implementation of the ZS1 algorithm found in [ZS98].

REFERENCES:

EXAMPLES:

```
sage: from sage.combinat.partitions import ZS1_iterator
sage: it = ZS1_iterator(4)
sage: next(it)
[4]
sage: type(_)
<type 'list'>
```

sage.combinat.partitions.**ZS1_iterator_nk**(*n*, *k*)

An iterator for the partitions of `n` of length at most `k` (in the decreasing lexicographic order) which returns lists and not objects of type `Partition`.

The algorithm is a mild variation on `ZS1_iterator()`; I would not vow for its speed.

EXAMPLES:

```
sage: from sage.combinat.partitions import ZS1_iterator_nk
sage: it = ZS1_iterator_nk(4, 3)
sage: next(it)
[4]
sage: type(_)
<type 'list'>
```

sage.combinat.partitions.**number_of_partitions**(*n*)

Returns the number of partitions of the integer n.

EXAMPLES:

```
sage: from sage.combinat.partitions import number_of_partitions
sage: number_of_partitions(3)
3
sage: number_of_partitions(10)
42
sage: number_of_partitions(40)
37338
sage: number_of_partitions(100)
190569292
sage: number_of_partitions(100000)
2749351056977569651267751632098635268817342931598005475820312598430214732811496417305550507416607
```

TESTS:

```
sage: n = 500 + randint(0,500)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1500 + randint(0,1500)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
```

```
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 100000000 + randint(0,100000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0  # long time (4s on sage.math, 2011)
True
```

Another consistency test for n up to 500:

```
sage: len([n for n in [1..500] if number_of_partitions(n) != Partitions(n).cardinality(algorithm
0
```

sage.combinat.partitions.**run_tests**(*longtest=False*, *forever=False*)

Test Bober's algorithm.

EXAMPLES:

```
sage: from sage.combinat.partitions import run_tests
sage: run_tests(False, False)
Computing p(1)... OK.
...
Done.
```

5.1.146 Perfect matchings

A perfect matching of a set S is a partition into 2-element sets. If S is the set $\{1, ..., n\}$, it is equivalent to fixpoint-free involutions. These simple combinatorial objects appear in different domains such as combinatoric of orthogonal polynomials and of the hyperoctaedral groups (see [MV], [McD] and also [CM]):

AUTHOR:

- Valentin Feray, 2010 : initial version

EXAMPLES:

Create a perfect matching:

```
sage: m = PerfectMatching([('a','e'),('b','c'),('d','f')]);m
[('a', 'e'), ('b', 'c'), ('d', 'f')]
```

Count its crossings, if the ground set is totally ordered:

```
sage: n = PerfectMatching([3,8,1,7,6,5,4,2]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
sage: n.number_of_crossings()
1
```

List the perfect matchings of a given ground set:

```
sage: PerfectMatchings(4).list()
[[(1, 2), (3, 4)], [(1, 3), (2, 4)], [(1, 4), (2, 3)]]
```

REFERENCES:

class sage.combinat.perfect_matching.**PerfectMatching**

Bases: sage.structure.element_wrapper.ElementWrapper

Class of perfect matching.

An instance of the class can be created from a list of pairs or from a fixed point-free involution as follows:

```
sage: m = PerfectMatching([('a','e'),('b','c'),('d','f')]);m
[('a', 'e'), ('b', 'c'), ('d', 'f')]
sage: n = PerfectMatching([3,8,1,7,6,5,4,2]);n
[(1, 3), (2, 8), (4, 7), (5, 6)]
sage: isinstance(m,PerfectMatching)
True
```

The parent, which is the set of perfect matchings of the ground set, is automatically created:

```
sage: n.parent()
Set of perfect matchings of {1, 2, 3, 4, 5, 6, 7, 8}
```

If the ground set is ordered, one can, for example, ask if the matching is non crossing:

```
sage: PerfectMatching([(1, 4), (2, 3), (5, 6)]).is_non_crossing()
True
```

TESTS:

```
sage: m = PerfectMatching([]); m
[]
sage: m.parent()
Set of perfect matchings of {}
```

Weingarten_function(*d*, *other=None*)

Returns the Weingarten function of two pairings.

This function is the value of some integrals over the orthogonal groups O_N. With the convention of [CM], the method returns $Wg^{O(d)}(other, self)$.

EXAMPLES:

```
sage: var('N')
N
sage: m = PerfectMatching([(1,3),(2,4)])
sage: n = PerfectMatching([(1,2),(3,4)])
sage: factor(m.Weingarten_function(N,n))
-1/((N + 2)*(N - 1)*N)
```

conjugate_by_permutation(*p*)

Returns the conjugate of the perfect matching `self` by the permutation `p` of the ground set.

EXAMPLE:

```
sage: m = PerfectMatching([(1,4),(2,6),(3,5)])
sage: m.conjugate_by_permutation(Permutation([4,1,5,6,3,2]))
[(4, 6), (1, 2), (5, 3)]
```

TEST:

```
sage: PerfectMatching([]).conjugate_by_permutation(Permutation([]))
[]
```

crossings()

INPUT:

A perfect matching on a *totally ordered* ground set.

OUTPUT:

We place the element of a ground set and draw the perfect matching by linking the elements of the same pair in the upper half-plane. This function returns the list of the pairs of crossing lines (as a line correspond to a pair, it returns a list of pairs of pairs).

EXAMPLES:

```
sage: n = PerfectMatching([3,8,1,7,6,5,4,2]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
sage: n.crossings()
[((1, 3), (2, 8))]
```

TESTS:

```
sage: m = PerfectMatching([]); m.crossings()
[]
```

crossings_iterator()

INPUT:

A perfect matching on a *totally ordered* ground set.

OUTPUT:

We place the element of a ground set and draw the perfect matching by linking the elements of the same pair in the upper half-plane. This function returns an iterator over the pairs of crossing lines (as a line correspond to a pair, the iterator produces pairs of pairs).

EXAMPLES:

```
sage: n = PerfectMatching([3,8,1,7,6,5,4,2]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
sage: it = n.crossings_iterator();
sage: next(it)
((1, 3), (2, 8))
sage: next(it)
Traceback (most recent call last):
...
StopIteration
```

is_non_crossing()

INPUT:

A perfect matching on a *totally ordered* ground set.

OUTPUT:

We place the element of a ground set and draw the perfect matching by linking the elements of the same pair in the upper half-plane. This function returns `True` if the picture obtained this way has no crossings.

EXAMPLES:

```
sage: n = PerfectMatching([3,8,1,7,6,5,4,2]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
sage: n.is_non_crossing()
False
sage: PerfectMatching([(1, 4), (2, 3), (5, 6)]).is_non_crossing()
True
```

is_non_nesting()

INPUT:

A perfect matching on a *totally ordered* ground set.

OUTPUT:

We place the element of a ground set and draw the perfect matching by linking the elements of the same pair in the upper half-plane. This function returns `True` if the picture obtained this way has no nestings.

EXAMPLES:

```
sage: n = PerfectMatching([3,8,1,7,6,5,4,2]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
sage: n.is_non_nesting()
False
sage: PerfectMatching([(1, 3), (2, 5), (4, 6)]).is_non_nesting()
True
```

loop_type(*other=None*)

INPUT:

- `other` – a perfect matching of the same set of `self`. (if the second argument is empty, the method `an_element()` is called on the parent of the first)

OUTPUT:

If we draw the two perfect matchings simultaneously as edges of a graph, the graph obtained is a union of cycles of even lengths. The function returns the ordered list of the semi-length of these cycles (considered as a partition)

EXAMPLES:

```
sage: m = PerfectMatching([('a','e'),('b','c'),('d','f')])
sage: n = PerfectMatching([('a','b'),('d','f'),('e','c')])
sage: m.loop_type(n)
[2, 1]
```

TESTS:

```
sage: m = PerfectMatching([]); m.loop_type()
[]
```

loops(*other=None*)

INPUT:

- `other` – a perfect matching of the same set of `self`. (if the second argument is empty, the method `an_element()` is called on the parent of the first)

OUTPUT:

If we draw the two perfect matchings simultaneously as edges of a graph, the graph obtained is a union of cycles of even lengths. The function returns the list of these cycles (each cycle is given as a list).

EXAMPLES:

```
sage: m = PerfectMatching([('a','e'),('b','c'),('d','f')])
sage: n = PerfectMatching([('a','b'),('d','f'),('e','c')])
sage: m.loops(n)
[['a', 'e', 'c', 'b'], ['d', 'f']]

sage: o = PerfectMatching([(1, 7), (2, 4), (3, 8), (5, 6)])
sage: p = PerfectMatching([(1, 6), (2, 7), (3, 4), (5, 8)])
sage: o.loops(p)
[[1, 7, 2, 4, 3, 8, 5, 6]]
```

loops_iterator(*other=None*)

INPUT:

- other – a perfect matching of the same set of self. (if the second argument is empty, the method an_element() is called on the parent of the first)

OUTPUT:

If we draw the two perfect matchings simultaneously as edges of a graph, the graph obtained is a union of cycles of even lengths. The function returns an iterator for these cycles (each cycle is given as a list).

EXAMPLES:

```
sage: o = PerfectMatching([(1, 7), (2, 4), (3, 8), (5, 6)])
sage: p = PerfectMatching([(1, 6), (2, 7), (3, 4), (5, 8)])
sage: it = o.loops_iterator(p)
sage: next(it)
[1, 7, 2, 4, 3, 8, 5, 6]
sage: next(it)
Traceback (most recent call last):
...
StopIteration
```

nestings()

INPUT:

A perfect matching on a *totally ordered* ground set.

OUTPUT:

We place the element of a ground set and draw the perfect matching by linking the elements of the same pair in the upper half-plane. This function returns the list of the pairs of nesting lines (as a line correspond to a pair, it returns a list of pairs of pairs).

EXAMPLES:

```
sage: m = PerfectMatching([(1, 6), (2, 7), (3, 5), (4, 8)])
sage: m.nestings()
[((1, 6), (3, 5)), ((2, 7), (3, 5))]

sage: n = PerfectMatching([3,8,1,7,6,5,4,2]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
sage: n.nestings()
[((2, 8), (4, 7)), ((2, 8), (5, 6)), ((4, 7), (5, 6))]
```

TESTS:

```
sage: m = PerfectMatching([]); m.nestings()
[]
```

nestings_iterator()

INPUT:

A perfect matching on a *totally ordered* ground set.

OUTPUT:

We place the element of a ground set and draw the perfect matching by linking the elements of the same pair in the upper half-plane. This function returns an iterator over the pairs of nesting lines (as a line correspond to a pair, the iterator produces pairs of pairs).

EXAMPLES:

```
sage: n = PerfectMatching([(1, 6), (2, 7), (3, 5), (4, 8)])
sage: it = n.nestings_iterator();
sage: next(it)
((1, 6), (3, 5))
sage: next(it)
((2, 7), (3, 5))
sage: next(it)
Traceback (most recent call last):
...
StopIteration
```

number_of_crossings()

INPUT:

A perfect matching on a *totally ordered* ground set.

OUTPUT:

We place the element of a ground set and draw the perfect matching by linking the elements of the same pair in the upper half-plane. This function returns the number the pairs of crossing lines.

EXAMPLES:

```
sage: n = PerfectMatching([3,8,1,7,6,5,4,2]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
sage: n.number_of_crossings()
1
```

number_of_loops(*other=None*)

INPUT:

- `other` – a perfect matching of the same set of `self`. (if the second argument is empty, the method `an_element()` is called on the parent of the first)

OUTPUT:

If we draw the two perfect matchings simultaneously as edges of a graph, the graph obtained is a union of cycles of even lengths. The function returns their numbers.

EXAMPLES:

```
sage: m = PerfectMatching([('a','e'),('b','c'),('d','f')])
sage: n = PerfectMatching([('a','b'),('d','f'),('e','c')])
sage: m.number_of_loops(n)
2
```

number_of_nestings()

INPUT:

A perfect matching on a *totally ordered* ground set.

OUTPUT:

We place the element of a ground set and draw the perfect matching by linking the elements of the same pair in the upper half-plane. This function returns the number the pairs of nesting lines.

EXAMPLES:

```
sage: n = PerfectMatching([3,8,1,7,6,5,4,2]); n
[(1, 3), (2, 8), (4, 7), (5, 6)]
sage: n.number_of_nestings()
3
```

partner(*x*)
Returns the element in the same pair than `x` in the matching `self`.

EXAMPLES:

```
sage: m = PerfectMatching([(-3, 1), (2, 4), (-2, 7)]); m.partner(4)
2
sage: n = PerfectMatching([('c','b'),('d','f'),('e','a')])
sage: n.partner('c')
'b'
```

size()
Returns the size of the perfect matching `self`, i.e. the number of elements in the ground set.

EXAMPLES:

```
sage: m = PerfectMatching([(-3, 1), (2, 4), (-2, 7)]); m.size()
6
```

to_graph()
Returns the graph corresponding to the perfect matching.

OUTPUT:

The realization of `self` as a graph.

EXAMPLES:

```
sage: PerfectMatching([[1,3], [4,2]]).to_graph().edges(labels=False)
[(1, 3), (2, 4)]
sage: PerfectMatching([[1,4], [3,2]]).to_graph().edges(labels=False)
[(1, 4), (2, 3)]
sage: PerfectMatching([]).to_graph().edges(labels=False)
[]
```

to_non_crossing_set_partition()
Returns the noncrossing set partition (on half as many elements) corresponding to the perfect matching if the perfect matching is noncrossing, and otherwise gives an error.

OUTPUT:

The realization of `self` as a noncrossing set partition.

EXAMPLES:

```
sage: PerfectMatching([[1,3], [4,2]]).to_non_crossing_set_partition()
Traceback (most recent call last):
...
ValueError: matching must be non-crossing
sage: PerfectMatching([[1,4], [3,2]]).to_non_crossing_set_partition()
{{1, 2}}
sage: PerfectMatching([]).to_non_crossing_set_partition()
{}
```

to_permutation()
Returns the permutation corresponding to the perfect matching.

OUTPUT:

The realization of `self` as a permutation.

EXAMPLES:

```
sage: PerfectMatching([[1,3], [4,2]]).to_permutation()
[3, 4, 1, 2]
sage: PerfectMatching([[1,4], [3,2]]).to_permutation()
[4, 3, 2, 1]
sage: PerfectMatching([]).to_permutation()
[]
```

class sage.combinat.perfect_matching.**PerfectMatchings**(*objects*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Class of perfect matchings of a ground set. At the creation, the set can be given as any iterable object. If the argument is an integer n, it will be transformed into $[1..n]$:

```
sage: M = PerfectMatchings(6);M
Set of perfect matchings of {1, 2, 3, 4, 5, 6}
sage: PerfectMatchings([-1, -3, 1, 2])
Set of perfect matchings of {1, 2, -3, -1}
```

One can ask for the list, the cardinality or an element of a set of perfect matching:

```
sage: PerfectMatchings(4).list()
[[(1, 2), (3, 4)], [(1, 3), (2, 4)], [(1, 4), (2, 3)]]
sage: PerfectMatchings(8).cardinality()
105
sage: M = PerfectMatchings(('a', 'e', 'b', 'f', 'c', 'd'))
sage: M.an_element()
[('a', 'b'), ('f', 'e'), ('c', 'd')]
sage: all([PerfectMatchings(i).an_element() in PerfectMatchings(i)
...        for i in range(2,11,2)])
True
```

TESTS:

```
sage: PerfectMatchings(5).list()
[]
sage: TestSuite(PerfectMatchings(6)).run()
sage: TestSuite(PerfectMatchings([])).run()
```

Element

alias of PerfectMatching

Weingarten_matrix(*N*)

Returns the Weingarten matrix corresponding to the set of PerfectMatchings self.

It is a useful theoretical tool to compute polynomial integral over the orthogonal group O_N (see [CM]).

EXAMPLES:

```
sage: M = PerfectMatchings(4).Weingarten_matrix(var('N'))
sage: N*(N-1)*(N+2)*M.apply_map(factor)
[N + 1    -1    -1]
[   -1 N + 1    -1]
[   -1    -1 N + 1]
```

an_element()

Returns a random element of self.

EXAMPLES:

```
sage: M = PerfectMatchings(('a', 'e', 'b', 'f', 'c', 'd'))
sage: M.an_element()
[('a', 'b'), ('f', 'e'), ('c', 'd')]
sage: all([PerfectMatchings(2*i).an_element() in PerfectMatchings(2*i)
...        for i in range(2,11,2)])
True
```

TESTS:

```
sage: p = PerfectMatchings(13).random_element()
Traceback (most recent call last):
...
ValueError: there is no perfect matching on an odd number of elements
```

cardinality()

Returns the cardinality of the set of perfect matching `self`

This is $1 * 3 * 5 * ... * (2n - 1)$, where $2n$ is the size of the ground set.

EXAMPLES:

```
sage: PerfectMatchings(8).cardinality()
105
```

random_element()

Returns a random element of self.

EXAMPLES:

```
sage: M = PerfectMatchings(('a', 'e', 'b', 'f', 'c', 'd'))
sage: M.an_element()
[('a', 'b'), ('f', 'e'), ('c', 'd')]
sage: all([PerfectMatchings(2*i).an_element() in PerfectMatchings(2*i)
...        for i in range(2,11,2)])
True
```

TESTS:

```
sage: p = PerfectMatchings(13).random_element()
Traceback (most recent call last):
...
ValueError: there is no perfect matching on an odd number of elements
```

5.1.147 Permutations

The Permutations module. Use `Permutation?` to get information about the Permutation class, and `Permutations?` to get information about the combinatorial class of permutations.

Warning: This file defined `Permutation` which depends upon `CombinatorialElement` despite it being deprecated (see trac ticket #13742). This is dangerous. In particular, the `Permutation._left_to_right_multiply_on_right()` method (which can be called trough multiplication) disables the input checks (see `Permutation()`). This should not happen. Do not trust the results.

What does this file define ?

The main part of this file consists in the definition of permutation objects, i.e. the `Permutation()` method and the `Permutation` class. Global options for elements of the permutation class can be set through the `PermutationOptions` object.

Below are listed all methods and classes defined in this file.

Methods of Permutations objects

`left_action_product()`	Returns the product of `self` with another permutation, in which the oth
`right_action_product()`	Returns the product of `self` with another permutation, in which `self`
`size()`	Returns the size of the permutation `self`.
`cycle_string()`	Returns the disjoint-cycles representation of `self` as string.
`next()`	Returns the permutation that follows `self` in lexicographic order (in the
`prev()`	Returns the permutation that comes directly before `self` in lexicograph
`to_tableau_by_shape()`	Returns a tableau of shape `shape` with the entries in `self`.
`to_cycles()`	Returns the permutation `self` as a list of disjoint cycles.
`forget_cycles()`	Return `self` under the forget cycle map.
`to_permutation_group_element()`	Returns a `PermutationGroupElement` equal to `self`.
`signature()`	Returns the signature of the permutation `sef`.
`is_even()`	Returns `True` if the permutation `self` is even, and `False` otherwise.
`to_matrix()`	Returns a matrix representing the permutation `self`.
`rank()`	Returns the rank of `self` in lexicographic ordering (on the symmetric g
`to_inversion_vector()`	Returns the inversion vector of a permutation `self`.
`inversions()`	Returns a list of the inversions of permutation `self`.
`show()`	Displays the permutation as a drawing.
`number_of_inversions()`	Returns the number of inversions in the permutation `self`.
`noninversions()`	Returns the `k`-noninversions in the permutation `self`.
`number_of_noninversions()`	Returns the number of `k`-noninversions in the permutation `self`.
`length()`	Returns the Coxeter length of a permutation `self`.
`inverse()`	Returns the inverse of a permutation `self`.
`ishift()`	Returns the `i`-shift of `self`.
`iswitch()`	Returns the `i`-switch of `self`.
`runs()`	Returns a list of the runs in the permutation `self`.
`longest_increasing_subsequence_length()`	Returns the length of the longest increasing subsequences of `self`.
`longest_increasing_subsequences()`	Returns the list of the longest increasing subsequences of `self`.
`cycle_type()`	Returns the cycle type of `self` as a partition of `len(self)`.
`foata_bijection()`	Returns the image of the permutation `self` under the Foata bijection ϕ.
`destandardization()`	Return destandardization of `self` with respect to `weight` and `ordere`
`to_lehmer_code()`	Returns the Lehmer code of the permutation `self`.
`to_lehmer_cocode()`	Returns the Lehmer cocode of `self`.
`reduced_word()`	Returns the reduced word of the permutation `self`.
`reduced_words()`	Returns a list of the reduced words of the permutation `self`.
`reduced_word_lexmin()`	Returns a lexicographically minimal reduced word of a permutation `sel`
`fixed_points()`	Returns a list of the fixed points of the permutation `self`.
`number_of_fixed_points()`	Returns the number of fixed points of the permutation `self`.
`recoils()`	Returns the list of the positions of the recoils of the permutation `self`.
`number_of_recoils()`	Returns the number of recoils of the permutation `self`.
`recoils_composition()`	Returns the composition corresponding to the recoils of `self`.
`descents()`	Returns the list of the descents of the permutation `self`.
`idescents()`	Returns a list of the idescents of `self`.
`idescents_signature()`	Returns the list obtained by mapping each position in `self` to -1 if it is
`number_of_descents()`	Returns the number of descents of the permutation `self`.

Table 5.1 – continued from previous page

number_of_idescents()	Returns the number of idescents of the permutation self.
descents_composition()	Returns the composition corresponding to the descents of self.
descent_polynomial()	Returns the descent polynomial of the permutation self.
major_index()	Returns the major index of the permutation self.
imajor_index()	Returns the inverse major index of the permutation self.
to_major_code()	Returns the major code of the permutation self.
peaks()	Returns a list of the peaks of the permutation self.
number_of_peaks()	Returns the number of peaks of the permutation self.
saliances()	Returns a list of the saliances of the permutation self.
number_of_saliances()	Returns the number of saliances of the permutation self.
bruhat_lequal()	Returns True if self is less or equal to p2 in the Bruhat order.
weak_excedences()	Returns all the numbers self[i] such that self[i] >= i+1.
bruhat_inversions()	Returns the list of inversions of self such that the application of this in
bruhat_inversions_iterator()	Returns an iterator over Bruhat inversions of self.
bruhat_succ()	Returns a list of the permutations covering self in the Bruhat order.
bruhat_succ_iterator()	An iterator for the permutations covering self in the Bruhat order.
bruhat_pred()	Returns a list of the permutations covered by self in the Bruhat order.
bruhat_pred_iterator()	An iterator for the permutations covered by self in the Bruhat order.
bruhat_smaller()	Returns the combinatorial class of permutations smaller than or equal to
bruhat_greater()	Returns the combinatorial class of permutations greater than or equal to
permutohedron_lequal()	Returns True if self is less or equal to p2 in the permutohedron order
permutohedron_succ()	Returns a list of the permutations covering self in the permutohedron o
permutohedron_pred()	Returns a list of the permutations covered by self in the permutohedro
permutohedron_smaller()	Returns a list of permutations smaller than or equal to self in the perm
permutohedron_greater()	Returns a list of permutations greater than or equal to self in the permu
right_permutohedron_interval_iterator()	Returns an iterator over permutations in an interval of the permutohedro
right_permutohedron_interval()	Returns a list of permutations in an interval of the permutohedron order.
has_pattern()	Tests whether the permutation self matches the pattern.
avoids()	Tests whether the permutation self avoids the pattern.
pattern_positions()	Returns the list of positions where the pattern patt appears in self.
reverse()	Returns the permutation obtained by reversing the 1-line notation of sel
complement()	Returns the complement of the permutation which is obtained by replaci
permutation_poset()	Returns the permutation poset of self.
dict()	Returns a dictionary corresponding to the permutation self.
action()	Returns the action of the permutation self on a list.
robinson_schensted()	Returns the pair of standard tableaux obtained by running the Robinson-
left_tableau()	Returns the left standard tableau after performing the RSK algorithm.
right_tableau()	Returns the right standard tableau after performing the RSK algorithm.
increasing_tree()	Returns the increasing tree of self.
increasing_tree_shape()	Returns the shape of the increasing tree of self.
binary_search_tree()	Returns the binary search tree of self.
sylvester_class()	Iterates over the equivalence class of self under sylvester congruence
RS_partition()	Returns the shape of the tableaux obtained by the RSK algorithm.
remove_extra_fixed_points()	Returns the permutation obtained by removing any fixed points at the en
retract_plain()	Returns the plain retract of self to a smaller symmetric group S_m.
retract_direct_product()	Returns the direct-product retract of self to a smaller symmetric group
retract_okounkov_vershik()	Returns the Okounkov-Vershik retract of self to a smaller symmetric g
hyperoctahedral_double_coset_type()	Returns the coset-type of self as a partition.
binary_search_tree_shape()	Returns the shape of the binary search tree of self (a non labelled bina
shifted_concatenation()	Returns the right (or left) shifted concatenation of self with a permutat
shifted_shuffle()	Returns the shifted shuffle of self with a permutation other.

Other classes defined in this file

Permutations	
Permutations_nk	
Permutations_mset	
Permutations_set	
Permutations_msetk	
Permutations_setk	
Arrangements	
Arrangements_msetk	
Arrangements_setk	
StandardPermutations_all	
StandardPermutations_n_abstract	
StandardPermutations_n	
StandardPermutations_descents	
StandardPermutations_recoilsfiner	
StandardPermutations_recoilsfatter	
StandardPermutations_recoils	
StandardPermutations_bruhat_smaller	
StandardPermutations_bruhat_greater	
CyclicPermutations	
CyclicPermutationsOfPartition	
StandardPermutations_avoiding_12	
StandardPermutations_avoiding_21	
StandardPermutations_avoiding_132	
StandardPermutations_avoiding_123	
StandardPermutations_avoiding_321	
StandardPermutations_avoiding_231	
StandardPermutations_avoiding_312	
StandardPermutations_avoiding_213	
StandardPermutations_avoiding_generic	
PatternAvoider	

Functions defined in this file

from_major_code()	Returns the permutation corresponding to major code mc.
from_permutation_group_element()	Returns a Permutation give a PermutationGroupElement pge.
from_rank()	Returns the permutation with the specified lexicographic rank.
from_inversion_vector()	Returns the permutation corresponding to inversion vector iv.
from_cycles()	Returns the permutation with given disjoint-cycle representation cycles.
from_lehmer_code()	Returns the permutation with Lehmer code lehmer.
from_reduced_word()	Returns the permutation corresponding to the reduced word rw.
bistochastic_as_sum_of_permutations()	Returns a given bistochastic matrix as a nonnegative linear combination of permutations.
descents_composition_list()	Returns a list of all the permutations in a given descent class (i. e., having a given descents composition).
descents_composition_first()	Returns the smallest element of a descent class.
descents_composition_last()	Returns the largest element of a descent class.
bruhat_lequal()	Returns True if p1 is less or equal to p2 in the Bruhat order.
permutohedron_lequal()	Returns True if p1 is less or equal to p2 in the permutohedron order.
to_standard()	Returns a standard permutation corresponding to the permutation self.

AUTHORS:

- Mike Hansen

- Dan Drake (2008-04-07): allow Permutation() to take lists of tuples
- Sebastien Labbe (2009-03-17): added robinson_schensted_inverse
- Travis Scrimshaw:
  - (2012-08-16): `to_standard()` no longer modifies input
  - (2013-01-19): Removed RSK implementation and moved to `rsk`.
  - (2013-07-13): Removed `CombinatorialClass` and moved permutations to the category framework.
- Darij Grinberg (2013-09-07): added methods; ameliorated trac ticket #14885 by exposing and documenting methods for global-independent multiplication.
- Travis Scrimshaw (2014-02-05): Made `StandardPermutations_n` a finite Weyl group to make it more uniform with `SymmetricGroup`. Added ability to compute the conjugacy classes.

Classes and methods

class sage.combinat.permutation.**Arrangements**

Bases: sage.combinat.permutation.Permutations

An arrangement of a multiset `mset` is an ordered selection without repetitions. It is represented by a list that contains only elements from `mset`, but maybe in a different order.

`Arrangements` returns the combinatorial class of arrangements of the multiset `mset` that contain `k` elements.

EXAMPLES:

```
sage: mset = [1,1,2,3,4,4,5]
sage: Arrangements(mset,2).list()
[[1, 1],
 [1, 2],
 [1, 3],
 [1, 4],
 [1, 5],
 [2, 1],
 [2, 3],
 [2, 4],
 [2, 5],
 [3, 1],
 [3, 2],
 [3, 4],
 [3, 5],
 [4, 1],
 [4, 2],
 [4, 3],
 [4, 4],
 [4, 5],
 [5, 1],
 [5, 2],
 [5, 3],
 [5, 4]]
 sage: Arrangements(mset,2).cardinality()
 22
 sage: Arrangements( ["c","a","t"], 2 ).list()
 [['c', 'a'], ['c', 't'], ['a', 'c'], ['a', 't'], ['t', 'c'], ['t', 'a']]
 sage: Arrangements( ["c","a","t"], 3 ).list()
 [['c', 'a', 't'],
  ['c', 't', 'a'],
```

```
['a', 'c', 't'],
['a', 't', 'c'],
['t', 'c', 'a'],
['t', 'a', 'c']]
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: A = Arrangements([1,1,2,3,4,4,5], 2)
sage: A.cardinality()
22
```

class sage.combinat.permutation.**Arrangements_msetk**(*mset*, *k*)

Bases: sage.combinat.permutation.Arrangements, sage.combinat.permutation.Permutations_mset

Arrangements of length k of a multiset M.

class sage.combinat.permutation.**Arrangements_setk**(*s*, *k*)

Bases: sage.combinat.permutation.Arrangements, sage.combinat.permutation.Permutations_setk

Arrangements of length k of a set S.

class sage.combinat.permutation.**CyclicPermutations**(*mset*)

Bases: sage.combinat.permutation.Permutations_mset

Return the class of all cyclic permutations of `mset` in cycle notation. These are the same as necklaces.

INPUT:

- `mset` – A multiset

EXAMPLES:

```
sage: CyclicPermutations(range(4)).list()
[[0, 1, 2, 3],
 [0, 1, 3, 2],
 [0, 2, 1, 3],
 [0, 2, 3, 1],
 [0, 3, 1, 2],
 [0, 3, 2, 1]]
sage: CyclicPermutations([1,1,1]).list()
[[1, 1, 1]]
```

iterator(*distinct=False*)

EXAMPLES:

```
sage: CyclicPermutations(range(4)).list() # indirect doctest
[[0, 1, 2, 3],
 [0, 1, 3, 2],
 [0, 2, 1, 3],
 [0, 2, 3, 1],
 [0, 3, 1, 2],
 [0, 3, 2, 1]]
 sage: CyclicPermutations([1,1,1]).list()
 [[1, 1, 1]]
 sage: CyclicPermutations([1,1,1]).list(distinct=True)
 [[1, 1, 1], [1, 1, 1]]
```

list(*distinct=False*)

EXAMPLES:

```
sage: CyclicPermutations(range(4)).list()
[[0, 1, 2, 3],
 [0, 1, 3, 2],
 [0, 2, 1, 3],
 [0, 2, 3, 1],
 [0, 3, 1, 2],
 [0, 3, 2, 1]]
```

class sage.combinat.permutation.**CyclicPermutationsOfPartition**(*partition*)

Bases: sage.combinat.permutation.Permutations

Combinations of cyclic permutations of each cell of a given partition.

This is the same as a Cartesian product of necklaces.

EXAMPLES:

```
sage: CyclicPermutationsOfPartition([[1,2,3,4],[5,6,7]]).list()
[[[1, 2, 3, 4], [5, 6, 7]],
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]

sage: CyclicPermutationsOfPartition([[1,2,3,4],[4,4,4]]).list()
[[[1, 2, 3, 4], [4, 4, 4]],
 [[1, 2, 4, 3], [4, 4, 4]],
 [[1, 3, 2, 4], [4, 4, 4]],
 [[1, 3, 4, 2], [4, 4, 4]],
 [[1, 4, 2, 3], [4, 4, 4]],
 [[1, 4, 3, 2], [4, 4, 4]]]

sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list()
[[[1, 2, 3], [4, 4, 4]], [[1, 3, 2], [4, 4, 4]]]

sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list(distinct=True)
[[[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]],
 [[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]]]
```

class Element

Bases: sage.structure.list_clone.ClonableArray

A cyclic permutation of a partition.

check()

Check that self is a valid element.

EXAMPLES:

```
sage: CP = CyclicPermutationsOfPartition([[1,2,3,4],[5,6,7]])
sage: elt = CP[0]
sage: elt.check()
```

CyclicPermutationsOfPartition.**iterator**(*distinct=False*)

AUTHORS:

- Robert Miller

EXAMPLES:

```
sage: CyclicPermutationsOfPartition([[1,2,3,4],[5,6,7]]).list() # indirect doctest
[[[1, 2, 3, 4], [5, 6, 7]],
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]

sage: CyclicPermutationsOfPartition([[1,2,3,4],[4,4,4]]).list()
[[[1, 2, 3, 4], [4, 4, 4]],
 [[1, 2, 4, 3], [4, 4, 4]],
 [[1, 3, 2, 4], [4, 4, 4]],
 [[1, 3, 4, 2], [4, 4, 4]],
 [[1, 4, 2, 3], [4, 4, 4]],
 [[1, 4, 3, 2], [4, 4, 4]]]

sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list()
[[[1, 2, 3], [4, 4, 4]], [[1, 3, 2], [4, 4, 4]]]

sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list(distinct=True)
[[[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]],
 [[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]]]
```

CyclicPermutationsOfPartition.**list**(*distinct=False*)

EXAMPLES:

```
sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list()
[[[1, 2, 3], [4, 4, 4]], [[1, 3, 2], [4, 4, 4]]]
sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list(distinct=True)
[[[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]],
 [[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]]]
```

class sage.combinat.permutation.**PatternAvoider**(*parent*, *patterns*)

Bases: sage.combinat.backtrack.GenericBacktracker

EXAMPLES:

```
sage: from sage.combinat.permutation import PatternAvoider
sage: P = Permutations(4)
sage: p = PatternAvoider(P, [[1,2,3]])
sage: loads(dumps(p))
<sage.combinat.permutation.PatternAvoider object at 0x...>
```

class sage.combinat.permutation.**Permutation**(*parent*, *l*, *check_input=True*)

Bases: sage.combinat.combinat.CombinatorialElement

A permutation.

Converts `l` to a permutation on $\{1, 2, \ldots, n\}$.

INPUT:

- `l` – Can be any one of the following:
  - an instance of `Permutation`,
  - list of integers, viewed as one-line permutation notation. The construction checks that you give an acceptable entry. To avoid the check, use the `check_input` option.
  - string, expressing the permutation in cycle notation.
  - list of tuples of integers, expressing the permutation in cycle notation.
  - a `PermutationGroupElement`
  - a pair of two standard tableaux of the same shape. This yields the permutation obtained from the pair using the inverse of the Robinson-Schensted algorithm.
- **`check_input` (boolean) – whether to check that input is correct. Slows** the function down, but ensures that nothing bad happens. This is set to `True` by default.

Warning: Since trac ticket #13742 the input is checked for correctness : it is not accepted unless it actually is a permutation on $\{1, \ldots, n\}$. It means that some `Permutation()` objects cannot be created anymore without setting `check_input = False`, as there is no certainty that its functions can handle them, and this should be fixed in a much better way ASAP (the functions should be rewritten to handle those cases, and new tests be added).

Warning: There are two possible conventions for multiplying permutations, and the one currently enabled in Sage by default is the one which has $(pq)(i) = q(p(i))$ for any permutations $p \in S_n$ and $q \in S_n$ and any $1 \leq i \leq n$. (This equation looks less strange when the action of permutations on numbers is written from the right: then it takes the form $i^{pq} = (i^p)^q$, which is an associativity law). There is an alternative convention, which has $(pq)(i) = p(q(i))$ instead. The conventions can be switched at runtime using `sage.combinat.permutation.Permutations.global_options()`. It is best for code not to rely on this setting being set to a particular standard, but rather use the methods `left_action_product()` and `right_action_product()` for multiplying permutations (these methods don't depend on the setting). See trac ticket #14885 for more details.

Note: The `bruhat*` methods refer to the *strong* Bruhat order. To use the *weak* Bruhat order, look under `permutohedron*`.

EXAMPLES:

```
sage: Permutation([2,1])
[2, 1]
sage: Permutation([2, 1, 4, 5, 3])
[2, 1, 4, 5, 3]
sage: Permutation('(1,2)')
[2, 1]
sage: Permutation('(1,2)(3,4,5)')
[2, 1, 4, 5, 3]
sage: Permutation( ((1,2),(3,4,5)) )
[2, 1, 4, 5, 3]
sage: Permutation( [(1,2),(3,4,5)] )
[2, 1, 4, 5, 3]
```

```
sage: Permutation( ((1,2)) )
[2, 1]
sage: Permutation( (1,2) )
[2, 1]
sage: Permutation( ((1,2),) )
[2, 1]
sage: Permutation( ((1,),) )
[1]
sage: Permutation( (1,) )
[1]
sage: Permutation( () )
[]
sage: Permutation( ((),) )
[]
sage: p = Permutation((1, 2, 5)); p
[2, 5, 3, 4, 1]
sage: type(p)
<class 'sage.combinat.permutation.StandardPermutations_n_with_category.element_class'>
```

Construction from a string in cycle notation:

```
sage: p = Permutation( '(4,5)' ); p
[1, 2, 3, 5, 4]
```

The size of the permutation is the maximum integer appearing; add a 1-cycle to increase this:

```
sage: p2 = Permutation( '(4,5)(10)' ); p2
[1, 2, 3, 5, 4, 6, 7, 8, 9, 10]
sage: len(p); len(p2)
5
10
```

We construct a `Permutation` from a `PermutationGroupElement`:

```
sage: g = PermutationGroupElement([2,1,3])
sage: Permutation(g)
[2, 1, 3]
```

From a pair of tableaux of the same shape. This uses the inverse of the Robinson-Schensted algorithm:

```
sage: p = [[1, 4, 7], [2, 5], [3], [6]]
sage: q = [[1, 2, 5], [3, 6], [4], [7]]
sage: P = Tableau(p)
sage: Q = Tableau(q)
sage: Permutation( (p, q) )
[3, 6, 5, 2, 7, 4, 1]
sage: Permutation( [p, q] )
[3, 6, 5, 2, 7, 4, 1]
sage: Permutation( (P, Q) )
[3, 6, 5, 2, 7, 4, 1]
sage: Permutation( [P, Q] )
[3, 6, 5, 2, 7, 4, 1]
```

TESTS:

```
sage: Permutation([()])
[]
sage: Permutation('()')
[]
sage: Permutation(())
```

```
[]
sage: Permutation( [1] )
[1]
```

From a pair of empty tableaux

```
sage: Permutation( ([], []) )
[]
sage: Permutation( [[], []] )
[]
```

_left_to_right_multiply_on_right(*rp*)

Return the permutation obtained by composing `self` with `rp` in such an order that `self` is applied first and `rp` is applied afterwards.

This is usually denoted by either `self * rp` or `rp * self` depending on the conventions used by the author. If the value of a permutation $p \in S_n$ on an integer $i \in \{1, 2, \cdots, n\}$ is denoted by $p(i)$, then this should be denoted by `rp * self` in order to have associativity (i.e., in order to have $(p \cdot q)(i) = p(q(i))$ for all p, q and i). If, on the other hand, the value of a permutation $p \in S_n$ on an integer $i \in \{1, 2, \cdots, n\}$ is denoted by i^p, then this should be denoted by `self * rp` in order to have associativity (i.e., in order to have $i^{p \cdot q} = (i^p)^q$ for all p, q and i).

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: q = Permutation([3,1,2])
sage: p.right_action_product(q)
[1, 3, 2]
sage: q.right_action_product(p)
[3, 2, 1]
```

_left_to_right_multiply_on_left(*lp*)

Return the permutation obtained by composing `self` with `lp` in such an order that `lp` is applied first and `self` is applied afterwards.

This is usually denoted by either `self * lp` or `lp * self` depending on the conventions used by the author. If the value of a permutation $p \in S_n$ on an integer $i \in \{1, 2, \cdots, n\}$ is denoted by $p(i)$, then this should be denoted by `self * lp` in order to have associativity (i.e., in order to have $(p \cdot q)(i) = p(q(i))$ for all p, q and i). If, on the other hand, the value of a permutation $p \in S_n$ on an integer $i \in \{1, 2, \cdots, n\}$ is denoted by i^p, then this should be denoted by `lp * self` in order to have associativity (i.e., in order to have $i^{p \cdot q} = (i^p)^q$ for all p, q and i).

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: q = Permutation([3,1,2])
sage: p.left_action_product(q)
[3, 2, 1]
sage: q.left_action_product(p)
[1, 3, 2]
```

RS_partition()

Return the shape of the tableaux obtained by applying the RSK algorithm to `self`.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).RS_partition()
[2, 1, 1]
```

action(*a*)

Return the action of the permutation `self` on a list `a`.

The action of a permutation $p \in S_n$ on an n-element list $(a_1, a_2, \ldots, a_n)$ is defined to be $(a_{p(1)}, a_{p(2)}, \ldots, a_{p(n)})$.

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: a = range(3)
sage: p.action(a)
[1, 0, 2]
sage: b = [1,2,3,4]
sage: p.action(b)
Traceback (most recent call last):
...
ValueError: len(a) must equal len(self)

sage: q = Permutation([2,3,1])
sage: a = range(3)
sage: q.action(a)
[1, 2, 0]
```

avoids(*patt*)

Test whether the permutation `self` avoids the pattern `patt`.

EXAMPLES:

```
sage: Permutation([6,2,5,4,3,1]).avoids([4,2,3,1])
False
sage: Permutation([6,1,2,5,4,3]).avoids([4,2,3,1])
True
sage: Permutation([6,1,2,5,4,3]).avoids([3,4,1,2])
True
```

binary_search_tree(*left_to_right=True*)

Return the binary search tree associated to `self`.

If w is a word, then the binary search tree associated to w is defined as the result of starting with an empty binary tree, and then inserting the letters of w one by one into this tree. Here, the insertion is being done according to the method `binary_search_insert()`, and the word w is being traversed from left to right.

A permutation is regarded as a word (using one-line notation), and thus a binary search tree associated to a permutation is defined.

If the optional keyword variable `left_to_right` is set to `False`, the word w is being traversed from right to left instead.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).binary_search_tree()
1[., 4[3[2[., .], .], .]]
sage: Permutation([4,1,3,2]).binary_search_tree()
4[1[., 3[2[., .], .]], .]
```

By passing the option `left_to_right=False` one can have the insertion going from right to left:

```
sage: Permutation([1,4,3,2]).binary_search_tree(False)
2[1[., .], 3[., 4[., .]]]
sage: Permutation([4,1,3,2]).binary_search_tree(False)
2[1[., .], 3[., 4[., .]]]
```

TESTS:

```
sage: Permutation([]).binary_search_tree()
.
```

binary_search_tree_shape(*left_to_right=True*)

Return the shape of the binary search tree of the permutation (a non labelled binary tree).

EXAMPLES:

```
sage: Permutation([1,4,3,2]).binary_search_tree_shape()
[., [[[., .], .], .]]
sage: Permutation([4,1,3,2]).binary_search_tree_shape()
[[., [[., .], .]], .]
```

By passing the option `left_to_right=False` one can have the insertion going from right to left:

```
sage: Permutation([1,4,3,2]).binary_search_tree_shape(False)
[[., .], [., [., .]]]
sage: Permutation([4,1,3,2]).binary_search_tree_shape(False)
[[., .], [., [., .]]]
```

bruhat_greater()

Returns the combinatorial class of permutations greater than or equal to `self` in the Bruhat order (on the symmetric group containing `self`).

See `bruhat_lequal()` for the definition of the Bruhat order.

EXAMPLES:

```
sage: Permutation([4,1,2,3]).bruhat_greater().list()
[[4, 1, 2, 3],
 [4, 1, 3, 2],
 [4, 2, 1, 3],
 [4, 2, 3, 1],
 [4, 3, 1, 2],
 [4, 3, 2, 1]]
```

bruhat_inversions()

Return the list of inversions of `self` such that the application of this inversion to `self` decreases its number of inversions by exactly 1.

Equivalently, it returns the list of pairs (i, j) such that $i < j$, such that $p(i) > p(j)$ and such that there exists no k (strictly) between i and j satisfying $p(i) > p(k) > p(j)$.

EXAMPLES:

```
sage: Permutation([5,2,3,4,1]).bruhat_inversions()
[[0, 1], [0, 2], [0, 3], [1, 4], [2, 4], [3, 4]]
sage: Permutation([6,1,4,5,2,3]).bruhat_inversions()
[[0, 1], [0, 2], [0, 3], [2, 4], [2, 5], [3, 4], [3, 5]]
```

bruhat_inversions_iterator()

Return the iterator for the inversions of `self` such that the application of this inversion to `self` decreases its number of inversions by exactly 1.

EXAMPLES:

```
sage: list(Permutation([5,2,3,4,1]).bruhat_inversions_iterator())
[[0, 1], [0, 2], [0, 3], [1, 4], [2, 4], [3, 4]]
sage: list(Permutation([6,1,4,5,2,3]).bruhat_inversions_iterator())
[[0, 1], [0, 2], [0, 3], [2, 4], [2, 5], [3, 4], [3, 5]]
```

bruhat_lequal(*p2*)

Return `True` if `self` is less or equal to `p2` in the Bruhat order.

The Bruhat order (also called strong Bruhat order or Chevalley order) on the symmetric group S_n is the partial order on S_n determined by the following condition: If p is a permutation, and i and j are two indices satisfying $p(i) > p(j)$ and $i < j$ (that is, (i, j) is an inversion of p with $i < j$), then $p \circ (i, j)$ (the permutation obtained by first switching i with j and then applying p) is smaller than p in the Bruhat order.

One can show that a permutation $p \in S_n$ is less or equal to a permutation $q \in S_n$ in the Bruhat order if and only if for every $i \in \{0, 1, \cdots, n\}$ and $j \in \{1, 2, \cdots, n\}$, the number of the elements among $p(1), p(2), \cdots, p(j)$ that are greater than i is $\leq$ to the number of the elements among $q(1), q(2), \cdots, q(j)$ that are greater than i.

This method assumes that `self` and `p2` are permutations of the same integer n.

EXAMPLES:

```
sage: Permutation([2,4,3,1]).bruhat_lequal(Permutation([3,4,2,1]))
True

sage: Permutation([2,1,3]).bruhat_lequal(Permutation([2,3,1]))
True
sage: Permutation([2,1,3]).bruhat_lequal(Permutation([3,1,2]))
True
sage: Permutation([2,1,3]).bruhat_lequal(Permutation([1,2,3]))
False
sage: Permutation([1,3,2]).bruhat_lequal(Permutation([2,1,3]))
False
sage: Permutation([1,3,2]).bruhat_lequal(Permutation([2,3,1]))
True
sage: Permutation([2,3,1]).bruhat_lequal(Permutation([1,3,2]))
False
sage: sorted( [len([b for b in Permutations(3) if a.bruhat_lequal(b)])
....:          for a in Permutations(3)] )
[1, 2, 2, 4, 4, 6]

sage: Permutation([]).bruhat_lequal(Permutation([]))
True
```

bruhat_pred()

Return a list of the permutations strictly smaller than `self` in the Bruhat order (on the symmetric group containing `self`) such that there is no permutation between one of those and `self`.

See `bruhat_lequal()` for the definition of the Bruhat order.

EXAMPLES:

```
sage: Permutation([6,1,4,5,2,3]).bruhat_pred()
[[1, 6, 4, 5, 2, 3],
 [4, 1, 6, 5, 2, 3],
 [5, 1, 4, 6, 2, 3],
 [6, 1, 2, 5, 4, 3],
 [6, 1, 3, 5, 2, 4],
 [6, 1, 4, 2, 5, 3],
 [6, 1, 4, 3, 2, 5]]
```

bruhat_pred_iterator()

An iterator for the permutations strictly smaller than `self` in the Bruhat order (on the symmetric group containing `self`) such that there is no permutation between one of those and `self`.

See `bruhat_lequal()` for the definition of the Bruhat order.

EXAMPLES:

```
sage: [x for x in Permutation([6,1,4,5,2,3]).bruhat_pred_iterator()]
[[1, 6, 4, 5, 2, 3],
 [4, 1, 6, 5, 2, 3],
 [5, 1, 4, 6, 2, 3],
 [6, 1, 2, 5, 4, 3],
 [6, 1, 3, 5, 2, 4],
 [6, 1, 4, 2, 5, 3],
 [6, 1, 4, 3, 2, 5]]
```

bruhat_smaller()

Return the combinatorial class of permutations smaller than or equal to `self` in the Bruhat order (on the symmetric group containing `self`).

See `bruhat_lequal()` for the definition of the Bruhat order.

EXAMPLES:

```
sage: Permutation([4,1,2,3]).bruhat_smaller().list()
[[1, 2, 3, 4],
 [1, 2, 4, 3],
 [1, 3, 2, 4],
 [1, 4, 2, 3],
 [2, 1, 3, 4],
 [2, 1, 4, 3],
 [3, 1, 2, 4],
 [4, 1, 2, 3]]
```

bruhat_succ()

Return a list of the permutations strictly greater than `self` in the Bruhat order (on the symmetric group containing `self`) such that there is no permutation between one of those and `self`.

See `bruhat_lequal()` for the definition of the Bruhat order.

EXAMPLES:

```
sage: Permutation([6,1,4,5,2,3]).bruhat_succ()
[[6, 4, 1, 5, 2, 3],
 [6, 2, 4, 5, 1, 3],
 [6, 1, 5, 4, 2, 3],
 [6, 1, 4, 5, 3, 2]]
```

bruhat_succ_iterator()

An iterator for the permutations that are strictly greater than `self` in the Bruhat order (on the symmetric group containing `self`) such that there is no permutation between one of those and `self`.

See `bruhat_lequal()` for the definition of the Bruhat order.

EXAMPLES:

```
sage: [x for x in Permutation([6,1,4,5,2,3]).bruhat_succ_iterator()]
[[6, 4, 1, 5, 2, 3],
 [6, 2, 4, 5, 1, 3],
 [6, 1, 5, 4, 2, 3],
 [6, 1, 4, 5, 3, 2]]
```

complement()

Return the complement of the permutation `self`.

The complement of a permutation $w \in S_n$ is defined as the permutation in S_n sending each i to $n+1-w(i)$.

EXAMPLES:

```
sage: Permutation([1,2,3]).complement()
[3, 2, 1]
sage: Permutation([1, 3, 2]).complement()
[3, 1, 2]
```

cycle_string (*singletons=False*)

Returns a string of the permutation in cycle notation.

If `singletons=True`, it includes 1-cycles in the string.

EXAMPLES:

```
sage: Permutation([1,2,3]).cycle_string()
'()'
sage: Permutation([2,1,3]).cycle_string()
'(1,2)'
sage: Permutation([2,3,1]).cycle_string()
'(1,2,3)'
sage: Permutation([2,1,3]).cycle_string(singletons=True)
'(1,2)(3)'
```

cycle_tuples (*singletons=True*)

Return the permutation `self` as a list of disjoint cycles.

The cycles are returned in the order of increasing smallest elements, and each cycle is returned as a tuple which starts with its smallest element.

If `singletons=False` is given, the list does not contain the singleton cycles.

EXAMPLES:

```
sage: Permutation([2,1,3,4]).to_cycles()
[(1, 2), (3,), (4,)]
sage: Permutation([2,1,3,4]).to_cycles(singletons=False)
[(1, 2)]

sage: Permutation([4,1,5,2,6,3]).to_cycles()
[(1, 4, 2), (3, 5, 6)]
```

The algorithm is of complexity $O(n)$ where n is the size of the given permutation.

TESTS:

```
sage: from sage.combinat.permutation import from_cycles
sage: for n in range(1,6):
....:     for p in Permutations(n):
....:         if from_cycles(n, p.to_cycles()) != p:
....:             print "There is a problem with ",p
....:             break
sage: size = 10000
sage: sample = (Permutations(size).random_element() for i in range(5))
sage: all(from_cycles(size, p.to_cycles()) == p for p in sample)
True
```

Note: there is an alternative implementation called `_to_cycle_set` which could be slightly (10%) faster for some input (typically for permutations of size in the range [100, 10000]). You can run the following benchmarks. For small permutations:

```
sage: for size in range(9): # not tested
....:  print size
....:  lp = Permutations(size).list()
```

```
....:  timeit('[p.to_cycles(False) for p in lp]')
....:  timeit('[p._to_cycles_set(False) for p in lp]')
....:  timeit('[p._to_cycles_list(False) for p in lp]')
....:  timeit('[p._to_cycles_orig(False) for p in lp]')
```

and larger ones:

```
sage: for size in [10, 20, 50, 75, 100, 200, 500, 1000, # not tested
....:       2000, 5000, 10000, 15000, 20000, 30000,
....:       50000, 80000, 100000]:
....:    print(size)
....:    lp = [Permutations(size).random_element() for i in range(20)]
....:    timeit("[p.to_cycles() for p in lp]")
....:    timeit("[p._to_cycles_set() for p in lp]")
....:    timeit("[p._to_cycles_list() for p in lp]")
```

cycle_type()

Return a partition of `len(self)` corresponding to the cycle type of `self`. This is a non-increasing sequence of the cycle lengths of `self`.

EXAMPLES:

```
sage: Permutation([3,1,2,4]).cycle_type()
[3, 1]
```

decreasing_runs(*as_tuple=False*)

Decreasing runs of the permutation.

INPUT:

- `as_tuple` – boolean (default: `False`) choice of output format

OUTPUT:

a list of lists or a tuple of tuples

See also:

`runs()`

EXAMPLES:

```
sage: s = Permutation([2,8,3,9,6,4,5,1,7])
sage: s.decreasing_runs()
[[2], [8, 3], [9, 6, 4], [5, 1], [7]]
sage: s.decreasing_runs(as_tuple=True)
((2,), (8, 3), (9, 6, 4), (5, 1), (7,))
```

descent_polynomial()

Return the descent polynomial of the permutation `self`.

The descent polynomial of a permutation p is the product of all the `z[p[i]]` where `i` ranges over the descents of `p`.

A descent of a permutation `p` is an integer `i` such that `p[i] > p[i+1]`. Here, Python's indexing convention is used, so `p[i]` means $p(i+1)$.

REFERENCES:

EXAMPLES:

```
sage: Permutation([2,1,3]).descent_polynomial()
z1
```

```
sage: Permutation([4,3,2,1]).descent_polynomial()
z1*z2^2*z3^3
```

Todo

This docstring needs to be fixed. First, the definition does not match the implementation (or the examples). Second, this doesn't seem to be defined in [GarStan1984] (the descent monomial in their (7.23) is different).

---

descents (*final_descent=False*)

Return the list of the descents of `self`.

A descent of a permutation `p` is an integer `i` such that `p[i] > p[i+1]`. Here, Python's indexing convention is used, so `p[i]` means $p(i+1)$.

With the `final_descent` option, the last position of a non-empty permutation is also considered as a descent.

EXAMPLES:

```
sage: Permutation([3,1,2]).descents()
[0]
sage: Permutation([1,4,3,2]).descents()
[1, 2]
sage: Permutation([1,4,3,2]).descents(final_descent=True)
[1, 2, 3]
```

descents_composition()

Return the descent composition of `self`.

The descent composition of a permutation $p \in S_n$ is defined as the composition of n whose descent set equals the descent set of p. Here, the descent set of p is defined as the set of all $i \in \{1, 2, \ldots, n-1\}$ satisfying $p(i) > p(i+1)$ (note that this differs from the output of the `descents()` method, since the latter uses Python's indexing which starts at 0 instead of 1). The descent set of a composition $c = (i_1, i_2, \ldots, i_k)$ is defined as the set $\{i_1, i_1+i_2, i_1+i_2+i_3, \ldots, i_1+i_2+\cdots+i_{k-1}\}$.

EXAMPLES:

```
sage: Permutation([1,3,2,4]).descents_composition()
[2, 2]
sage: Permutation([4,1,6,7,2,3,8,5]).descents_composition()
[1, 3, 3, 1]
sage: Permutation([]).descents_composition()
[]
```

destandardize (*weight*, *ordered_alphabet=None*)

Return destandardization of `self` with respect to `weight` and `ordered_alphabet`.

INPUT:

- `weight` – list or tuple of nonnegative integers that sum to n if `self` is a permutation in S_n.
- `ordered_alphabet` – (default: None) a list or tuple specifying the ordered alphabet the destandardized word is over

OUTPUT: word over the `ordered_alphabet` which standardizes to `self`

Let $weight = (w_1, w_2, \ldots, w_\ell)$. Then this methods looks for an increasing sequence of $1, 2, \ldots, w_1$ and labels all letters in it by 1, then an increasing sequence of $w_1+1, w_1+2, \ldots, w_1+w_2$ and labels all these letters by 2, etc.. If an increasing sequence for the specified `weight` does not exist, an error is

returned. The output is a word `w` over the specified ordered alphabet with evaluation `weight` such that `w.standard_permutation()` is `self`.

EXAMPLES:

```
sage: p = Permutation([1,2,5,3,6,4])
sage: p.destandardize([3,1,2])
word: 113132
sage: p = Permutation([2,1,3])
sage: p.destandardize([2,1])
Traceback (most recent call last):
...
ValueError: Standardization with weight [2, 1] is not possible!
```

TESTS:

```
sage: p = Permutation([4,1,2,3,5,6])
sage: p.destandardize([2,1,3], ordered_alphabet = [1,'a',3])
word: 311a33
sage: p.destandardize([2,1,3], ordered_alphabet = [1,'a'])
Traceback (most recent call last):
...
ValueError: Not enough letters in the alphabet are specified compared to the weight
```

dict()

Returns a dictionary corresponding to the permutation.

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: d = p.dict()
sage: d[1]
2
sage: d[2]
1
sage: d[3]
3
```

fixed_points()

Return a list of the fixed points of `self`.

EXAMPLES:

```
sage: Permutation([1,3,2,4]).fixed_points()
[1, 4]
sage: Permutation([1,2,3,4]).fixed_points()
[1, 2, 3, 4]
```

foata_bijection()

Return the image of the permutation `self` under the Foata bijection ϕ.

The bijection shows that maj and inv are equidistributed: if $\phi(P) = Q$, then $\mathrm{maj}(P) = \mathrm{inv}(Q)$.

The Foata bijection ϕ is a bijection on the set of words with no two equal letters. It can be defined by induction on the size of the word: Given a word $w_1 w_2 \cdots w_n$, start with $\phi(w_1) = w_1$. At the i-th step, if $\phi(w_1 w_2 \cdots w_i) = v_1 v_2 \cdots v_i$, we define $\phi(w_1 w_2 \cdots w_i w_{i+1})$ by placing w_{i+1} on the end of the word $v_1 v_2 \cdots v_i$ and breaking the word up into blocks as follows. If $w_{i+1} > v_i$, place a vertical line to the right of each v_k for which $w_{i+1} > v_k$. Otherwise, if $w_{i+1} < v_i$, place a vertical line to the right of each v_k for which $w_{i+1} < v_k$. In either case, place a vertical line at the start of the word as well. Now, within each block between vertical lines, cyclically shift the entries one place to the right.

For instance, to compute $\phi([1, 4, 2, 5, 3])$, the sequence of words is

- 1,
- $|1|4 \to 14$,
- $|14|2 \to 412$,
- $|4|1|2|5 \to 4125$,
- $|4|125|3 \to 45123$.

So $\phi([1,4,2,5,3]) = [4,5,1,2,3]$.

See section 2 of [FoSc78].

REFERENCES:

EXAMPLES:

```
sage: Permutation([1,2,4,3]).foata_bijection()
[4, 1, 2, 3]
sage: Permutation([2,5,1,3,4]).foata_bijection()
[2, 1, 3, 5, 4]

sage: P = Permutation([2,5,1,3,4])
sage: P.major_index() == P.foata_bijection().number_of_inversions()
True

sage: all( P.major_index() == P.foata_bijection().number_of_inversions()
....:      for P in Permutations(4) )
True
```

The example from [FoSc78]:

```
sage: Permutation([7,4,9,2,6,1,5,8,3]).foata_bijection()
[4, 7, 2, 6, 1, 9, 5, 8, 3]
```

Border cases:

```
sage: Permutation([]).foata_bijection()
[]
sage: Permutation([1]).foata_bijection()
[1]
```

forget_cycles()

Return the image of `self` under the map which forgets cycles.

Consider a permutation σ written in standard cyclic form:

$$\sigma = (a_{1,1}, \ldots, a_{1,k_1})(a_{2,1}, \ldots, a_{2,k_2}) \cdots (a_{m,1}, \ldots, a_{m,k_m}),$$

where $a_{1,1} < a_{2,1} < \cdots < a_{m,1}$ and $a_{j,1} < a_{j,i}$ for all $1 \leq j \leq m$ and $2 \leq i \leq k_j$ where we include cycles of length 1 as well. The image of the forget cycle map ϕ is given by

$$\phi(\sigma) = [a_{1,1}, \ldots, a_{1,k_1}, a_{2,1} \ldots, a_{2,k_2}, \ldots, a_{m,1}, \ldots, a_{m,k_m}],$$

considered as a permutation in 1-line notation.

EXAMPLES:

```
sage: P = Permutations(5)
sage: x = P([1, 5, 3, 4, 2])
sage: x.forget_cycles()
[1, 2, 5, 3, 4]
```

We select all permutations with a cycle composition of $[2,3,1]$ in S_6:

```
sage: P = Permutations(6)
sage: l = [p for p in P if [len(t) for t in p.to_cycles()] == [1,3,2]]
```

Next we apply ϕ and then take the inverse, and then view the results as a poset under the Bruhat order:

```
sage: l = [p.forget_cycles().inverse() for p in l]
sage: B = Poset([l, lambda x,y: x.bruhat_lequal(y)])
sage: R.<q> = QQ[]
sage: sum(q^B.rank_function()(x) for x in B)
q^5 + 2*q^4 + 3*q^3 + 3*q^2 + 2*q + 1
```

We check the statement in [CC13] that the posets $C_{[1,3,1,1]}$ and $C_{[1,3,2]}$ are isomorphic:

```
sage: l2 = [p for p in P if [len(t) for t in p.to_cycles()] == [1,3,1,1]]
sage: l2 = [p.forget_cycles().inverse() for p in l2]
sage: B2 = Poset([l2, lambda x,y: x.bruhat_lequal(y)])
sage: B.is_isomorphic(B2)
True
```

REFERENCES:

has_pattern(*patt*)

Test whether the permutation `self` contains the pattern `patt`.

EXAMPLES:

```
sage: Permutation([3,5,1,4,6,2]).has_pattern([1,3,2])
True
```

hyperoctahedral_double_coset_type()

Return the coset-type of `self` as a partition.

`self` must be a permutation of even size $2n$. The coset-type determines the double class of the permutation, that is its image in $H_n \backslash S_{2n} / H_n$, where H_n is the n-th hyperoctahedral group.

The coset-type is determined as follows. Consider the perfect matching $\{\{1,2\},\{3,4\},\ldots,\{2n-1,2n\}\}$ and its image by `self`, and draw them simultaneously as edges of a graph whose vertices are labeled by $1,2,\ldots,2n$. The coset-type is the ordered sequence of the semi-lengths of the cycles of this graph (see Chapter VII of [Mcd] for more details, particularly Section VII.2).

EXAMPLE:

```
sage: Permutation([3, 4, 6, 1, 5, 7, 2, 8]).hyperoctahedral_double_coset_type()
[3, 1]
sage: all([p.hyperoctahedral_double_coset_type() ==
....:      p.inverse().hyperoctahedral_double_coset_type()
....:       for p in Permutations(4)])
True
sage: Permutation([]).hyperoctahedral_double_coset_type()
[]
sage: Permutation([3,1,2]).hyperoctahedral_double_coset_type()
Traceback (most recent call last):
...
ValueError: [3, 1, 2] is a permutation of odd size and has no coset-type
```

REFERENCES:

idescents(*final_descent=False*)

Return a list of the idescents of `self`, that is the list of the descents of `self`'s inverse.

A descent of a permutation `p` is an integer `i` such that `p[i] > p[i+1]`. Here, Python's indexing convention is used, so `p[i]` means $p(i+1)$.

With the `final_descent` option, the last position of a non-empty permutation is also considered as a descent.

EXAMPLES:

```
sage: Permutation([2,3,1]).idescents()
[0]
sage: Permutation([1,4,3,2]).idescents()
[1, 2]
sage: Permutation([1,4,3,2]).idescents(final_descent=True)
[1, 2, 3]
```

idescents_signature(*final_descent=False*)

Return the list obtained as follows: Each position in `self` is mapped to -1 if it is an idescent and 1 if it is not an idescent.

See `idescents()` for a definition of idescents.

With the `final_descent` option, the last position of a non-empty permutation is also considered as a descent.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).idescents()
[1, 2]
sage: Permutation([1,4,3,2]).idescents_signature()
[1, -1, -1, 1]
```

imajor_index(*final_descent=False*)

Return the inverse major index of the permutation `self`, which is the major index of the inverse of `self`.

The major index of a permutation p is the sum of the descents of p. Since our permutation indices are 0-based, we need to add the number of descents.

With the `final_descent` option, the last position of a non-empty permutation is also considered as a descent.

EXAMPLES:

```
sage: Permutation([2,1,3]).imajor_index()
1
sage: Permutation([3,4,1,2]).imajor_index()
2
sage: Permutation([4,3,2,1]).imajor_index()
6
```

increasing_tree(*compare=<built-in function min>*)

Return the increasing tree associated to `self`.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).increasing_tree()
1[., 2[3[4[., .], .], .]]
sage: Permutation([4,1,3,2]).increasing_tree()
1[4[., .], 2[3[., .], .]]
```

By passing the option `compare=max` one can have the decreasing tree instead:

```
sage: Permutation([2,3,4,1]).increasing_tree(max)
4[3[2[., .], .], 1[., .]]
```

```
sage: Permutation([2,3,1,4]).increasing_tree(max)
4[3[2[., .], 1[., .]], .]
```

increasing_tree_shape(*compare=<built-in function min>*)

Return the shape of the increasing tree associated with the permutation.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).increasing_tree_shape()
[., [[[., .], .], .]]
sage: Permutation([4,1,3,2]).increasing_tree_shape()
[[., .], [[., .], .]]
```

By passing the option `compare=max` one can have the decreasing tree instead:

```
sage: Permutation([2,3,4,1]).increasing_tree_shape(max)
[[[., .], .], [., .]]
sage: Permutation([2,3,1,4]).increasing_tree_shape(max)
[[[., .], [., .]], .]
```

inverse()

Return the inverse of `self`.

EXAMPLES:

```
sage: Permutation([3,8,5,10,9,4,6,1,7,2]).inverse()
[8, 10, 1, 6, 3, 7, 9, 2, 5, 4]
sage: Permutation([2, 4, 1, 5, 3]).inverse()
[3, 1, 5, 2, 4]
sage: ~Permutation([2, 4, 1, 5, 3])
[3, 1, 5, 2, 4]
```

inversions()

Return a list of the inversions of `self`.

An inversion of a permutation p is a pair (i, j) such that $i < j$ and $p(i) > p(j)$.

EXAMPLES:

```
sage: Permutation([3,2,4,1,5]).inversions()
[(1, 2), (1, 4), (2, 4), (3, 4)]
```

is_even()

Return `True` if the permutation `self` is even and `False` otherwise.

EXAMPLES:

```
sage: Permutation([1,2,3]).is_even()
True
sage: Permutation([2,1,3]).is_even()
False
```

ishift(*i*)

Return the `i`-shift of `self`. If an `i`-shift of `self` can't be performed, then `self` is returned.

An i-shift can be applied when i is not inbetween $i-1$ and $i+1$. The i-shift moves i to the other side, and leaves the relative positions of $i-1$ and $i+1$ in place. All other entries of the permutations are also left in place.

EXAMPLES:

Here, 2 is to the left of both 1 and 3. A 2-shift can be applied which moves the 2 to the right and leaves 1 and 3 in their same relative order:

```
sage: Permutation([2,1,3]).ishift(2)
[1, 3, 2]
```

All entries other than i, $i-1$ and $i+1$ are unchanged:

```
sage: Permutation([2,4,1,3]).ishift(2)
[1, 4, 3, 2]
```

Since 2 is between 1 and 3 in `[1,2,3]`, a 2-shift cannot be applied to `[1,2,3]`

```
sage: Permutation([1,2,3]).ishift(2)
[1, 2, 3]
```

iswitch(*i*)

Return the `i`-switch of `self`. If an `i`-switch of `self` can't be performed, then `self` is returned.

An i-switch can be applied when the subsequence of `self` formed by the entries $i-1$, i and $i+1$ is neither increasing nor decreasing. In this case, this subsequence is reversed (i. e., its leftmost element and its rightmost element switch places), while all other letters of `self` are kept in place.

EXAMPLES:

Here, 2 is to the left of both 1 and 3. A 2-switch can be applied which moves the 2 to the right and switches the relative order between 1 and 3:

```
sage: Permutation([2,1,3]).iswitch(2)
[3, 1, 2]
```

All entries other than $i-1$, i and $i+1$ are unchanged:

```
sage: Permutation([2,4,1,3]).iswitch(2)
[3, 4, 1, 2]
```

Since 2 is between 1 and 3 in `[1,2,3]`, a 2-switch cannot be applied to `[1,2,3]`

```
sage: Permutation([1,2,3]).iswitch(2)
[1, 2, 3]
```

left_action_product(*lp*)

Return the permutation obtained by composing `self` with `lp` in such an order that `lp` is applied first and `self` is applied afterwards.

This is usually denoted by either `self * lp` or `lp * self` depending on the conventions used by the author. If the value of a permutation $p \in S_n$ on an integer $i \in \{1, 2, \cdots, n\}$ is denoted by $p(i)$, then this should be denoted by `self * lp` in order to have associativity (i.e., in order to have $(p \cdot q)(i) = p(q(i))$ for all p, q and i). If, on the other hand, the value of a permutation $p \in S_n$ on an integer $i \in \{1, 2, \cdots, n\}$ is denoted by i^p, then this should be denoted by `lp * self` in order to have associativity (i.e., in order to have $i^{p \cdot q} = (i^p)^q$ for all p, q and i).

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: q = Permutation([3,1,2])
sage: p.left_action_product(q)
[3, 2, 1]
sage: q.left_action_product(p)
[1, 3, 2]
```

left_tableau()
Return the left standard tableau after performing the RSK algorithm on `self`.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).left_tableau()
[[1, 2], [3], [4]]
```

length()
Return the Coxeter length of `self`.

The length of a permutation p is given by the number of inversions of p.

EXAMPLES:

```
sage: Permutation([5, 1, 3, 4, 2]).length()
6
```

longest_increasing_subsequence_length()
Return the length of the longest increasing subsequences of `self`.

EXAMPLES:

```
sage: Permutation([2,3,1,4]).longest_increasing_subsequence_length()
3
sage: all([i.longest_increasing_subsequence_length() == len(RSK(i)[0][0]) for i in Permutati
True
sage: Permutation([]).longest_increasing_subsequence_length()
0
```

longest_increasing_subsequences()
Return the list of the longest increasing subsequences of `self`.

Note: The algorithm is not optimal.

EXAMPLES:

```
sage: Permutation([2,3,4,1]).longest_increasing_subsequences()
[[2, 3, 4]]
sage: Permutation([5, 7, 1, 2, 6, 4, 3]).longest_increasing_subsequences()
[[1, 2, 6], [1, 2, 4], [1, 2, 3]]
```

major_index(*final_descent=False*)
Return the major index of `self`.

The major index of a permutation p is the sum of the descents of p. Since our permutation indices are 0-based, we need to add the number of descents.

With the `final_descent` option, the last position of a non-empty permutation is also considered as a descent.

EXAMPLES:

```
sage: Permutation([2,1,3]).major_index()
1
sage: Permutation([3,4,1,2]).major_index()
2
sage: Permutation([4,3,2,1]).major_index()
6
```

next()
Return the permutation that follows `self` in lexicographic order on the symmetric group containing

`self`. If `self` is the last permutation, then `next` returns `False`.

EXAMPLES:

```
sage: p = Permutation([1, 3, 2])
sage: next(p)
[2, 1, 3]
sage: p = Permutation([4,3,2,1])
sage: next(p)
False
```

TESTS:

```
sage: p = Permutation([])
sage: next(p)
False
```

noninversions(*k*)

Return the list of all `k`-noninversions in `self`.

If k is an integer and $p \in S_n$ is a permutation, then a k-noninversion in p is defined as a strictly increasing sequence $(i_1, i_2, \ldots, i_k)$ of elements of $\{1, 2, \ldots, n\}$ satisfying $p(i_1) < p(i_2) < \cdots < p(i_k)$. (In other words, a k-noninversion in p can be regarded as a k-element subset of $\{1, 2, \ldots, n\}$ on which p restricts to an increasing map.)

EXAMPLES:

```
sage: p = Permutation([3, 2, 4, 1, 5])
sage: p.noninversions(1)
[[3], [2], [4], [1], [5]]
sage: p.noninversions(2)
[[3, 4], [3, 5], [2, 4], [2, 5], [4, 5], [1, 5]]
sage: p.noninversions(3)
[[3, 4, 5], [2, 4, 5]]
sage: p.noninversions(4)
[]
sage: p.noninversions(5)
[]
```

TESTS:

```
sage: q = Permutation([])
sage: q.noninversions(1)
[]
```

number_of_descents(*final_descent=False*)

Return the number of descents of `self`.

With the `final_descent` option, the last position of a non-empty permutation is also considered as a descent.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).number_of_descents()
2
sage: Permutation([1,4,3,2]).number_of_descents(final_descent=True)
3
```

number_of_fixed_points()

Return the number of fixed points of `self`.

EXAMPLES:

```
sage: Permutation([1,3,2,4]).number_of_fixed_points()
2
sage: Permutation([1,2,3,4]).number_of_fixed_points()
4
```

number_of_idescents(*final_descent=False*)

Return the number of idescents of `self`.

See `idescents()` for a definition of idescents.

With the `final_descent` option, the last position of a non-empty permutation is also considered as a descent.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).number_of_idescents()
2
sage: Permutation([1,4,3,2]).number_of_idescents(final_descent=True)
3
```

number_of_inversions()

Return the number of inversions in `self`.

An inversion of a permutation is a pair of elements (i, j) with $i < j$ and $p(i) > p(j)$.

REFERENCES:

- http://mathworld.wolfram.com/PermutationInversion.html

EXAMPLES:

```
sage: Permutation([3, 2, 4, 1, 5]).number_of_inversions()
4
sage: Permutation([1, 2, 6, 4, 7, 3, 5]).number_of_inversions()
6
```

number_of_noninversions(k)

Return the number of `k`-noninversions in `self`.

If k is an integer and $p \in S_n$ is a permutation, then a k-noninversion in p is defined as a strictly increasing sequence $(i_1, i_2, \ldots, i_k)$ of elements of $\{1, 2, \ldots, n\}$ satisfying $p(i_1) < p(i_2) < \cdots < p(i_k)$. (In other words, a k-noninversion in p can be regarded as a k-element subset of $\{1, 2, \ldots, n\}$ on which p restricts to an increasing map.)

The number of k-noninversions in p has been denoted by $\mathrm{noninv}_k(p)$ in [RSW2011], where conjectures and results regarding this number have been stated.

REFERENCES:

EXAMPLES:

```
sage: p = Permutation([3, 2, 4, 1, 5])
sage: p.number_of_noninversions(1)
5
sage: p.number_of_noninversions(2)
6
sage: p.number_of_noninversions(3)
2
sage: p.number_of_noninversions(4)
0
sage: p.number_of_noninversions(5)
0
```

The number of 2-noninversions of a permutation $p \in S_n$ is $\binom{n}{2}$ minus its number of inversions:

```
sage: b = binomial(5, 2)
sage: all( x.number_of_noninversions(2) == b - x.number_of_inversions()
....:      for x in Permutations(5) )
True
```

We also check some corner cases:

```
sage: all( x.number_of_noninversions(1) == 5 for x in Permutations(5) )
True
sage: all( x.number_of_noninversions(0) == 1 for x in Permutations(5) )
True
sage: Permutation([]).number_of_noninversions(1)
0
sage: Permutation([]).number_of_noninversions(0)
1
sage: Permutation([2, 1]).number_of_noninversions(3)
0
```

number_of_peaks()

Return the number of peaks of the permutation `self`.

A peak of a permutation p is an integer i such that $p(i-1) < p(i)$ and $p(i) > p(i+1)$.

EXAMPLES:

```
sage: Permutation([1,3,2,4,5]).number_of_peaks()
1
sage: Permutation([4,1,3,2,6,5]).number_of_peaks()
2
```

number_of_recoils()

Return the number of recoils of the permutation `self`.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).number_of_recoils()
2
```

number_of_saliances()

Return the number of saliances of `self`.

A saliance of a permutation p is an integer i such that $p(i) > p(j)$ for all $j > i$.

EXAMPLES:

```
sage: Permutation([2,3,1,5,4]).number_of_saliances()
2
sage: Permutation([5,4,3,2,1]).number_of_saliances()
5
```

pattern_positions(*patt*)

Return the list of positions where the pattern `patt` appears in the permutation `self`.

EXAMPLES:

```
sage: Permutation([3,5,1,4,6,2]).pattern_positions([1,3,2])
[[0, 1, 3], [2, 3, 5], [2, 4, 5]]
```

peaks()

Return a list of the peaks of the permutation `self`.

A peak of a permutation p is an integer i such that $p(i-1) < p(i)$ and $p(i) > p(i+1)$.

EXAMPLES:

```
sage: Permutation([1,3,2,4,5]).peaks()
[1]
sage: Permutation([4,1,3,2,6,5]).peaks()
[2, 4]
sage: Permutation([]).peaks()
[]
```

permutation_poset()

Return the permutation poset of `self`.

The permutation poset of a permutation p is the poset with vertices $(i, p(i))$ for $i = 1, 2, \ldots, n$ (where n is the size of p) and order inherited from $\mathbf{Z} \times \mathbf{Z}$.

EXAMPLES:

```
sage: Permutation([3,1,5,4,2]).permutation_poset().cover_relations()
[[(2, 1), (5, 2)],
 [(2, 1), (3, 5)],
 [(2, 1), (4, 4)],
 [(1, 3), (3, 5)],
 [(1, 3), (4, 4)]]
sage: Permutation([]).permutation_poset().cover_relations()
[]
sage: Permutation([1,3,2]).permutation_poset().cover_relations()
[[(1, 1), (2, 3)], [(1, 1), (3, 2)]]
sage: Permutation([1,2]).permutation_poset().cover_relations()
[[(1, 1), (2, 2)]]
sage: P = Permutation([1,5,2,4,3])
sage: P.permutation_poset().greene_shape() == P.RS_partition()    # This should hold for any
True
```

permutohedron_greater(*side='right'*)

Return a list of permutations greater than or equal to `self` in the permutohedron order.

By default, the computations are done in the right permutohedron. If you pass the option `side='left'`, then they will be done in the left permutohedron.

See `permutohedron_lequal()` for the definition of the permutohedron orders.

EXAMPLES:

```
sage: Permutation([4,2,1,3]).permutohedron_greater()
[[4, 2, 1, 3], [4, 2, 3, 1], [4, 3, 2, 1]]
sage: Permutation([4,2,1,3]).permutohedron_greater(side='left')
[[4, 2, 1, 3], [4, 3, 1, 2], [4, 3, 2, 1]]
```

permutohedron_join(*other*, *side='right'*)

Return the join of the permutations `self` and `other` in the right permutohedron order (or, if `side` is set to `'left'`, in the left permutohedron order).

The permutohedron orders (see `permutohedron_lequal()`) are lattices; the join operation refers to this lattice structure. In more elementary terms, the join of two permutations π and ψ in the symmetric group S_n is the permutation in S_n whose set of inversion is the transitive closure of the union of the set of inversions of π with the set of inversions of ψ.

See also:

`permutohedron_lequal()`, `permutohedron_meet()`.

ALGORITHM:

It is enough to construct the join of any two permutations π and ψ in S_n with respect to the right weak order. (The join of π and ψ with respect to the left weak order is the inverse of the join of π^{-1} and ψ^{-1} with respect to the right weak order.) Start with an empty list l (denoted `xs` in the actual code). For $i = 1, 2, \ldots, n$ (in this order), we insert i into this list in the rightmost possible position such that any letter in $\{1, 2, ..., i-1\}$ which appears further right than i in either π or ψ (or both) must appear further right than i in the resulting list. After all numbers are inserted, we are left with a list which is precisely the join of π and ψ (in one-line notation). This algorithm is due to Markowsky, [Mark94] (Theorem 1 (a)).

REFERENCES:

AUTHORS:

Viviane Pons and Darij Grinberg, 18 June 2014.

EXAMPLES:

```
sage: p = Permutation([3,1,2])
sage: q = Permutation([1,3,2])
sage: p.permutohedron_join(q)
[3, 1, 2]
sage: r = Permutation([2,1,3])
sage: r.permutohedron_join(p)
[3, 2, 1]

sage: p = Permutation([3,2,4,1])
sage: q = Permutation([4,2,1,3])
sage: p.permutohedron_join(q)
[4, 3, 2, 1]
sage: r = Permutation([3,1,2,4])
sage: p.permutohedron_join(r)
[3, 2, 4, 1]
sage: q.permutohedron_join(r)
[4, 3, 2, 1]
sage: s = Permutation([1,4,2,3])
sage: s.permutohedron_join(r)
[4, 3, 1, 2]
```

The universal property of the join operation is satisfied:

```
sage: def test_uni_join(p, q):
....:     j = p.permutohedron_join(q)
....:     if not p.permutohedron_lequal(j):
....:         return False
....:     if not q.permutohedron_lequal(j):
....:         return False
....:     for r in p.permutohedron_greater():
....:         if q.permutohedron_lequal(r) and not j.permutohedron_lequal(r):
....:             return False
....:     return True
sage: all( test_uni_join(p, q) for p in Permutations(3) for q in Permutations(3) )
True
sage: test_uni_join(Permutation([6, 4, 7, 3, 2, 5, 8, 1]), Permutation([7, 3, 1, 2, 5, 4, 6,
True
```

Border cases:

```
sage: p = Permutation([])
sage: p.permutohedron_join(p)
[]
```

```
sage: p = Permutation([1])
sage: p.permutohedron_join(p)
[1]
```

The left permutohedron:

sage: p = Permutation([3,1,2]) sage: q = Permutation([1,3,2]) sage: p.permutohedron_join(q, side="left") [3, 2, 1] sage: r = Permutation([2,1,3]) sage: r.permutohedron_join(p, side="left") [3, 1, 2]

permutohedron_lequal(*p2*, *side='right'*)

Return `True` if `self` is less or equal to `p2` in the permutohedron order.

By default, the computations are done in the right permutohedron. If you pass the option `side='left'`, then they will be done in the left permutohedron.

For every nonnegative integer n, the right (resp. left) permutohedron order (also called the right (resp. left) weak order, or the right (resp. left) weak Bruhat order) is a partial order on the symmetric group S_n. It can be defined in various ways, including the following ones:

- Two permutations u and v in S_n satisfy $u \leq v$ in the right (resp. left) permutohedron order if and only if the (Coxeter) length of the permutation $v^{-1} \circ u$ (resp. of the permutation $u \circ v^{-1}$) equals the length of v minus the length of u. Here, $p \circ q$ means the permutation obtained by applying q first and then p. (Recall that the Coxeter length of a permutation is its number of inversions.)
- Two permutations u and v in S_n satisfy $u \leq v$ in the right (resp. left) permutohedron order if and only if every pair (i, j) of elements of $\{1, 2, \cdots, n\}$ such that $i < j$ and $u^{-1}(i) > u^{-1}(j)$ (resp. $u(i) > u(j)$) also satisfies $v^{-1}(i) > v^{-1}(j)$ (resp. $v(i) > v(j)$).
- A permutation $v \in S_n$ covers a permutation $u \in S_n$ in the right (resp. left) permutohedron order if and only if we have $v = u \circ (i, i+1)$ (resp. $v = (i, i+1) \circ u$) for some $i \in \{1, 2, \cdots, n-1\}$ satisfying $u(i) < u(i+1)$ (resp. $u^{-1}(i) < u^{-1}(i+1)$). Here, again, $p \circ q$ means the permutation obtained by applying q first and then p.

The right and the left permutohedron order are mutually isomorphic, with the isomorphism being the map sending every permutation to its inverse. Each of these orders endows the symmetric group S_n with the structure of a graded poset (the rank function being the Coxeter length).

Warning: The permutohedron order is not to be mistaken for the strong Bruhat order (`bruhat_lequal()`), despite both orders being occasionally referred to as the Bruhat order.

EXAMPLES:

```
sage: p = Permutation([3,2,1,4])
sage: p.permutohedron_lequal(Permutation([4,2,1,3]))
False
sage: p.permutohedron_lequal(Permutation([4,2,1,3]), side='left')
True
sage: p.permutohedron_lequal(p)
True

sage: Permutation([2,1,3]).permutohedron_lequal(Permutation([2,3,1]))
True
sage: Permutation([2,1,3]).permutohedron_lequal(Permutation([3,1,2]))
False
sage: Permutation([2,1,3]).permutohedron_lequal(Permutation([1,2,3]))
False
sage: Permutation([1,3,2]).permutohedron_lequal(Permutation([2,1,3]))
False
```

```
sage: Permutation([1,3,2]).permutohedron_lequal(Permutation([2,3,1]))
False
sage: Permutation([2,3,1]).permutohedron_lequal(Permutation([1,3,2]))
False
sage: Permutation([2,1,3]).permutohedron_lequal(Permutation([2,3,1]), side='left')
False
sage: sorted( [len([b for b in Permutations(3) if a.permutohedron_lequal(b)])
....:          for a in Permutations(3)] )
[1, 2, 2, 3, 3, 6]
sage: sorted( [len([b for b in Permutations(3) if a.permutohedron_lequal(b, side="left")])
....:          for a in Permutations(3)] )
[1, 2, 2, 3, 3, 6]

sage: Permutation([]).permutohedron_lequal(Permutation([]))
True
```

permutohedron_meet(*other*, *side='right'*)

Return the meet of the permutations `self` and `other` in the right permutohedron order (or, if `side` is set to `'left'`, in the left permutohedron order).

The permutohedron orders (see `permutohedron_lequal()`) are lattices; the meet operation refers to this lattice structure. It is connected to the join operation by the following simple symmetry property: If π and ψ are two permutations π and ψ in the symmetric group S_n, and if w_0 denotes the permutation $(n, n-1, \ldots, 1) \in S_n$, then

$$\pi \wedge \psi = w_0 \circ ((w_0 \circ \pi) \vee (w_0 \circ \psi)) = ((\pi \circ w_0) \vee (\psi \circ w_0)) \circ w_0$$

and

$$\pi \vee \psi = w_0 \circ ((w_0 \circ \pi) \wedge (w_0 \circ \psi)) = ((\pi \circ w_0) \wedge (\psi \circ w_0)) \circ w_0,$$

where $\wedge$ means meet and $\vee$ means join.

See also:

`permutohedron_lequal()`, `permutohedron_join()`.

AUTHORS:

Viviane Pons and Darij Grinberg, 18 June 2014.

EXAMPLES:

```
sage: p = Permutation([3,1,2])
sage: q = Permutation([1,3,2])
sage: p.permutohedron_meet(q)
[1, 3, 2]
sage: r = Permutation([2,1,3])
sage: r.permutohedron_meet(p)
[1, 2, 3]

sage: p = Permutation([3,2,4,1])
sage: q = Permutation([4,2,1,3])
sage: p.permutohedron_meet(q)
[2, 1, 3, 4]
sage: r = Permutation([3,1,2,4])
sage: p.permutohedron_meet(r)
[3, 1, 2, 4]
sage: q.permutohedron_meet(r)
[1, 2, 3, 4]
sage: s = Permutation([1,4,2,3])
```

```
sage: s.permutohedron_meet(r)
[1, 2, 3, 4]
```

The universal property of the meet operation is satisfied:

```
sage: def test_uni_meet(p, q):
....:     m = p.permutohedron_meet(q)
....:     if not m.permutohedron_lequal(p):
....:         return False
....:     if not m.permutohedron_lequal(q):
....:         return False
....:     for r in p.permutohedron_smaller():
....:         if r.permutohedron_lequal(q) and not r.permutohedron_lequal(m):
....:             return False
....:     return True
sage: all( test_uni_meet(p, q) for p in Permutations(3) for q in Permutations(3) )
True
sage: test_uni_meet(Permutation([6, 4, 7, 3, 2, 5, 8, 1]), Permutation([7, 3, 1, 2, 5, 4, 6,
True
```

Border cases:

```
sage: p = Permutation([])
sage: p.permutohedron_meet(p)
[]
sage: p = Permutation([1])
sage: p.permutohedron_meet(p)
[1]
```

The left permutohedron:

sage: p = Permutation([3,1,2]) sage: q = Permutation([1,3,2]) sage: p.permutohedron_meet(q, side="left") [1, 2, 3] sage: r = Permutation([2,1,3]) sage: r.permutohedron_meet(p, side="left") [2, 1, 3]

permutohedron_pred(*side='right'*)

Return a list of the permutations strictly smaller than `self` in the permutohedron order such that there is no permutation between any of those and `self`.

By default, the computations are done in the right permutohedron. If you pass the option `side='left'`, then they will be done in the left permutohedron.

See `permutohedron_lequal()` for the definition of the permutohedron orders.

EXAMPLES:

```
sage: p = Permutation([4,2,1,3])
sage: p.permutohedron_pred()
[[2, 4, 1, 3], [4, 1, 2, 3]]
sage: p.permutohedron_pred(side='left')
[[4, 1, 2, 3], [3, 2, 1, 4]]
```

permutohedron_smaller(*side='right'*)

Return a list of permutations smaller than or equal to `self` in the permutohedron order.

By default, the computations are done in the right permutohedron. If you pass the option `side='left'`, then they will be done in the left permutohedron.

See `permutohedron_lequal()` for the definition of the permutohedron orders.

EXAMPLES:

```
sage: Permutation([4,2,1,3]).permutohedron_smaller()
[[1, 2, 3, 4],
 [1, 2, 4, 3],
 [1, 4, 2, 3],
 [2, 1, 3, 4],
 [2, 1, 4, 3],
 [2, 4, 1, 3],
 [4, 1, 2, 3],
 [4, 2, 1, 3]]

sage: Permutation([4,2,1,3]).permutohedron_smaller(side='left')
[[1, 2, 3, 4],
 [1, 3, 2, 4],
 [2, 1, 3, 4],
 [2, 3, 1, 4],
 [3, 1, 2, 4],
 [3, 2, 1, 4],
 [4, 1, 2, 3],
 [4, 2, 1, 3]]
```

permutohedron_succ(*side='right'*)

Return a list of the permutations strictly greater than `self` in the permutohedron order such that there is no permutation between any of those and `self`.

By default, the computations are done in the right permutohedron. If you pass the option `side='left'`, then they will be done in the left permutohedron.

See `permutohedron_lequal()` for the definition of the permutohedron orders.

EXAMPLES:

```
sage: p = Permutation([4,2,1,3])
sage: p.permutohedron_succ()
[[4, 2, 3, 1]]
sage: p.permutohedron_succ(side='left')
[[4, 3, 1, 2]]
```

prev()

Return the permutation that comes directly before `self` in lexicographic order on the symmetric group containing `self`. If `self` is the first permutation, then it returns `False`.

EXAMPLES:

```
sage: p = Permutation([1,2,3])
sage: p.prev()
False
sage: p = Permutation([1,3,2])
sage: p.prev()
[1, 2, 3]
```

TESTS:

```
sage: p = Permutation([])
sage: p.prev()
False
```

Check that trac ticket #16913 is fixed:

```
sage: Permutation([1,4,3,2]).prev()
[1, 4, 2, 3]
```

rank()
Return the rank of `self` in the lexicographic ordering on the symmetric group to which `self` belongs.

EXAMPLES:

```
sage: Permutation([1,2,3]).rank()
0
sage: Permutation([1, 2, 4, 6, 3, 5]).rank()
10
sage: perms = Permutations(6).list()
sage: [p.rank() for p in perms ] == range(factorial(6))
True
```

recoils()
Return the list of the positions of the recoils of `self`.

A recoil of a permutation p is an integer i such that $i+1$ appears to the left of i in p. Here, the positions are being counted starting at 0. (Note that it is the positions, not the recoils themselves, which are being listed.)

EXAMPLES:

```
sage: Permutation([1,4,3,2]).recoils()
[2, 3]
sage: Permutation([]).recoils()
[]
```

recoils_composition()
Return the recoils composition of `self`.

The recoils composition of a permutation $p \in S_n$ is the composition of n whose descent set is the set of the recoils of p (not their positions). In other words, this is the descents composition of p^{-1}.

EXAMPLES:

```
sage: Permutation([1,3,2,4]).recoils_composition()
[2, 2]
sage: Permutation([]).recoils_composition()
[]
```

reduced_word()
Return a reduced word of the permutation `self`.

See `reduced_words()` for the definition of reduced words and a way to compute them all.

Warning: This does not respect the multiplication convention.

EXAMPLES:

```
sage: Permutation([3,5,4,6,2,1]).reduced_word()
[2, 1, 4, 3, 2, 4, 3, 5, 4, 5]

Permutation([1]).reduced_word_lexmin()
[]
Permutation([]).reduced_word_lexmin()
[]
```

reduced_word_lexmin()
Return a lexicographically minimal reduced word of the permutation `self`.

See `reduced_words()` for the definition of reduced words and a way to compute them all.

EXAMPLES:

```
sage: Permutation([3,4,2,1]).reduced_word_lexmin()
[1, 2, 1, 3, 2]

Permutation([1]).reduced_word_lexmin()
[]
Permutation([]).reduced_word_lexmin()
[]
```

reduced_words()

Return a list of the reduced words of `self`.

The notion of a reduced word is based on the well-known fact that every permutation can be written as a product of adjacent transpositions. In more detail: If n is a nonnegative integer, we can define the transpositions $s_i = (i, i+1) \in S_n$ for all $i \in \{1, 2, \ldots, n-1\}$, and every $p \in S_n$ can then be written as a product $s_{i_1} s_{i_2} \cdots s_{i_k}$ for some sequence $(i_1, i_2, \ldots, i_k)$ of elements of $\{1, 2, \ldots, n-1\}$ (here $\{1, 2, \ldots, n-1\}$ denotes the empty set when $n \leq 1$). Fixing a p, the sequences $(i_1, i_2, \ldots, i_k)$ of smallest length satisfying $p = s_{i_1} s_{i_2} \cdots s_{i_k}$ are called the reduced words of p. (Their length is the Coxeter length of p, and can be computed using `length()`.)

Note that the product of permutations is defined here in such a way that $(pq)(i) = p(q(i))$ for all permutations p and q and each $i \in \{1, 2, \ldots, n\}$ (this is the same convention as in `left_action_product()`, but not the default semantics of the $*$ operator on permutations in Sage). Thus, for instance, $s_2 s_1$ is the permutation obtained by first transposing 1 with 2 and then transposing 2 with 3.

See also:

`reduced_word()`, `reduced_word_lexmin()`

EXAMPLES:

```
sage: Permutation([2,1,3]).reduced_words()
[[1]]
sage: Permutation([3,1,2]).reduced_words()
[[2, 1]]
sage: Permutation([3,2,1]).reduced_words()
[[1, 2, 1], [2, 1, 2]]
sage: Permutation([3,2,4,1]).reduced_words()
[[1, 2, 3, 1], [1, 2, 1, 3], [2, 1, 2, 3]]

Permutation([1]).reduced_words()
[[]]
Permutation([]).reduced_words()
[[]]
```

remove_extra_fixed_points()

Return the permutation obtained by removing any fixed points at the end of `self`.

EXAMPLES:

```
sage: Permutation([2,1,3]).remove_extra_fixed_points()
[2, 1]
sage: Permutation([1,2,3,4]).remove_extra_fixed_points()
[1]
```

See also:

`retract_plain()`

retract_direct_product(m)

Return the direct-product retract of the permutation `self` $\in S_n$ to S_m, where $m \leq n$. If this retract is

undefined, then `None` is returned.

If $p \in S_n$ is a permutation, and m is a nonnegative integer less or equal to n, then the direct-product retract of p to S_m is defined only if $p([m]) = [m]$, where $[m]$ denotes the interval $\{1, 2, \ldots, m\}$. In this case, it is defined as the permutation written $(p(1), p(2), \ldots, p(m))$ in one-line notation.

EXAMPLES:

```
sage: Permutation([4,1,2,3,5]).retract_direct_product(4)
[4, 1, 2, 3]
sage: Permutation([4,1,2,3,5]).retract_direct_product(3)

sage: Permutation([1,4,2,3,6,5]).retract_direct_product(5)
sage: Permutation([1,4,2,3,6,5]).retract_direct_product(4)
[1, 4, 2, 3]
sage: Permutation([1,4,2,3,6,5]).retract_direct_product(3)
sage: Permutation([1,4,2,3,6,5]).retract_direct_product(2)
sage: Permutation([1,4,2,3,6,5]).retract_direct_product(1)
[1]
sage: Permutation([1,4,2,3,6,5]).retract_direct_product(0)
[]

sage: all( p.retract_direct_product(3) == p for p in Permutations(3) )
True
```

See also:

`retract_plain()`, `retract_okounkov_vershik()`

retract_okounkov_vershik(*m*)

Return the Okounkov-Vershik retract of the permutation `self` $\in S_n$ to S_m, where $m \leq n$.

If $p \in S_n$ is a permutation, and m is a nonnegative integer less or equal to n, then the Okounkov-Vershik retract of p to S_m is defined as the permutation in S_m which sends every $i \in \{1, 2, \ldots, m\}$ to $p^{k_i}(i)$, where k_i is the smallest positive integer k satisfying $p^k(i) \leq m$.

In other words, the Okounkov-Vershik retract of p is the permutation whose disjoint cycle decomposition is obtained by removing all letters strictly greater than m from the decomposition of p into disjoint cycles (and removing all cycles which are emptied in the process).

When $m = n - 1$, the Okounkov-Vershik retract (as a map $S_n \to S_{n-1}$) is the map $\widetilde{p}_n$ introduced in Section 7 of [OkounkovVershik2], and appears as (3.20) in [CST10]. In the general case, the Okounkov-Vershik retract of a permutation in S_n to S_m can be obtained by first taking its Okounkov-Vershik retract to S_{n-1}, then that of the resulting permutation to S_{n-2}, etc. until arriving in S_m.

REFERENCES:

EXAMPLES:

```
sage: Permutation([4,1,2,3,5]).retract_okounkov_vershik(4)
[4, 1, 2, 3]
sage: Permutation([4,1,2,3,5]).retract_okounkov_vershik(3)
[3, 1, 2]
sage: Permutation([4,1,2,3,5]).retract_okounkov_vershik(2)
[2, 1]
sage: Permutation([4,1,2,3,5]).retract_okounkov_vershik(1)
[1]
sage: Permutation([4,1,2,3,5]).retract_okounkov_vershik(0)
[]

sage: Permutation([1,4,2,3,6,5]).retract_okounkov_vershik(5)
[1, 4, 2, 3, 5]
```

```
sage: Permutation([1,4,2,3,6,5]).retract_okounkov_vershik(4)
[1, 4, 2, 3]
sage: Permutation([1,4,2,3,6,5]).retract_okounkov_vershik(3)
[1, 3, 2]
sage: Permutation([1,4,2,3,6,5]).retract_okounkov_vershik(2)
[1, 2]
sage: Permutation([1,4,2,3,6,5]).retract_okounkov_vershik(1)
[1]
sage: Permutation([1,4,2,3,6,5]).retract_okounkov_vershik(0)
[]

sage: Permutation([6,5,4,3,2,1]).retract_okounkov_vershik(5)
[1, 5, 4, 3, 2]
sage: Permutation([6,5,4,3,2,1]).retract_okounkov_vershik(4)
[1, 2, 4, 3]

sage: Permutation([1,5,2,6,3,7,4,8]).retract_okounkov_vershik(4)
[1, 3, 2, 4]

sage: all( p.retract_direct_product(3) == p for p in Permutations(3) )
True
```

See also:

retract_plain(), retract_direct_product()

retract_plain(*m*)

Return the plain retract of the permutation `self` in S_n to S_m, where $m \leq n$. If this retract is undefined, then `None` is returned.

If $p \in S_n$ is a permutation, and m is a nonnegative integer less or equal to n, then the plain retract of p to S_m is defined only if every $i > m$ satisfies $p(i) = i$. In this case, it is defined as the permutation written $(p(1), p(2), \ldots, p(m))$ in one-line notation.

EXAMPLES:

```
sage: Permutation([4,1,2,3,5]).retract_plain(4)
[4, 1, 2, 3]
sage: Permutation([4,1,2,3,5]).retract_plain(3)

sage: Permutation([1,3,2,4,5,6]).retract_plain(3)
[1, 3, 2]
sage: Permutation([1,3,2,4,5,6]).retract_plain(2)

sage: Permutation([1,2,3,4,5]).retract_plain(1)
[1]
sage: Permutation([1,2,3,4,5]).retract_plain(0)
[]

sage: all( p.retract_plain(3) == p for p in Permutations(3) )
True
```

See also:

retract_direct_product(), retract_okounkov_vershik(), remove_extra_fixed_points()

reverse()

Returns the permutation obtained by reversing the list.

EXAMPLES:

```
sage: Permutation([3,4,1,2]).reverse()
[2, 1, 4, 3]
sage: Permutation([1,2,3,4,5]).reverse()
[5, 4, 3, 2, 1]
```

right_action_product(*rp*)

Return the permutation obtained by composing `self` with `rp` in such an order that `self` is applied first and `rp` is applied afterwards.

This is usually denoted by either `self * rp` or `rp * self` depending on the conventions used by the author. If the value of a permutation $p \in S_n$ on an integer $i \in \{1, 2, \cdots, n\}$ is denoted by $p(i)$, then this should be denoted by `rp * self` in order to have associativity (i.e., in order to have $(p \cdot q)(i) = p(q(i))$ for all p, q and i). If, on the other hand, the value of a permutation $p \in S_n$ on an integer $i \in \{1, 2, \cdots, n\}$ is denoted by i^p, then this should be denoted by `self * rp` in order to have associativity (i.e., in order to have $i^{p \cdot q} = (i^p)^q$ for all p, q and i).

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: q = Permutation([3,1,2])
sage: p.right_action_product(q)
[1, 3, 2]
sage: q.right_action_product(p)
[3, 2, 1]
```

right_permutohedron_interval(*other*)

Return the list of the permutations belonging to the right permutohedron interval where `self` is the minimal element and `other` the maximal element.

See `permutohedron_lequal()` for the definition of the permutohedron orders.

EXAMPLES:

```
sage: Permutation([2, 1, 4, 5, 3]).right_permutohedron_interval(Permutation([2, 5, 4, 1, 3]))
[[2, 1, 4, 5, 3], [2, 1, 5, 4, 3], [2, 4, 1, 5, 3], [2, 4, 5, 1, 3], [2, 5, 1, 4, 3], [2, 5,
```

TESTS:

```
sage: Permutation([]).right_permutohedron_interval(Permutation([]))
[[]]
sage: Permutation([3, 1, 2]).right_permutohedron_interval(Permutation([3, 1, 2]))
[[3, 1, 2]]
sage: Permutation([1, 3, 2, 4]).right_permutohedron_interval(Permutation([3, 4, 2, 1]))
[[1, 3, 2, 4], [1, 3, 4, 2], [3, 1, 2, 4], [3, 1, 4, 2], [3, 2, 1, 4], [3, 2, 4, 1], [3, 4,
sage: Permutation([2, 1, 4, 5, 3]).right_permutohedron_interval(Permutation([2, 5, 4, 1, 3]))
[[2, 1, 4, 5, 3], [2, 1, 5, 4, 3], [2, 4, 1, 5, 3], [2, 4, 5, 1, 3], [2, 5, 1, 4, 3], [2, 5,
sage: Permutation([2, 5, 4, 1, 3]).right_permutohedron_interval(Permutation([2, 1, 4, 5, 3]))
Traceback (most recent call last):
...
ValueError: [2, 5, 4, 1, 3] must be lower or equal than [2, 1, 4, 5, 3] for the right permut
sage: Permutation([2, 4, 1, 3]).right_permutohedron_interval(Permutation([2, 1, 4, 5, 3]))
Traceback (most recent call last):
...
ValueError: len([2, 4, 1, 3]) and len([2, 1, 4, 5, 3]) must be equal
```

right_permutohedron_interval_iterator(*other*)

Return an iterator on the permutations (represented as integer lists) belonging to the right permutohedron interval where `self` is the minimal element and `other` the maximal element.

See permutohedron_lequal() for the definition of the permutohedron orders.

EXAMPLES:

```
sage: Permutation([2, 1, 4, 5, 3]).right_permutohedron_interval(Permutation([2, 5, 4, 1, 3])
[[2, 1, 4, 5, 3], [2, 1, 5, 4, 3], [2, 4, 1, 5, 3], [2, 4, 5, 1, 3], [2, 5, 1, 4, 3], [2, 5,
```

right_tableau()

Return the right standard tableau after performing the RSK algorithm on self.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).right_tableau()
[[1, 2], [3], [4]]
```

robinson_schensted()

Return the pair of standard tableaux obtained by running the Robinson-Schensted algorithm on self.

This can also be done by running RSK() on self (with the optional argument check_standard=True to return standard Young tableaux).

EXAMPLES:

```
sage: Permutation([6,2,3,1,7,5,4]).robinson_schensted()
[[[1, 3, 4], [2, 5], [6, 7]], [[1, 3, 5], [2, 6], [4, 7]]]
```

runs(*as_tuple=False*)

Return a list of the runs in the nonempty permutation self.

A run in a permutation is defined to be a maximal (with respect to inclusion) nonempty increasing substring (i. e., contiguous subsequence). For instance, the runs in the permutation [6,1,7,3,4,5,2] are [6], [1,7], [3,4,5] and [2].

Runs in an empty permutation are not defined.

INPUT:

- as_tuple – boolean (default: False) choice of output format

OUTPUT:

a list of lists or a tuple of tuples

REFERENCES:

- http://mathworld.wolfram.com/PermutationRun.html

EXAMPLES:

```
sage: Permutation([1,2,3,4]).runs()
[[1, 2, 3, 4]]
sage: Permutation([4,3,2,1]).runs()
[[4], [3], [2], [1]]
sage: Permutation([2,4,1,3]).runs()
[[2, 4], [1, 3]]
sage: Permutation([1]).runs()
[[1]]
```

The example from above:

```
sage: Permutation([6,1,7,3,4,5,2]).runs()
[[6], [1, 7], [3, 4, 5], [2]]
sage: Permutation([6,1,7,3,4,5,2]).runs(as_tuple=True)
((6,), (1, 7), (3, 4, 5), (2,))
```

The number of runs in a nonempty permutation equals its number of descents plus 1:

```
sage: all( len(p.runs()) == p.number_of_descents() + 1
....:      for p in Permutations(6) )
True
```

saliances()

Return a list of the saliances of the permutation `self`.

A saliance of a permutation p is an integer i such that $p(i) > p(j)$ for all $j > i$.

EXAMPLES:

```
sage: Permutation([2,3,1,5,4]).saliances()
[3, 4]
sage: Permutation([5,4,3,2,1]).saliances()
[0, 1, 2, 3, 4]
```

shifted_concatenation(*other*, *side='right'*)

Return the right (or left) shifted concatenation of `self` with a permutation `other`. These operations are also known as the Loday-Ronco over and under operations.

INPUT:

- `other` – a permutation, a list, a tuple, or any iterable representing a permutation.
- `side` – (default: `"right"`) the string "left" or "right".

OUTPUT:

If `side` is `"right"`, the method returns the permutation obtained by concatenating `self` with the letters of `other` incremented by the size of `self`. This is what is called `side / other` in [LodRon0102066], and denoted as the "over" operation. Otherwise, i. e., when `side` is `"left"`, the method returns the permutation obtained by concatenating the letters of `other` incremented by the size of `self` with `self`. This is what is called `side \ other` in [LodRon0102066] (which seems to use the $(\sigma\pi)(i) = \pi(\sigma(i))$ convention for the product of permutations).

EXAMPLES:

```
sage: Permutation([]).shifted_concatenation(Permutation([]), "right")
[]
sage: Permutation([]).shifted_concatenation(Permutation([]), "left")
[]
sage: Permutation([2, 4, 1, 3]).shifted_concatenation(Permutation([3, 1, 2]), "right")
[2, 4, 1, 3, 7, 5, 6]
sage: Permutation([2, 4, 1, 3]).shifted_concatenation(Permutation([3, 1, 2]), "left")
[7, 5, 6, 2, 4, 1, 3]
```

REFERENCES:

shifted_shuffle(*other*)

Return the shifted shuffle of two permutations `self` and `other`.

INPUT:

- `other` – a permutation, a list, a tuple, or any iterable representing a permutation.

OUTPUT:

The list of the permutations appearing in the shifted shuffle of the permutations `self` and `other`.

EXAMPLES:

```
sage: Permutation([]).shifted_shuffle(Permutation([]))
[[]]
sage: Permutation([1, 2, 3]).shifted_shuffle(Permutation([1]))
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [4, 1, 2, 3]]
sage: Permutation([1, 2]).shifted_shuffle(Permutation([2, 1]))
[[1, 2, 4, 3], [1, 4, 2, 3], [1, 4, 3, 2], [4, 1, 2, 3], [4, 1, 3, 2], [4, 3, 1, 2]]
sage: Permutation([1]).shifted_shuffle([1])
[[1, 2], [2, 1]]
sage: len(Permutation([3, 1, 5, 4, 2]).shifted_shuffle(Permutation([2, 1, 4, 3])))
126
```

The shifted shuffle product is associative. We can test this on an admittedly toy example:

```
sage: all( all( all( sorted(flatten([abs.shifted_shuffle(c)
....:                                for abs in a.shifted_shuffle(b)]))
....:                == sorted(flatten([a.shifted_shuffle(bcs)
....:                                   for bcs in b.shifted_shuffle(c)]))
....:                for c in Permutations(2) )
....:           for b in Permutations(2) )
....:      for a in Permutations(2) )
True
```

The `shifted_shuffle` method on permutations gives the same permutations as the `shifted_shuffle` method on words (but is faster):

```
sage: all( all( sorted(p1.shifted_shuffle(p2))
....:           == sorted([Permutation(p) for p in
....:                      Word(p1).shifted_shuffle(Word(p2))])
....:           for p2 in Permutations(3) )
....:      for p1 in Permutations(2) )
True
```

show(*representation='cycles'*, *orientation='landscape'*, ***args*)

Display the permutation as a drawing.

INPUT:

- `representation` – different kinds of drawings are available
  - `"cycles"` (default) – the permutation is displayed as a collection of directed cycles
  - `"braid"` – the permutation is displayed as segments linking each element $1, ..., n$ to its image on a parallel line.

    When using this drawing, it is also possible to display the permutation horizontally (`orientation = "landscape"`, default option) or vertically (`orientation = "portrait"`).
  - `"chord-diagram"` – the permutation is displayed as a directed graph, all of its vertices being located on a circle.

All additional arguments are forwarded to the `show` subcalls.

EXAMPLES:

```
sage: Permutations(20).random_element().show(representation = "cycles")
sage: Permutations(20).random_element().show(representation = "chord-diagram")
sage: Permutations(20).random_element().show(representation = "braid")
sage: Permutations(20).random_element().show(representation = "braid", orientation='portrait
```

TESTS:

```
sage: Permutations(20).random_element().show(representation = "modern_art")
Traceback (most recent call last):
...
ValueError: The value of 'representation' must be equal to 'cycles', 'chord-diagram' or 'bra
```

sign()

Return the signature of the permutation `self`. This is $(-1)^l$, where l is the number of inversions of `self`.

Note: `sign()` can be used as an alias for `signature()`.

EXAMPLES:

```
sage: Permutation([4, 2, 3, 1, 5]).signature()
-1
sage: Permutation([1,3,2,5,4]).sign()
1
sage: Permutation([]).sign()
1
```

signature()

Return the signature of the permutation `self`. This is $(-1)^l$, where l is the number of inversions of `self`.

Note: `sign()` can be used as an alias for `signature()`.

EXAMPLES:

```
sage: Permutation([4, 2, 3, 1, 5]).signature()
-1
sage: Permutation([1,3,2,5,4]).sign()
1
sage: Permutation([]).sign()
1
```

simion_schmidt(*avoid=[1, 2, 3]*)

Implements the Simion-Schmidt map which sends an arbitrary permutation to a pattern avoiding permutation, where the permutation pattern is one of four length-three patterns. This method also implements the bijection between (for example) `[1,2,3]`- and `[1,3,2]`-avoiding permutations.

INPUT:

- `avoid` – one of the patterns `[1,2,3]`, `[1,3,2]`, `[3,1,2]`, `[3,2,1]`.

EXAMPLES:

```
sage: P=Permutations(6)
sage: p=P([4,5,1,6,3,2])
sage: pl= [ [1,2,3], [1,3,2], [3,1,2], [3,2,1] ]
sage: for q in pl:
....:     s=p.simion_schmidt(q)
....:     print s, s.has_pattern(q)
....:
[4, 6, 1, 5, 3, 2] False
[4, 2, 1, 3, 5, 6] False
[4, 5, 3, 6, 2, 1] False
[4, 5, 1, 6, 2, 3] False
```

size()

Return the size of `self`.

EXAMPLES:

```
sage: Permutation([3,4,1,2,5]).size()
5
```

sylvester_class(*left_to_right=False*)

Iterate over the equivalence class of the permutation `self` under sylvester congruence.

Sylvester congruence is an equivalence relation on the set S_n of all permutations of n. It is defined as the smallest equivalence relation such that every permutation of the form $uacvbw$ with u, v and w being words and a, b and c being letters satisfying $a \leq b < c$ is equivalent to the permutation $ucavbw$. (Here, permutations are regarded as words by way of one-line notation.) This definition comes from [HNT05], Definition 8, where it is more generally applied to arbitrary words.

The equivalence class of a permutation $p \in S_n$ under sylvester congruence is called the *sylvester class* of p. It is an interval in the right permutohedron order (see `permutohedron_lequal()`) on S_n.

This is related to the `sylvester_class()` method in that the equivalence class of a permutation π under sylvester congruence is the sylvester class of the right-to-left binary search tree of π. However, the present method yields permutations, while the method on labelled binary trees yields plain lists.

If the variable `left_to_right` is set to `True`, the method instead iterates over the equivalence class of `self` with respect to the *left* sylvester congruence. The left sylvester congruence is easiest to define by saying that two permutations are equivalent under it if and only if their reverses (`reverse()`) are equivalent under (standard) sylvester congruence.

EXAMPLES:

The sylvester class of a permutation in S_5:

```
sage: p = Permutation([3, 5, 1, 2, 4])
sage: sorted(p.sylvester_class())
[[1, 3, 2, 5, 4],
 [1, 3, 5, 2, 4],
 [1, 5, 3, 2, 4],
 [3, 1, 2, 5, 4],
 [3, 1, 5, 2, 4],
 [3, 5, 1, 2, 4],
 [5, 1, 3, 2, 4],
 [5, 3, 1, 2, 4]]
```

The sylvester class of a permutation p contains p:

```
sage: all( p in p.sylvester_class() for p in Permutations(4) )
True
```

Small cases:

```
sage: list(Permutation([]).sylvester_class())
[[]]

sage: list(Permutation([1]).sylvester_class())
[[1]]
```

The sylvester classes in S_3:

```
sage: [sorted(p.sylvester_class()) for p in Permutations(3)]
[[[1, 2, 3]],
 [[1, 3, 2], [3, 1, 2]],
 [[2, 1, 3]],
 [[2, 3, 1]],
```

```
 [[1, 3, 2], [3, 1, 2]],
 [[3, 2, 1]]]
```

The left sylvester classes in S_3:

```
sage: [sorted(p.sylvester_class(left_to_right=True)) for p in Permutations(3)]
[[[1, 2, 3]],
 [[1, 3, 2]],
 [[2, 1, 3], [2, 3, 1]],
 [[2, 1, 3], [2, 3, 1]],
 [[3, 1, 2]],
 [[3, 2, 1]]]
```

A left sylvester class in S_5:

```
sage: p = Permutation([4, 2, 1, 5, 3])
sage: sorted(p.sylvester_class(left_to_right=True))
[[4, 2, 1, 3, 5],
 [4, 2, 1, 5, 3],
 [4, 2, 3, 1, 5],
 [4, 2, 3, 5, 1],
 [4, 2, 5, 1, 3],
 [4, 2, 5, 3, 1],
 [4, 5, 2, 1, 3],
 [4, 5, 2, 3, 1]]
```

to_alternating_sign_matrix()

Return a matrix representing the permutation in the AlternatingSignMatrix class.

EXAMPLES:

```
sage: m = Permutation([1,2,3]).to_alternating_sign_matrix(); m
[1 0 0]
[0 1 0]
[0 0 1]
sage: parent(m)
Alternating sign matrices of size 3
```

to_cycles(*singletons=True*)

Return the permutation `self` as a list of disjoint cycles.

The cycles are returned in the order of increasing smallest elements, and each cycle is returned as a tuple which starts with its smallest element.

If `singletons=False` is given, the list does not contain the singleton cycles.

EXAMPLES:

```
sage: Permutation([2,1,3,4]).to_cycles()
[(1, 2), (3,), (4,)]
sage: Permutation([2,1,3,4]).to_cycles(singletons=False)
[(1, 2)]

sage: Permutation([4,1,5,2,6,3]).to_cycles()
[(1, 4, 2), (3, 5, 6)]
```

The algorithm is of complexity $O(n)$ where n is the size of the given permutation.

TESTS:

```
sage: from sage.combinat.permutation import from_cycles
sage: for n in range(1,6):
....:     for p in Permutations(n):
....:         if from_cycles(n, p.to_cycles()) != p:
....:             print "There is a problem with ",p
....:             break
sage: size = 10000
sage: sample = (Permutations(size).random_element() for i in range(5))
sage: all(from_cycles(size, p.to_cycles()) == p for p in sample)
True
```

Note: there is an alternative implementation called `_to_cycle_set` which could be slightly (10%) faster for some input (typically for permutations of size in the range [100, 10000]). You can run the following benchmarks. For small permutations:

```
sage: for size in range(9): # not tested
....:  print size
....:  lp = Permutations(size).list()
....:  timeit('[p.to_cycles(False) for p in lp]')
....:  timeit('[p._to_cycles_set(False) for p in lp]')
....:  timeit('[p._to_cycles_list(False) for p in lp]')
....:  timeit('[p._to_cycles_orig(False) for p in lp]')
```

and larger ones:

```
sage: for size in [10, 20, 50, 75, 100, 200, 500, 1000, # not tested
....:        2000, 5000, 10000, 15000, 20000, 30000,
....:        50000, 80000, 100000]:
....:     print(size)
....:     lp = [Permutations(size).random_element() for i in range(20)]
....:     timeit("[p.to_cycles() for p in lp]")
....:     timeit("[p._to_cycles_set() for p in lp]")
....:     timeit("[p._to_cycles_list() for p in lp]")
```

to_inversion_vector()

Return the inversion vector of `self`.

The inversion vector of a permutation $p \in S_n$ is defined as the vector $(v_1, v_2, \ldots, v_n)$, where v_i is the number of elements larger than i that appear to the left of i in the permutation p.

The algorithm is of complexity $O(n \log(n))$ where n is the size of the given permutation.

EXAMPLES:

```
sage: Permutation([5,9,1,8,2,6,4,7,3]).to_inversion_vector()
[2, 3, 6, 4, 0, 2, 2, 1, 0]
sage: Permutation([8,7,2,1,9,4,6,5,10,3]).to_inversion_vector()
[3, 2, 7, 3, 4, 3, 1, 0, 0, 0]
sage: Permutation([3,2,4,1,5]).to_inversion_vector()
[3, 1, 0, 0, 0]
```

TESTS:

```
sage: from sage.combinat.permutation import from_inversion_vector
sage: all(from_inversion_vector(p.to_inversion_vector()) == p
....:   for n in range(6) for p in Permutations(n))
True

sage: P = Permutations(1000)
sage: sample = (P.random_element() for i in range(5))
```

```
sage: all(from_inversion_vector(p.to_inversion_vector()) == p
....:   for p in sample)
True
```

to_lehmer_cocode()

Return the Lehmer cocode of the permutation `self`.

The Lehmer cocode of a permutation p is defined as the list $(c_1, c_2, \ldots, c_n)$, where c_i is the number of $j < i$ such that $p(j) > p(i)$.

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: p.to_lehmer_cocode()
[0, 1, 0]
sage: q = Permutation([3,1,2])
sage: q.to_lehmer_cocode()
[0, 1, 1]
```

to_lehmer_code()

Return the Lehmer code of the permutation `self`.

The Lehmer code of a permutation p is defined as the list $[c[1], c[2], ..., c[n]]$, where $c[i]$ is the number of $j > i$ such that $p(j) < p(i)$.

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: p.to_lehmer_code()
[1, 0, 0]
sage: q = Permutation([3,1,2])
sage: q.to_lehmer_code()
[2, 0, 0]

sage: Permutation([1]).to_lehmer_code()
[0]
sage: Permutation([]).to_lehmer_code()
[]
```

TESTS:

```
sage: from sage.combinat.permutation import from_lehmer_code
sage: all(from_lehmer_code(p.to_lehmer_code()) == p
....:   for n in range(6) for p in Permutations(n))
True

sage: P = Permutations(1000)
sage: sample = (P.random_element() for i in range(5))
sage: all(from_lehmer_code(p.to_lehmer_code()) == p
....:   for p in sample)
True
```

to_major_code(*final_descent=False*)

Return the major code of the permutation `self`.

The major code of a permutation p is defined as the sequence $(m_1 - m_2, m_2 - m_3, \ldots, m_n)$, where m_i is the major index of the permutation obtained by erasing all letters smaller than i from p.

With the `final_descent` option, the last position of a non-empty permutation is also considered as a descent. This has an effect on the computation of major indices.

REFERENCES:

•Carlitz, L. *q-Bernoulli and Eulerian Numbers*. Trans. Amer. Math. Soc. 76 (1954) 332-350. http://www.ams.org/journals/tran/1954-076-02/S0002-9947-1954-0060538-2/

•Skandera, M. *An Eulerian Partner for Inversions*. Sem. Lothar. Combin. 46 (2001) B46d. http://www.lehigh.edu/~mas906/papers/partner.ps

EXAMPLES:

```
sage: Permutation([9,3,5,7,2,1,4,6,8]).to_major_code()
[5, 0, 1, 0, 1, 2, 0, 1, 0]
sage: Permutation([2,8,4,3,6,7,9,5,1]).to_major_code()
[8, 3, 3, 1, 4, 0, 1, 0, 0]
```

to_matrix()

Return a matrix representing the permutation.

EXAMPLES:

```
sage: Permutation([1,2,3]).to_matrix()
[1 0 0]
[0 1 0]
[0 0 1]

sage: Permutation([1,3,2]).to_matrix()
[1 0 0]
[0 0 1]
[0 1 0]
```

Notice that matrix multiplication corresponds to permutation multiplication only when the permutation option mult='r2l'

```
sage: PermutationOptions(mult='r2l')
sage: p = Permutation([2,1,3])
sage: q = Permutation([3,1,2])
sage: (p*q).to_matrix()
[0 0 1]
[0 1 0]
[1 0 0]
sage: p.to_matrix()*q.to_matrix()
[0 0 1]
[0 1 0]
[1 0 0]
sage: PermutationOptions(mult='l2r')
sage: (p*q).to_matrix()
[1 0 0]
[0 0 1]
[0 1 0]
```

to_permutation_group_element()

Returns a PermutationGroupElement equal to self.

EXAMPLES:

```
sage: Permutation([2,1,4,3]).to_permutation_group_element()
(1,2)(3,4)
sage: Permutation([1,2,3]).to_permutation_group_element()
()
```

to_tableau_by_shape(*shape*)

Return a tableau of shape `shape` with the entries in `self`. The tableau is such that the reading word (i.

e., the word obtained by reading the tableau row by row, starting from the top row in English notation, with each row being read from left to right) is `self`.

EXAMPLES:

```
sage: Permutation([3,4,1,2,5]).to_tableau_by_shape([3,2])
[[1, 2, 5], [3, 4]]
sage: Permutation([3,4,1,2,5]).to_tableau_by_shape([3,2]).reading_word_permutation()
[3, 4, 1, 2, 5]
```

weak_excedences()

Return all the numbers `self[i]` such that `self[i] >= i+1`.

EXAMPLES:

```
sage: Permutation([1,4,3,2,5]).weak_excedences()
[1, 4, 3, 5]
```

class sage.combinat.permutation.**Permutations**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Permutations.

`Permutations(n)` returns the class of permutations of `n`, if `n` is an integer, list, set, or string.

`Permutations(n, k)` returns the class of length-`k` partial permutations of `n` (where `n` is any of the above things); `k` must be a nonnegative integer. A length-k partial permutation of n is defined as a k-tuple of pairwise distinct elements of $\{1, 2, \ldots, n\}$.

Valid keyword arguments are: 'descents', 'bruhat_smaller', 'bruhat_greater', 'recoils_finer', 'recoils_fatter', 'recoils', and 'avoiding'. With the exception of 'avoiding', you cannot specify `n` or `k` along with a keyword.

`Permutations(descents=(list,n))` returns the class of permutations of n with descents in the positions specified by `list`. This uses the slightly nonstandard convention that the images of $1, 2, ..., n$ under the permutation are regarded as positions $0, 1, ..., n-1$, so for example the presence of 1 in `list` signifies that the permutations π should satisfy $\pi(2) > \pi(3)$. Note that `list` is supposed to be a list of positions of the descents, not the descents composition. It does *not* return the class of permutations with descents composition `list`.

`Permutations(bruhat_smaller=p)` and `Permutations(bruhat_greater=p)` return the class of permutations smaller-or-equal or greater-or-equal, respectively, than the given permutation `p` in the Bruhat order. (The Bruhat order is defined in `bruhat_lequal()`. It is also referred to as the *strong* Bruhat order.)

`Permutations(recoils=p)` returns the class of permutations whose recoils composition is `p`. Unlike the `descents=(list, n)` syntax, this actually takes a *composition* as input.

`Permutations(recoils_fatter=p)` and `Permutations(recoils_finer=p)` return the class of permutations whose recoils composition is fatter or finer, respectively, than the given composition `p`.

`Permutations(n, avoiding=P)` returns the class of permutations of `n` avoiding `P`. Here `P` may be a single permutation or a list of permutations; the returned class will avoid all patterns in `P`.

EXAMPLES:

```
sage: p = Permutations(3); p
Standard permutations of 3
sage: p.list()
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

sage: p = Permutations(3, 2); p
Permutations of {1,...,3} of length 2
sage: p.list()
[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]
```

```
sage: p = Permutations(['c', 'a', 't']); p
Permutations of the set ['c', 'a', 't']
sage: p.list()
[['c', 'a', 't'],
 ['c', 't', 'a'],
 ['a', 'c', 't'],
 ['a', 't', 'c'],
 ['t', 'c', 'a'],
 ['t', 'a', 'c']]

sage: p = Permutations(['c', 'a', 't'], 2); p
Permutations of the set ['c', 'a', 't'] of length 2
sage: p.list()
[['c', 'a'], ['c', 't'], ['a', 'c'], ['a', 't'], ['t', 'c'], ['t', 'a']]

sage: p = Permutations([1,1,2]); p
Permutations of the multi-set [1, 1, 2]
sage: p.list()
[[1, 1, 2], [1, 2, 1], [2, 1, 1]]

sage: p = Permutations([1,1,2], 2); p
Permutations of the multi-set [1, 1, 2] of length 2
sage: p.list()
[[1, 1], [1, 2], [2, 1]]

sage: p = Permutations(descents=([1], 4)); p
Standard permutations of 4 with descents [1]
sage: p.list()
[[1, 3, 2, 4], [1, 4, 2, 3], [2, 3, 1, 4], [2, 4, 1, 3], [3, 4, 1, 2]]

sage: p = Permutations(bruhat_smaller=[1,3,2,4]); p
Standard permutations that are less than or equal to [1, 3, 2, 4] in the Bruhat order
sage: p.list()
[[1, 2, 3, 4], [1, 3, 2, 4]]

sage: p = Permutations(bruhat_greater=[4,2,3,1]); p
Standard permutations that are greater than or equal to [4, 2, 3, 1] in the Bruhat order
sage: p.list()
[[4, 2, 3, 1], [4, 3, 2, 1]]

sage: p = Permutations(recoils_finer=[2,1]); p
Standard permutations whose recoils composition is finer than [2, 1]
sage: p.list()
[[1, 2, 3], [1, 3, 2], [3, 1, 2]]

sage: p = Permutations(recoils_fatter=[2,1]); p
Standard permutations whose recoils composition is fatter than [2, 1]
sage: p.list()
[[1, 3, 2], [3, 1, 2], [3, 2, 1]]

sage: p = Permutations(recoils=[2,1]); p
Standard permutations whose recoils composition is [2, 1]
sage: p.list()
[[1, 3, 2], [3, 1, 2]]
```

```
sage: p = Permutations(4, avoiding=[1,3,2]); p
Standard permutations of 4 avoiding [1, 3, 2]
sage: p.list()
[[4, 1, 2, 3],
 [4, 2, 1, 3],
 [4, 2, 3, 1],
 [4, 3, 1, 2],
 [4, 3, 2, 1],
 [3, 4, 1, 2],
 [3, 4, 2, 1],
 [2, 3, 4, 1],
 [3, 2, 4, 1],
 [1, 2, 3, 4],
 [2, 1, 3, 4],
 [2, 3, 1, 4],
 [3, 1, 2, 4],
 [3, 2, 1, 4]]

sage: p = Permutations(5, avoiding=[[3,4,1,2], [4,2,3,1]]); p
Standard permutations of 5 avoiding [[3, 4, 1, 2], [4, 2, 3, 1]]
sage: p.cardinality()
88
sage: p.random_element()
[5, 1, 2, 4, 3]
```

Element
alias of Permutation

global_options(**get_value*, ***set_value*)
Set the global options for elements of the permutation class. The defaults are for permutations to be displayed in list notation and the multiplication done from left to right (like in GAP) – that is, $(\pi\psi)(i) = \psi(\pi(i))$ for all i.

Note: These options have no effect on permutation group elements.

OPTIONS:

- `display` – (default: `list`) Specifies how the permutations should be printed
  - `cycle` – the permutations are displayed in cycle notation (i. e., as products of disjoint cycles)
  - `list` – the permutations are displayed in list notation (aka 1-line notation)
  - `reduced_expression` – alias for `reduced_word`
  - `reduced_word` – the permutations are displayed as reduced words
  - `singleton` – the permutations are displayed in cycle notation with singleton cycles shown as well
  - `word` – alias for `reduced_word`
- `generator_name` – (default: `s`) the letter used in latexing the reduced word
- `latex` – (default: `list`) Specifies how the permutations should be latexed
  - `cycle` – latex in cycle notation
  - `list` – latex as a list in one-line notation
  - `oneline` – alias for `list`

– `reduced_expression` – alias for `reduced_word`

– `reduced_word` – latex as reduced words

– `singleton` – latex in cycle notation with singleton cycles shown as well

– `twoline` – latex in two-line notation

– `word` – alias for `reduced_word`

- `latex_empty_str` – (default: `1`) The LaTeX representation of a reduced word when said word is empty
- `mult` – (default: `l2r`) The multiplication of permutations

  – `l2r` – left to right: $(p_1 \cdot p_2)(x) = p_2(p_1(x))$

  – `r2l` – right to left: $(p_1 \cdot p_2)(x) = p_1(p_2(x))$

EXAMPLES:

```
sage: p213 = Permutation([2,1,3])
sage: p312 = Permutation([3,1,2])
sage: Permutations.global_options(mult='l2r', display='list')
sage: Permutations.global_options['display']
'list'
sage: p213
[2, 1, 3]
sage: Permutations.global_options(display='cycle')
sage: p213
(1,2)
sage: Permutations.global_options(display='singleton')
sage: p213
(1,2)(3)
sage: Permutations.global_options(display='list')

sage: Permutations.global_options['mult']
'l2r'
sage: p213*p312
[1, 3, 2]
sage: Permutations.global_options(mult='r2l')
sage: p213*p312
[3, 2, 1]
sage: Permutations.global_options.reset()
```

See `GlobalOptions` for more features of these options.

class sage.combinat.permutation.**PermutationsNK**(*s*, *k*)

Bases: sage.combinat.permutation.Permutations_setk

This exists solely for unpickling `PermutationsNK` objects created with Sage <= 6.3.

class sage.combinat.permutation.**Permutations_mset**(*mset*)

Bases: sage.combinat.permutation.Permutations

Permutations of a multiset M.

A permutation of a multiset M is represented by a list that contains exactly the same elements as M (with the same multiplicities), but possibly in different order. If M is a proper set there are $|M|!$ such permutations. Otherwise, if the first element appears k_1 times, the second element appears k_2 times and so on, the number of permutations is $|M|!/(k_1!k_2!\ldots)$, which is sometimes called a multinomial coefficient.

EXAMPLES:

```
sage: mset = [1,1,2,2,2]
sage: from sage.combinat.permutation import Permutations_mset
sage: P = Permutations_mset(mset); P
Permutations of the multi-set [1, 1, 2, 2, 2]
sage: sorted(P)
[[1, 1, 2, 2, 2],
 [1, 2, 1, 2, 2],
 [1, 2, 2, 1, 2],
 [1, 2, 2, 2, 1],
 [2, 1, 1, 2, 2],
 [2, 1, 2, 1, 2],
 [2, 1, 2, 2, 1],
 [2, 2, 1, 1, 2],
 [2, 2, 1, 2, 1],
 [2, 2, 2, 1, 1]]
sage: MS = MatrixSpace(GF(2),2,2)
sage: A = MS([1,0,1,1])
sage: rows = A.rows()
sage: rows[0].set_immutable()
sage: rows[1].set_immutable()
sage: P = Permutations_mset(rows); P
Permutations of the multi-set [(1, 0), (1, 1)]
sage: sorted(P)
[[(1, 0), (1, 1)], [(1, 1), (1, 0)]]
```

class Element

Bases: sage.structure.list_clone.ClonableArray

A permutation of an arbitrary multiset.

check()

Verify that self is a valid permutation of the underlying multiset.

EXAMPLES:

```
sage: S = Permutations(['c','a','c'])
sage: elt = S(['c','c','a'])
sage: elt.check()
```

Permutations_mset.**cardinality**()

EXAMPLES:

```
sage: Permutations([1,2,2]).cardinality()
3
sage: Permutations([1,1,2,2,2]).cardinality()
10
```

class sage.combinat.permutation.**Permutations_msetk**(*mset*, *k*)

Bases: sage.combinat.permutation.Permutations_mset

Length-k partial permutations of a multiset.

A length-k partial permutation of a multiset M is represented by a list of length k whose entries are elements of M, appearing in the list with a multiplicity not higher than their respective multiplicity in M.

class sage.combinat.permutation.**Permutations_nk**(n, k)

Bases: sage.combinat.permutation.Permutations

Length-k partial permutations of $\{1, 2, \ldots, n\}$.

class Element

Bases: sage.structure.list_clone.ClonableArray

A length-k partial permutation of $[n]$.

check()
Verify that `self` is a valid length-k partial permutation of $[n]$.

EXAMPLES:

```
sage: S = Permutations(4, 2)
sage: elt = S([3, 1])
sage: elt.check()
```

Permutations_nk.**cardinality**()
EXAMPLES:

```
sage: Permutations(3,0).cardinality()
1
sage: Permutations(3,1).cardinality()
3
sage: Permutations(3,2).cardinality()
6
sage: Permutations(3,3).cardinality()
6
sage: Permutations(3,4).cardinality()
0
```

Permutations_nk.**random_element**()
EXAMPLES:

```
sage: Permutations(3,2).random_element()
[0, 1]
```

class sage.combinat.permutation.**Permutations_set**(*s*)
Bases: sage.combinat.permutation.Permutations

Permutations of an arbitrary given finite set.

Here, a "permutation of a finite set S" means a list of the elements of S in which every element of S occurs exactly once. This is not to be confused with bijections from S to S, which are also often called permutations in literature.

class Element
Bases: sage.structure.list_clone.ClonableArray

A permutation of an arbitrary set.

check()
Verify that `self` is a valid permutation of the underlying set.

EXAMPLES:

```
sage: S = Permutations(['c','a','t'])
sage: elt = S(['t','c','a'])
sage: elt.check()
```

Permutations_set.**cardinality**()
EXAMPLES:

```
sage: Permutations([1,2,3]).cardinality()
6
```

Permutations_set.**random_element**()
EXAMPLES:

```
sage: Permutations([1,2,3]).random_element()
[1, 2, 3]
```

class sage.combinat.permutation.**Permutations_setk**(*s*, *k*)
Bases: sage.combinat.permutation.Permutations_set

Length-k partial permutations of an arbitrary given finite set.

Here, a "length-k partial permutation of a finite set S" means a list of length k whose entries are pairwise distinct and all belong to S.

random_element()
EXAMPLES:

```
sage: Permutations([1,2,3],2).random_element()
[1, 2]
```

class sage.combinat.permutation.**StandardPermutations_all**
Bases: sage.combinat.permutation.Permutations

All standard permutations.

class sage.combinat.permutation.**StandardPermutations_avoiding_12**(*n*)
Bases: sage.combinat.permutation.StandardPermutations_avoiding_generic

TESTS:

```
sage: P = Permutations(3, avoiding=[1, 2])
sage: TestSuite(P).run()
```

cardinality()
Return the cardinality of `self`.

EXAMPLES:

```
sage: P = Permutations(3, avoiding=[1, 2])
sage: P.cardinality()
1
```

class sage.combinat.permutation.**StandardPermutations_avoiding_123**(*n*)
Bases: sage.combinat.permutation.StandardPermutations_avoiding_generic

TESTS:

```
sage: P = Permutations(3, avoiding=[2, 1, 3])
sage: TestSuite(P).run()
```

cardinality()
EXAMPLES:

```
sage: Permutations(5, avoiding=[1, 2, 3]).cardinality()
42
sage: len( Permutations(5, avoiding=[1, 2, 3]).list() )
42
```

class sage.combinat.permutation.**StandardPermutations_avoiding_132**(*n*)
Bases: sage.combinat.permutation.StandardPermutations_avoiding_generic

TESTS:

```
sage: P = Permutations(3, avoiding=[1, 3, 2])
sage: TestSuite(P).run()
```

cardinality()
EXAMPLES:
```
sage: Permutations(5, avoiding=[1, 3, 2]).cardinality()
42
sage: len( Permutations(5, avoiding=[1, 3, 2]).list() )
42
```

class sage.combinat.permutation.**StandardPermutations_avoiding_21**(*n*)
Bases: sage.combinat.permutation.StandardPermutations_avoiding_generic

TESTS:
```
sage: P = Permutations(3, avoiding=[2, 1])
sage: TestSuite(P).run()
```

cardinality()
Return the cardinality of `self`.

EXAMPLES:
```
sage: P = Permutations(3, avoiding=[2, 1])
sage: P.cardinality()
1
```

class sage.combinat.permutation.**StandardPermutations_avoiding_213**(*n*)
Bases: sage.combinat.permutation.StandardPermutations_avoiding_generic

TESTS:
```
sage: P = Permutations(3, avoiding=[2, 1, 3])
sage: TestSuite(P).run()
```

cardinality()
EXAMPLES:
```
sage: Permutations(5, avoiding=[2, 1, 3]).cardinality()
42
sage: len( Permutations(5, avoiding=[2, 1, 3]).list() )
42
```

class sage.combinat.permutation.**StandardPermutations_avoiding_231**(*n*)
Bases: sage.combinat.permutation.StandardPermutations_avoiding_generic

TESTS:
```
sage: P = Permutations(3, avoiding=[2, 3, 1])
sage: TestSuite(P).run()
```

cardinality()
EXAMPLES:
```
sage: Permutations(5, avoiding=[2, 3, 1]).cardinality()
42
sage: len( Permutations(5, avoiding=[2, 3, 1]).list() )
42
```

class sage.combinat.permutation.**StandardPermutations_avoiding_312**(*n*)
Bases: sage.combinat.permutation.StandardPermutations_avoiding_generic

TESTS:

```
sage: P = Permutations(3, avoiding=[3, 1, 2])
sage: TestSuite(P).run()
```

cardinality()

EXAMPLES:

```
sage: Permutations(5, avoiding=[3, 1, 2]).cardinality()
42
sage: len( Permutations(5, avoiding=[3, 1, 2]).list() )
42
```

class sage.combinat.permutation.**StandardPermutations_avoiding_321**(n)

Bases: sage.combinat.permutation.StandardPermutations_avoiding_generic

TESTS:

```
sage: P = Permutations(3, avoiding=[3, 2, 1])
sage: TestSuite(P).run()
```

cardinality()

EXAMPLES:

```
sage: Permutations(5, avoiding=[3, 2, 1]).cardinality()
42
sage: len( Permutations(5, avoiding=[3, 2, 1]).list() )
42
```

class sage.combinat.permutation.**StandardPermutations_avoiding_generic**(n, a)

Bases: sage.combinat.permutation.StandardPermutations_n_abstract

Generic class for subset of permutations avoiding a particular pattern.

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: P = Permutations(3, avoiding=[[2, 1, 3],[1,2,3]])
sage: P.cardinality()
4
```

class sage.combinat.permutation.**StandardPermutations_bruhat_greater**(p)

Bases: sage.combinat.permutation.Permutations

Permutations of $\{1, \ldots, n\}$ that are greater than or equal to a permutation p in the Bruhat order.

class sage.combinat.permutation.**StandardPermutations_bruhat_smaller**(p)

Bases: sage.combinat.permutation.Permutations

Permutations of $\{1, \ldots, n\}$ that are less than or equal to a permutation p in the Bruhat order.

class sage.combinat.permutation.**StandardPermutations_descents**(d, n)

Bases: sage.combinat.permutation.StandardPermutations_n_abstract

Permutations of $\{1, \ldots, n\}$ with a fixed set of descents.

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: P = Permutations(descents=([1,0,2],5))
sage: P.cardinality()
4
```

first()
Return the first permutation with descents d.

EXAMPLES:

```
sage: Permutations(descents=([1,0,4,8],12)).first()
[3, 2, 1, 4, 6, 5, 7, 8, 10, 9, 11, 12]
```

last()
Return the last permutation with descents d.

EXAMPLES:

```
sage: Permutations(descents=([1,0,4,8],12)).last()
[12, 11, 8, 9, 10, 4, 5, 6, 7, 1, 2, 3]
```

class sage.combinat.permutation.**StandardPermutations_n**(n)
Bases: sage.combinat.permutation.StandardPermutations_n_abstract

Permutations of the set $\{1, 2, \ldots, n\}$.

These are also called permutations of size n, or the elements of the n-th symmetric group.

Todo

Have a `reduced_word()` which works in both multiplication conventions.

---

class Element(*parent*, *l*, *check_input=True*)
Bases: sage.combinat.permutation.Permutation

Constructor. Checks that INPUT is not a mess, and calls `CombinatorialElement`. It should not, because `CombinatorialElement` is deprecated.

INPUT:

- `l` – a list of `int` variables
- `check_input` (boolean) – whether to check that input is correct. Slows the function down, but ensures that nothing bad happens.

  This is set to `True` by default.

TESTS:

```
sage: Permutation([1,2,3])
[1, 2, 3]
sage: Permutation([1,2,2,4])
Traceback (most recent call last):
...
ValueError: An element appears twice in the input. It should not.
sage: Permutation([1,2,4,-1])
Traceback (most recent call last):
...
ValueError: the elements must be strictly positive integers
sage: Permutation([1,2,4,5])
Traceback (most recent call last):
...
ValueError: The permutation has length 4 but its maximal element is
```

```
5. Some element may be repeated, or an element is missing, but there
is something wrong with its length.
```

has_left_descent(*i*, *mult=None*)

Check if `i` is a left descent of `self`.

A *left descent* of a permutation $\pi \in S_n$ means an index $i \in \{1, 2, \ldots, n-1\}$ such that $s_i \circ \pi$ has smaller length than π. Here, $\circ$ denotes the multiplication of S_n. How it is defined depends on the `mult` variable in `Permutations.global_options()`. If this variable is set to `'l2r'`, then the multiplication is defined by the rule $(\alpha\beta)(x) = \beta(\alpha(x))$ for $\alpha, \beta \in S_n$ and $x \in \{1, 2, \ldots, n\}$; then, a left descent of π is an index $i \in \{1, 2, \ldots, n-1\}$ satisfying $\pi(i) > \pi(i+1)$. If this variable is set to `'r2l'`, then the multiplication is defined by the rule $(\alpha\beta)(x) = \alpha(\beta(x))$ for $\alpha, \beta \in S_n$ and $x \in \{1, 2, \ldots, n\}$; then, a left descent of π is an index $i \in \{1, 2, \ldots, n-1\}$ satisfying $\pi^{-1}(i) > \pi^{-1}(i+1)$.

The optional parameter `mult` can be set to `'l2r'` or `'r2l'`; if so done, it is used instead of the `mult` variable in `Permutations.global_options()`. Anyone using this method in a non-interactive environment is encouraged to do so in order to have code behave reliably.

Warning: The methods `descents()` and `idescents()` behave differently than their Weyl group counterparts. In particular, the indexing is 0-based. This could lead to errors. Instead, construct the descent set as in the example.

Warning: The optional input `mult` might disappear once trac ticket #14881 is fixed.

EXAMPLES:

```
sage: P = Permutations(4)
sage: x = P([3, 2, 4, 1])
sage: x.descents() # known bug - different indices
[1, 3]
sage: [i for i in P.index_set() if x.has_left_descent(i)]
[1, 3]
sage: [i for i in P.index_set() if x.has_left_descent(i, mult="l2r")]
[1, 3]
sage: [i for i in P.index_set() if x.has_left_descent(i, mult="r2l")]
[1, 2]
```

has_right_descent(*i*, *mult=None*)

Check if `i` is a right descent of `self`.

A *right descent* of a permutation $\pi \in S_n$ means an index $i \in \{1, 2, \ldots, n-1\}$ such that $\pi \circ s_i$ has smaller length than π. Here, $\circ$ denotes the multiplication of S_n. How it is defined depends on the `mult` variable in `Permutations.global_options()`. If this variable is set to `'l2r'`, then the multiplication is defined by the rule $(\alpha\beta)(x) = \beta(\alpha(x))$ for $\alpha, \beta \in S_n$ and $x \in \{1, 2, \ldots, n\}$; then, a right descent of π is an index $i \in \{1, 2, \ldots, n-1\}$ satisfying $\pi^{-1}(i) > \pi^{-1}(i+1)$. If this variable is set to `'r2l'`, then the multiplication is defined by the rule $(\alpha\beta)(x) = \alpha(\beta(x))$ for $\alpha, \beta \in S_n$ and $x \in \{1, 2, \ldots, n\}$; then, a right descent of π is an index $i \in \{1, 2, \ldots, n-1\}$ satisfying $\pi(i) > \pi(i+1)$.

The optional parameter `mult` can be set to `'l2r'` or `'r2l'`; if so done, it is used instead of the `mult` variable in `Permutations.global_options()`. Anyone using this method in a non-interactive environment is encouraged to do so in order to have code behave reliably.

Warning: The methods `descents()` and `idescents()` behave differently than their Weyl group counterparts. In particular, the indexing is 0-based. This could lead to errors. Instead, construct the descent set as in the example.

Warning: The optional input `mult` might disappear once trac ticket #14881 is fixed.

EXAMPLES:

```
sage: P = Permutations(4)
sage: x = P([3, 2, 4, 1])
sage: (~x).descents() # known bug - different indices
[1, 2]
sage: [i for i in P.index_set() if x.has_right_descent(i)]
[1, 2]
sage: [i for i in P.index_set() if x.has_right_descent(i, mult="l2r")]
[1, 2]
sage: [i for i in P.index_set() if x.has_right_descent(i, mult="r2l")]
[1, 3]
```

inverse()

Return the inverse of `self`.

EXAMPLES:

```
sage: P = Permutations(4)
sage: w0 = P([4,3,2,1])
sage: w0.inverse() == w0
True
sage: w0.inverse().parent() is P
True
sage: P([3,2,4,1]).inverse()
[4, 2, 1, 3]
```

StandardPermutations_n.**algebra**(*base_ring*, *category=None*)

Return the symmetric group algebra associated to `self`.

INPUT:

- `base_ring` – a ring
- `category` – a category (default: the category of `self`)

EXAMPLES:

```
sage: P = Permutations(4)
sage: A = P.algebra(QQ); A
Symmetric group algebra of order 4 over Rational Field

sage: A.category()
Join of Category of coxeter group algebras over Rational Field
    and Category of finite group algebras over Rational Field
sage: A = P.algebra(QQ, category=Monoids())
sage: A.category()
Category of finite dimensional monoid algebras over Rational Field
```

StandardPermutations_n.**cardinality**()

Return the number of permutations of size n, which is $n!$.

EXAMPLES:

```
sage: Permutations(0).cardinality()
1
sage: Permutations(3).cardinality()
6
sage: Permutations(4).cardinality()
24
```

StandardPermutations_n.**cartan_type**()

Return the Cartan type of `self`.

The symmetric group S_n is a Coxeter group of type A_{n-1}.

EXAMPLES:

```
sage: A = SymmetricGroup([2,3,7]); A.cartan_type()
['A', 2]
sage: A = SymmetricGroup([]); A.cartan_type()
['A', 0]
```

StandardPermutations_n.**conjugacy_class**(*g*)

Return the conjugacy class of `g` in `self`.

INPUT:

- `g` – a partition or an element of `self`

EXAMPLES:

```
sage: G = Permutations(5)
sage: g = G([2,3,4,1,5])
sage: G.conjugacy_class(g)
Conjugacy class of cycle type [4, 1] in Standard permutations of 5
sage: G.conjugacy_class(Partition([2, 1, 1, 1]))
Conjugacy class of cycle type [2, 1, 1, 1] in Standard permutations of 5
```

StandardPermutations_n.**conjugacy_classes**()

Return a list of the conjugacy classes of `self`.

EXAMPLES:

```
sage: G = Permutations(4)
sage: G.conjugacy_classes()
[Conjugacy class of cycle type [1, 1, 1, 1] in Standard permutations of 4,
 Conjugacy class of cycle type [2, 1, 1] in Standard permutations of 4,
 Conjugacy class of cycle type [2, 2] in Standard permutations of 4,
 Conjugacy class of cycle type [3, 1] in Standard permutations of 4,
 Conjugacy class of cycle type [4] in Standard permutations of 4]
```

StandardPermutations_n.**conjugacy_classes_iterator**()

Iterate over the conjugacy classes of `self`.

EXAMPLES:

```
sage: G = Permutations(4)
sage: list(G.conjugacy_classes_iterator()) == G.conjugacy_classes()
True
```

StandardPermutations_n.**conjugacy_classes_representatives**()

Return a complete list of representatives of conjugacy classes in `self`.

Let S_n be the symmetric group on n letters. The conjugacy classes are indexed by partitions λ of n. The ordering of the conjugacy classes is reverse lexicographic order of the partitions.

EXAMPLES:

```
sage: G = Permutations(5)
sage: G.conjugacy_classes_representatives()
[[1, 2, 3, 4, 5],
 [2, 1, 3, 4, 5],
 [2, 1, 4, 3, 5],
 [2, 3, 1, 4, 5],
 [2, 3, 1, 5, 4],
 [2, 3, 4, 1, 5],
 [2, 3, 4, 5, 1]]
```

TESTS:

Check some border cases:

```
sage: S = Permutations(0)
sage: S.conjugacy_classes_representatives()
[[]]
sage: S = Permutations(1)
sage: S.conjugacy_classes_representatives()
[[1]]
```

StandardPermutations_n.**element_in_conjugacy_classes**(*nu*)

Return a permutation with cycle type `nu`.

If the size of `nu` is smaller than the size of permutations in `self`, then some fixed points are added.

EXAMPLES:

```
sage: PP = Permutations(5)
sage: PP.element_in_conjugacy_classes([2,2])
[2, 1, 4, 3, 5]
```

StandardPermutations_n.**identity**()

Return the identity permutation of size n.

EXAMPLES:

```
sage: Permutations(4).identity()
[1, 2, 3, 4]
sage: Permutations(0).identity()
[]
```

StandardPermutations_n.**index_set**()

Return the index set for the descents of the symmetric group `self`.

This is $\{1, 2, \ldots, n-1\}$, where `self` is S_n.

EXAMPLES:

```
sage: P = Permutations(8)
sage: P.index_set()
(1, 2, 3, 4, 5, 6, 7)
```

StandardPermutations_n.**one**()

Return the identity permutation of size n.

EXAMPLES:

```
sage: Permutations(4).identity()
[1, 2, 3, 4]
```

```
sage: Permutations(0).identity()
[]
```

StandardPermutations_n.**random_element**()

EXAMPLES:

```
sage: Permutations(4).random_element()
[1, 2, 4, 3]
```

StandardPermutations_n.**rank**(*p*)

EXAMPLES:

```
sage: SP3 = Permutations(3)
sage: map(SP3.rank, SP3)
[0, 1, 2, 3, 4, 5]
sage: SP0 = Permutations(0)
sage: map(SP0.rank, SP0)
[0]
```

StandardPermutations_n.**simple_reflection**(*i*)

For i in the index set of `self` (that is, for i in $\{1, 2, \ldots, n-1\}$, where `self` is S_n), this returns the elementary transposition $s_i = (i, i+1)$.

EXAMPLES:

```
sage: P = Permutations(4)
sage: P.simple_reflection(2)
[1, 3, 2, 4]
sage: P.simple_reflections()
Finite family {1: [2, 1, 3, 4], 2: [1, 3, 2, 4], 3: [1, 2, 4, 3]}
```

StandardPermutations_n.**unrank**(*r*)

EXAMPLES:

```
sage: SP3 = Permutations(3)
sage: l = map(SP3.unrank, range(6))
sage: l == SP3.list()
True
sage: SP0 = Permutations(0)
sage: l = map(SP0.unrank, range(1))
sage: l == SP0.list()
True
```

class sage.combinat.permutation.**StandardPermutations_n_abstract**(*n*, *category=None*)

Bases: sage.combinat.permutation.Permutations

Abstract base class for subsets of permutations of the set $\{1, 2, \ldots, n\}$.

Warning: Anyone inheriting from this class should override the __contains__ method.

class sage.combinat.permutation.**StandardPermutations_recoils**(*recoils*)

Bases: sage.combinat.permutation.Permutations

Permutations of $\{1, \ldots, n\}$ with a fixed recoils composition.

class sage.combinat.permutation.**StandardPermutations_recoilsfatter**(*recoils*)

Bases: sage.combinat.permutation.Permutations

TESTS:

```
sage: P = Permutations(recoils_fatter=[2,2])
sage: TestSuite(P).run()
```

class sage.combinat.permutation.**StandardPermutations_recoilsfiner**(*recoils*)
Bases: sage.combinat.permutation.Permutations

TESTS:

```
sage: P = Permutations(recoils_finer=[2,2])
sage: TestSuite(P).run()
```

sage.combinat.permutation.**bistochastic_as_sum_of_permutations**(*M*, *check=True*)
Return the positive sum of permutations corresponding to the bistochastic matrix `M`.

A stochastic matrix is a matrix with nonnegative real entries such that the sum of the elements of any row is equal to 1. A bistochastic matrix is a stochastic matrix whose transpose matrix is also stochastic (there are conditions both on the rows and on the columns).

According to the Birkhoff-von Neumann Theorem, any bistochastic matrix can be written as a convex combination of permutation matrices, which also means that the polytope of bistochastic matrices is integer.

As a non-bistochastic matrix can obviously not be written as a convex combination of permutations, this theorem is an equivalence.

This function, given a bistochastic matrix, returns the corresponding decomposition.

INPUT:

- `M` – A bistochastic matrix
- `check` (boolean) – set to `True` (default) to check that the matrix is indeed bistochastic

OUTPUT:

- An element of `CombinatorialFreeModule`, which is a free F-module (where F is the ground ring of the given matrix) whose basis is indexed by the permutations.

Note:

- In this function, we just assume 1 to be any constant : for us a matrix M is bistochastic if there exists $c > 0$ such that M/c is bistochastic.
- You can obtain a sequence of pairs `(permutation,coeff)`, where `permutation` is a Sage `Permutation` instance, and `coeff` its corresponding coefficient from the result of this function by applying the `list` function.
- If you are interested in the matrix corresponding to a `Permutation` you will be glad to learn about the `Permutation.to_matrix()` method.
- The base ring of the matrix can be anything that can be coerced to `RR`.

See also:

- `as_sum_of_permutations()` to use this method through the `Matrix` class.

EXAMPLES:

We create a bistochastic matrix from a convex sum of permutations, then try to deduce the decomposition from the matrix:

```
sage: from sage.combinat.permutation import bistochastic_as_sum_of_permutations
sage: L = []
sage: L.append((9,Permutation([4, 1, 3, 5, 2])))
sage: L.append((6,Permutation([5, 3, 4, 1, 2])))
sage: L.append((3,Permutation([3, 1, 4, 2, 5])))
sage: L.append((2,Permutation([1, 4, 2, 3, 5])))
sage: M = sum([c * p.to_matrix() for (c,p) in L])
sage: decomp = bistochastic_as_sum_of_permutations(M)
sage: print decomp
2*B[[1, 4, 2, 3, 5]] + 3*B[[3, 1, 4, 2, 5]] + 9*B[[4, 1, 3, 5, 2]] + 6*B[[5, 3, 4, 1, 2]]
```

An exception is raised when the matrix is not positive and bistochastic:

```
sage: M = Matrix([[2,3],[2,2]])
sage: decomp = bistochastic_as_sum_of_permutations(M)
Traceback (most recent call last):
...
ValueError: The matrix is not bistochastic

sage: bistochastic_as_sum_of_permutations(Matrix(GF(7), 2, [2,1,1,2]))
Traceback (most recent call last):
...
ValueError: The base ring of the matrix must have a coercion map to RR

sage: bistochastic_as_sum_of_permutations(Matrix(ZZ, 2, [2,-1,-1,2]))
Traceback (most recent call last):
...
ValueError: The matrix should have nonnegative entries
```

sage.combinat.permutation.**bruhat_lequal**(*p1*, *p2*)

Return `True` if `p1` is less than `p2` in the Bruhat order.

Algorithm from mupad-combinat.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.bruhat_lequal([2,4,3,1],[3,4,2,1])
True
```

sage.combinat.permutation.**descents_composition_first**(*dc*)

Compute the smallest element of a descent class having a descent composition `dc`.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.descents_composition_first([1,1,3,4,3])
[3, 2, 1, 4, 6, 5, 7, 8, 10, 9, 11, 12]
```

sage.combinat.permutation.**descents_composition_last**(*dc*)

Return the largest element of a descent class having a descent composition `dc`.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.descents_composition_last([1,1,3,4,3])
[12, 11, 8, 9, 10, 4, 5, 6, 7, 1, 2, 3]
```

sage.combinat.permutation.**descents_composition_list**(*dc*)

Return a list of all the permutations that have the descent composition `dc`.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.descents_composition_list([1,2,2])
[[2, 1, 4, 3, 5],
 [2, 1, 5, 3, 4],
 [3, 1, 4, 2, 5],
 [3, 1, 5, 2, 4],
 [4, 1, 3, 2, 5],
 [5, 1, 3, 2, 4],
 [4, 1, 5, 2, 3],
 [5, 1, 4, 2, 3],
 [3, 2, 4, 1, 5],
 [3, 2, 5, 1, 4],
 [4, 2, 3, 1, 5],
 [5, 2, 3, 1, 4],
 [4, 2, 5, 1, 3],
 [5, 2, 4, 1, 3],
 [4, 3, 5, 1, 2],
 [5, 3, 4, 1, 2]]
```

sage.combinat.permutation.**from_cycles**(*n*, *cycles*, *parent=None*)

Return the permutation in the n-th symmetric group whose decomposition into disjoint cycles is `cycles`.

This function checks that its input is correct (i.e. that the cycles are disjoint and their elements integers among $1...n$). It raises an exception otherwise.

Warning: It assumes that the elements are of `int` type.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.from_cycles(4, [[1,2]])
[2, 1, 3, 4]
sage: permutation.from_cycles(4, [[1,2,4]])
[2, 4, 3, 1]
sage: permutation.from_cycles(10, [[3,1],[4,5],[6,8,9]])
[3, 2, 1, 5, 4, 8, 7, 9, 6, 10]
sage: permutation.from_cycles(10, ((2, 5), (6, 1, 3)))
[3, 5, 6, 4, 2, 1, 7, 8, 9, 10]
sage: permutation.from_cycles(4, [])
[1, 2, 3, 4]
sage: permutation.from_cycles(4, [[]])
[1, 2, 3, 4]
sage: permutation.from_cycles(0, [])
[]
```

Bad input (see trac ticket #13742):

```
sage: Permutation("(-12,2)(3,4)")
Traceback (most recent call last):
...
ValueError: All elements should be strictly positive integers, and I just found a non-positive c
sage: Permutation("(1,2)(2,4)")
Traceback (most recent call last):
...
ValueError: An element appears twice. It should not.
sage: permutation.from_cycles(4, [[1,18]])
Traceback (most recent call last):
```

```
...
ValueError: You claimed that this was a permutation on 1...4 but it contains 18
```

sage.combinat.permutation.**from_inversion_vector**(*iv*, *parent=None*)

Return the permutation corresponding to inversion vector `iv`.

See *sage.combinat.permutation.Permutation.to_inversion_vector* for a definition of the inversion vector of a permutation.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.from_inversion_vector([3,1,0,0,0])
[3, 2, 4, 1, 5]
sage: permutation.from_inversion_vector([2,3,6,4,0,2,2,1,0])
[5, 9, 1, 8, 2, 6, 4, 7, 3]
sage: permutation.from_inversion_vector([0])
[1]
sage: permutation.from_inversion_vector([])
[]
```

sage.combinat.permutation.**from_lehmer_code**(*lehmer*, *parent=None*)

Return the permutation with Lehmer code `lehmer`.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: lc = Permutation([2,1,5,4,3]).to_lehmer_code(); lc
[1, 0, 2, 1, 0]
sage: permutation.from_lehmer_code(lc)
[2, 1, 5, 4, 3]
```

sage.combinat.permutation.**from_major_code**(*mc*, *final_descent=False*)

Return the permutation with major code `mc`.

The major code of a permutation is defined in `to_major_code()`.

Warning: This function creates illegal permutations (i.e. `Permutation([9])`, and this is dangerous as the `Permutation()` class is only designed to handle permutations on $1...n$. This will have to be changed when Sage permutations will be able to handle anything, but right now this should be fixed. Be careful with the results.

Warning: If `mc` is not a major index of a permutation, then the return value of this method can be anything. Garbage in, garbage out!

REFERENCES:

•Skandera, M. *An Eulerian Partner for Inversions*. Sem. Lothar. Combin. 46 (2001) B46d.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.from_major_code([5, 0, 1, 0, 1, 2, 0, 1, 0])
[9, 3, 5, 7, 2, 1, 4, 6, 8]
sage: permutation.from_major_code([8, 3, 3, 1, 4, 0, 1, 0, 0])
[2, 8, 4, 3, 6, 7, 9, 5, 1]
sage: Permutation([2,1,6,4,7,3,5]).to_major_code()
[3, 2, 0, 2, 2, 0, 0]
sage: permutation.from_major_code([3, 2, 0, 2, 2, 0, 0])
[2, 1, 6, 4, 7, 3, 5]
```

TESTS:

```
sage: permutation.from_major_code([])
[]

sage: all( permutation.from_major_code(p.to_major_code()) == p
....:      for p in Permutations(5) )
True
```

sage.combinat.permutation.**from_permutation_group_element**(*pge*, *parent=None*)

Return a Permutation given a PermutationGroupElement pge.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: pge = PermutationGroupElement([(1,2),(3,4)])
sage: permutation.from_permutation_group_element(pge)
[2, 1, 4, 3]
```

sage.combinat.permutation.**from_rank**(*n*, *rank*)

Return the permutation of the set $\{1, ..., n\}$ with lexicographic rank rank. This is the permutation whose Lehmer code is the factoradic representation of rank. In particular, the permutation with rank 0 is the identity permutation.

The permutation is computed without iterating through all of the permutations with lower rank. This makes it efficient for large permutations.

Note: The variable rank is not checked for being in the interval from 0 to $n! - 1$. When outside this interval, it acts as its residue modulo $n!$.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: Permutation([3, 6, 5, 4, 2, 1]).rank()
359
sage: [permutation.from_rank(3, i) for i in range(6)]
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
sage: Permutations(6)[10]
[1, 2, 4, 6, 3, 5]
sage: permutation.from_rank(6,10)
[1, 2, 4, 6, 3, 5]
```

sage.combinat.permutation.**from_reduced_word**(*rw*, *parent=None*)

Return the permutation corresponding to the reduced word rw.

See reduced_words() for a definition of reduced words and the convention on the order of multiplication used.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.from_reduced_word([3,2,3,1,2,3,1])
[3, 4, 2, 1]
sage: permutation.from_reduced_word([])
[]
```

sage.combinat.permutation.**permutohedron_lequal**(*p1*, *p2*, *side='right'*)

Return True if p1 is less than or equal to p2 in the permutohedron order.

By default, the computations are done in the right permutohedron. If you pass the option `side='left'`, then they will be done in the left permutohedron.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.permutohedron_lequal(Permutation([3,2,1,4]),Permutation([4,2,1,3]))
False
sage: permutation.permutohedron_lequal(Permutation([3,2,1,4]),Permutation([4,2,1,3]), side='left
True
```

sage.combinat.permutation.**to_standard**(*p*)

Return a standard permutation corresponding to the list `p`.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.to_standard([4,2,7])
[2, 1, 3]
sage: permutation.to_standard([1,2,3])
[1, 2, 3]
sage: permutation.to_standard([])
[]
```

TESTS:

Does not mutate the list:

```
sage: a = [1,2,4]
sage: permutation.to_standard(a)
[1, 2, 3]
sage: a
[1, 2, 4]
```

5.1.148 Permutations (Cython file)

This is a nearly-straightforward implementation of what Knuth calls "Algorithm P" in TAOCP 7.2.1.2. The intent is to be able to enumerate permutation by "plain changes", or multiplication by adjacent transpositions, as a generator. This is useful when a class of objects is inherently enumerated by permutations, but it is faster to swap items in a permutation than construct the next object directly from the next permutation in a list. The backtracking algorithm in sage/graphs/genus.pyx is an example of this.

The lowest level is implemented as a struct with auxilliary methods. This is because Cython does not allow pointers to class instances, so a list of these objects is inherently slower than a list of structs. The author prefers ugly code to slow code.

For those willing to sacrifice a (very small) amount of speed, we provide a class that wraps our struct.

sage.combinat.permutation_cython.**left_action_product**(*S*, *lp*)

Return the permutation obtained by composing a permutation `S` with a permutation `lp` in such an order that `lp` is applied first and `S` is applied afterwards.

See also:

`sage.combinat.permutation.Permutation.left_action_product()`

EXAMPLES:

```
sage: p = [2,1,3,4]
sage: q = [3,1,2]
sage: from sage.combinat.permutation_cython import left_action_product
```

```
sage: left_action_product(p, q)
[3, 2, 1, 4]
sage: left_action_product(q, p)
[1, 3, 2, 4]
sage: q
[3, 1, 2]
```

sage.combinat.permutation_cython.**left_action_same_n**(*S*, *lp*)

Return the permutation obtained by composing a permutation `S` with a permutation `lp` in such an order that `lp` is applied first and `S` is applied afterwards and `S` and `lp` are of the same length.

See also:

`sage.combinat.permutation.Permutation.left_action_product()`

EXAMPLES:

```
sage: p = [2,1,3]
sage: q = [3,1,2]
sage: from sage.combinat.permutation_cython import left_action_same_n
sage: left_action_same_n(p, q)
[3, 2, 1]
sage: left_action_same_n(q, p)
[1, 3, 2]
```

sage.combinat.permutation_cython.**permutation_iterator_transposition_list**(*n*)

Returns a list of transposition indices to enumerate the permutations on n letters by adjacent transpositions. Assumes zero-based lists. We artificially limit the argument to $n < 12$ to avoid overflowing 32-bit pointers. While the algorithm works for larger n, the user is encouraged to avoid filling anything more than 4GB of memory with the output of this function.

EXAMPLES:

```
sage: import sage.combinat.permutation_cython
sage: from sage.combinat.permutation_cython import permutation_iterator_transposition_list
sage: permutation_iterator_transposition_list(4)
[2, 1, 0, 2, 0, 1, 2, 0, 2, 1, 0, 2, 0, 1, 2, 0, 2, 1, 0, 2, 0, 1, 2]
sage: permutation_iterator_transposition_list(200)
Traceback (most recent call last):
...
ValueError: Cowardly refusing to enumerate the permutations on more than 12 letters.
sage: permutation_iterator_transposition_list(1)
[]

sage: # Generate the permutations of [1,2,3,4] fixing 4.
sage: Q = [1,2,3,4]
sage: L = [copy(Q)]
sage: for t in permutation_iterator_transposition_list(3):
...       Q[t], Q[t+1] = Q[t+1], Q[t]
...       L.append(copy(Q))
sage: print L
[[1, 2, 3, 4], [1, 3, 2, 4], [3, 1, 2, 4], [3, 2, 1, 4], [2, 3, 1, 4], [2, 1, 3, 4]]
```

sage.combinat.permutation_cython.**right_action_product**(*S*, *rp*)

Return the permutation obtained by composing a permutation `S` with a permutation `rp` in such an order that `S` is applied first and `rp` is applied afterwards.

See also:

`sage.combinat.permutation.Permutation.right_action_product()`

EXAMPLES:

```
sage: p = [2,1,3,4]
sage: q = [3,1,2]
sage: from sage.combinat.permutation_cython import right_action_product
sage: right_action_product(p, q)
[1, 3, 2, 4]
sage: right_action_product(q, p)
[3, 2, 1, 4]
sage: q
[3, 1, 2]
```

sage.combinat.permutation_cython.**right_action_same_n**(*S*, *rp*)

Return the permutation obtained by composing a permutation `S` with a permutation `rp` in such an order that `S` is applied first and `rp` is applied afterwards and `S` and `rp` are of the same length.

See also:

`sage.combinat.permutation.Permutation.right_action_product()`

EXAMPLES:

```
sage: p = [2,1,3]
sage: q = [3,1,2]
sage: from sage.combinat.permutation_cython import right_action_same_n
sage: right_action_same_n(p, q)
[1, 3, 2]
sage: right_action_same_n(q, p)
[3, 2, 1]
```

5.1.149 Posets

Common posets can be accessed through `posets.<tab>` and are listed in the posets catalog:

- *Catalog of posets*

Poset-related classes:

- *Finite posets*
- *Finite lattices and semilattices*
- *Linear Extensions of Posets*
- *Incidence Algebras*
- *Cartesian products of Posets*
- *Möbius Algebras*
- *Generalized Tamari lattices*
- *Tamari Interval-posets*
- *Shard intersection order*

If you are looking for Poset-related *categories*, see `Posets`, `FinitePosets`, `LatticePosets` and `FiniteLatticePosets`.

5.1.150 Poset features that are imported by default in the interpreter namespace

5.1.151 Cartesian products of Posets

AUTHORS:

- Daniel Krenn (2015)

class sage.combinat.posets.cartesian_product.**CartesianProductPoset**(*sets*, *category*, *order=None*, ***kwargs*)

Bases: sage.sets.cartesian_product.CartesianProduct

A class implementing Cartesian products of posets (and elements thereof). Compared to CartesianProduct you are able to specify an order for comparison of the elements.

INPUT:

- sets – a tuple of parents.
- category – a subcategory of Sets().CartesianProducts() & Posets().
- order – a string or function specifying an order less or equal. It can be one of the following:
  - 'native' – elements are ordered by their native ordering, i.e., the order the wrapped elements (tuples) provide.
  - 'lex' – elements are ordered lexicographically.
  - 'product' – an element is less or equal to another element, if less or equal is true for all its components (Cartesian projections).
  - A function which performs the comparison $\leq$. It takes two input arguments and outputs a boolean.

Other keyword arguments (kwargs) are passed to the constructor of CartesianProduct.

EXAMPLES:

```
sage: P = Poset((srange(3), lambda left, right: left <= right))
sage: Cl = cartesian_product((P, P), order='lex')
sage: Cl((1, 1)) <= Cl((2, 0))
True
sage: Cp = cartesian_product((P, P), order='product')
sage: Cp((1, 1)) <= Cp((2, 0))
False
sage: def le_sum(left, right):
....:     return (sum(left) < sum(right) or
....:             sum(left) == sum(right) and left[0] <= right[0])
sage: Cs = cartesian_product((P, P), order=le_sum)
sage: Cs((1, 1)) <= Cs((2, 0))
True
```

TESTS:

```
sage: Cl.category()
Join of Category of finite posets and
Category of Cartesian products of finite enumerated sets
sage: TestSuite(Cl).run()
sage: Cp.category()
Join of Category of finite posets and
Category of Cartesian products of finite enumerated sets
sage: TestSuite(Cp).run()
```

class Element

Bases: sage.sets.cartesian_product.CartesianProduct.Element

CartesianProductPoset.**le**(*left*, *right*)

Test whether left is less than or equal to right.

INPUT:

- left – an element.
- right – an element.

OUTPUT:

A boolean.

Note: This method uses the order defined on creation of this Cartesian product. See CartesianProductPoset.

EXAMPLES:

```
sage: P = Posets.ChainPoset(10)
sage: def le_sum(left, right):
....:     return (sum(left) < sum(right) or
....:             sum(left) == sum(right) and left[0] <= right[0])
sage: C = cartesian_product((P, P), order=le_sum)
sage: C.le(C((1, 6)), C((6, 1)))
True
sage: C.le(C((6, 1)), C((1, 6)))
False
sage: C.le(C((1, 6)), C((6, 6)))
True
sage: C.le(C((6, 6)), C((1, 6)))
False
```

CartesianProductPoset.**le_lex**(*left*, *right*)

Test whether left is lexicographically smaller or equal to right.

INPUT:

- left – an element.
- right – an element.

OUTPUT:

A boolean.

EXAMPLES:

```
sage: P = Poset((srange(2), lambda left, right: left <= right))
sage: Q = cartesian_product((P, P), order='lex')
sage: T = [Q((0, 0)), Q((1, 1)), Q((0, 1)), Q((1, 0))]
sage: for a in T:
....:     for b in T:
....:         assert(Q.le(a, b) == (a <= b))
....:         print '%s <= %s = %s' % (a, b, a <= b)
(0, 0) <= (0, 0) = True
(0, 0) <= (1, 1) = True
(0, 0) <= (0, 1) = True
(0, 0) <= (1, 0) = True
(1, 1) <= (0, 0) = False
(1, 1) <= (1, 1) = True
```

```
(1, 1) <= (0, 1) = False
(1, 1) <= (1, 0) = False
(0, 1) <= (0, 0) = False
(0, 1) <= (1, 1) = True
(0, 1) <= (0, 1) = True
(0, 1) <= (1, 0) = True
(1, 0) <= (0, 0) = False
(1, 0) <= (1, 1) = True
(1, 0) <= (0, 1) = False
(1, 0) <= (1, 0) = True
```

TESTS:

Check that trac ticket #19999 is resolved:

```
sage: P = Poset((srange(2), lambda left, right: left <= right))
sage: Q = cartesian_product((P, P), order='product')
sage: R = cartesian_product((Q, P), order='lex')
sage: R(((1, 0), 0)) <= R(((0, 1), 0))
False
sage: R(((0, 1), 0)) <= R(((1, 0), 0))
False
```

CartesianProductPoset.**le_native**(*left*, *right*)

Test whether `left` is smaller or equal to `right` in the order provided by the elements themselves.

INPUT:

- `left` – an element.
- `right` – an element.

OUTPUT:

A boolean.

EXAMPLES:

```
sage: P = Poset((srange(2), lambda left, right: left <= right))
sage: Q = cartesian_product((P, P), order='native')
sage: T = [Q((0, 0)), Q((1, 1)), Q((0, 1)), Q((1, 0))]
sage: for a in T:
....:     for b in T:
....:         assert(Q.le(a, b) == (a <= b))
....:         print '%s <= %s = %s' % (a, b, a <= b)
(0, 0) <= (0, 0) = True
(0, 0) <= (1, 1) = True
(0, 0) <= (0, 1) = True
(0, 0) <= (1, 0) = True
(1, 1) <= (0, 0) = False
(1, 1) <= (1, 1) = True
(1, 1) <= (0, 1) = False
(1, 1) <= (1, 0) = False
(0, 1) <= (0, 0) = False
(0, 1) <= (1, 1) = True
(0, 1) <= (0, 1) = True
(0, 1) <= (1, 0) = True
(1, 0) <= (0, 0) = False
(1, 0) <= (1, 1) = True
(1, 0) <= (0, 1) = False
(1, 0) <= (1, 0) = True
```

CartesianProductPoset.**le_product**(*left*, *right*)

Test whether `left` is component-wise smaller or equal to `right`.

INPUT:

- `left` – an element.
- `right` – an element.

OUTPUT:

A boolean.

The comparison is `True` if the result of the comparision in each component is `True`.

EXAMPLES:

```
sage: P = Poset((srange(2), lambda left, right: left <= right))
sage: Q = cartesian_product((P, P), order='product')
sage: T = [Q((0, 0)), Q((1, 1)), Q((0, 1)), Q((1, 0))]
sage: for a in T:
....:     for b in T:
....:         assert(Q.le(a, b) == (a <= b))
....:         print '%s <= %s = %s' % (a, b, a <= b)
(0, 0) <= (0, 0) = True
(0, 0) <= (1, 1) = True
(0, 0) <= (0, 1) = True
(0, 0) <= (1, 0) = True
(1, 1) <= (0, 0) = False
(1, 1) <= (1, 1) = True
(1, 1) <= (0, 1) = False
(1, 1) <= (1, 0) = False
(0, 1) <= (0, 0) = False
(0, 1) <= (1, 1) = True
(0, 1) <= (0, 1) = True
(0, 1) <= (1, 0) = False
(1, 0) <= (0, 0) = False
(1, 0) <= (1, 1) = True
(1, 0) <= (0, 1) = False
(1, 0) <= (1, 0) = True
```

5.1.152 Elements of posets, lattices, semilattices, etc.

class sage.combinat.posets.elements.**JoinSemilatticeElement**(*poset*, *element*, *vertex*)

Bases: sage.combinat.posets.elements.PosetElement

Establishes the parent-child relationship between `poset` and `element`, where `element` is associated to the vertex `vertex` of the Hasse diagram of the poset.

INPUT:

- `poset` - a poset object
- `element` - any object
- `vertex` - a vertex of the Hasse diagram of the poset

TESTS:

```
sage: from sage.combinat.posets.elements import PosetElement
sage: P = Poset([[1,2],[4],[3],[4],[]], facade = False)
sage: e = P(0)
```

```
sage: e.parent() is P
True
sage: TestSuite(e).run()
```

class sage.combinat.posets.elements.**LatticePosetElement**(*poset*, *element*, *vertex*)
Bases: sage.combinat.posets.elements.MeetSemilatticeElement, sage.combinat.posets.elements.JoinSemilatticeElement

Establishes the parent-child relationship between `poset` and `element`, where `element` is associated to the vertex `vertex` of the Hasse diagram of the poset.

INPUT:

- `poset` - a poset object
- `element` - any object
- `vertex` - a vertex of the Hasse diagram of the poset

TESTS:

```
sage: from sage.combinat.posets.elements import PosetElement
sage: P = Poset([[1,2],[4],[3],[4],[]], facade = False)
sage: e = P(0)
sage: e.parent() is P
True
sage: TestSuite(e).run()
```

class sage.combinat.posets.elements.**MeetSemilatticeElement**(*poset*, *element*, *vertex*)
Bases: sage.combinat.posets.elements.PosetElement

Establishes the parent-child relationship between `poset` and `element`, where `element` is associated to the vertex `vertex` of the Hasse diagram of the poset.

INPUT:

- `poset` - a poset object
- `element` - any object
- `vertex` - a vertex of the Hasse diagram of the poset

TESTS:

```
sage: from sage.combinat.posets.elements import PosetElement
sage: P = Poset([[1,2],[4],[3],[4],[]], facade = False)
sage: e = P(0)
sage: e.parent() is P
True
sage: TestSuite(e).run()
```

class sage.combinat.posets.elements.**PosetElement**(*poset*, *element*, *vertex*)
Bases: sage.structure.element.Element

Establishes the parent-child relationship between `poset` and `element`, where `element` is associated to the vertex `vertex` of the Hasse diagram of the poset.

INPUT:

- `poset` - a poset object
- `element` - any object
- `vertex` - a vertex of the Hasse diagram of the poset

TESTS:

```
sage: from sage.combinat.posets.elements import PosetElement
sage: P = Poset([[1,2],[4],[3],[4],[]], facade = False)
sage: e = P(0)
sage: e.parent() is P
True
sage: TestSuite(e).run()
```

5.1.153 Hasse diagrams of posets

antichains()	Returns all antichains of self, organized as a prefix tree
antichains_iterator()	Return an iterator over the antichains of the poset.
are_comparable()	Returns whether i and j are comparable in the poset
are_incomparable()	Returns whether i and j are incomparable in the poset
bottom()	Returns the bottom element of the poset, if it exists.
cardinality()	Returns the number of elements in the poset.
chains()	Return all chains of self, organized as a prefix tree.
complements()	Deprecated.
cover_relations()	Return the list of cover relations.
cover_relations_iterator()	Iterate over cover relations.
covers()	Returns True if y covers x and False otherwise.
deprecated_function_alias()	Create an aliased version of a function or a method which raise a deprecation warning mes
dual()	Returns a poset that is dual to the given poset.
frattini_sublattice()	Return the list of elements of the Frattini sublattice of the lattice.
has_bottom()	Returns True if the poset has a unique minimal element.
has_top()	Returns True if the poset contains a unique maximal element, and False otherwise.
interval()	Return a list of the elements z of self such that $x \leq z \leq y$. The order is that induced by
is_bounded()	Returns True if the poset contains a unique maximal element and a unique minimal elemer
is_chain()	Returns True if the poset is totally ordered, and False otherwise.
is_complemented_lattice()	Return True if self is the Hasse diagram of a complemented lattice, and False otherw
is_distributive_lattice()	Returns True if self is the Hasse diagram of a distributive lattice, and False otherwise
is_gequal()	Returns True if x is greater than or equal to y, and False otherwise.
is_graded()	Deprecated, has conflicting definition of "graded" vs. "ranked" with posets.
is_greater_than()	Returns True if x is greater than but not equal to y, and False otherwise.
is_join_semilattice()	Returns True if self has a join operation, and False otherwise.
is_lequal()	Returns True if i is less than or equal to j in the poset, and False otherwise.
is_less_than()	Returns True if x is less than or equal to y in the poset, and False otherwise.
is_linear_extension()	Test if an ordering is a linear extension.
is_meet_semilattice()	Returns True if self has a meet operation, and False otherwise.
is_ranked()	Returns True if the poset is ranked, and False otherwise.
join_matrix()	Returns the matrix of joins of self. The (x,y)-entry of this matrix is the join of x and y
lequal_matrix()	Returns the matrix whose (i,j) entry is 1 if i is less than j in the poset, and 0 otherwise
linear_extension()	Return a linear extension
linear_extensions()	Return all linear extensions
lower_covers_iterator()	Returns the list of elements that are covered by element.
maximal_elements()	Returns a list of the maximal elements of the poset.
maximal_sublattices()	Return maximal sublattices of the lattice.
meet_matrix()	Returns the matrix of meets of self. The (x,y)-entry of this matrix is the meet of x and
minimal_elements()	Returns a list of the minimal elements of the poset.
moebius_function()	Returns the value of the Möbius function of the poset on the elements i and j.
moebius_function_matrix()	Returns the matrix of the Möbius function of this poset

Table 5.2 – continued from previous page

open_interval()	Return a list of the elements z of self such that $x < z < y$. The order is that induced by
order_filter()	Return the order filter generated by a list of elements.
order_ideal()	Return the order ideal generated by a list of elements.
principal_order_filter()	Returns the order filter generated by i.
principal_order_ideal()	Returns the order ideal generated by i.
pseudocomplement()	Return the pseudocomplement of element, if it exists.
rank()	Returns the rank of element, or the rank of the poset if element is None. (The rank of
rank_function()	Return the (normalized) rank function of the poset, if it exists.
top()	Returns the top element of the poset, if it exists.
upper_covers_iterator()	Returns the list of elements that cover element.

class sage.combinat.posets.hasse_diagram.**HasseDiagram**(*data=None*, *pos=None*, *loops=None*, *format=None*, *weighted=None*, *implementation='c_graph'*, *data_structure='sparse'*, *vertex_labels=True*, *name=None*, *multiedges=None*, *convert_empty_dict_labels_to_None=None*, *sparse=True*, *immutable=False*)

Bases: sage.graphs.digraph.DiGraph

The Hasse diagram of a poset. This is just a transitively-reduced, directed, acyclic graph without loops or multiple edges.

Note: We assume that range(n) is a linear extension of the poset. That is, range(n) is the vertex set and a topological sort of the digraph.

This should not be called directly, use Poset instead; all type checking happens there.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]}); H
Hasse diagram of a poset containing 4 elements
sage: TestSuite(H).run()
```

antichains(*element_class=<type 'list'>*)

Returns all antichains of self, organized as a prefix tree

INPUT:

- element_class – (default:list) an iterable type

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: H = P._hasse_diagram
sage: A = H.antichains()
sage: list(A)
[[], [0], [1], [1, 2], [1, 3], [2], [3], [4]]
sage: A.cardinality()
8
sage: [1,3] in A
True
sage: [1,4] in A
False
```

TESTS:

```
sage: TestSuite(A).run(skip = "_test_pickling")
```

Note: It's actually the pickling of the cached method `coxeter_transformation()` that fails ...

TESTS:

```
sage: A = Poset()._hasse_diagram.antichains()
sage: list(A)
[[]]
sage: TestSuite(A).run()
```

antichains_iterator()

Return an iterator over the antichains of the poset.

Note: The algorithm is based on Freese-Jezek-Nation p. 226. It does a depth first search through the set of all antichains organized in a prefix tree.

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: H = P._hasse_diagram
sage: H.antichains_iterator()
<generator object antichains_iterator at ...>
sage: list(H.antichains_iterator())
[[], [4], [3], [2], [1], [1, 3], [1, 2], [0]]

sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[4],2:[3],3:[4]})
sage: list(H.antichains_iterator())
[[], [4], [3], [2], [1], [1, 3], [1, 2], [0]]

sage: H = HasseDiagram({0:[],1:[],2:[]})
sage: list(H.antichains_iterator())
[[], [2], [1], [1, 2], [0], [0, 2], [0, 1], [0, 1, 2]]

sage: H = HasseDiagram({0:[1],1:[2],2:[3],3:[4]})
sage: list(H.antichains_iterator())
[[], [4], [3], [2], [1], [0]]
```

TESTS:

```
sage: H = Poset()._hasse_diagram
sage: list(H.antichains_iterator())
[[]]
```

are_comparable(i, j)

Returns whether `i` and `j` are comparable in the poset

INPUT:

- `i`, `j` – vertices of this Hasse diagram

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: H = P._hasse_diagram
sage: H.are_comparable(1,2)
False
```

```
sage: [ (i,j) for i in H.vertices() for j in H.vertices() if H.are_comparable(i,j)]
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 0), (1, 1), (1, 4), (2, 0), (2, 2), (2, 3), (2,
```

are_incomparable(*i*, *j*)

Returns whether `i` and `j` are incomparable in the poset

INPUT:

- `i`, `j` – vertices of this Hasse diagram

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: H = P._hasse_diagram
sage: H.are_incomparable(1,2)
True
sage: [ (i,j) for i in H.vertices() for j in H.vertices() if H.are_incomparable(i,j)]
[(1, 2), (1, 3), (2, 1), (3, 1)]
```

bottom()

Returns the bottom element of the poset, if it exists.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.bottom() is None
True
sage: Q = Poset({0:[1],1:[]})
sage: Q.bottom()
0
```

cardinality()

Returns the number of elements in the poset.

EXAMPLES:

```
sage: Poset([[1,2,3],[4],[4],[4],[]]).cardinality()
5
```

TESTS:

For a time, this function was named `size()`, which would override the same-named method of the underlying digraph. trac ticket #8735 renamed this method to `cardinality()` with a deprecation warning. trac ticket #11214 removed the warning since code for graphs was raising the warning inadvertently. This tests that `size()` for a Hasse diagram returns the number of edges in the digraph.

```
sage: L = Posets.BooleanLattice(5)
sage: H = L.hasse_diagram()
sage: H.size()
80
sage: H.size() == H.num_edges()
True
```

chains(*element_class=<type 'list'>*, *exclude=None*)

Return all chains of `self`, organized as a prefix tree.

INPUT:

- `element_class` – (default: `list`) an iterable type
- `exclude` – elements of the poset to be excluded (default: `None`)

OUTPUT:

The enumerated set (with a forest structure given by prefix ordering) consisting of all chains of `self`, each of which is given as an `element_class`.

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: H = P._hasse_diagram
sage: A = H.chains()
sage: list(A)
[[], [0], [0, 1], [0, 1, 4], [0, 2], [0, 2, 3], [0, 2, 3, 4], [0, 2, 4], [0, 3], [0, 3, 4],
sage: A.cardinality()
20
sage: [1,3] in A
False
sage: [1,4] in A
True
```

One can exclude some vertices:

```
sage: list(H.chains(exclude=[4, 3]))
[[], [0], [0, 1], [0, 2], [1], [2]]
```

The `element_class` keyword determines how the chains are being returned:

sage: P = Poset({1: [2, 3], 2: [4]}) sage: list(P._hasse_diagram.chains(element_class=tuple)) [(), (0,), (0, 1), (0, 1, 2), (0, 2), (0, 3), (1,), (1, 2), (2,), (3,)] sage: list(P._hasse_diagram.chains()) [[], [0], [0, 1], [0, 1, 2], [0, 2], [0, 3], [1], [1, 2], [2], [3]]

(Note that taking the Hasse diagram has renamed the vertices.)

sage: list(P._hasse_diagram.chains(element_class=tuple, exclude=[0])) [(), (1,), (1, 2), (2,), (3,)]

See also:

`antichains()`

closed_interval(*x*, *y*)

Return a list of the elements z of `self` such that $x \leq z \leq y$. The order is that induced by the ordering in `self.linear_extension`.

INPUT:

- `x` – any element of the poset
- `y` – any element of the poset

EXAMPLES:

```
sage: uc = [[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]]
sage: dag = DiGraph(dict(zip(range(len(uc)),uc)))
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram(dag)
sage: I = set([2,5,6,4,7])
sage: I == set(H.interval(2,7))
True
```

complements()

Deprecated.

cover_relations()

Return the list of cover relations.

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[2,3], 1:[3,4], 2:[5], 3:[5], 4:[5]})
sage: H.cover_relations()
[(0, 2), (0, 3), (1, 3), (1, 4), (2, 5), (3, 5), (4, 5)]
```

cover_relations_iterator()

Iterate over cover relations.

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[2,3], 1:[3,4], 2:[5], 3:[5], 4:[5]})
sage: list(H.cover_relations_iterator())
[(0, 2), (0, 3), (1, 3), (1, 4), (2, 5), (3, 5), (4, 5)]
```

covers(*x*, *y*)

Returns True if y covers x and False otherwise.

EXAMPLES:

```
sage: Q = Poset([[1,5],[2,6],[3],[4],[],[6,3],[4]])
sage: Q.covers(Q(1),Q(6))
True
sage: Q.covers(Q(1),Q(4))
False
```

coxeter_transformation()

Returns the matrix of the Auslander-Reiten translation acting on the Grothendieck group of the derived category of modules on the poset, in the basis of simple modules.

EXAMPLES:

```
sage: M = Posets.PentagonPoset()._hasse_diagram.coxeter_transformation(); M
[ 0  0  0  0 -1]
[ 0  0  0  1 -1]
[ 0  1  0  0 -1]
[-1  1  1  0 -1]
[-1  1  0  1 -1]
```

TESTS:

```
sage: M = Posets.PentagonPoset()._hasse_diagram.coxeter_transformation()
sage: M**8 == 1
True
```

deprecated_function_alias(*trac_number*, *func*)

Create an aliased version of a function or a method which raise a deprecation warning message.

If f is a function or a method, write `g = deprecated_function_alias(trac_number, f)` to make a deprecated aliased version of f.

INPUT:

- `trac_number` – integer. The trac ticket number where the deprecation is introduced.
- `func` – the function or method to be aliased

EXAMPLES:

```
sage: from sage.misc.superseded import deprecated_function_alias
sage: g = deprecated_function_alias(13109, number_of_partitions)
sage: g(5)
```

```
doctest:...: DeprecationWarning: g is deprecated. Please use sage.combinat.partition.number_
See http://trac.sagemath.org/13109 for details.
7
```

This also works for methods:

```
sage: class cls(object):
....:     def new_meth(self): return 42
....:     old_meth = deprecated_function_alias(13109, new_meth)
sage: cls().old_meth()
doctest:...: DeprecationWarning: old_meth is deprecated. Please use new_meth instead.
See http://trac.sagemath.org/13109 for details.
42
```

trac ticket #11585:

```
sage: def a(): pass
sage: b = deprecated_function_alias(13109, a)
sage: b()
doctest:...: DeprecationWarning: b is deprecated. Please use a instead.
See http://trac.sagemath.org/13109 for details.
```

AUTHORS:

- Florent Hivert (2009-11-23), with the help of Mike Hansen.
- Luca De Feo (2011-07-11), printing the full module path when different from old path

dual()

Returns a poset that is dual to the given poset.

EXAMPLES:

```
sage: P = Posets.IntegerPartitions(4)
sage: H = P._hasse_diagram; H
Hasse diagram of a poset containing 5 elements
sage: H.dual()
Hasse diagram of a poset containing 5 elements
```

TESTS:

```
sage: H = Posets.IntegerPartitions(4)._hasse_diagram
sage: H.is_isomorphic( H.dual().dual() )
True
sage: H.is_isomorphic( H.dual() )
False
```

frattini_sublattice()

Return the list of elements of the Frattini sublattice of the lattice.

EXAMPLES:

```
sage: H = Posets.PentagonPoset()._hasse_diagram
sage: H.frattini_sublattice()
[0, 4]
```

has_bottom()

Returns True if the poset has a unique minimal element.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.has_bottom()
False
sage: Q = Poset({0:[1],1:[]})
sage: Q.has_bottom()
True
```

has_top()

Returns `True` if the poset contains a unique maximal element, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4,5],4:[],5:[]})
sage: P.has_top()
False
sage: Q = Poset({0:[1],1:[]})
sage: Q.has_top()
True
```

interval(x, y)

Return a list of the elements z of `self` such that $x \leq z \leq y$. The order is that induced by the ordering in `self.linear_extension`.

INPUT:

- x – any element of the poset
- y – any element of the poset

EXAMPLES:

```
sage: uc = [[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]]
sage: dag = DiGraph(dict(zip(range(len(uc)),uc)))
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram(dag)
sage: I = set([2,5,6,4,7])
sage: I == set(H.interval(2,7))
True
```

is_bounded()

Returns True if the poset contains a unique maximal element and a unique minimal element, and False otherwise.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4,5],4:[],5:[]})
sage: P.is_bounded()
False
sage: Q = Poset({0:[1],1:[]})
sage: Q.is_bounded()
True
```

is_chain()

Returns True if the poset is totally ordered, and False otherwise.

EXAMPLES:

```
sage: L = Poset({0:[1],1:[2],2:[3],3:[4]})
sage: L.is_chain()
True
sage: V = Poset({0:[1,2]})
```

```
sage: V.is_chain()
False
```

TESTS:

Check trac ticket #15330:

```
sage: p = Poset(DiGraph({0:[1],2:[1]}))
sage: p.is_chain()
False
```

is_complemented_lattice()

Return `True` if `self` is the Hasse diagram of a complemented lattice, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1, 2, 3], 1:[4], 2:[4], 3:[4]})
sage: H.is_complemented_lattice()
True

sage: H = HasseDiagram({0:[1, 2], 1:[3], 2:[3], 3:[4]})
sage: H.is_complemented_lattice()
False
```

is_distributive_lattice()

Returns `True` if `self` is the Hasse diagram of a distributive lattice, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: H.is_distributive_lattice()
False
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3]})
sage: H.is_distributive_lattice()
True
sage: H = HasseDiagram({0:[1,2,3],1:[4],2:[4],3:[4]})
sage: H.is_distributive_lattice()
False
```

is_gequal(*x*, *y*)

Returns `True` if `x` is greater than or equal to `y`, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: Q = HasseDiagram({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: x,y,z = 0,1,4
sage: Q.is_gequal(x,y)
False
sage: Q.is_gequal(y,x)
False
sage: Q.is_gequal(x,z)
False
sage: Q.is_gequal(z,x)
True
sage: Q.is_gequal(z,y)
True
sage: Q.is_gequal(z,z)
True
```

is_graded()
Deprecated, has conflicting definition of "graded" vs. "ranked" with posets.

Return `True` if the Hasse diagram is ranked. For definition of ranked see `rank_function()`.

is_greater_than(*x*, *y*)
Returns `True` if `x` is greater than but not equal to `y`, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: Q = HasseDiagram({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: x,y,z = 0,1,4
sage: Q.is_greater_than(x,y)
False
sage: Q.is_greater_than(y,x)
False
sage: Q.is_greater_than(x,z)
False
sage: Q.is_greater_than(z,x)
True
sage: Q.is_greater_than(z,y)
True
sage: Q.is_greater_than(z,z)
False
```

is_join_semilattice()
Returns `True` if `self` has a join operation, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: H.is_join_semilattice()
True
sage: H = HasseDiagram({0:[2,3],1:[2,3]})
sage: H.is_join_semilattice()
False
sage: H = HasseDiagram({0:[2,3],1:[2,3],2:[4],3:[4]})
sage: H.is_join_semilattice()
False
```

is_lequal(*i*, *j*)
Returns True if i is less than or equal to j in the poset, and False otherwise.

Note: If the `lequal_matrix()` has been computed, then this method is redefined to use the cached matrix (see `_alternate_is_lequal()`).

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: x,y,z = 0, 1, 4
sage: H.is_lequal(x,y)
False
sage: H.is_lequal(y,x)
False
sage: H.is_lequal(x,z)
True
sage: H.is_lequal(y,z)
True
```

```
sage: H.is_lequal(z,z)
True
```

is_less_than(*x*, *y*)

Returns True if x is less than or equal to y in the poset, and False otherwise.

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: x,y,z = 0, 1, 4
sage: H.is_less_than(x,y)
False
sage: H.is_less_than(y,x)
False
sage: H.is_less_than(x,z)
True
sage: H.is_less_than(y,z)
True
sage: H.is_less_than(z,z)
False
```

is_linear_extension(*lin_ext=None*)

Test if an ordering is a linear extension.

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]})
sage: H.is_linear_extension(range(4))
True
sage: H.is_linear_extension([3,2,1,0])
False
```

is_meet_semilattice()

Returns `True` if `self` has a meet operation, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: H.is_meet_semilattice()
True

sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]})
sage: H.is_meet_semilattice()
True

sage: H = HasseDiagram({0:[2,3],1:[2,3]})
sage: H.is_meet_semilattice()
False
```

is_ranked()

Returns True if the poset is ranked, and False otherwise.

A poset is *ranked* if it admits a rank function. For more information about the rank function, see `rank_function()` and `is_graded()`.

EXAMPLES:

```
sage: P = Poset([[1],[2],[3],[4],[]])
sage: P.is_ranked()
True
sage: Q = Poset([[1,5],[2,6],[3],[4],[],[6,3],[4]])
sage: Q.is_ranked()
False
```

join_matrix()

Returns the matrix of joins of `self`. The `(x,y)`-entry of this matrix is the join of `x` and `y` in `self`.

This algorithm is modelled after the algorithm of Freese-Jezek-Nation (p217). It can also be found on page 140 of [Gec81].

Note: Once the matrix has been computed, it is stored in `_join_matrix()`. Delete this attribute if you want to recompute the matrix.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: H.join_matrix()
[0 1 2 3 4 5 6 7]
[1 1 4 7 4 7 7 7]
[2 4 2 6 4 5 6 7]
[3 7 6 3 7 7 6 7]
[4 4 4 7 4 7 7 7]
[5 7 5 7 7 5 7 7]
[6 7 6 6 7 7 6 7]
[7 7 7 7 7 7 7 7]
```

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[2,3],1:[2,3]})
sage: H.join_matrix()
Traceback (most recent call last):
...
ValueError: Not a join-semilattice: no top element.

sage: H = HasseDiagram({0:[2,3],1:[2,3],2:[4],3:[4]})
sage: H.join_matrix()
Traceback (most recent call last):
...
ValueError: No join for x=...
```

lequal_matrix()

Returns the matrix whose `(i,j)` entry is 1 if `i` is less than `j` in the poset, and 0 otherwise; and redefines `__lt__` to use this matrix.

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: H = P._hasse_diagram
sage: H.lequal_matrix()
[1 1 1 1 1 1 1 1]
[0 1 0 1 0 0 0 1]
[0 0 1 1 1 0 1 1]
[0 0 0 1 0 0 0 1]
[0 0 0 0 1 0 0 1]
```

```
[0 0 0 0 0 1 1 1]
[0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 0 1]
```

TESTS:

```
sage: H.lequal_matrix().is_immutable()
True
```

linear_extension()

Return a linear extension

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]})
sage: H.linear_extension()
[0, 1, 2, 3]
```

linear_extensions()

Return all linear extensions

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]})
sage: H.linear_extensions()
[[0, 1, 2, 3], [0, 2, 1, 3]]
```

lower_covers_iterator(*element*)

Returns the list of elements that are covered by `element`.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: list(H.lower_covers_iterator(0))
[]
sage: list(H.lower_covers_iterator(4))
[1, 2]
```

maximal_elements()

Returns a list of the maximal elements of the poset.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.maximal_elements()
[4]
```

maximal_sublattices()

Return maximal sublattices of the lattice.

EXAMPLES:

```
sage: L = Posets.PentagonPoset()
sage: ms = L._hasse_diagram.maximal_sublattices()
sage: sorted(ms, key=sorted)
[{0, 1, 2, 4}, {0, 1, 3, 4}, {0, 2, 3, 4}]
```

meet_matrix()

Returns the matrix of meets of `self`. The `(x,y)`-entry of this matrix is the meet of `x` and `y` in `self`.

This algorithm is modelled after the algorithm of Freese-Jezek-Nation (p217). It can also be found on page 140 of [Gec81].

Note: Once the matrix has been computed, it is stored in `_meet_matrix()`. Delete this attribute if you want to recompute the matrix.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: H.meet_matrix()
[0 0 0 0 0 0 0 0]
[0 1 0 0 1 0 0 1]
[0 0 2 0 2 2 2 2]
[0 0 0 3 0 0 3 3]
[0 1 2 0 4 2 2 4]
[0 0 2 0 2 5 2 5]
[0 0 2 3 2 2 6 6]
[0 1 2 3 4 5 6 7]
```

REFERENCE:

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[2,3],1:[2,3]})
sage: H.meet_matrix()
Traceback (most recent call last):
...
ValueError: Not a meet-semilattice: no bottom element.

sage: H = HasseDiagram({0:[1,2],1:[3,4],2:[3,4]})
sage: H.meet_matrix()
Traceback (most recent call last):
...
ValueError: No meet for x=...
```

minimal_elements()

Returns a list of the minimal elements of the poset.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P(0) in P.minimal_elements()
True
sage: P(1) in P.minimal_elements()
True
sage: P(2) in P.minimal_elements()
True
```

mobius_function(**args*, ***kwds*)

Deprecated: Use `moebius_function()` instead. See trac ticket #19855 for details.

mobius_function_matrix(**args*, ***kwds*)

Deprecated: Use `moebius_function_matrix()` instead. See trac ticket #19855 for details.

moebius_function(*i*, *j*)

Returns the value of the Möbius function of the poset on the elements `i` and `j`.

EXAMPLES:

```
sage: P = Poset([[1,2,3],[4],[4],[4],[]])
sage: H = P._hasse_diagram
sage: H.moebius_function(0,4)
2
sage: for u,v in P.cover_relations_iterator():
....:     if P.moebius_function(u,v) != -1:
....:         print "Bug in moebius_function!"
```

moebius_function_matrix()

Returns the matrix of the Möbius function of this poset

This returns the sparse matrix over $\mathbf{Z}$ whose `(x, y)` entry is the value of the Möbius function of `self` evaluated on `x` and `y`, and redefines `moebius_function()` to use it.

Note: The result is cached in `_moebius_function_matrix()`.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: H.moebius_function_matrix()
[ 1 -1 -1 -1  1  0  1  0]
[ 0  1  0  0 -1  0  0  0]
[ 0  0  1  0 -1 -1 -1  2]
[ 0  0  0  1  0  0 -1  0]
[ 0  0  0  0  1  0  0 -1]
[ 0  0  0  0  0  1  0 -1]
[ 0  0  0  0  0  0  1 -1]
[ 0  0  0  0  0  0  0  1]
```

TESTS:

```
sage: H.moebius_function_matrix().is_immutable()
True
sage: hasattr(H,'_moebius_function_matrix')
True

sage: H.moebius_function == H._moebius_function_from_matrix
True
```

open_interval(x, y)

Return a list of the elements z of `self` such that $x < z < y$. The order is that induced by the ordering in `self.linear_extension`.

EXAMPLES:

```
sage: uc = [[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]]
sage: dag = DiGraph(dict(zip(range(len(uc)),uc)))
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram(dag)
sage: set([5,6,4]) == set(H.open_interval(2,7))
True
sage: H.open_interval(7,2)
[]
```

order_filter(*elements*)

Return the order filter generated by a list of elements.

I is an order filter if, for any x in I and y such that $y \geq x$, then y is in I.

EXAMPLES:

```
sage: H = Posets.BooleanLattice(4)._hasse_diagram
sage: H.order_filter([3,8])
[3, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

order_ideal(*elements*)

Return the order ideal generated by a list of elements.

I is an order ideal if, for any x in I and y such that $y \leq x$, then y is in I.

EXAMPLES:

```
sage: H = Posets.BooleanLattice(4)._hasse_diagram
sage: H.order_ideal([7,10])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 10]
```

principal_order_filter(*i*)

Returns the order filter generated by `i`.

EXAMPLES:

```
sage: H = Posets.BooleanLattice(4)._hasse_diagram
sage: H.principal_order_filter(2)
[2, 3, 6, 7, 10, 11, 14, 15]
```

principal_order_ideal(*i*)

Returns the order ideal generated by i.

EXAMPLES:

```
sage: H = Posets.BooleanLattice(4)._hasse_diagram
sage: H.principal_order_ideal(6)
[0, 2, 4, 6]
```

pseudocomplement(*element*)

Return the pseudocomplement of `element`, if it exists.

The pseudocomplement is the greatest element whose meet with given element is the bottom element. It may not exist, and then the function returns `None`.

INPUT:

- `element` – an element of the lattice.

OUTPUT:

An element of the Hasse diagram, i.e. an integer, or `None` if the pseudocomplement does not exist.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0: [1, 2], 1: [3], 2: [4], 3: [4]})
sage: H.pseudocomplement(2)
3

sage: H = HasseDiagram({0: [1, 2, 3], 1: [4], 2: [4], 3: [4]})
sage: H.pseudocomplement(2) is None
True
```

rank(*element=None*)

Returns the rank of `element`, or the rank of the poset if `element` is `None`. (The rank of a poset is the length of the longest chain of elements of the poset.)

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: H.rank(5)
2
sage: H.rank()
3
sage: Q = HasseDiagram({0:[1,2],1:[3],2:[],3:[]})
sage: Q.rank()
2
sage: Q.rank(1)
1
```

rank_function()

Return the (normalized) rank function of the poset, if it exists.

A *rank function* of a poset P is a function r that maps elements of P to integers and satisfies: $r(x) = r(y) + 1$ if x covers y. The function r is normalized such that its minimum value on every connected component of the Hasse diagram of P is 0. This determines the function r uniquely (when it exists).

OUTPUT:

- a lambda function, if the poset admits a rank function
- None, if the poset does not admit a rank function

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: P.rank_function() is not None
True
sage: P = Poset(([1,2,3,4],[[1,4],[2,3],[3,4]]), facade = True)
sage: P.rank_function() is not None
True
sage: P = Poset(([1,2,3,4,5],[[1,2],[2,3],[3,4],[1,5],[5,4]]), facade = True)
sage: P.rank_function() is not None
False
sage: P = Poset(([1,2,3,4,5,6,7,8],[[1,4],[2,3],[3,4],[5,7],[6,7]]), facade = True)
sage: f = P.rank_function(); f is not None
True
sage: f(5)
0
sage: f(2)
0
```

TESTS:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: r = P.rank_function()
sage: for u,v in P.cover_relations_iterator():
...     if r(v) != r(u) + 1:
...         print "Bug in rank_function!"

sage: Q = Poset([[1,2],[4],[3],[4],[]])
sage: Q.rank_function() is None
True
```

test for ticket trac ticket #14006:

```
sage: H = Poset()._hasse_diagram
sage: s = dumps(H)
```

```
sage: f = H.rank_function()
sage: s = dumps(H)
```

top()
Returns the top element of the poset, if it exists.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4,5],4:[],5:[]})
sage: P.top() is None
True
sage: Q = Poset({0:[1],1:[]})
sage: Q.top()
1
```

upper_covers_iterator(*element*)
Returns the list of elements that cover `element`.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: list(H.upper_covers_iterator(0))
[1, 2, 3]
sage: list(H.upper_covers_iterator(7))
[]
```

5.1.154 Incidence Algebras

class sage.combinat.posets.incidence_algebras.**IncidenceAlgebra**(*R*, *P*, *prefix='I'*)
Bases: sage.combinat.free_module.CombinatorialFreeModule

The incidence algebra of a poset.

Let P be a poset and R be a commutative unital associative ring. The *incidence algebra* I_P is the algebra of functions $\alpha\colon P \times P \to R$ such that $\alpha(x, y) = 0$ if $x \not\leq y$ where multiplication is given by convolution:

$$(\alpha * \beta)(x, y) = \sum_{x \leq k \leq y} \alpha(x, k)\beta(k, y).$$

This has a natural basis given by indicator functions for the interval $[a, b]$, i.e. $X_{a,b}(x, y) = \delta_{ax}\delta_{by}$. The incidence algebra is a unital algebra with the identity given by the Kronecker delta $\delta(x, y) = \delta_{xy}$. The Möbius function of P is another element of I_p whose inverse is the ζ function of the poset (so $\zeta(x, y) = 1$ for every interval $[x, y]$).

Todo

Implement the incidence coalgebra.

REFERENCES:

- Wikipedia article Incidence_algebra

class Element(*M*, *x*)
Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

An element of an incidence algebra.

is_unit()
Return if `self` is a unit.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: mu = I.moebius()
sage: mu.is_unit()
True
sage: zeta = I.zeta()
sage: zeta.is_unit()
True
sage: x = mu - I.zeta() + I[2,2]
sage: x.is_unit()
False
sage: y = I.moebius() + I.zeta()
sage: y.is_unit()
True
```

This depends on the base ring:

```
sage: I = P.incidence_algebra(ZZ)
sage: y = I.moebius() + I.zeta()
sage: y.is_unit()
False
```

to_matrix()
Return `self` as a matrix.

We define a matrix $M_{xy} = \alpha(x, y)$ for some element $\alpha \in I_P$ in the incidence algebra I_P and we order the elements $x, y \in P$ by some linear extension of P. This defines an algebra (iso)morphism; in particular, multiplication in the incidence algebra goes to matrix multiplication.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: I.moebius().to_matrix()
[ 1 -1 -1  1]
[ 0  1  0 -1]
[ 0  0  1 -1]
[ 0  0  0  1]
sage: I.zeta().to_matrix()
[1 1 1 1]
[0 1 0 1]
[0 0 1 1]
[0 0 0 1]
```

TESTS:

We check that this is an algebra (iso)morphism:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: mu = I.moebius()
sage: (mu*mu).to_matrix() == mu.to_matrix() * mu.to_matrix()
True
```

IncidenceAlgebra.**delta**()
Return the element 1 in `self` (which is the Kronecker delta $\delta(x, y)$).

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.one()
I[0, 0] + I[1, 1] + I[2, 2] + I[3, 3] + I[4, 4] + I[5, 5]
 + I[6, 6] + I[7, 7] + I[8, 8] + I[9, 9] + I[10, 10]
 + I[11, 11] + I[12, 12] + I[13, 13] + I[14, 14] + I[15, 15]
```

IncidenceAlgebra.**deprecated_function_alias**(*trac_number*, *func*)

Create an aliased version of a function or a method which raise a deprecation warning message.

If f is a function or a method, write `g = deprecated_function_alias(trac_number, f)` to make a deprecated aliased version of f.

INPUT:

- `trac_number` – integer. The trac ticket number where the deprecation is introduced.
- `func` – the function or method to be aliased

EXAMPLES:

```
sage: from sage.misc.superseded import deprecated_function_alias
sage: g = deprecated_function_alias(13109, number_of_partitions)
sage: g(5)
doctest:...: DeprecationWarning: g is deprecated. Please use sage.combinat.partition.number_
See http://trac.sagemath.org/13109 for details.
7
```

This also works for methods:

```
sage: class cls(object):
....:     def new_meth(self): return 42
....:     old_meth = deprecated_function_alias(13109, new_meth)
sage: cls().old_meth()
doctest:...: DeprecationWarning: old_meth is deprecated. Please use new_meth instead.
See http://trac.sagemath.org/13109 for details.
42
```

trac ticket #11585:

```
sage: def a(): pass
sage: b = deprecated_function_alias(13109, a)
sage: b()
doctest:...: DeprecationWarning: b is deprecated. Please use a instead.
See http://trac.sagemath.org/13109 for details.
```

AUTHORS:

- Florent Hivert (2009-11-23), with the help of Mike Hansen.
- Luca De Feo (2011-07-11), printing the full module path when different from old path

IncidenceAlgebra.**mobius**(**args*, ***kwds*)

Deprecated: Use `moebius()` instead. See trac ticket #19855 for details.

IncidenceAlgebra.**moebius**()

Return the Möbius function of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: I.moebius()
```

```
I[0, 0] - I[0, 1] - I[0, 2] + I[0, 3] + I[1, 1]
 - I[1, 3] + I[2, 2] - I[2, 3] + I[3, 3]
```

IncidenceAlgebra.**one**()

Return the element 1 in `self` (which is the Kronecker delta $\delta(x, y)$).

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.one()
I[0, 0] + I[1, 1] + I[2, 2] + I[3, 3] + I[4, 4] + I[5, 5]
 + I[6, 6] + I[7, 7] + I[8, 8] + I[9, 9] + I[10, 10]
 + I[11, 11] + I[12, 12] + I[13, 13] + I[14, 14] + I[15, 15]
```

IncidenceAlgebra.**poset**()

Return the defining poset of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.poset()
Finite lattice containing 16 elements
sage: I.poset() == P
True
```

IncidenceAlgebra.**product_on_basis**(*A*, *B*)

Return the product of basis elements indexed by `A` and `B`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.product_on_basis((1, 3), (3, 11))
I[1, 11]
sage: I.product_on_basis((1, 3), (2, 2))
0
```

IncidenceAlgebra.**reduced_subalgebra**(*prefix='R'*)

Return the reduced incidence subalgebra.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.reduced_subalgebra()
Reduced incidence algebra of Finite lattice containing 16 elements
 over Rational Field
```

IncidenceAlgebra.**some_elements**()

Return a list of elements of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(1)
sage: I = P.incidence_algebra(QQ)
sage: I.some_elements()
[2*I[0, 0] + 2*I[0, 1] + 3*I[1, 1],
 I[0, 0] - I[0, 1] + I[1, 1],
 I[0, 0] + I[0, 1] + I[1, 1]]
```

IncidenceAlgebra.**zeta**()
Return the ζ function in self.

The ζ function on a poset P is given by

$$\zeta(x,y) = \begin{cases} 1 & x \leq y, \\ 0 & x \not\leq y. \end{cases}$$

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.zeta() * I.moebius() == I.one()
True
```

class sage.combinat.posets.incidence_algebras.**ReducedIncidenceAlgebra**(*I*, *prefix='R'*)
Bases: sage.combinat.free_module.CombinatorialFreeModule

The reduced incidence algebra of a poset.

The reduced incidence algebra R_P is a subalgebra of the incidence algebra I_P where $\alpha(x,y) = \alpha(x',y')$ when $[x,y]$ is isomorphic to $[x',y']$ as posets. Thus the delta, Möbius, and zeta functions are all elements of R_P.

class **Element**(*M*, *x*)
Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

An element of a reduced incidence algebra.

is_unit()
Return if self is a unit.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: x = R.an_element()
sage: x.is_unit()
True
```

lift()
Return the lift of self to the ambient space.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: R = I.reduced_subalgebra()
sage: x = R.an_element(); x
2*R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)]
sage: x.lift()
2*I[0, 0] + 2*I[0, 1] + 2*I[0, 2] + 3*I[0, 3] + 2*I[1, 1]
 + 2*I[1, 3] + 2*I[2, 2] + 2*I[2, 3] + 2*I[3, 3]
```

to_matrix()
Return self as a matrix.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: mu = R.moebius()
sage: mu.to_matrix()
[ 1 -1 -1  1]
```

```
[ 0  1  0 -1]
[ 0  0  1 -1]
[ 0  0  0  1]
```

ReducedIncidenceAlgebra.**delta**()

Return the Kronecker delta function in `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.delta()
R[(0, 0)]
```

ReducedIncidenceAlgebra.**deprecated_function_alias**(*trac_number*, *func*)

Create an aliased version of a function or a method which raise a deprecation warning message.

If f is a function or a method, write `g = deprecated_function_alias(trac_number, f)` to make a deprecated aliased version of f.

INPUT:

- `trac_number` – integer. The trac ticket number where the deprecation is introduced.
- `func` – the function or method to be aliased

EXAMPLES:

```
sage: from sage.misc.superseded import deprecated_function_alias
sage: g = deprecated_function_alias(13109, number_of_partitions)
sage: g(5)
doctest:...: DeprecationWarning: g is deprecated. Please use sage.combinat.partition.number_
See http://trac.sagemath.org/13109 for details.
7
```

This also works for methods:

```
sage: class cls(object):
....:     def new_meth(self): return 42
....:     old_meth = deprecated_function_alias(13109, new_meth)
sage: cls().old_meth()
doctest:...: DeprecationWarning: old_meth is deprecated. Please use new_meth instead.
See http://trac.sagemath.org/13109 for details.
42
```

trac ticket #11585:

```
sage: def a(): pass
sage: b = deprecated_function_alias(13109, a)
sage: b()
doctest:...: DeprecationWarning: b is deprecated. Please use a instead.
See http://trac.sagemath.org/13109 for details.
```

AUTHORS:

- Florent Hivert (2009-11-23), with the help of Mike Hansen.
- Luca De Feo (2011-07-11), printing the full module path when different from old path

ReducedIncidenceAlgebra.**lift**()

Return the lift morphism from `self` to the ambient space.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.lift
Generic morphism:
  From: Reduced incidence algebra of Finite lattice containing 4 elements over Rational Fiel
  To:   Incidence algebra of Finite lattice containing 4 elements over Rational Field
sage: R.an_element() - R.one()
R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)]
sage: R.lift(R.an_element() - R.one())
I[0, 0] + 2*I[0, 1] + 2*I[0, 2] + 3*I[0, 3] + I[1, 1]
 + 2*I[1, 3] + I[2, 2] + 2*I[2, 3] + I[3, 3]
```

ReducedIncidenceAlgebra.**mobius**(**args*, ***kwds*)

Deprecated: Use `moebius()` instead. See trac ticket #19855 for details.

ReducedIncidenceAlgebra.**moebius**()

Return the Möbius function of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.moebius()
R[(0, 0)] - R[(0, 1)] + R[(0, 3)] - R[(0, 7)] + R[(0, 15)]
```

ReducedIncidenceAlgebra.**one_basis**()

Return the index of the element 1 in `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.one_basis()
(0, 0)
```

ReducedIncidenceAlgebra.**poset**()

Return the defining poset of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.poset()
Finite lattice containing 16 elements
sage: R.poset() == P
True
```

ReducedIncidenceAlgebra.**some_elements**()

Return a list of elements of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.some_elements()
[2*R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)],
 R[(0, 0)] - R[(0, 1)] + R[(0, 3)] - R[(0, 7)] + R[(0, 15)],
 R[(0, 0)] + R[(0, 1)] + R[(0, 3)] + R[(0, 7)] + R[(0, 15)]]
```

ReducedIncidenceAlgebra.**zeta**()

Return the ζ function in `self`.

The ζ function on a poset P is given by

$$\zeta(x,y) = \begin{cases} 1 & x \leq y, \\ 0 & x \not\leq y. \end{cases}$$

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.zeta()
R[(0, 0)] + R[(0, 1)] + R[(0, 3)] + R[(0, 7)] + R[(0, 15)]
```

5.1.155 Finite lattices and semilattices

This module implements finite (semi)lattices. It defines:

`LatticePoset()`	Construct a lattice.
`MeetSemilattice()`	Construct a meet semi-lattice.
`JoinSemilattice()`	Construct a join semi-lattice.
`FiniteLatticePoset`	A class for finite lattices.
`FiniteMeetSemilattice`	A class for finite meet semilattices.
`FiniteJoinSemilattice`	A class for finite join semilattices.

List of (semi)lattice methods

Meet and join

`meet()`	Return the meet of given elements.
`join()`	Return the join of given elements.
`meet_matrix()`	Return the matrix of meets of all elements of the meet semi-lattice.
`join_matrix()`	Return the matrix of joins of all elements of the join semi-lattice.

Properties of the lattice

`is_distributive()`	Return `True` if the lattice is distributive.
`is_modular()`	Return `True` if the lattice is modular.
`is_lower_semimodular()`	Return `True` if the lattice is lower semimodular.
`is_upper_semimodular()`	Return `True` if the lattice is upper semimodular.
`is_atomic()`	Return `True` if every element of the lattice can be written as a join of atoms.
`is_geometric()`	Return `True` if the lattice is atomic and upper semimodular.
`is_complemented()`	Return `True` if every element of the lattice has at least one complement.
`is_pseudocomplemented()`	Return `True` if every element of the lattice has a pseudocomplement.
`is_supersolvable()`	Return `True` if the lattice is supersolvable.
`is_planar()`	Return `True` if the lattice has an upward planar drawing.
`is_dismantlable()`	Return `True` if the lattice is dismantlable.
`breadth()`	Return the breadth of the lattice.

Elements and sublattices

`complements()`	Return the list of complements of an element, or the dictionary of complements for all elements.
`pseudocomplement()`	Return the pseudocomplement of an element.
`is_modular_element()`	Return `True` if given element is modular in the lattice.
`sublattice()`	Return sublattice generated by list of elements.
`maximal_sublattices()`	Return maximal sublattices of the lattice.
`frattini_sublattice()`	Return the intersection of maximal sublattices of the lattice.

Miscellaneous

moebius_algebra()	Return the quantum Möbius algebra of the lattice.
quantum_moebius_algebra()	Return the quantum Möbius algebra of the lattice.

class sage.combinat.posets.lattices.**FiniteJoinSemilattice**(*hasse_diagram*, *elements*, *category*, *facade*, *key*)

Bases: sage.combinat.posets.posets.FinitePoset

We assume that the argument passed to FiniteJoinSemilattice is the poset of a join-semilattice (i.e. a poset with least upper bound for each pair of elements).

TESTS:

```
sage: J = JoinSemilattice([[1,2],[3],[3]])
sage: TestSuite(J).run()

sage: P = Poset([[1,2],[3],[3]])
sage: J = JoinSemilattice(P)
sage: TestSuite(J).run()
```

Element

alias of JoinSemilatticeElement

join(*x*, *y=None*)

Return the join of given elements in the lattice.

INPUT:

- x, y - two elements of the (semi)lattice OR
- x - a list or tuple of elements

EXAMPLES:

```
sage: D = Posets.DiamondPoset(5)
sage: D.join(1, 2)
4
sage: D.join(1, 1)
1
sage: D.join(1, 4)
4
sage: D.join(1, 0)
1
```

Using list of elements as an argument. Join of empty list is the bottom element:

```
sage: B4=Posets.BooleanLattice(4)
sage: B4.join([2,4,8])
14
sage: B4.join([])
0
```

Test that this method also works for facade lattices:

```
sage: L = LatticePoset([[1,2],[3],[3]], facade = True)
sage: L.join(1, 0)
1
sage: L.join(1, 2)
3
```

join_matrix()

Return a matrix whose (i,j) entry is k, where self.linear_extension()[k] is the join (least upper bound) of self.linear_extension()[i] and self.linear_extension()[j].

EXAMPLES:

```
sage: P = LatticePoset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]], facade = False)
sage: J = P.join_matrix(); J
[0 1 2 3 4 5 6 7]
[1 1 3 3 7 7 7 7]
[2 3 2 3 4 6 6 7]
[3 3 3 3 7 7 7 7]
[4 7 4 7 4 7 7 7]
[5 7 6 7 7 5 6 7]
[6 7 6 7 7 6 6 7]
[7 7 7 7 7 7 7 7]
sage: J[P(4).vertex,P(3).vertex] == P(7).vertex
True
sage: J[P(5).vertex,P(2).vertex] == P(5).vertex
True
sage: J[P(5).vertex,P(2).vertex] == P(2).vertex
False
```

class sage.combinat.posets.lattices.**FiniteLatticePoset**(*hasse_diagram*, *elements*, *category*, *facade*, *key*)

Bases: sage.combinat.posets.lattices.FiniteMeetSemilattice, sage.combinat.posets.lattices.FiniteJoinSemilattice

We assume that the argument passed to FiniteLatticePoset is the poset of a lattice (i.e. a poset with greatest lower bound and least upper bound for each pair of elements).

TESTS:

```
sage: L = LatticePoset([[1,2],[3],[3]])
sage: TestSuite(L).run()

sage: P = Poset([[1,2],[3],[3]])
sage: L = LatticePoset(P)
sage: TestSuite(L).run()
```

Element

alias of LatticePosetElement

breadth(*certificate=False*)

Return the breadth of the lattice.

The breadth of a lattice is the largest integer n such that any join of elements $x_1, x_2, \ldots, x_{n+1}$ is join of a proper subset of x_i.

INPUT:

- certificate – (boolean; default: False) – whether to return an integer (the breadth) or a certificate, i.e. a biggest set whose join differs from the join of any subset.

EXAMPLES:

```
sage: D10 = Posets.DiamondPoset(10)
sage: D10.breadth()
2

sage: B3 = Posets.BooleanLattice(3)
sage: B3.breadth()
3
sage: B3.breadth(certificate=True)
[1, 2, 4]
```

Smallest example of a lattice with breadth 4:

```
sage: L = LatticePoset(DiGraph('O]???w?K_@S?E_??Q?@_?D??I??W?B??@??C??O?@???'))
sage: L.breadth()
4
```

ALGORITHM:

For a lattice to have breadth at least n, it must have an n-element antichain A with join j. Element j must cover at least n elements. There must also be $n-2$ levels of elements between A and j. So we start by searching elements that could be our j and then just check possible antichains A.

TESTS:

```
sage: Posets.ChainPoset(0).breadth()
0
sage: Posets.ChainPoset(1).breadth()
1
```

complements(*element=None*)

Return the list of complements of an element in the lattice, or the dictionary of complements for all elements.

Elements x and y are complements if their meet and join are respectively the bottom and the top element of the lattice.

INPUT:

- `element` - an element of the lattice whose complement is returned. If `None` (default) then dictionary of complements for all elements having at least one complement is returned.

EXAMPLES:

```
sage: L=LatticePoset({0:['a','b','c'], 'a':[1], 'b':[1], 'c':[1]})
sage: C = L.complements()
```

Let us check that $'a'$ and $'b'$ are complements of each other:

```
sage: 'a' in C['b']
True
sage: 'b' in C['a']
True
```

Full list of complements:

```
sage: L.complements() # random
{0: [1], 1: [0], 'a': ['b', 'c'], 'b': ['c', 'a'], 'c': ['b', 'a']}

sage: L=LatticePoset({0:[1,2],1:[3],2:[3],3:[4]})
sage: L.complements() # random
{0: [4], 4: [0]}
sage: L.complements(1)
[]
```

TESTS:

```
sage: L=LatticePoset({0:['a','b','c'], 'a':[1], 'b':[1], 'c':[1]})
sage: for v,v_complements in L.complements().items():
....:     for v_c in v_complements:
....:         assert L.meet(v,v_c) == L.bottom()
....:         assert L.join(v,v_c) == L.top()
```

frattini_sublattice()
Return the Frattini sublattice of the lattice.

The Frattini sublattice $\Phi(L)$ is the intersection of all proper maximal sublattices of L. It is also the set of "non-generators" - if the sublattice generated by set S of elements is whole lattice, then also $S \setminus \Phi(L)$ generates whole lattice.

EXAMPLES:

```
sage: L = LatticePoset(( [], [[1,2],[1,17],[1,8],[2,3],[2,22],
....:                         [2,5],[2,7],[17,22],[17,13],[8,7],
....:                         [8,13],[3,16],[3,9],[22,16],[22,18],
....:                         [22,10],[5,18],[5,14],[7,9],[7,14],
....:                         [7,10],[13,10],[16,6],[16,19],[9,19],
....:                         [18,6],[18,33],[14,33],[10,19],
....:                         [10,33],[6,4],[19,4],[33,4]] ))
sage: sorted(L.frattini_sublattice().list())
[1, 2, 4, 10, 19, 22, 33]
```

is_atomic()
Returns `True` if `self` is an atomic lattice and `False` otherwise.

A lattice is atomic if every element can be written as a join of atoms.

EXAMPLES:

```
sage: L = LatticePoset({0:[1,2,3],1:[4],2:[4],3:[4]})
sage: L.is_atomic()
True

sage: L = LatticePoset({0:[1,2],1:[3],2:[3],3:[4]})
sage: L.is_atomic()
False
```

NOTES:

See [Sta97], Section 3.3 for a discussion of atomic lattices.

REFERENCES:

is_complemented()
Returns `True` if `self` is a complemented lattice, and `False` otherwise.

EXAMPLES:

```
sage: L = LatticePoset({0:[1,2,3],1:[4],2:[4],3:[4]})
sage: L.is_complemented()
True

sage: L = LatticePoset({0:[1,2],1:[3],2:[3],3:[4]})
sage: L.is_complemented()
False
```

is_dismantlable(*certificate=False*)
Return `True` if the lattice is dismantlable, and `False` otherwise.

An n-element lattice L_n is dismantlable if there is a sublattice chain $L_{n-1} \supset L_{n-2}, \supset \cdots, \supset L_0$ so that every L_i is a sublattice of L_{i+1} with one element less, and L_0 is the empty lattice. In other words, a dismantlable lattice can be reduced to empty lattice removing doubly irreducible element one by one.

INPUT:

- `certificate` (boolean) – Whether to return a certificate.

– If `certificate = False` (default), returns `True` or `False` accordingly.

– If `certificate = True`, returns:

* `(True, elms)` when the lattice is dismantlable, where `elms` is elements listed in a possible removing order.

* `(False, crown)` when the lattice is not dismantlable, where `crown` is a subposet of $2k$ elements $a_1, \ldots, a_k, b_1, \ldots, b_k$ with covering relations $a_i \lessdot b_i$ and $a_i \lessdot b_{i+1}$ for $i \in [1, \ldots, k-1]$, and $a_k \lessdot b_1$.

EXAMPLES:

```
sage: DL12 = LatticePoset((divisors(12), attrcall("divides")))
sage: DL12.is_dismantlable()
True
sage: DL12.is_dismantlable(certificate=True)
(True, [4, 2, 1, 3, 6, 12])

sage: B3 = Posets.BooleanLattice(3)
sage: B3.is_dismantlable()
False
sage: B3.is_dismantlable(certificate=True)
(False, Finite poset containing 6 elements)
```

Every planar lattice is dismantlable. Converse is not true:

```
sage: L = LatticePoset( ([], [[0, 1], [0, 2], [0, 3], [0, 4],
....:                        [1, 7], [2, 6], [3, 5], [4, 5],
....:                        [4, 6], [4, 7], [5, 8], [6, 8],
....:                        [7, 8]]) )
sage: L.is_dismantlable()
True
sage: L.is_planar()
False
```

TESTS:

```
sage: Posets.ChainPoset(0).is_dismantlable()
True
sage: Posets.ChainPoset(1).is_dismantlable()
True

sage: L = LatticePoset(DiGraph('L@_?W?E?@CCO?A?@??_?O?Jw????C?'))
sage: L.is_dismantlable()
False
sage: c = L.is_dismantlable(certificate=True)[1]
sage: (3 in c, 12 in c, 9 in c)
(True, False, True)
```

is_distributive()

Return `True` if the lattice is distributive, and `False` otherwise.

A lattice $(L, \vee, \wedge)$ is distributive if meet distributes over join: $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ for every $x, y, z \in L$ just like $x \cdot (y + z) = x \cdot y + x \cdot z$ in normal arithmetic. For duality in lattices it follows that then also join distributes over meet.

EXAMPLES:

```
sage: L = LatticePoset({0:[1,2],1:[3],2:[3]})
sage: L.is_distributive()
True
```

```
sage: L = LatticePoset({0:[1,2,3],1:[4],2:[4],3:[4]})
sage: L.is_distributive()
False
```

is_geometric()

Return `True` if the lattice is geometric, and `False` otherwise.

A lattice is geometric if it is both atomic and upper semimodular.

EXAMPLES:

Canonical example is the lattice of partitions of finite set ordered by refinement:

```
sage: S = SetPartitions(3)
sage: L = LatticePoset( (S, lambda a, b: S.is_less_than(a, b)) )
sage: L.is_geometric()
True
```

Smallest example of geometric lattice that is not modular:

```
sage: L = LatticePoset(DiGraph('K]?@g@S?q?M?@?@?@?@?@?@??'))
sage: L.is_geometric()
True
sage: L.is_modular()
False
```

Two non-examples:

```
sage: L = LatticePoset({1:[2, 3, 4], 2:[5, 6], 3:[5], 4:[6], 5:[7], 6:[7]})
sage: L.is_geometric()  # Graded, but not upper semimodular
False
sage: L = Posets.ChainPoset(3)
sage: L.is_geometric()  # Modular, but not atomic
False
```

TESTS:

```
sage: LatticePoset({}).is_geometric()
True
sage: LatticePoset({1:[]}).is_geometric()
True
```

is_lower_semimodular()

Return `True` if the lattice is lower semimodular and `False` otherwise.

A lattice is lower semimodular if for any x in the lattice that covers y and z, both y and z cover their meet.

See also `is_modular()` and `is_upper_semimodular()`.

See Wikipedia article Semimodular_lattice

EXAMPLES:

```
sage: L = posets.DiamondPoset(5)
sage: L.is_lower_semimodular()
True

sage: L = posets.PentagonPoset()
sage: L.is_lower_semimodular()
False

sage: L = posets.ChainPoset(6)
```

```
sage: L.is_lower_semimodular()
True
```

ALGORITHM:

Based on pp. 286-287 of Enumerative Combinatorics, Vol 1 [EnumComb1].

is_modular(*L=None*)

Return `True` if the lattice is modular and `False` otherwise.

Using the parameter `L`, this can also be used to check that some subset of elements are all modular.

INPUT:

- `L` – (default: `None`) a list of elements to check being modular, if `L` is `None`, then this checks the entire lattice

An element x in a lattice L is *modular* if $x \leq b$ implies

$$x \vee (a \wedge b) = (x \vee a) \wedge b$$

for every $a, b \in L$. We say L is modular if x is modular for all $x \in L$. There are other equivalent definitions, see Wikipedia article Modular_lattice.

See also:

`is_upper_semimodular()`, `is_lower_semimodular()` and `is_modular_element()`

EXAMPLES:

```
sage: L = posets.DiamondPoset(5)
sage: L.is_modular()
True

sage: L = posets.PentagonPoset()
sage: L.is_modular()
False

sage: L = posets.ChainPoset(6)
sage: L.is_modular()
True

sage: L = LatticePoset({1:[2,3],2:[4,5],3:[5,6],4:[7],5:[7],6:[7]})
sage: L.is_modular()
False
sage: [L.is_modular([x]) for x in L]
[True, True, False, True, True, False, True]
```

ALGORITHM:

Based on pp. 286-287 of Enumerative Combinatorics, Vol 1 [EnumComb1].

is_modular_element(*x*)

Return `True` if `x` is a modular element and `False` otherwise.

INPUT:

- `x` – an element of the lattice

An element x in a lattice L is *modular* if $x \leq b$ implies

$$x \vee (a \wedge b) = (x \vee a) \wedge b$$

for every $a, b \in L$.

See also:

`is_modular()` to check modularity for the full lattice or some set of elements

EXAMPLES:

```
sage: L = LatticePoset({1:[2,3],2:[4,5],3:[5,6],4:[7],5:[7],6:[7]})
sage: L.is_modular()
False
sage: [L.is_modular_element(x) for x in L]
[True, True, False, True, True, False, True]
```

is_planar()

Return `True` if the lattice is *upward* planar, and `False` otherwise.

A lattice is upward planar if its Hasse diagram has a planar drawing in the $\mathbb{R}^2$ plane, in such a way that x is strictly below y (on the vertical axis) whenever $x < y$ in the lattice.

Note that the scientific litterature on posets often omits "upward" and shortens it to "planar lattice" (e.g. [GW14]), which can cause confusion with the notion of graph planarity in graph theory.

Note: Not all lattices which are planar – in the sense of graph planarity – admit such a planar drawing (see example below).

ALGORITHM:

Using the result from [Platt76], this method returns its result by testing that the Hasse diagram of the lattice is planar (in the sense of graph theory) when an edge is added between the top and bottom elements.

EXAMPLES:

The Boolean lattice of 2^3 elements is not upward planar, even if it's covering relations graph is planar:

```
sage: B3 = Posets.BooleanLattice(3)
sage: B3.is_planar()
False
sage: G = B3.cover_relations_graph()
sage: G.is_planar()
True
```

Ordinal product of planar lattices is obviously planar. Same does not apply to Cartesian products:

```
sage: P = Posets.PentagonPoset()
sage: Pc = P.product(P)
sage: Po = P.ordinal_product(P)
sage: Pc.is_planar()
False
sage: Po.is_planar()
True
```

TESTS:

```
sage: Posets.ChainPoset(0).is_planar()
True
sage: Posets.ChainPoset(1).is_planar()
True
```

REFERENCES:

is_pseudocomplemented()

Return `True` if the lattice is pseudocomplemented, and `False` otherwise.

A lattice is pseudocomplemented if every element e has a pseudocomplement $e^\star$, i.e. the greatest element such that the meet of e and $e^\star$ is the bottom element.

See Wikipedia article Pseudocomplement.

EXAMPLES:

```
sage: L = LatticePoset({1: [2, 5], 2: [3, 6], 3: [4], 4: [7],
....:                   5: [6], 6: [7]})
sage: L.is_pseudocomplemented()
True

sage: L = LatticePoset({1: [2, 3], 2: [4, 5, 6], 3: [6], 4: [7],
....:                   5: [7], 6: [7]})
sage: L.is_pseudocomplemented()  # Element 3 has no pseudocomplement
False
```

See also:

`sage.combinat.posets.lattices.FiniteMeetSemilattice.pseudocomplement()`.

TESTS:

```
sage: LatticePoset({}).is_pseudocomplemented()
True
```

is_supersolvable()

Return `True` if `self` is a supersolvable lattice and `False` otherwise.

A lattice L is *supersolvable* if there exists a maximal chain C such that every $x \in C$ is a modular element in L.

EXAMPLES:

```
sage: L = posets.DiamondPoset(5)
sage: L.is_supersolvable()
True

sage: L = posets.PentagonPoset()
sage: L.is_supersolvable()
False

sage: L = posets.ChainPoset(6)
sage: L.is_supersolvable()
True

sage: L = LatticePoset({1:[2,3],2:[4,5],3:[5,6],4:[7],5:[7],6:[7]})
sage: L.is_supersolvable()
True
sage: L.is_modular()
False

sage: L = LatticePoset({0: [1, 2, 3, 4], 1: [5, 6, 7],
....:                   2: [5, 8, 9], 3: [6, 8, 10], 4: [7, 9, 10],
....:                   5: [11], 6: [11], 7: [11], 8: [11],
....:                   9: [11], 10: [11]})
sage: L.is_supersolvable()
False
```

is_upper_semimodular()

Return `True` if the lattice is upper semimodular and `False` otherwise.

A lattice is upper semimodular if for any x in the lattice that is covered by y and z, both y and z are covered by their join.

See also `is_modular()` and `is_lower_semimodular()`.

See Wikipedia article Semimodular_lattice

EXAMPLES:

```
sage: L = posets.DiamondPoset(5)
sage: L.is_upper_semimodular()
True

sage: L = posets.PentagonPoset()
sage: L.is_upper_semimodular()
False

sage: L = posets.ChainPoset(6)
sage: L.is_upper_semimodular()
True

sage: L = LatticePoset(posets.IntegerPartitions(4))
sage: L.is_upper_semimodular()
True
```

ALGORITHM:

Based on pp. 286-287 of Enumerative Combinatorics, Vol 1 [EnumComb1].

maximal_sublattices()

Return maximal (proper) sublattices of the lattice.

EXAMPLES:

```
sage: L = LatticePoset(( [], [[1,2],[1,17],[1,8],[2,3],[2,22],
....:                         [2,5],[2,7],[17,22],[17,13],[8,7],
....:                         [8,13],[3,16],[3,9],[22,16],[22,18],
....:                         [22,10],[5,18],[5,14],[7,9],[7,14],
....:                         [7,10],[13,10],[16,6],[16,19],[9,19],
....:                         [18,6],[18,33],[14,33],[10,19],
....:                         [10,33],[6,4],[19,4],[33,4]] ))
sage: maxs = L.maximal_sublattices()
sage: len(maxs)
7
sage: sorted(maxs[0].list())
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14, 16, 18, 19, 22, 33]
```

moebius_algebra(*R*)

Return the Möbius algebra of `self` over `R`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: L.moebius_algebra(QQ)
Moebius algebra of Finite lattice containing 16 elements over Rational Field
```

quantum_moebius_algebra(*q=None*)

Return the quantum Möbius algebra of `self` with parameter `q`.

INPUT:

- `q` – (optional) the deformation parameter q

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: L.quantum_moebius_algebra()
Quantum Moebius algebra of Finite lattice containing 16 elements
 with q=q over Univariate Laurent Polynomial Ring in q over Integer Ring
```

sublattice(*elms*)

Return the smallest sublattice containing elements on the given list.

INPUT:

- elms – a list of elements of the lattice.

EXAMPLES:

```
sage: L=LatticePoset(( [], [[1,2],[1,17],[1,8],[2,3],[2,22],[2,5],[2,7],[17,22],[17,13],[8,7
sage: L.sublattice([14, 13, 22]).list()
[1, 2, 8, 7, 14, 17, 13, 22, 10, 33]

sage: L = Posets.BooleanLattice(3)
sage: L.sublattice([3,5,6,7])
Finite lattice containing 8 elements
```

Note: This is very unoptimal algorithm. Better one is described on "Computing the sublattice of a lattice generated by a set of elements" by K. Bertet and M. Morvan. Feel free to implement it.

---

class sage.combinat.posets.lattices.**FiniteMeetSemilattice**(*hasse_diagram*, *elements*, *category*, *facade*, *key*)

Bases: sage.combinat.posets.posets.FinitePoset

Note: We assume that the argument passed to MeetSemilattice is the poset of a meet-semilattice (i.e. a poset with greatest lower bound for each pair of elements).

TESTS:

```
sage: M = MeetSemilattice([[1,2],[3],[3]])
sage: TestSuite(M).run()

sage: P = Poset([[1,2],[3],[3]])
sage: M = MeetSemilattice(P)
sage: TestSuite(M).run()
```

Element

alias of MeetSemilatticeElement

meet(*x*, *y=None*)

Return the meet of given elements in the lattice.

EXAMPLES:

```
sage: D = Posets.DiamondPoset(5)
sage: D.meet(1, 2)
0
sage: D.meet(1, 1)
1
sage: D.meet(1, 0)
0
sage: D.meet(1, 4)
1
```

Using list of elements as an argument. Meet of empty list is the bottom element:

```
sage: B4=Posets.BooleanLattice(4)
sage: B4.meet([3,5,6])
0
sage: B4.meet([])
15
```

Test that this method also works for facade lattices:

```
sage: L = LatticePoset([[1,2],[3],[3]], facade = True)
sage: L.meet(2, 3)
2
sage: L.meet(1, 2)
0
```

meet_matrix()

Return a matrix whose `(i,j)` entry is `k`, where `self.linear_extension()[k]` is the meet (greatest lower bound) of `self.linear_extension()[i]` and `self.linear_extension()[j]`.

EXAMPLES:

```
sage: P = LatticePoset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]], facade = False)
sage: M = P.meet_matrix(); M
[0 0 0 0 0 0 0 0]
[0 1 0 1 0 0 0 1]
[0 0 2 2 2 0 2 2]
[0 1 2 3 2 0 2 3]
[0 0 2 2 4 0 2 4]
[0 0 0 0 0 5 5 5]
[0 0 2 2 2 5 6 6]
[0 1 2 3 4 5 6 7]
sage: M[P(4).vertex,P(3).vertex] == P(0).vertex
True
sage: M[P(5).vertex,P(2).vertex] == P(2).vertex
True
sage: M[P(5).vertex,P(2).vertex] == P(5).vertex
False
```

pseudocomplement(*element*)

Return the pseudocomplement of `element`, if it exists.

The pseudocomplement is the greatest element whose meet with given element is the bottom element. I.e. in a meet-semilattice with bottom element $\hat{0}$ the pseudocomplement of an element e is the element $e^\star$ such that $e \wedge e^\star = \hat{0}$ and $e' \leq e^\star$ if $e \wedge e' = \hat{0}$.

See Wikipedia article Pseudocomplement.

INPUT:

- `element` – an element of the lattice.

OUTPUT:

An element of the lattice or `None` if the pseudocomplement does not exist.

EXAMPLES:

The pseudocompelement's pseudocomplement is not always the original element:

```
sage: L = LatticePoset({1: [2, 3], 2: [4], 3: [5], 4: [6], 5: [6]})
sage: L.pseudocomplement(2)
5
```

```
sage: L.pseudocomplement(5)
4
```

An element can have complements but no pseudocomplement, or vice versa:

```
sage: L = LatticePoset({0: [1, 2], 1: [3, 4, 5], 2: [5], 3: [6],
....:                   4: [6], 5: [6]})
sage: L.complements(1), L.pseudocomplement(1)
([], 2)
sage: L.complements(2), L.pseudocomplement(2)
([3, 4], None)
```

See also:

sage.combinat.posets.lattices.FiniteLatticePoset.is_pseudocomplemented().

TESTS:

```
sage: L = LatticePoset({'a': []})
sage: L.pseudocomplement('a')
'a'
sage: L = LatticePoset({'a': ['b'], 'b': ['c']})
sage: [L.pseudocomplement(e) for e in ['a', 'b', 'c']]
['c', 'a', 'a']
```

sage.combinat.posets.lattices.**JoinSemilattice**(*data*, **args*, ***options*)

Construct a join semi-lattice from various forms of input data.

INPUT:

- data, *args, **options – data and options that will be passed down to Poset() to construct a poset that is also a join semilattice.

See also:

Poset(), MeetSemilattice(), LatticePoset()

EXAMPLES:

Using data that defines a poset:

```
sage: JoinSemilattice([[1,2],[3],[3]])
Finite join-semilattice containing 4 elements

sage: JoinSemilattice([[1,2],[3],[3]], cover_relations = True)
Finite join-semilattice containing 4 elements
```

Using a previously constructed poset:

```
sage: P = Poset([[1,2],[3],[3]])
sage: J = JoinSemilattice(P); J
Finite join-semilattice containing 4 elements
sage: type(J)
<class 'sage.combinat.posets.lattices.FiniteJoinSemilattice_with_category'>
```

If the data is not a lattice, then an error is raised:

```
sage: elms = [1,2,3,4,5,6,7]
sage: rels = [[1,2],[3,4],[4,5],[2,5]]
sage: JoinSemilattice((elms, rels))
Traceback (most recent call last):
...
ValueError: Not a join semilattice.
```

sage.combinat.posets.lattices.**LatticePoset**(*data*, **args*, ***options*)

Construct a lattice from various forms of input data.

INPUT:

- `data`, `*args`, `**options` – data and options that will be passed down to `Poset()` to construct a poset that is also a lattice.

OUTPUT:

FiniteLatticePoset – an instance of `FiniteLatticePoset`

See also:

`Posets`, `FiniteLatticePosets`, `JoinSemiLattice()`, `MeetSemiLattice()`

EXAMPLES:

Using data that defines a poset:

```
sage: LatticePoset([[1,2],[3],[3]])
Finite lattice containing 4 elements

sage: LatticePoset([[1,2],[3],[3]], cover_relations = True)
Finite lattice containing 4 elements
```

Using a previously constructed poset:

```
sage: P = Poset([[1,2],[3],[3]])
sage: L = LatticePoset(P); L
Finite lattice containing 4 elements
sage: type(L)
<class 'sage.combinat.posets.lattices.FiniteLatticePoset_with_category'>
```

If the data is not a lattice, then an error is raised:

```
sage: elms = [1,2,3,4,5,6,7]
sage: rels = [[1,2],[3,4],[4,5],[2,5]]
sage: LatticePoset((elms, rels))
Traceback (most recent call last):
...
ValueError: Not a lattice.
```

Creating a facade lattice:

```
sage: L = LatticePoset([[1,2],[3],[3]], facade = True)
sage: L.category()
Join of Category of finite lattice posets and Category of finite enumerated sets and Category of
sage: parent(L[0])
Integer Ring
sage: TestSuite(L).run(skip = ['_test_an_element']) # is_parent_of is not yet implemented
```

sage.combinat.posets.lattices.**MeetSemilattice**(*data*, **args*, ***options*)

Construct a meet semi-lattice from various forms of input data.

INPUT:

- `data`, `*args`, `**options` – data and options that will be passed down to `Poset()` to construct a poset that is also a meet semilattice.

See also:

Poset(), JoinSemilattice(), LatticePoset()

EXAMPLES:

Using data that defines a poset:

```
sage: MeetSemilattice([[1,2],[3],[3]])
Finite meet-semilattice containing 4 elements

sage: MeetSemilattice([[1,2],[3],[3]], cover_relations = True)
Finite meet-semilattice containing 4 elements
```

Using a previously constructed poset:

```
sage: P = Poset([[1,2],[3],[3]])
sage: L = MeetSemilattice(P); L
Finite meet-semilattice containing 4 elements
sage: type(L)
<class 'sage.combinat.posets.lattices.FiniteMeetSemilattice_with_category'>
```

If the data is not a lattice, then an error is raised:

```
sage: elms = [1,2,3,4,5,6,7]
sage: rels = [[1,2],[3,4],[4,5],[2,5]]
sage: MeetSemilattice((elms, rels))
Traceback (most recent call last):
...
ValueError: Not a meet semilattice.
```

5.1.156 Linear Extensions of Posets

This module defines two classes:

- LinearExtensionOfPoset
- LinearExtensionsOfPoset

Classes and methods

class sage.combinat.posets.linear_extensions.**LinearExtensionOfPoset**

Bases: sage.structure.list_clone.ClonableArray

A linear extension of a finite poset P of size n is a total ordering $\pi := \pi_0\pi_1\ldots\pi_{n-1}$ of its elements such that $i < j$ whenever $\pi_i < \pi_j$ in the poset P.

When the elements of P are indexed by $\{1, 2, \ldots, n\}$, π denotes a permutation of the elements of P in one-line notation.

INPUT:

- linear_extension – a list of the elements of P
- poset – the underlying poset P

See also:

Poset, LinearExtensionsOfPoset

EXAMPLES:

```
sage: P = Poset(([1,2,3,4], [[1,3],[1,4],[2,3]]), linear_extension=True, facade=False)
sage: p = P.linear_extension([1,4,2,3]); p
[1, 4, 2, 3]
sage: p.parent()
The set of all linear extensions of Finite poset containing 4 elements with distinguished linear
sage: p[0], p[1], p[2], p[3]
(1, 4, 2, 3)
```

Following Schützenberger and later Haiman and Malvenuto-Reutenauer, Stanley [Stan2009] defined a promotion and evacuation operator on any finite poset P using operators τ_i on the linear extensions of P:

```
sage: p.promotion()
[1, 2, 3, 4]
sage: Q = p.promotion().to_poset()
sage: Q.cover_relations()
[[1, 3], [1, 4], [2, 3]]
sage: Q == P
True

sage: p.promotion(3)
[1, 4, 2, 3]
sage: Q = p.promotion(3).to_poset()
sage: Q == P
False
sage: Q.cover_relations()
[[1, 2], [1, 4], [3, 4]]
```

check()

Checks whether `self` is indeed a linear extension of the underlying poset.

TESTS:

```
sage: P = Poset(([1,2,3,4], [[1,3],[1,4],[2,3]]))
sage: P.linear_extension([1,4,2,3])
[1, 4, 2, 3]
sage: P.linear_extension([4,3,2,1])
Traceback (most recent call last):
...
ValueError: [4, 3, 2, 1] is not a linear extension of Finite poset containing 4 elements
```

evacuation()

Computes evacuation on the linear extension of a poset.

Evacuation on a linear extension π of length n is defined as $\pi(\tau_1 \cdots \tau_{n-1})(\tau_1 \cdots \tau_{n-2}) \cdots (\tau_1)$. For more details see [Stan2009].

See also:

`tau()`, `promotion()`

EXAMPLES:

```
sage: P = Poset(([1,2,3,4,5,6,7], [[1,2],[1,4],[2,3],[2,5],[3,6],[4,7],[5,6]]))
sage: p = P.linear_extension([1,2,3,4,5,6,7])
sage: p.evacuation()
[1, 4, 2, 3, 7, 5, 6]
sage: p.evacuation().evacuation() == p
True
```

poset()

Returns the underlying original poset.

EXAMPLES:

```
sage: P = Poset(([1,2,3,4], [[1,2],[2,3],[1,4]]))
sage: p = P.linear_extension([1,2,4,3])
sage: p.poset()
Finite poset containing 4 elements
```

promotion(*i=1*)

Computes the (generalized) promotion on the linear extension of a poset.

INPUT:

- i – an integer between 1 and $n-1$, where n is the cardinality of the poset (default: 1)

The i-th generalized promotion operator ∂_i on a linear extension π is defined as $\pi\tau_i\tau_{i+1}\cdots\tau_{n-1}$, where n is the size of the linear extension (or size of the underlying poset).

For more details see [Stan2009].

See also:

`tau()`, `evacuation()`

EXAMPLES:

```
sage: P = Poset(([1,2,3,4,5,6,7], [[1,2],[1,4],[2,3],[2,5],[3,6],[4,7],[5,6]]))
sage: p = P.linear_extension([1,2,3,4,5,6,7])
sage: q = p.promotion(4); q
[1, 2, 3, 5, 6, 4, 7]
sage: p.to_poset() == q.to_poset()
False
sage: p.to_poset().is_isomorphic(q.to_poset())
True
```

tau(*i*)

Returns the operator τ_i on linear extensions `self` of a poset.

INPUT:

- i – an integer between 1 and $n-1$, where n is the cardinality of the poset.

The operator τ_i on a linear extension π of a poset P interchanges positions i and $i+1$ if the result is again a linear extension of P, and otherwise acts trivially. For more details, see [Stan2009].

EXAMPLES:

```
sage: P = Poset(([1,2,3,4], [[1,3],[1,4],[2,3]]), linear_extension=True)
sage: L = P.linear_extensions()
sage: l = L.an_element(); l
[1, 2, 3, 4]
sage: l.tau(1)
[2, 1, 3, 4]
sage: for p in L:
....:     for i in range(1,4):
....:         print i, p, p.tau(i)
....:
1 [1, 2, 3, 4] [2, 1, 3, 4]
2 [1, 2, 3, 4] [1, 2, 3, 4]
3 [1, 2, 3, 4] [1, 2, 4, 3]
1 [1, 2, 4, 3] [2, 1, 4, 3]
2 [1, 2, 4, 3] [1, 4, 2, 3]
3 [1, 2, 4, 3] [1, 2, 3, 4]
```

```
1 [1, 4, 2, 3] [1, 4, 2, 3]
2 [1, 4, 2, 3] [1, 2, 4, 3]
3 [1, 4, 2, 3] [1, 4, 2, 3]
1 [2, 1, 3, 4] [1, 2, 3, 4]
2 [2, 1, 3, 4] [2, 1, 3, 4]
3 [2, 1, 3, 4] [2, 1, 4, 3]
1 [2, 1, 4, 3] [1, 2, 4, 3]
2 [2, 1, 4, 3] [2, 1, 4, 3]
3 [2, 1, 4, 3] [2, 1, 3, 4]
```

TESTS:

```
sage: type(l.tau(1))
<class 'sage.combinat.posets.linear_extensions.LinearExtensionsOfPoset_with_category.element
sage: l.tau(2) == l
True
```

to_poset()

Return the poset associated to the linear extension `self`.

This method returns the poset obtained from the original poset P by relabelling the i-th element of `self` to the i-th element of the original poset, while keeping the linear extension of the original poset.

For a poset with default linear extension $1, \ldots, n$, `self` can be interpreted as a permutation, and the relabelling is done according to the inverse of this permutation.

EXAMPLES:

```
sage: P = Poset(([1,2,3,4], [[1,2],[1,3],[3,4]]), linear_extension=True, facade=False)
sage: p = P.linear_extension([1,3,4,2])
sage: Q = p.to_poset(); Q
Finite poset containing 4 elements with distinguished linear extension
sage: P == Q
False
```

The default linear extension remains the same:

```
sage: list(P)
[1, 2, 3, 4]
sage: list(Q)
[1, 2, 3, 4]
```

But the relabelling can be seen on cover relations:

```
sage: P.cover_relations()
[[1, 2], [1, 3], [3, 4]]
sage: Q.cover_relations()
[[1, 2], [1, 4], [2, 3]]

sage: p = P.linear_extension([1,2,3,4])
sage: Q = p.to_poset()
sage: P == Q
True
```

class sage.combinat.posets.linear_extensions.**LinearExtensionsOfPoset**(*poset*, *facade*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The set of all linear extensions of a finite poset

INPUT:

- poset – a poset P of size n
- facade – a boolean (default: False)

See also:

- sage.combinat.posets.posets.FinitePoset.linear_extensions()
- sage.graphs.linearextensions.LinearExtensions

EXAMPLES:

```
sage: elms = [1,2,3,4]
sage: rels = [[1,3],[1,4],[2,3]]
sage: P = Poset((elms, rels), linear_extension=True)
sage: L = P.linear_extensions(); L
The set of all linear extensions of Finite poset containing 4 elements with distinguished linear
sage: L.cardinality()
5
sage: L.list()
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [2, 1, 3, 4], [2, 1, 4, 3]]
sage: L.an_element()
[1, 2, 3, 4]
sage: L.poset()
Finite poset containing 4 elements with distinguished linear extension
```

Element
alias of LinearExtensionOfPoset

markov_chain_digraph(*action='promotion'*, *labeling='identity'*)
Returns the digraph of the action of generalized promotion or tau on self

INPUT:

- action – 'promotion' or 'tau' (default: 'promotion')
- labeling – 'identity' or 'source' (default: 'identity')

Todo

- generalize this feature by accepting a family of operators as input
- move up in some appropriate category

This method creates a graph with vertices being the linear extensions of a given finite poset and an edge from π to π' if $\pi' = \pi\partial_i$ where ∂_i is the promotion operator (see promotion()) if action is set to promotion and τ_i (see tau()) if action is set to tau. The label of the edge is i (resp. π_i) if labeling is set to identity (resp. source).

EXAMPLES:

```
sage: P = Poset(([1,2,3,4], [[1,3],[1,4],[2,3]]), linear_extension = True)
sage: L = P.linear_extensions()
sage: G = L.markov_chain_digraph(); G
Looped multi-digraph on 5 vertices
sage: sorted(G.vertices(), key = repr)
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [2, 1, 3, 4], [2, 1, 4, 3]]
sage: sorted(G.edges(), key = repr)
[([1, 2, 3, 4], [1, 2, 3, 4], 4), ([1, 2, 3, 4], [1, 2, 4, 3], 2), ([1, 2, 3, 4], [1, 2, 4,
([1, 2, 3, 4], [2, 1, 4, 3], 1), ([1, 2, 4, 3], [1, 2, 3, 4], 3), ([1, 2, 4, 3], [1, 2, 4, 3
```

```
([1, 2, 4, 3], [1, 4, 2, 3], 2), ([1, 2, 4, 3], [2, 1, 3, 4], 1), ([1, 4, 2, 3], [1, 2, 3, 4
([1, 4, 2, 3], [1, 2, 3, 4], 2), ([1, 4, 2, 3], [1, 4, 2, 3], 3), ([1, 4, 2, 3], [1, 4, 2, 3
([2, 1, 3, 4], [1, 2, 4, 3], 1), ([2, 1, 3, 4], [2, 1, 3, 4], 4), ([2, 1, 3, 4], [2, 1, 4, 3
([2, 1, 3, 4], [2, 1, 4, 3], 3), ([2, 1, 4, 3], [1, 4, 2, 3], 1), ([2, 1, 4, 3], [2, 1, 3, 4
([2, 1, 4, 3], [2, 1, 3, 4], 3), ([2, 1, 4, 3], [2, 1, 4, 3], 4)]

sage: G = L.markov_chain_digraph(labeling = 'source')
sage: sorted(G.vertices(), key = repr)
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [2, 1, 3, 4], [2, 1, 4, 3]]
sage: sorted(G.edges(), key = repr)
[([1, 2, 3, 4], [1, 2, 3, 4], 4), ([1, 2, 3, 4], [1, 2, 4, 3], 2), ([1, 2, 3, 4], [1, 2, 4,
([1, 2, 3, 4], [2, 1, 4, 3], 1), ([1, 2, 4, 3], [1, 2, 3, 4], 4), ([1, 2, 4, 3], [1, 2, 4, 3
([1, 2, 4, 3], [1, 4, 2, 3], 2), ([1, 2, 4, 3], [2, 1, 3, 4], 1), ([1, 4, 2, 3], [1, 2, 3, 4
([1, 4, 2, 3], [1, 2, 3, 4], 4), ([1, 4, 2, 3], [1, 4, 2, 3], 2), ([1, 4, 2, 3], [1, 4, 2, 3
([2, 1, 3, 4], [1, 2, 4, 3], 2), ([2, 1, 3, 4], [2, 1, 3, 4], 4), ([2, 1, 3, 4], [2, 1, 4, 3
([2, 1, 3, 4], [2, 1, 4, 3], 3), ([2, 1, 4, 3], [1, 4, 2, 3], 2), ([2, 1, 4, 3], [2, 1, 3, 4
([2, 1, 4, 3], [2, 1, 3, 4], 4), ([2, 1, 4, 3], [2, 1, 4, 3], 3)]
```

The edges of the graph are by default colored using blue for edge 1, red for edge 2, green for edge 3, and yellow for edge 4:

```
sage: view(G) # optional - dot2tex graphviz, not tested (opens external window)
```

Alternatively, one may get the graph of the action of the `tau` operator:

```
sage: G = L.markov_chain_digraph(action='tau'); G
Looped multi-digraph on 5 vertices
sage: sorted(G.vertices(), key = repr)
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [2, 1, 3, 4], [2, 1, 4, 3]]
sage: sorted(G.edges(), key = repr)
[([1, 2, 3, 4], [1, 2, 3, 4], 2), ([1, 2, 3, 4], [1, 2, 4, 3], 3), ([1, 2, 3, 4], [2, 1, 3,
([1, 2, 4, 3], [1, 2, 3, 4], 3), ([1, 2, 4, 3], [1, 4, 2, 3], 2), ([1, 2, 4, 3], [2, 1, 4, 3
([1, 4, 2, 3], [1, 2, 4, 3], 2), ([1, 4, 2, 3], [1, 4, 2, 3], 1), ([1, 4, 2, 3], [1, 4, 2, 3
([2, 1, 3, 4], [1, 2, 3, 4], 1), ([2, 1, 3, 4], [2, 1, 3, 4], 2), ([2, 1, 3, 4], [2, 1, 4, 3
([2, 1, 4, 3], [1, 2, 4, 3], 1), ([2, 1, 4, 3], [2, 1, 3, 4], 3), ([2, 1, 4, 3], [2, 1, 4, 3
sage: view(G) # optional - dot2tex graphviz, not tested (opens external window)
```

See also:

`markov_chain_transition_matrix()`, `promotion()`, `tau()`

TESTS:

```
sage: P = Poset(([1,2,3,4], [[1,3],[1,4],[2,3]]), linear_extension = True, facade = True)
sage: L = P.linear_extensions()
sage: G = L.markov_chain_digraph(labeling = 'source'); G
Looped multi-digraph on 5 vertices
```

markov_chain_transition_matrix(*action='promotion'*, *labeling='identity'*)

Returns the transition matrix of the Markov chain for the action of generalized promotion or tau on `self`

INPUT:

- `action` – 'promotion' or 'tau' (default: 'promotion')
- `labeling` – 'identity' or 'source' (default: 'identity')

This method yields the transition matrix of the Markov chain defined by the action of the generalized promotion operator ∂_i (resp. τ_i) on the set of linear extensions of a finite poset. Here the transition from the linear extension π to π', where $\pi' = \pi\partial_i$ (resp. $\pi' = \pi\tau_i$) is counted with weight x_i (resp. x_{π_i} if `labeling` is set to `source`).

EXAMPLES:

```
sage: P = Poset(([1,2,3,4], [[1,3],[1,4],[2,3]]), linear_extension = True)
sage: L = P.linear_extensions()
sage: L.markov_chain_transition_matrix()
[-x0 - x1 - x2            x2       x0 + x1             0             0]
[      x1 + x2 -x0 - x1 - x2             0            x0             0]
[            0            x1      -x0 - x1             0            x0]
[            0            x0             0 -x0 - x1 - x2       x1 + x2]
[           x0             0             0       x1 + x2 -x0 - x1 - x2]

sage: L.markov_chain_transition_matrix(labeling = 'source')
[-x0 - x1 - x2            x3       x0 + x3             0             0]
[      x1 + x2 -x0 - x1 - x3             0            x1             0]
[            0            x1      -x0 - x3             0            x1]
[            0            x0             0 -x0 - x1 - x2       x0 + x3]
[           x0             0             0       x0 + x2 -x0 - x1 - x3]

sage: L.markov_chain_transition_matrix(action = 'tau')
[     -x0 - x2            x2             0            x0             0]
[           x2 -x0 - x1 - x2            x1             0            x0]
[            0            x1           -x1             0             0]
[           x0             0             0      -x0 - x2            x2]
[            0            x0             0            x2      -x0 - x2]

sage: L.markov_chain_transition_matrix(action = 'tau', labeling = 'source')
[     -x0 - x2            x3             0            x1             0]
[           x2 -x0 - x1 - x3            x3             0            x1]
[            0            x1           -x3             0             0]
[           x0             0             0      -x1 - x2            x3]
[            0            x0             0            x2      -x1 - x3]
```

See also:

markov_chain_digraph(),promotion(),tau()

poset()

Returns the underlying original poset.

EXAMPLES:

```
sage: P = Poset(([1,2,3,4], [[1,2],[2,3],[1,4]]))
sage: L = P.linear_extensions()
sage: L.poset()
Finite poset containing 4 elements
```

5.1.157 Möbius Algebras

class sage.combinat.posets.moebius_algebra.**BasisAbstract**(*R*, *basis_keys*, *element_class=None*, *category=None*, *prefix='B'*, ***kwds*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

Abstract base class for a basis.

class sage.combinat.posets.moebius_algebra.**MoebiusAlgebra**(*R*, *L*)

Bases: sage.structure.parent.Parent, sage.structure.unique_representation.UniqueRepresenta

The Möbius algebra of a lattice.

Let L be a lattice. The *Möbius algebra* M_L was originally constructed by Solomon and has a natural basis $\{E_x \mid x \in L\}$ with multiplication given by $E_x \cdot E_y = E_{x \vee y}$. Moreover this has a basis given by orthogonal idempotents $\{I_x \mid x \in L\}$ (so $I_x I_y = \delta_{xy} I_x$ where δ is the Kronecker delta) related to the natural basis by

$$I_x = \sum_{y \leq x} \mu_L(y, x) E_x,$$

where μ_L is the Möbius function of L.

REFERENCES:

class **E**(*M*, *prefix='E'*)

Bases: sage.combinat.posets.moebius_algebra.BasisAbstract

The natural basis of a Möbius algebra.

Let E_x and E_y be basis elements of M_L for some lattice L. Multiplication is given by $E_x E_y = E_{x \vee y}$.

one()

Return the element `1` of `self`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.moebius_algebra(QQ).E()
sage: E.one()
E[0]
```

product_on_basis(*x*, *y*)

Return the product of basis elements indexed by `x` and `y`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.moebius_algebra(QQ).E()
sage: E.product_on_basis(5, 14)
E[15]
sage: E.product_on_basis(2, 8)
E[10]
```

class MoebiusAlgebra.**I**(*M*, *prefix='I'*)

Bases: sage.combinat.posets.moebius_algebra.BasisAbstract

The (orthogonal) idempotent basis of a Möbius algebra.

Let I_x and I_y be basis elements of M_L for some lattice L. Multiplication is given by $I_x I_y = \delta_{xy} I_x$ where δ_{xy} is the Kronecker delta.

one()

Return the element `1` of `self`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: I = L.moebius_algebra(QQ).I()
sage: I.one()
I[0] + I[1] + I[2] + I[3] + I[4] + I[5] + I[6] + I[7] + I[8]
 + I[9] + I[10] + I[11] + I[12] + I[13] + I[14] + I[15]
```

product_on_basis(*x*, *y*)

Return the product of basis elements indexed by `x` and `y`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: I = L.moebius_algebra(QQ).I()
sage: I.product_on_basis(5, 14)
0
sage: I.product_on_basis(2, 2)
I[2]
```

MoebiusAlgebra.**a_realization**()

Return a particular realization of `self` (the B-basis).

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.moebius_algebra(QQ)
sage: M.a_realization()
Moebius algebra of Finite lattice containing 16 elements
 over Rational Field in the natural basis
```

MoebiusAlgebra.**lattice**()

Return the defining lattice of `self`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.moebius_algebra(QQ)
sage: M.lattice()
Finite lattice containing 16 elements
sage: M.lattice() == L
True
```

class sage.combinat.posets.moebius_algebra.**MoebiusAlgebraBases**(*parent_with_realization*)

Bases: sage.categories.realizations.Category_realization_of_parent

The category of bases of a Möbius algebra.

INPUT:

- `base` – a Möbius algebra

TESTS:

```
sage: from sage.combinat.posets.moebius_algebra import MoebiusAlgebraBases
sage: M = posets.BooleanLattice(4).moebius_algebra(QQ)
sage: bases = MoebiusAlgebraBases(M)
sage: M.E() in bases
True
```

class ElementMethods

class MoebiusAlgebraBases.**ParentMethods**

one()

Return the element `1` of `self`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: C = L.quantum_moebius_algebra().C()
sage: all(C.one() * b == b for b in C.basis())
True
```

product_on_basis(*x*, *y*)

Return the product of basis elements indexed by `x` and `y`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: C = L.quantum_moebius_algebra().C()
sage: C.product_on_basis(5, 14)
q^3*C[15]
sage: C.product_on_basis(2, 8)
q^4*C[10]
```

MoebiusAlgebraBases.**super_categories**()

The super categories of `self`.

EXAMPLES:

```
sage: from sage.combinat.posets.moebius_algebra import MoebiusAlgebraBases
sage: M = posets.BooleanLattice(4).moebius_algebra(QQ)
sage: bases = MoebiusAlgebraBases(M)
sage: bases.super_categories()
[Category of finite dimensional commutative algebras with basis over Rational Field,
 Category of realizations of Moebius algebra of Finite lattice
    containing 16 elements over Rational Field]
```

class sage.combinat.posets.moebius_algebra.**QuantumMoebiusAlgebra**(*L*, *q=None*)

Bases: sage.structure.parent.Parent, sage.structure.unique_representation.UniqueRepresenta

The quantum Möbius algebra of a lattice.

Let L be a lattice, and we define the *quantum Möbius algebra* $M_L(q)$ as the algebra with basis $\{E_x \mid x \in L\}$ with multiplication given by

$$E_x E_y = \sum_{z \geq a \geq x \vee y} \mu_L(a, z) q^{\operatorname{crk} a} E_z,$$

where μ_L is the Möbius function of L and crk is the corank function (i.e., $\operatorname{crk} a = \operatorname{rank} L - \operatorname{rank}$ a). At $q = 1$, this reduces to the multiplication formula originally given by Solomon.

class C(*M*, *prefix='C'*)

Bases: sage.combinat.posets.moebius_algebra.BasisAbstract

The characteristic basis of a quantum Möbius algebra.

The characteristic basis $\{C_x \mid x \in L\}$ of M_L for some lattice L is defined by

$$C_x = \sum_{a \geq x} P(F^x; q) E_a,$$

where $F^x = \{y \in L \mid y \geq x\}$ is the principal order filter of x and $P(F^x; q)$ is the characteristic polynomial of the (sub)poset F^x.

class QuantumMoebiusAlgebra.**E**(*M*, *prefix='E'*)

Bases: sage.combinat.posets.moebius_algebra.BasisAbstract

The natural basis of a quantum Möbius algebra.

Let E_x and E_y be basis elements of M_L for some lattice L. Multiplication is given by

$$E_x E_y = \sum_{z \geq a \geq x \vee y} \mu_L(a, z) q^{\operatorname{crk} a} E_z,$$

where μ_L is the Möbius function of L and crk is the corank function (i.e., $\operatorname{crk} a = \operatorname{rank} L - \operatorname{rank}$ a).

one()

Return the element `1` of `self`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.quantum_moebius_algebra().E()
sage: all(E.one() * b == b for b in E.basis())
True
```

product_on_basis(*x*, *y*)

Return the product of basis elements indexed by `x` and `y`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.quantum_moebius_algebra().E()
sage: E.product_on_basis(5, 14)
E[15]
sage: E.product_on_basis(2, 8)
q^2*E[10] + (q-q^2)*E[11] + (q-q^2)*E[14] + (1-2*q+q^2)*E[15]
```

class QuantumMoebiusAlgebra.**KL**(*M*, *prefix='KL'*)

Bases: sage.combinat.posets.moebius_algebra.BasisAbstract

The Kazhdan-Lusztig basis of a quantum Möbius algebra.

The Kazhdan-Lusztig basis $\{B_x \mid x \in L\}$ of M_L for some lattice L is defined by

$$B_x = \sum_{y \geq x} P_{x,y}(q) E_a,$$

where $P_{x,y}(q)$ is the Kazhdan-Lusztig polynomial of L, following the definition given in [EPW14].

EXAMPLES:

We construct some examples of Proposition 4.5 of [EPW14]:

```
sage: M = posets.BooleanLattice(4).quantum_moebius_algebra()
sage: KL = M.KL()
sage: KL[4] * KL[5]
(q^2+q^3)*KL[5] + (q+2*q^2+q^3)*KL[7] + (q+2*q^2+q^3)*KL[13]
 + (1+3*q+3*q^2+q^3)*KL[15]
sage: KL[4] * KL[15]
(1+3*q+3*q^2+q^3)*KL[15]
sage: KL[4] * KL[10]
(q+3*q^2+3*q^3+q^4)*KL[14] + (1+4*q+6*q^2+4*q^3+q^4)*KL[15]
```

QuantumMoebiusAlgebra.**a_realization**()

Return a particular realization of `self` (the B-basis).

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.quantum_moebius_algebra()
sage: M.a_realization()
Quantum Moebius algebra of Finite lattice containing 16 elements
 with q=q over Univariate Laurent Polynomial Ring in q
 over Integer Ring in the natural basis
```

QuantumMoebiusAlgebra.**lattice**()

Return the defining lattice of `self`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.quantum_moebius_algebra()
sage: M.lattice()
```

```
Finite lattice containing 16 elements
sage: M.lattice() == L
True
```

5.1.158 A catalog of posets and lattices.

Some common posets can be accessed through the `posets.<tab>` object:

```
sage: posets.PentagonPoset()
Finite lattice containing 5 elements
```

Moreover, the set of all posets of order n is represented by `Posets(n)`:

```
sage: Posets(5)
Posets containing 5 vertices
```

Catalog of common posets:

`AntichainPoset()`	Return an antichain on n elements.
`BooleanLattice()`	Return the Boolean lattice on 2^n elements.
`ChainPoset()`	Return a chain on n elements.
`DiamondPoset()`	Return the lattice of rank two on n elements.
`IntegerCompositions()`	Return the poset of integer compositions of n.
`IntegerPartitions()`	Return the poset of integer partitions of `n`.
`PentagonPoset()`	Return the Pentagon poset.
`RandomPoset()`	Return a random poset on n vertices according to a probability p.
`RestrictedIntegerPartitions()`	Return the poset of integer partitions of n, ordered by restricted refinement.
`ShardPoset()`	Return the shard intersection order.
`SSTPoset()`	Return the poset on semistandard tableaux of shape s and largest entry f that is ordered by componentwise comparison.
`StandardExample()`	Return the standard example of a poset with dimension n.
`SymmetricGroupBruhatIntervalPoset()`	The poset of permutations with respect to Bruhat order.
`SymmetricGroupBruhatOrderPoset()`	The poset of permutations with respect to Bruhat order.
`SymmetricGroupWeakOrderPoset()`	The poset of permutations of $\{1, 2, \ldots, n\}$ with respect to the weak order.
`TamariLattice()`	Return the Tamari lattice.
`TetrahedralPoset()`	Return the Tetrahedral poset with $n - 1$ layers based on the input colors.

Constructions

class `sage.combinat.posets.poset_examples.`**`Posets`**

Bases: `object`

A collection of posets and lattices.

EXAMPLES:

```
sage: Posets.BooleanLattice(3)
Finite lattice containing 8 elements
sage: Posets.ChainPoset(3)
Finite lattice containing 3 elements
sage: Posets.RandomPoset(17,.15)
Finite poset containing 17 elements
```

The category of all posets:

```
sage: Posets()
Category of posets
```

The enumerated set of all posets on 3 vertices, up to an isomorphism:

```
sage: Posets(3)
Posets containing 3 vertices
```

See also:

`Posets`, `FinitePosets`, `Poset()`

TESTS:

```
sage: P = Posets
sage: TestSuite(P).run()
```

static `AntichainPoset` (n)

Returns an antichain (a poset with no comparable elements) containing n elements.

EXAMPLES:

```
sage: A = Posets.AntichainPoset(6); A
Finite poset containing 6 elements
sage: for i in range(5):
...       for j in range(5):
...           if A.covers(A(i),A(j)):
...              print "TEST FAILED"
```

TESTS:

Check that trac ticket #8422 is solved:

```
sage: Posets.AntichainPoset(0)
Finite poset containing 0 elements
sage: C = Posets.AntichainPoset(1); C
Finite poset containing 1 elements
sage: C.cover_relations()
[]
sage: C = Posets.AntichainPoset(2); C
Finite poset containing 2 elements
sage: C.cover_relations()
[]
```

static `BooleanLattice` (n)

Returns the Boolean lattice containing 2^n elements.

EXAMPLES:

```
sage: Posets.BooleanLattice(5)
Finite lattice containing 32 elements
```

static `ChainPoset` (n)

Returns a chain (a totally ordered poset) containing `n` elements.

EXAMPLES:

```
sage: C = Posets.ChainPoset(6); C
Finite lattice containing 6 elements
sage: C.linear_extension()
[0, 1, 2, 3, 4, 5]
sage: for i in range(5):
```

```
...       for j in range(5):
...           if C.covers(C(i),C(j)) and j != i+1:
...              print "TEST FAILED"
```

TESTS:

Check that trac ticket #8422 is solved:

```
sage: Posets.ChainPoset(0)
Finite lattice containing 0 elements
sage: C = Posets.ChainPoset(1); C
Finite lattice containing 1 elements
sage: C.cover_relations()
[]
sage: C = Posets.ChainPoset(2); C
Finite lattice containing 2 elements
sage: C.cover_relations()
[[0, 1]]
```

static DiamondPoset(*n*, *facade=None*)

Return the lattice of rank two containing `n` elements.

INPUT:

- `n` - number of vertices, an integer at least 3.
- `facade` (boolean) – whether to make the returned poset a facade poset (see `sage.categories.facade_sets`). The default behaviour is the same as the default behaviour of the `Poset()` constructor).

EXAMPLES:

```
sage: Posets.DiamondPoset(7)
Finite lattice containing 7 elements
```

static IntegerCompositions(*n*)

Returns the poset of integer compositions of the integer `n`.

A composition of a positive integer n is a list of positive integers that sum to n. The order is reverse refinement: $[p_1, p_2, ..., p_l] < [q_1, q_2, ..., q_m]$ if q consists of an integer composition of p_1, followed by an integer composition of p_2, and so on.

EXAMPLES:

```
sage: P = Posets.IntegerCompositions(7); P
Finite poset containing 64 elements
sage: len(P.cover_relations())
192
```

static IntegerPartitions(*n*)

Returns the poset of integer partitions on the integer `n`.

A partition of a positive integer n is a non-increasing list of positive integers that sum to n. If p and q are integer partitions of n, then p covers q if and only if q is obtained from p by joining two parts of p (and sorting, if necessary).

EXAMPLES:

```
sage: P = Posets.IntegerPartitions(7); P
Finite poset containing 15 elements
sage: len(P.cover_relations())
28
```

static PentagonPoset (*facade=None*)

Returns the Pentagon poset.

INPUT:

- facade (boolean) – whether to make the returned poset a facade poset (see sage.categories.facade_sets). The default behaviour is the same as the default behaviour of the Poset() constructor).

EXAMPLES:

```
sage: P = Posets.PentagonPoset(); P
Finite lattice containing 5 elements
sage: P.cover_relations()
[[0, 1], [0, 2], [1, 4], [2, 3], [3, 4]]
```

This is smallest lattice that is not modular:

```
sage: P.is_modular()
False
```

This poset and the DiamondPoset() are the two smallest lattices which are not distributive:

```
sage: P.is_distributive()
False
sage: Posets.DiamondPoset(5).is_distributive()
False
```

static RandomPoset (*n*, *p*)

Generate a random poset on n vertices according to a probability p.

INPUT:

- n - number of vertices, a non-negative integer
- p - a probability, a real number between 0 and 1 (inclusive)

OUTPUT:

A poset on n vertices. The construction decides to make an ordered pair of vertices comparable in the poset with probability p, however a pair is not made comparable if it would violate the defining properties of a poset, such as transitivity.

So in practice, once the probability exceeds a small number the generated posets may be very similar to a chain. So to create interesting examples, keep the probability small, perhaps on the order of $1/n$.

EXAMPLES:

```
sage: Posets.RandomPoset(17,.15)
Finite poset containing 17 elements
```

TESTS:

```
sage: Posets.RandomPoset('junk', 0.5)
Traceback (most recent call last):
...
TypeError: number of elements must be an integer, not junk

sage: Posets.RandomPoset(-6, 0.5)
Traceback (most recent call last):
...
ValueError: number of elements must be non-negative, not -6

sage: Posets.RandomPoset(6, 'garbage')
```

```
Traceback (most recent call last):
...
TypeError: probability must be a real number, not garbage

sage: Posets.RandomPoset(6, -0.5)
Traceback (most recent call last):
...
ValueError: probability must be between 0 and 1, not -0.5
```

static RestrictedIntegerPartitions(*n*)

Returns the poset of integer partitions on the integer n ordered by restricted refinement. That is, if p and q are integer partitions of n, then p covers q if and only if q is obtained from p by joining two distinct parts of p (and sorting, if necessary).

EXAMPLES:

```
sage: P = Posets.RestrictedIntegerPartitions(7); P
Finite poset containing 15 elements
sage: len(P.cover_relations())
17
```

static SSTPoset(*s*, *f=None*)

The poset on semistandard tableaux of shape `s` and largest entry `f` that is ordered by componentwise comparison of the entries.

INPUT:

- `s` - shape of the tableaux
- `f` - maximum fill number. This is an optional argument. If no maximal number is given, it will use the number of cells in the shape.

NOTE: This is a basic implementation and most certainly not the most efficient.

EXAMPLES:

```
sage: Posets.SSTPoset([2,1])
Finite poset containing 8 elements

sage: Posets.SSTPoset([2,1],4)
Finite poset containing 20 elements

sage: Posets.SSTPoset([2,1],2).cover_relations()
[[[[1, 1], [2]], [[1, 2], [2]]]]

sage: Posets.SSTPoset([3,2]).bottom()  # long time (6s on sage.math, 2012)
[[1, 1, 1], [2, 2]]

sage: Posets.SSTPoset([3,2],4).maximal_elements()
[[[3, 3, 4], [4, 4]]]
```

static ShardPoset(*n*)

Return the shard intersection order on permutations of size n.

This is defined on the set of permutations. To every permutation, one can attach a pre-order, using the descending runs and their relative positions.

The shard intersection order is given by the implication (or refinement) order on the set of pre-orders defined from all permutations.

This can also be seen in a geometrical way. Every pre-order defines a cone in a vector space of dimension n. The shard poset is given by the inclusion of these cones.

See also:

`shard_preorder_graph()`

EXAMPLES:

```
sage: P = posets.ShardPoset(4); P  # indirect doctest
Finite poset containing 24 elements
sage: P.chain_polynomial()
34*q^4 + 90*q^3 + 79*q^2 + 24*q + 1
sage: P.characteristic_polynomial()
q^3 - 11*q^2 + 23*q - 13
sage: P.zeta_polynomial()
17/3*q^3 - 6*q^2 + 4/3*q
sage: P.is_selfdual()
False
```

static `StandardExample`(*n*, *facade=None*)

Return the partially ordered set on `2n` elements with dimension `n`.

Let P be the poset on $\{0, 1, 2, \ldots, 2n-1\}$ whose defining relations are that $i < j$ for every $0 \leq i < n \leq j < 2n$ except when $i + n = j$. The poset P is the so-called *standard example* of a poset with dimension n.

INPUT:

- `n` – an integer ≥ 2, dimension of the constructed poset
- `facade` (boolean) – whether to make the returned poset a facade poset (see `sage.categories.facade_sets`); the default behaviour is the same as the default behaviour of the `Poset()` constructor

OUTPUT:

The standard example of a poset of dimension n.

EXAMPLES:

```
sage: A = Posets.StandardExample(3); A
Finite poset containing 6 elements
sage: A.dimension()
3
```

REFERENCES:

TESTS:

```
sage: A = Posets.StandardExample(10); A
Finite poset containing 20 elements
sage: len(A.cover_relations())
90

sage: P = Posets.StandardExample(5, facade=False)
sage: P(4) < P(3), P(4) > P(3)
(False, False)
```

static `SymmetricGroupBruhatIntervalPoset`(*start*, *end*)

The poset of permutations with respect to Bruhat order.

INPUT:

- `start` - list permutation
- `end` - list permutation (same n, of course)

Note: Must have `start` <= `end`.

EXAMPLES:

Any interval is rank symmetric if and only if it avoids these permutations:

```
sage: P1 = Posets.SymmetricGroupBruhatIntervalPoset([1,2,3,4], [3,4,1,2])
sage: P2 = Posets.SymmetricGroupBruhatIntervalPoset([1,2,3,4], [4,2,3,1])
sage: ranks1 = [P1.rank(v) for v in P1]
sage: ranks2 = [P2.rank(v) for v in P2]
sage: [ranks1.count(i) for i in uniq(ranks1)]
[1, 3, 5, 4, 1]
sage: [ranks2.count(i) for i in uniq(ranks2)]
[1, 3, 5, 6, 4, 1]
```

static SymmetricGroupBruhatOrderPoset(n)

The poset of permutations with respect to Bruhat order.

EXAMPLES:

```
sage: Posets.SymmetricGroupBruhatOrderPoset(4)
Finite poset containing 24 elements
```

static SymmetricGroupWeakOrderPoset(n, *labels='permutations'*, *side='right'*)

The poset of permutations of $\{1, 2, \ldots, n\}$ with respect to the weak order (also known as the permutohedron order, cf. `permutohedron_lequal()`).

The optional variable `labels` (default: `"permutations"`) determines the labelling of the elements if $n < 10$. The optional variable `side` (default: `"right"`) determines whether the right or the left permutohedron order is to be used.

EXAMPLES:

```
sage: Posets.SymmetricGroupWeakOrderPoset(4)
Finite poset containing 24 elements
```

static TamariLattice(n)

Return the n-th Tamari lattice.

INPUT:

- n a nonnegative integer

OUTPUT:

- a finite lattice

The elements of the lattice are `Dyck paths` in the $(n+1 \times n)$-rectangle.

See Tamari lattice for mathematical background.

EXAMPLES:

```
sage: posets.TamariLattice(3)
Finite lattice containing 5 elements
```

static TetrahedralPoset(n, **colors*, ***labels*)

Return the tetrahedral poset based on the input colors.

This method will return the tetrahedral poset with n-1 layers and covering relations based on the input colors of 'green', 'red', 'orange', 'silver', 'yellow' and 'blue' as defined in [Striker2011]. For particular color choices, the order ideals of the resulting tetrahedral poset will be isomorphic to known combinatorial objects.

For example, for the colors 'blue', 'yellow', 'orange', and 'green', the order ideals will be in bijection with alternating sign matrices. For the colors 'yellow', 'orange', and 'green', the order ideals will be in bijection with semistandard Young tableaux of staircase shape. For the colors 'red', 'orange', 'green', and optionally 'yellow', the order ideals will be in bijection with totally symmetric self-complementary plane partitions in a $2n \times 2n \times 2n$ box.

INPUT:

- `n` - Defines the number (n-1) of layers in the poset.
- `colors` - The colors that define the covering relations of the poset. Colors used are 'green', 'red', 'yellow', 'orange', 'silver', and 'blue'.
- `labels` - Keyword variable used to determine whether the poset is labeled with integers or tuples. To label with integers, the method should be called with `labels='integers'`. Otherwise, the labeling will default to tuples.

EXAMPLES:

```
sage: Posets.TetrahedralPoset(4,'green','red','yellow','silver','blue','orange')
Finite poset containing 10 elements

sage: Posets.TetrahedralPoset(4,'green','red','yellow','silver','blue','orange', labels='int
Finite poset containing 10 elements

sage: A = AlternatingSignMatrices(3)
sage: p = A.lattice()
sage: ji = p.join_irreducibles_poset()
sage: tet = Posets.TetrahedralPoset(3, 'green','yellow','blue','orange')
sage: ji.is_isomorphic(tet)
True
```

REFERENCES:

`sage.combinat.posets.poset_examples.`**`posets`**
alias of `Posets`

5.1.159 Finite posets

This module implements finite partially ordered sets. It defines:

`FinitePoset`	A class for finite posets
`FinitePosets_n`	A class for finite posets up to isomorphism (i.e. unlabeled posets)
`Poset()`	Construct a finite poset from various forms of input data.
`is_poset()`	Return `True` if a directed graph is acyclic and transitively reduced.

List of Poset methods

Comparing, intervals and relations

is_less_than()	Return True if x is strictly less than y in the poset.
is_greater_than()	Return True if x is strictly greater than y in the poset.
is_lequal()	Return True if x is less than or equal to y in the poset.
is_gequal()	Return True if x is greater than or equal to y in the poset.
compare_elements()	Compare two element of the poset.
closed_interval()	Return the list of elements in a closed interval of the poset.
open_interval()	Return the list of elements in an open interval of the poset.
relations()	Return the list of relations in the poset.
relations_iterator()	Return an iterator over relations in the poset.
order_filter()	Return the upper set generated by elements.
order_ideal()	Return the lower set generated by elements.

Covering

covers()	Return True if y covers x.
lower_covers()	Return elements covered by given element.
upper_covers()	Return elements covering given element.
cover_relations()	Return the list of cover relations.
lower_covers_iterator()	Return an iterator over elements covered by given element.
upper_covers_iterator()	Return an iterator over elements covering given element.
cover_relations_iterator()	Return an iterator over cover relations of the poset.

Properties of the poset

cardinality()	Return the number of elements in the poset.
height()	Return the number of elements in a longest chain of the poset.
width()	Return the number of elements in a longest antichain of the poset.
relations_number()	Return the number of relations in the poset.
dimension()	Return the dimension of the poset.
has_bottom()	Return True if the poset has a unique minimal element.
has_top()	Return True if the poset has a unique maximal element.
is_bounded()	Return True if the poset has both unique minimal and unique maximal element.
is_chain()	Return True if the poset is totally ordered.
is_connected()	Return True if the poset is connected.
is_graded()	Return True if all maximal chains of the poset has same length.
is_ranked()	Return True if the poset has a rank function.
is_rank_symmetric()	Return True if the poset is rank symmetric.
is_eulerian()	Return True if the poset is Eulerian.
is_incomparable_chain_free()	Return True if the poset is (m+n)-free.
is_slender()	Return True if the poset is slender.
is_join_semilattice()	Return True is the poset has a join operation.
is_meet_semilattice()	Return True if the poset has a meet operation.

Minimal and maximal elements

bottom()	Return the bottom element of the poset, if it exists.
top()	Return the top element of the poset, if it exists.
maximal_elements()	Return the list of the maximal elements of the poset.
minimal_elements()	Return the list of the minimal elements of the poset.

New posets from old ones

disjoint_union()	Return the disjoint union of the poset with other poset.
ordinal_sum()	Return the ordinal sum of the poset with other poset.
ordinal_product()	Return the ordinal product of the poset with other poset.
product()	Return the Cartesian product of the poset with other poset.
with_bounds()	Return the poset with bottom and top element adjoined.
dual()	Return the dual poset of this poset.
completion_by_cuts()	Return the Dedekind-MacNeille completion of the poset.
connected_components()	Return the connected components of the poset as subposets.
ordinal_summands()	Return the ordinal summands of the poset.
subposet()	Return the subposet containing elements with partial order induced by this poset.
random_subposet()	Return a random subposet that contains each element with given probability.
canonical_label()	Return copy of the poset canonically (re)labelled with elements $\{0, \ldots, n-1\}$.
relabel()	Return a copy of this poset with its elements relabelled.

Chains & antichains

is_chain_of_poset()	Return `True` if the given list is a chain of the poset.
chains()	Return the chains of the poset.
antichains()	Return the antichains of the poset.
maximal_chains()	Return the maximal chains of the poset.
maximal_antichains()	Return the maximal antichains of the poset.
antichains_iterator()	Return an iterator over the antichains of the poset.

Drawing

show()	Display the Hasse diagram of the poset.
plot()	Return a Graphic object corresponding the Hasse diagram of the poset.
graphviz_string()	Return a representation in the DOT language, ready to render in graphviz.

Comparing posets

is_isomorphic()	Return `True` if both posets are isomorphic.
is_induced_subposet()	Return `True` if given poset is an induced subposet of this poset.

Polynomials

chain_polynomial()	Return the chain polynomial of the poset.
characteristic_polynomial()	Return the characteristic polynomial of the poset.
f_polynomial()	Return the f-polynomial of a bounded poset.
flag_f_polynomial()	Return the flag f-polynomial of a bounded and ranked poset.
flag_h_polynomial()	Return the flag h-polynomial of a bounded and ranked poset.
h_polynomial()	Return the h-polynomial of a bounded poset.
kazhdan_lusztig_polynomial()	Return the Kazhdan-Lusztig polynomial $P_{x,y}(q)$ of `self`.
order_polynomial()	Return the order polynomial of the poset.
zeta_polynomial()	Return the zeta polynomial of the poset.

Polytopes

chain_polytope()	Return the chain polytope of the poset.
order_polytope()	Return the order polytope of the poset.

Graphs

hasse_diagram()	Return the Hasse diagram of the poset as a directed graph.
cover_relations_graph()	Return the (undirected) graph of cover relations.
comparability_graph()	Return the comparability graph of the poset.
incomparability_graph()	Return the incomparability graph of the poset.
linear_extensions_graph()	Return the linear extensions graph of the poset.

Other & not yet classified

isomorphic_subposets()	Return all subposets isomorphic to another poset.
isomorphic_subposets_iterator()	Return an iterator over the subposets isomorphic to another poset.
has_isomorphic_subposet()	Return True if the poset contains a subposet isomorphic to another poset.
moebius_function()	Return the value of Möbius function of given elements in the poset.
moebius_function_matrix()	Return a matrix whose (i, j) entry is the value of the Möbius function evaluated at
is_linear_extension()	Return whether l is a linear extension of self.
linear_extension()	Return a linear extension of this poset.
linear_extensions()	Return the enumerated set of all the linear extensions of this poset.
promotion()	Computes the (extended) promotion on the linear extension of the poset.
evacuation()	Computes evacuation on the linear extension associated to the poset.
coxeter_transformation()	Return the matrix of the Auslander-Reiten translation acting on the Grothendieck gro
coxeter_polynomial()	Return the characteristic polynomial of the Coxeter transformation.
list()	List the elements of the poset.
cuts()	Return the cuts of the given poset.
dilworth_decomposition()	Return a partition of the points into the minimal number of chains.
frank_network()	Return Frank's network (a DiGraph along with a cost function on its edges) associat
greene_shape()	Computes the Greene-Kleitman partition aka Greene shape of the poset self.
incidence_algebra()	Return the indicence algebra of self.
is_EL_labelling()	Return whether f is an EL labelling of the poset.
is_linear_extension()	Return whether l is a linear extension of self.
isomorphic_subposets_iterator()	Return an iterator over the subposets isomorphic to another poset.
isomorphic_subposets()	Return all subposets isomorphic to another poset.
lequal_matrix()	Computes the matrix whose (i, j) entry is 1 if self.linear_extension()[
level_sets()	Return a list l such that l[i+1] is the set of minimal elements of the poset obtained by
order_complex()	Return the order complex associated to this poset.
p_partition_enumerator()	Return a P-partition enumerator of the poset.
random_order_ideal()	Return a random order ideal of self with uniform probability.
rank()	Return the rank of an element, or the rank of the poset.
rank_function()	Return a rank function of the poset, if it exists.
unwrap()	Unwraps an element of this poset.
with_linear_extension()	Return a copy of self with a different default linear extension.

Classes and functions

class sage.combinat.posets.posets.**FinitePoset**(*hasse_diagram*, *elements*, *category*, *facade*, *key*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

A (finite) n-element poset constructed from a directed acyclic graph.

INPUT:

- hasse_diagram – an instance of FinitePoset, or a DiGraph that is transitively-reduced, acyclic, loop-free, and multiedge-free.
- elements – an optional list of elements, with element[i] corresponding to vertex i. If elements is None, then it is set to be the vertex set of the digraph. Note that if this option is set, then elements is considered as a specified linear extension of the poset and the $linear_extension$ attribute is set.
- category – FinitePosets, or a subcategory thereof.
- facade – a boolean or None (default); whether the FinitePoset's elements should be wrapped to make them aware of the Poset they belong to.

– If `facade = True`, the `FinitePoset`'s elements are exactly those given as input.

– If `facade = False`, the `FinitePoset`'s elements will become `PosetElement` objects.

– If `facade = None` (default) the expected behaviour is the behaviour of `facade = True`, unless the opposite can be deduced from the context (i.e. for instance if a `FinitePoset` is built from another `FinitePoset`, itself built with `facade = False`)

• `key` – any hashable value (default: `None`).

EXAMPLES:

```
sage: uc = [[2,3], [], [1], [1], [1], [3,4]]
sage: from sage.combinat.posets.posets import FinitePoset
sage: P = FinitePoset(DiGraph(dict([[i,uc[i]] for i in range(len(uc))])), facade=False); P
Finite poset containing 6 elements
sage: P.cover_relations()
[[5, 4], [5, 3], [4, 1], [0, 2], [0, 3], [2, 1], [3, 1]]
sage: TestSuite(P).run()
sage: P.category()
Join of Category of finite posets and Category of finite enumerated sets
sage: P.__class__
<class 'sage.combinat.posets.posets.FinitePoset_with_category'>

sage: Q = sage.combinat.posets.posets.FinitePoset(P, facade = False); Q
Finite poset containing 6 elements

sage: Q is P
True
```

We keep the same underlying Hasse diagram, but change the elements:

```
sage: Q = sage.combinat.posets.posets.FinitePoset(P, elements=[1,2,3,4,5,6], facade=False); Q
Finite poset containing 6 elements with distinguished linear extension
sage: Q.cover_relations()
[[1, 2], [1, 5], [2, 6], [3, 4], [3, 5], [4, 6], [5, 6]]
```

We test the facade argument:

```
sage: P = Poset(DiGraph({'a':['b'],'b':['c'],'c':['d']}), facade=False)
sage: P.category()
Join of Category of finite posets and Category of finite enumerated sets
sage: parent(P[0]) is P
True

sage: Q = Poset(DiGraph({'a':['b'],'b':['c'],'c':['d']}), facade=True)
sage: Q.category()
Join of Category of finite posets
    and Category of finite enumerated sets
    and Category of facade sets
sage: parent(Q[0]) is str
True
sage: TestSuite(Q).run(skip = ['_test_an_element']) # is_parent_of is not yet implemented
```

Changing a non facade poset to a facade poset:

```
sage: PQ = Poset(P, facade=True)
sage: PQ.category()
Join of Category of finite posets
    and Category of finite enumerated sets
    and Category of facade sets
sage: parent(PQ[0]) is str
```

```
True
sage: PQ is Q
True
```

Changing a facade poset to a non facade poset:

```
sage: QP = Poset(Q, facade = False)
sage: QP.category()
Join of Category of finite posets
    and Category of finite enumerated sets
sage: parent(QP[0]) is QP
True
```

Note: A class that inherits from this class needs to define `Element`. This is the class of the elements that the inheriting class contains. For example, for this class, `FinitePoset`, `Element` is `PosetElement`. It can also define `_dual_class` which is the class of dual posets of this class. E.g. `FiniteMeetSemilattice._dual_class` is `FiniteJoinSemilattice`.

TESTS:

Equality is derived from `UniqueRepresentation`. We check that this gives consistent results:

```
sage: P = Poset([[1,2],[3],[3]])
sage: P == P
True
sage: Q = Poset([[1,2],[],[1]])
sage: Q == P
False
sage: p1, p2 = Posets(2).list()
sage: p2 == p1, p1 != p2
(False, True)
sage: [[p1.__eq__(p2) for p1 in Posets(2)] for p2 in Posets(2)]
[[True, False], [False, True]]
sage: [[p2.__eq__(p1) for p1 in Posets(2)] for p2 in Posets(2)]
[[True, False], [False, True]]
sage: [[p2 == p1 for p1 in Posets(3)] for p2 in Posets(3)]
[[True, False, False, False, False],
 [False, True, False, False, False],
 [False, False, True, False, False],
 [False, False, False, True, False],
 [False, False, False, False, True]]

sage: [[p1.__ne__(p2) for p1 in Posets(2)] for p2 in Posets(2)]
[[False, True], [True, False]]
sage: P = Poset([[1,2,4],[3],[3]])
sage: Q = Poset([[1,2],[],[1],[4]])
sage: P != Q
True
sage: P != P
False
sage: Q != Q
False
sage: [[p1.__ne__(p2) for p1 in Posets(2)] for p2 in Posets(2)]
[[False, True], [True, False]]

sage: P = Poset((divisors(12), attrcall("divides")), linear_extension=True)
sage: Q = Poset(P)
sage: Q == P
```

```
False
sage: Q = Poset(P, linear_extension=True)
sage: Q == P
True
```

Element
alias of PosetElement

antichains(*element_constructor=<type 'list'>*)
Return the antichains of the poset.

An *antichain* of a poset is a set of elements of the poset that are pairwise incomparable.

INPUT:

- element_constructor – a function taking an iterable as argument (default: list)

OUTPUT:

The enumerated set (of type PairwiseCompatibleSubsets) of all antichains of the poset, each of which is given as an element_constructor.

EXAMPLES:

```
sage: A = Posets.PentagonPoset().antichains(); A
Set of antichains of Finite lattice containing 5 elements
sage: list(A)
[[], [0], [1], [1, 2], [1, 3], [2], [3], [4]]
sage: A.cardinality()
8
sage: A[3]
[1, 2]
```

To get the antichains as, say, sets, one may use the element_constructor option:

```
sage: list(Posets.ChainPoset(3).antichains(element_constructor=set))
[set(), {0}, {1}, {2}]
```

To get the antichains of a given size one can currently use:

```
sage: list(A.elements_of_depth_iterator(2))
[[1, 2], [1, 3]]
```

Eventually the following syntax will be accepted:

```
sage: A.subset(size = 2) # todo: not implemented
```

Note: Internally, this uses sage.combinat.subsets_pairwise.PairwiseCompatibleSubsets and SearchForest. At this point, iterating through this set is about twice slower than using antichains_iterator() (tested on posets.AntichainPoset(15)). The algorithm is the same (depth first search through the tree), but antichains_iterator() manually inlines things which apparently avoids some infrastructure overhead.

On the other hand, this returns a full featured enumerated set, with containment testing, etc.

See also:

maximal_antichains(), chains()

antichains_iterator()
Return an iterator over the antichains of the poset.

EXAMPLES:

```
sage: it = Posets.PentagonPoset().antichains_iterator(); it
<generator object antichains_iterator at ...>
sage: it.next(), it.next()
([], [4])
```

See also:

antichains()

bottom()

Return the unique minimal element of the poset, if it exists.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.bottom() is None
True
sage: Q = Poset({0:[1],1:[]})
sage: Q.bottom()
0
```

See also:

has_bottom(),top()

canonical_label()

Return the unique poset on the labels $\{0, \ldots, n-1\}$ (where n is the number of elements in the poset) that is isomorphic to this poset and invariant in the isomorphism class.

See also:

- Canonical labeling of directed graphs: canonical_label()

EXAMPLES:

```
sage: P = Poset((divisors(12), attrcall("divides")), linear_extension=True, facade=False)
sage: P.list()
[1, 2, 3, 4, 6, 12]
sage: P.cover_relations()
[[1, 2], [1, 3], [2, 4], [2, 6], [3, 6], [4, 12], [6, 12]]
sage: Q = P.canonical_label()
sage: Q.list() # random
[0, 1, 2, 3, 4, 5]
sage: Q.is_isomorphic(P)
True
```

As a facade:

```
sage: P = Poset((divisors(12), attrcall("divides")), facade=True, linear_extension=True)
sage: P.list()
[1, 2, 3, 4, 6, 12]
sage: P.cover_relations()
[[1, 2], [1, 3], [2, 4], [2, 6], [3, 6], [4, 12], [6, 12]]
sage: Q = P.canonical_label()
sage: sorted(Q.list())
[0, 1, 2, 3, 4, 5]
sage: Q.is_isomorphic(P)
True
```

Canonical labeling of (semi)lattice returns (semi)lattice:

```
sage: D=DiGraph({'a':['b','c']})
sage: P=Poset(D)
sage: ML=MeetSemilattice(D)
sage: P.canonical_label()
Finite poset containing 3 elements
sage: ML.canonical_label()
Finite meet-semilattice containing 3 elements
```

TESTS:

```
sage: P = Poset(digraphs.Path(10), linear_extension = True)
sage: Q = P.canonical_label()
sage: Q.linear_extension() # random
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: Q.cover_relations() # random
[[0, 1], [1, 2], [2, 3], [3, 4], [4, 5], [5, 6], [6, 7], [7, 8], [8, 9]]
sage: P = Poset(digraphs.Path(10))
sage: Q = P.canonical_label()
sage: Q.linear_extension() # random
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: Q.is_isomorphic(P)
True
```

cardinality()

Return the number of elements in the poset.

EXAMPLES:

```
sage: Poset([[1,2,3],[4],[4],[4],[]]).cardinality()
5
```

chain_polynomial()

Return the chain polynomial of `self`.

The coefficient of q^k is the number of chains of length k in `self`. The length of a chain is the number of elements.

Warning: This is not what has been called the chain polynomial in [St1986]. The latter is identical with the order polynomial (`order_polynomial()`).

EXAMPLES:

```
sage: P = Posets.ChainPoset(3)
sage: t = P.chain_polynomial(); t
q^3 + 3*q^2 + 3*q + 1
sage: t(1) == len(list(P.chains()))
True

sage: P = Posets.BooleanLattice(3)
sage: P.chain_polynomial()
6*q^4 + 18*q^3 + 19*q^2 + 8*q + 1

sage: P = Posets.AntichainPoset(5)
sage: P.chain_polynomial()
5*q + 1

sage: P = Poset({})
sage: P.chain_polynomial()
1
```

```
sage: parent(P.chain_polynomial())
Univariate Polynomial Ring in q over Integer Ring

sage: R = Poset({1: []})
sage: R.chain_polynomial()
q + 1
```

chain_polytope()

Return the chain polytope of the poset `self`.

The chain polytope of a finite poset P is defined as the subset of $\mathbf{R}^P$ consisting of all maps $x : P \to \mathbf{R}$ satisfying

$$x(p) \geq 0 \text{ for all } p \in P,$$

and

$$x(p_1) + x(p_2) + \ldots + x(p_k) \leq 1 \text{ for all chains } p_1 < p_2 < \ldots < p_k \text{ in } P.$$

This polytope was defined and studied in [St1986].

EXAMPLES:

```
sage: P = posets.AntichainPoset(3)
sage: Q = P.chain_polytope();Q
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 8 vertices
sage: P = posets.PentagonPoset()
sage: Q = P.chain_polytope();Q
A 5-dimensional polyhedron in QQ^5 defined as the convex hull of 8 vertices
```

chains(*element_constructor=<type 'list'>*, *exclude=None*)

Return the chains of the poset.

A *chain* of a poset is a set of elements of the poset that are pairwise comparable.

INPUT:

- `element_constructor` – a function taking an iterable as argument (default: `list`)
- `exclude` – elements of the poset to be excluded (default: `None`)

OUTPUT:

The enumerated set (of type `PairwiseCompatibleSubsets`) of all chains of the poset, each of which is given as an `element_constructor`.

EXAMPLES:

```
sage: C = Posets.PentagonPoset().chains(); C
Set of chains of Finite lattice containing 5 elements
sage: list(C)
[[], [0], [0, 1], [0, 1, 4], [0, 2], [0, 2, 3], [0, 2, 3, 4], [0, 2, 4], [0, 3], [0, 3, 4],
```

Exclusion of vertices, tuple (instead of list) as constructor:

```
sage: P = Poset({1: [2, 3], 2: [4], 3: [4, 5]})
sage: list(P.chains(element_constructor=tuple, exclude=[3]))
[(), (1,), (1, 2), (1, 2, 4), (1, 4), (1, 5), (2,), (2, 4), (4,), (5,)]
```

To get the chains of a given size one can currently use:

```
sage: list(C.elements_of_depth_iterator(2))
[[0, 1], [0, 2], [0, 3], [0, 4], [1, 4], [2, 3], [2, 4], [3, 4]]
```

Eventually the following syntax will be accepted:

```
sage: C.subset(size = 2) # todo: not implemented
```

See also:

maximal_chains(), antichains()

characteristic_polynomial()

Return the characteristic polynomial of the poset.

The poset is expected to be graded and have a bottom element.

If P is a graded poset with rank n and a unique minimal element $\hat{0}$, then the characteristic polynomial of P is defined to be

$$\sum_{x \in P} \mu(\hat{0}, x) q^{n-\rho(x)} \in \mathbf{Z}[q],$$

where ρ is the rank function, and μ is the Möbius function of P.

See section 3.10 of [EnumComb1].

EXAMPLES:

```
sage: P = Posets.DiamondPoset(5)
sage: P.characteristic_polynomial()
q^2 - 3*q + 2
sage: P = Poset({1:[2,3],2:[4],3:[5],4:[6],5:[6],6:[7]})
sage: P.characteristic_polynomial()
q^4 - 2*q^3 + q
sage: P = Poset({1: []})
sage: P.characteristic_polynomial()
1
```

closed_interval(*x*, *y*)

Return the list of elements z such that $x \leq z \leq y$ in the poset.

EXAMPLES:

```
sage: P = Poset((divisors(1000), attrcall("divides")))
sage: P.closed_interval(2, 100)
[2, 4, 10, 20, 50, 100]
```

See also:

open_interval()

TESTS:

```
sage: C = Posets.ChainPoset(10)
sage: C.closed_interval(3, 3)
[3]
sage: C.closed_interval(8, 5)
[]
sage: A = Posets.AntichainPoset(10)
sage: A.closed_interval(3, 7)
[]
```

comparability_graph()

Returns the comparability graph of `self`.

See Wikipedia article Comparability_graph

See also:

`incomparability_graph()`, `sage.graphs.comparability`

EXAMPLES:

```
sage: p = posets.ChainPoset(4)
sage: p.comparability_graph().is_isomorphic(graphs.CompleteGraph(4))
True

sage: p = posets.DiamondPoset(5)
sage: g = p.comparability_graph(); g
Comparability graph on 5 vertices
sage: g.size()
7
```

compare_elements(x, y)

Compare x and y in the poset.

- If $x < y$, return `-1`.
- If $x = y$, return `0`.
- If $x > y$, return `1`.
- If x and y are not comparable, return `None`.

EXAMPLES:

```
sage: P = Poset([[1, 2], [4], [3], [4], []])
sage: P.compare_elements(0, 0)
0
sage: P.compare_elements(0, 4)
-1
sage: P.compare_elements(4, 0)
1
sage: P.compare_elements(1, 2) is None
True
```

completion_by_cuts()

Return the completion by cuts of `self`.

This is a lattice, also called the Dedekind-MacNeille completion.

See the Wikipedia article Dedekind-MacNeille completion.

OUTPUT:

- a finite lattice

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: P.completion_by_cuts().is_isomorphic(P)
True

sage: P = posets.AntichainPoset(3)
sage: Q = P.completion_by_cuts()
sage: Q.is_isomorphic(posets.DiamondPoset(5))
True
```

```
sage: P = posets.SymmetricGroupBruhatOrderPoset(3)
sage: Q = P.completion_by_cuts(); Q
Finite lattice containing 7 elements
```

See also:

`cuts()`

connected_components()

Return the connected components of the poset as subposets.

EXAMPLES:

```
sage: P = Poset({1:[2,3], 3:[4,5]})
sage: CC = P.connected_components()
sage: CC[0] is P
True

sage: P = Poset({1:[2,3], 3:[4,5], 6:[7,8]})
sage: sorted(P.connected_components(), key=len)
[Finite poset containing 3 elements,
 Finite poset containing 5 elements]
```

TESTS:

```
sage: Poset().connected_components() # Test empty poset
[]
```

cover_relations()

Return the list of pairs `[x, y]` of elements of the poset such that `y` covers `x`.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.cover_relations()
[[1, 2], [0, 2], [2, 3], [3, 4]]
```

cover_relations_graph()

Return the graph of cover relations.

EXAMPLES:

```
sage: P = Poset({0:[1,2],1:[3],2:[3],3:[]})
sage: G = P.cover_relations_graph(); G
Graph on 4 vertices
sage: S = Poset()
sage: H = S.cover_relations_graph(); H
Graph on 0 vertices
```

Check that it is hashable and coincides with the Hasse diagram as a graph:

```
sage: hash(G) == hash(G)
True
sage: G == Graph(P.hasse_diagram())
True
```

cover_relations_iterator()

Return an iterator over the cover relations of the poset.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: type(P.cover_relations_iterator())
<type 'generator'>
sage: [z for z in P.cover_relations_iterator()]
[[1, 2], [0, 2], [2, 3], [3, 4]]
```

covers (*x*, *y*)

Return `True` if `y` covers `x` and `False` otherwise.

Element y covers x if $x < y$ and there is no z such that $x < z < y$.

EXAMPLES:

```
sage: P = Poset([[1,5], [2,6], [3], [4], [], [6,3], [4]])
sage: P.covers(1, 6)
True
sage: P.covers(1, 4)
False
sage: P.covers(1, 5)
False
```

coxeter_polynomial()

Return the Coxeter polynomial of the poset.

OUTPUT:

a polynomial in one variable

The output is the characteristic polynomial of the Coxeter transformation. This polynomial only depends on the derived category of modules on the poset.

See also:

`coxeter_transformation()`

EXAMPLES:

```
sage: P = posets.PentagonPoset()
sage: P.coxeter_polynomial()
x^5 + x^4 + x + 1

sage: p = posets.SymmetricGroupWeakOrderPoset(3)
sage: p.coxeter_polynomial()
x^6 + x^5 - x^3 + x + 1
```

coxeter_transformation()

Return the Coxeter transformation of the poset.

OUTPUT:

a square matrix with integer coefficients

The output is the matrix of the Auslander-Reiten translation acting on the Grothendieck group of the derived category of modules on the poset, in the basis of simple modules. This matrix is usually called the Coxeter transformation.

See also:

`coxeter_polynomial()`

EXAMPLES:

```
sage: Posets.PentagonPoset().coxeter_transformation()
[ 0  0  0  0 -1]
[ 0  0  0  1 -1]
[ 0  1  0  0 -1]
[-1  1  1  0 -1]
[-1  1  0  1 -1]
```

TESTS:

```
sage: M = posets.PentagonPoset().coxeter_transformation()
sage: M ** 8 == 1
True
```

cuts()

Return the list of cuts of the poset `self`.

A cut is a subset A of `self` such that the set of lower bounds of the set of upper bounds of A is exactly A.

The cuts are computed here using the maximal independent sets in the auxiliary graph defined as $P \times [0, 1]$ with an edge from $(x, 0)$ to $(y, 1)$ if and only if $x \not\geq_P y$. See the end of section 4 in [JRJ94].

EXAMPLES:

```
sage: P = posets.AntichainPoset(3)
sage: Pc = P.cuts()
sage: [list(c) for c in Pc]
[[0], [0, 1, 2], [], [1], [2]]
sage: Pc[0]
frozenset({0})
```

See also:

`completion_by_cuts()`

REFERENCES:

deprecated_function_alias(*trac_number*, *func*)

Create an aliased version of a function or a method which raise a deprecation warning message.

If f is a function or a method, write `g = deprecated_function_alias(trac_number, f)` to make a deprecated aliased version of f.

INPUT:

- `trac_number` – integer. The trac ticket number where the deprecation is introduced.
- `func` – the function or method to be aliased

EXAMPLES:

```
sage: from sage.misc.superseded import deprecated_function_alias
sage: g = deprecated_function_alias(13109, number_of_partitions)
sage: g(5)
doctest:...: DeprecationWarning: g is deprecated. Please use sage.combinat.partition.number_
See http://trac.sagemath.org/13109 for details.
7
```

This also works for methods:

```
sage: class cls(object):
....:     def new_meth(self): return 42
....:     old_meth = deprecated_function_alias(13109, new_meth)
sage: cls().old_meth()
```

```
doctest:...: DeprecationWarning: old_meth is deprecated. Please use new_meth instead.
See http://trac.sagemath.org/13109 for details.
42
```

trac ticket #11585:

```
sage: def a(): pass
sage: b = deprecated_function_alias(13109, a)
sage: b()
doctest:...: DeprecationWarning: b is deprecated. Please use a instead.
See http://trac.sagemath.org/13109 for details.
```

AUTHORS:

- Florent Hivert (2009-11-23), with the help of Mike Hansen.
- Luca De Feo (2011-07-11), printing the full module path when different from old path

dilworth_decomposition()

Return a partition of the points into the minimal number of chains.

According to Dilworth's theorem, the points of a poset can be partitioned into α chains, where α is the cardinality of its largest antichain. This method returns such a partition.

See Wikipedia article Dilworth's_theorem.

See also:

`width()` – return the width of the poset.

ALGORITHM:

We build a bipartite graph in which a vertex v of the poset is represented by two vertices v^-, v^+. For any two u, v such that $u < v$ in the poset we add an edge v^+u^-.

A matching in this graph is equivalent to a partition of the poset into chains: indeed, a chain $v_1...v_k$ gives rise to the matching $v_1^+v_2^-, v_2^+v_3^-$, ..., and from a matching one can build the union of chains.

According to Dilworth's theorem, the number of chains is equal to α (the posets' width).

EXAMPLES:

```
sage: p = posets.BooleanLattice(4)
sage: p.width()
6
sage: p.dilworth_decomposition()  # random
[[7, 6, 4], [11, 3], [12, 8, 0], [13, 9, 1], [14, 10, 2], [15, 5]]
```

TESTS:

```
sage: p = posets.IntegerCompositions(5)
sage: d = p.dilworth_decomposition()
sage: for chain in d:
....:     for i in range(len(chain)-1):
....:         assert p.is_greater_than(chain[i],chain[i+1])
sage: set(p) == set().union(*d)
True
```

dimension(*certificate=False*)

Return the dimension of the Poset.

The (Dushnik-Miller) dimension of a poset is the minimal number of total orders so that the poset can be defined as "intersection" of all of them. Mathematically said, dimension of a poset defined on a set X

of points is the smallest integer n such that there exists $P_1, ..., P_n$ linear extensions of P satisfying the following property:

$$u \leq_P v \text{ if and only if } \forall i, u \leq_{P_i} v$$

For more information, see the Wikipedia article Order_dimension.

INPUT:

- `certificate` (boolean; default:`False`) – whether to return an integer (the dimension) or a certificate, i.e. a smallest set of linear extensions.

Note: The speed of this function greatly improves when more efficient MILP solvers (e.g. Gurobi, CPLEX) are installed. See `MixedIntegerLinearProgram` for more information.

ALGORITHM:

As explained [FT00], the dimension of a poset is equal to the (weak) chromatic number of a hypergraph. More precisely:

> Let $inc(P)$ be the set of (ordered) pairs of incomparable elements of P, i.e. all uv and vu such that $u \not\leq_P v$ and $v \not\leq_P u$. Any linear extension of P is a total order on X that can be seen as the union of relations from P along with some relations from $inc(P)$. Thus, the dimension of P is the smallest number of linear extensions of P which *cover* all points of $inc(P)$.
>
> Consequently, $dim(P)$ is equal to the chromatic number of the hypergraph $\mathcal{H}_{inc}$, where $\mathcal{H}_{inc}$ is the hypergraph defined on $inc(P)$ whose sets are all $S \subseteq inc(P)$ such that $P \cup S$ is not acyclic.

We solve this problem through a `Mixed Integer Linear Program`.

EXAMPLES:

We create a poset, compute a set of linear extensions and check that we get back the poset from them:

```
sage: P = Poset([[1,4], [3], [4,5,3], [6], [], [6], []])
sage: P.dimension()
3
sage: L = P.dimension(certificate=True)
sage: L # random -- architecture-dependent
[[0, 2, 4, 5, 1, 3, 6], [2, 5, 0, 1, 3, 4, 6], [0, 1, 2, 3, 5, 6, 4]]
sage: Poset( (L[0], lambda x, y: all(l.index(x) < l.index(y) for l in L)) ) == P
True
```

According to Schnyder's theorem, the poset (of height 2) of a graph has dimension ≤ 3 if and only if the graph is planar:

```
sage: G = graphs.CompleteGraph(4)
sage: P = Poset(DiGraph({(u,v):[u,v] for u,v,_ in G.edges()}))
sage: P.dimension()
3

sage: G = graphs.CompleteBipartiteGraph(3,3)
sage: P = Poset(DiGraph({(u,v):[u,v] for u,v,_ in G.edges()}))
sage: P.dimension() # not tested - around 4s with CPLEX
4
```

TESTS:

Empty Poset:

```
sage: Poset().dimension()
0
```

```
sage: Poset().dimension(certificate=True)
[]
```

REFERENCES:

disjoint_union(*other*, *labels='pairs'*)

Return a poset isomorphic to disjoint union (also called direct sum) of the poset with `other`.

The disjoint union of P and Q is a poset that contains every element and relation from both P and Q, and where every element of P is incomparable to every element of Q.

Mathematically, it is only defined when P and Q have no common element; here we force that by giving them different names in the resulting poset.

INPUT:

- `other`, a poset.
- `labels` - (defaults to 'pairs') If set to 'pairs', each element `v` in this poset will be named `(0,v)` and each element `u` in `other` will be named `(1,u)` in the result. If set to 'integers', the elements of the result will be relabeled with consecutive integers.

EXAMPLES:

```
sage: P1 = Poset( (['a', 'b'], [['a', 'b']]) )
sage: P2 = Poset( (['c', 'd'], [['c', 'd']]) )
sage: P = P1.disjoint_union(P2); P
Finite poset containing 4 elements
sage: sorted(P.cover_relations())
[[(0, 'a'), (0, 'b')], [(1, 'c'), (1, 'd')]]
sage: P = P1.disjoint_union(P2, labels='integers');
sage: P.cover_relations()
[[2, 3], [0, 1]]

sage: N5 = Posets.PentagonPoset(); N5
Finite lattice containing 5 elements
sage: N5.disjoint_union(N5)  # Union of lattices is not a lattice
Finite poset containing 10 elements
```

We show how to get literally direct sum with elements untouched:

```
sage: P = P1.disjoint_union(P2).relabel(lambda x: x[1])
sage: sorted(P.cover_relations())
[['a', 'b'], ['c', 'd']]
```

dual()

Return the dual poset of the given poset.

EXAMPLES:

```
sage: P = Poset(([1,2,3],[[1,2],[1,3]]))
sage: P.cover_relations()
[[1, 2], [1, 3]]
sage: Q = P.dual()
sage: Q.cover_relations()
[[3, 1], [2, 1]]

sage: P = LatticePoset([[1,2],[3],[3]], facade = True)
sage: P.cover_relations()
[[0, 1], [0, 2], [1, 3], [2, 3]]
sage: Q = P.dual()
sage: Q.cover_relations()
```

```
[[3, 2], [3, 1], [2, 0], [1, 0]]
sage: Q.category()
Join of Category of finite lattice posets
    and Category of finite enumerated sets
    and Category of facade sets
sage: Q.__class__
<class 'sage.combinat.posets.lattices.FiniteLatticePoset_with_category'>

sage: P = MeetSemilattice([[1,2],[3],[3]])
sage: P.dual().__class__
<class 'sage.combinat.posets.lattices.FiniteJoinSemilattice_with_category'>
sage: P = JoinSemilattice([[1,2],[3],[3]])
sage: P.dual().__class__
<class 'sage.combinat.posets.lattices.FiniteMeetSemilattice_with_category'>
```

evacuation()

Compute evacuation on the linear extension associated to the poset `self`.

OUTPUT:

- an isomorphic poset, with the same default linear extension

Evacuation is defined on a poset `self` of size n by applying the evacuation operator $(\tau_1 \cdots \tau_{n-1})(\tau_1 \cdots \tau_{n-2}) \cdots (\tau_1)$, to the default linear extension π of `self` (see `evacuation()`), and relabeling `self` accordingly. For more details see [Stan2009].

See also:

- `linear_extension()`
- `with_linear_extension()` and the `linear_extension` option of `Poset()`
- `evacuation()`
- `promotion()`

REFERENCES:

EXAMPLES:

```
sage: P = Poset(([1,2], [[1,2]]), linear_extension=True, facade=False)
sage: P.evacuation()
Finite poset containing 2 elements with distinguished linear extension
sage: P.evacuation() == P
True

sage: P = Poset(([1,2,3,4,5,6,7], [[1,2],[1,4],[2,3],[2,5],[3,6],[4,7],[5,6]]), linear_exter
sage: P.list()
[1, 2, 3, 4, 5, 6, 7]
sage: Q = P.evacuation(); Q
Finite poset containing 7 elements with distinguished linear extension
sage: Q.cover_relations()
[[1, 2], [1, 3], [2, 5], [3, 4], [3, 6], [4, 7], [6, 7]]
```

Note that the results depend on the linear extension associated to the poset:

```
sage: P = Poset(([1,2,3,4,5,6,7], [[1,2],[1,4],[2,3],[2,5],[3,6],[4,7],[5,6]]))
sage: P.list()
[1, 2, 3, 5, 6, 4, 7]
sage: Q = P.evacuation(); Q
Finite poset containing 7 elements with distinguished linear extension
```

```
sage: Q.cover_relations()
[[1, 2], [1, 5], [2, 3], [5, 6], [5, 4], [6, 7], [4, 7]]
```

Here is an example of a poset where the vertices are not labelled by $\{1, 2, \ldots, n\}$:

```
sage: P = Poset((divisors(15), attrcall("divides")), linear_extension = True)
sage: P.list()
[1, 3, 5, 15]
sage: Q = P.evacuation(); Q
Finite poset containing 4 elements with distinguished linear extension
sage: Q.cover_relations()
[[1, 3], [1, 5], [3, 15], [5, 15]]
```

AUTHOR:

- Anne Schilling (2012-02-18)

f_polynomial()

Return the f-polynomial of a bounded poset `self`.

This is the f-polynomial of the order complex of the poset minus its bounds.

The coefficient of q^i is the number of chains of $i+1$ elements containing both bounds of the poset.

See also:

`is_bounded()`, `h_polynomial()`, `order_complex()`, `sage.homology.cell_complex.GenericCellC`

Warning: This is slightly different from the `fPolynomial` method in Macaulay2.

EXAMPLES:

```
sage: P = Posets.DiamondPoset(5)
sage: P.f_polynomial()
3*q^2 + q
sage: P = Poset({1:[2,3],2:[4],3:[5],4:[6],5:[7],6:[7]})
sage: P.f_polynomial()
q^4 + 4*q^3 + 5*q^2 + q

sage: P = Poset({2: []})
sage: P.f_polynomial()
1
```

flag_f_polynomial()

Return the flag f-polynomial of a bounded and ranked poset `self`.

This is the sum, over all chains containing both bounds, of a monomial encoding the ranks of the elements of the chain.

More precisely, if P is a bounded ranked poset, then the flag f-polynomial of P is defined as the polynomial

$$\sum_{\substack{p_0<p_1<\ldots<p_k, \\ p_0=\min P,\ p_k=\max P}} x_{\rho(p_1)} x_{\rho(p_2)} \cdots x_{\rho(p_k)} \in \mathbf{Z}[x_1, x_2, \cdots, x_n]$$

where $\min P$ and $\max P$ are (respectively) the minimum and the maximum of P, where ρ is the rank function of P (normalized to satisfy $\rho(\min P) = 0$), and where n is the rank of $\max P$. (Note that the indeterminate x_0 doesn't actually appear in the polynomial.)

For technical reasons, the polynomial is returned in the slightly larger ring $\mathbf{Z}[x_0, x_1, x_2, \cdots, x_{n+1}]$ by this method.

See Wikipedia article h-vector.

See also:

`is_bounded()`, `flag_h_polynomial()`

EXAMPLES:

```
sage: P = Posets.DiamondPoset(5)
sage: P.flag_f_polynomial()
3*x1*x2 + x2

sage: P = Poset({1:[2,3],2:[4],3:[5],4:[6],5:[6]})
sage: fl = P.flag_f_polynomial(); fl
2*x1*x2*x3 + 2*x1*x3 + 2*x2*x3 + x3
sage: q = polygen(ZZ,'q')
sage: fl(q,q,q,q) == P.f_polynomial()
True

sage: P = Poset({1:[2,3,4],2:[5],3:[5],4:[5],5:[6]})
sage: P.flag_f_polynomial()
3*x1*x2*x3 + 3*x1*x3 + x2*x3 + x3

sage: P = Poset({2: [3]})
sage: P.flag_f_polynomial()
x1

sage: P = Poset({2: []})
sage: P.flag_f_polynomial()
1
```

flag_h_polynomial()

Return the flag h-polynomial of a bounded and ranked poset `self`.

If P is a bounded ranked poset whose maximal element has rank n (where the minimal element is set to have rank 0), then the flag h-polynomial of P is defined as the polynomial

$$\prod_{k=1}^{n}(1-x_k)\cdot f\left(\frac{x_1}{1-x_1},\frac{x_2}{1-x_2},\cdots,\frac{x_n}{1-x_n}\right)\in\mathbf{Z}[x_1,x_2,\cdots,x_n],$$

where f is the flag f-polynomial of P (see `flag_f_polynomial()`).

For technical reasons, the polynomial is returned in the slightly larger ring $\mathbf{Q}[x_0,x_1,x_2,\cdots,x_{n+1}]$ by this method.

See Wikipedia article h-vector.

See also:

`is_bounded()`, `flag_f_polynomial()`

EXAMPLES:

```
sage: P = Posets.DiamondPoset(5)
sage: P.flag_h_polynomial()
2*x1*x2 + x2

sage: P = Poset({1:[2,3],2:[4],3:[5],4:[6],5:[6]})
sage: fl = P.flag_h_polynomial(); fl
-x1*x2*x3 + x1*x3 + x2*x3 + x3
sage: q = polygen(ZZ,'q')
sage: fl(q,q,q,q) == P.h_polynomial()
True
```

```
sage: P = Poset({1:[2,3,4],2:[5],3:[5],4:[5],5:[6]})
sage: P.flag_h_polynomial()
2*x1*x3 + x3

sage: P = posets.ChainPoset(4)
sage: P.flag_h_polynomial()
x3

sage: P = Poset({2: [3]})
sage: P.flag_h_polynomial()
x1

sage: P = Poset({2: []})
sage: P.flag_h_polynomial()
1
```

frank_network()

Computes Frank's network of the poset `self`.

This is defined in Section 8 of [BF1999].

OUTPUT:

A pair (G, e), where G is Frank's network of P encoded as a `DiGraph`, and e is the cost function on its edges encoded as a dictionary (indexed by these edges, which in turn are encoded as tuples of 2 vertices).

Note: Frank's network of P is a certain directed graph with $2|P| + 2$ vertices, defined in Section 8 of [BF1999]. Its set of vertices consists of two vertices $(0, p)$ and $(1, p)$ for each element p of P, as well as two vertices $(-1, 0)$ and $(2, 0)$. (These notations are not the ones used in [BF1999]; see the table below for their relation.) The edges are:

- for each p in P, an edge from $(-1, 0)$ to $(0, p)$;
- for each p in P, an edge from $(1, p)$ to $(2, 0)$;
- for each p and q in P such that $p \geq q$, an edge from $(0, p)$ to $(1, q)$.

We make this digraph into a network in the sense of flow theory as follows: The vertex $(-1, 0)$ is considered as the source of this network, and the vertex $(2, 0)$ as the sink. The cost function is defined to be 1 on the edge from $(0, p)$ to $(1, p)$ for each $p \in P$, and to be 0 on every other edge. The capacity is 1 on each edge. Here is how to translate this notations into that used in [BF1999]:

```
our notations                    [BF1999]
   (-1, 0)                         s
   (0, p)                        x_p
   (1, p)                        y_p
   (2, 0)                         t
    a[e]                        a(e)
```

REFERENCES:

EXAMPLES:

```
sage: ps = [[16,12,14,-13],[[12,14],[14,-13],[12,16],[16,-13]]]
sage: G, e = Poset(ps).frank_network()
sage: G.edges()
[((-1, 0), (0, -13), None), ((-1, 0), (0, 12), None), ((-1, 0), (0, 14), None), ((-1, 0), (0
sage: e
{((-1, 0), (0, -13)): 0,
 ((-1, 0), (0, 12)): 0,
```

```
 ((-1, 0), (0, 14)): 0,
 ((-1, 0), (0, 16)): 0,
 ((0, -13), (1, -13)): 1,
 ((0, -13), (1, 12)): 0,
 ((0, -13), (1, 14)): 0,
 ((0, -13), (1, 16)): 0,
 ((0, 12), (1, 12)): 1,
 ((0, 14), (1, 12)): 0,
 ((0, 14), (1, 14)): 1,
 ((0, 16), (1, 12)): 0,
 ((0, 16), (1, 16)): 1,
 ((1, -13), (2, 0)): 0,
 ((1, 12), (2, 0)): 0,
 ((1, 14), (2, 0)): 0,
 ((1, 16), (2, 0)): 0}
sage: qs = [[1,2,3,4,5,6,7,8,9],[[1,3],[3,4],[5,7],[1,9],[2,3]]]
sage: Poset(qs).frank_network()
(Digraph on 20 vertices,
 {((-1, 0), (0, 1)): 0,
  ((-1, 0), (0, 2)): 0,
  ((-1, 0), (0, 3)): 0,
  ((-1, 0), (0, 4)): 0,
  ((-1, 0), (0, 5)): 0,
  ((-1, 0), (0, 6)): 0,
  ((-1, 0), (0, 7)): 0,
  ((-1, 0), (0, 8)): 0,
  ((-1, 0), (0, 9)): 0,
  ((0, 1), (1, 1)): 1,
  ((0, 2), (1, 2)): 1,
  ((0, 3), (1, 1)): 0,
  ((0, 3), (1, 2)): 0,
  ((0, 3), (1, 3)): 1,
  ((0, 4), (1, 1)): 0,
  ((0, 4), (1, 2)): 0,
  ((0, 4), (1, 3)): 0,
  ((0, 4), (1, 4)): 1,
  ((0, 5), (1, 5)): 1,
  ((0, 6), (1, 6)): 1,
  ((0, 7), (1, 5)): 0,
  ((0, 7), (1, 7)): 1,
  ((0, 8), (1, 8)): 1,
  ((0, 9), (1, 1)): 0,
  ((0, 9), (1, 9)): 1,
  ((1, 1), (2, 0)): 0,
  ((1, 2), (2, 0)): 0,
  ((1, 3), (2, 0)): 0,
  ((1, 4), (2, 0)): 0,
  ((1, 5), (2, 0)): 0,
  ((1, 6), (2, 0)): 0,
  ((1, 7), (2, 0)): 0,
  ((1, 8), (2, 0)): 0,
  ((1, 9), (2, 0)): 0})
```

AUTHOR:

- Darij Grinberg (2013-05-09)

ge (*x*, *y*)

Return `True` if x is greater than or equal to y in the poset, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_gequal(3, 1)
True
sage: P.is_gequal(2, 2)
True
sage: P.is_gequal(0, 1)
False
```

See also:

`is_greater_than()`, `is_lequal()`.

graphviz_string(*graph_string='graph'*, *edge_string='–'*)

Returns a representation in the DOT language, ready to render in graphviz.

REFERENCES:

- http://www.graphviz.org/doc/info/lang.html

EXAMPLES:

```
sage: P = Poset({'a':['b'],'b':['d'],'c':['d'],'d':['f'],'e':['f'],'f':[]})
sage: print P.graphviz_string()
graph {
"f";"d";"b";"a";"c";"e";
"f"--"e";"d"--"c";"b"--"a";"d"--"b";"f"--"d";
}
```

greene_shape()

Return the Greene-Kleitman partition of `self`.

The Greene-Kleitman partition of a finite poset P is the partition $(c_1 - c_0, c_2 - c_1, c_3 - c_2, \ldots)$, where c_k is the maximum cardinality of a union of k chains of P. Equivalently, this is the conjugate of the partition $(a_1 - a_0, a_2 - a_1, a_3 - a_2, \ldots)$, where a_k is the maximum cardinality of a union of k antichains of P.

See many sources, e. g., [BF1999], for proofs of this equivalence.

EXAMPLES:

```
sage: P = Poset([[3,2,1],[[3,1],[2,1]]])
sage: P.greene_shape()
[2, 1]
sage: P = Poset([[1,2,3,4],[[1,4],[2,4],[4,3]]])
sage: P.greene_shape()
[3, 1]
sage: P = Poset([[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22],[[1,4],[2,4],[4,
sage: P.greene_shape()
[3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
sage: P = Poset([[],[]])
sage: P.greene_shape()
[]
```

AUTHOR:

- Darij Grinberg (2013-05-09)

gt(*x*, *y*)

Return `True` if x is greater than but not equal to y in the poset, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_greater_than(3, 1)
True
sage: P.is_greater_than(1, 2)
False
sage: P.is_greater_than(3, 3)
False
sage: P.is_greater_than(0, 1)
False
```

For non-facade posets also > works:

```
sage: P = Poset({3: [1, 2]}, facade=False)
sage: P(2) > P(3)
True
```

See also:

is_gequal(), is_less_than().

h_polynomial()

Return the h-polynomial of a bounded poset `self`.

This is the h-polynomial of the order complex of the poset minus its bounds.

This is related to the f-polynomial by a simple change of variables:

$$h(q) = (1-q)^{\deg f} f\left(\frac{q}{1-q}\right),$$

where f and h denote the f-polynomial and the h-polynomial, respectively.

See Wikipedia article h-vector.

See also:

is_bounded(), f_polynomial(), order_complex(), sage.homology.simplicial_complex.Simpli

Warning: This is slightly different from the `hPolynomial` method in Macaulay2.

EXAMPLES:

```
sage: P = Posets.AntichainPoset(3).order_ideals_lattice()
sage: P.h_polynomial()
q^3 + 4*q^2 + q
sage: P = Posets.DiamondPoset(5)
sage: P.h_polynomial()
2*q^2 + q
sage: P = Poset({1: []})
sage: P.h_polynomial()
1
```

has_bottom()

Return `True` if the poset has a unique minimal element, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[3], 1:[3], 2:[3], 3:[4], 4:[]})
sage: P.has_bottom()
False
sage: Q = Poset({0:[1], 1:[]})
```

```
sage: Q.has_bottom()
True
```

See also:

bottom(), has_top(), is_bounded()

TESTS:

```
sage: Poset().has_top()  # Test empty poset
False
```

has_isomorphic_subposet(*other*)

Return True if the poset contains a subposet isomorphic to other.

By subposet we mean that there exist a set X of elements such that self.subposet(X) is isomorphic to other.

INPUT:

- other – a finite poset

EXAMPLES:

```
sage: D = Poset({1:[2,3], 2:[4], 3:[4]})
sage: T = Poset({1:[2,3], 2:[4,5], 3:[6,7]})
sage: N5 = Posets.PentagonPoset()

sage: N5.has_isomorphic_subposet(T)
False
sage: N5.has_isomorphic_subposet(D)
True

sage: len([P for P in Posets(5) if P.has_isomorphic_subposet(D)])
11
```

has_top()

Return True if the poset has a unique maximal element, and False otherwise.

EXAMPLES:

```
sage: P = Poset({0:[3], 1:[3], 2:[3], 3:[4, 5], 4:[], 5:[]})
sage: P.has_top()
False
sage: Q = Poset({0:[3], 1:[3], 2:[3], 3:[4], 4:[]})
sage: Q.has_top()
True
```

See also:

top(), has_bottom(), is_bounded()

TESTS:

```
sage: Poset().has_top()  # Test empty poset
False
```

hasse_diagram(*wrapped=True*)

Return the Hasse diagram of self as a Sage DiGraph. If dot2tex is installed, then this sets the Hasse diagram's latex options to use the dot2tex formatting.

Todo

Should the vertices of the diagram have the poset as parent?

EXAMPLES:

```
sage: Q = Poset({5:[2,3], 1:[3,4], 2:[0], 3:[0], 4:[0]}, facade = False)
sage: Q.hasse_diagram()
Digraph on 6 vertices

sage: P = Poset({'a':['b'],'b':['d'],'c':['d'],'d':['f'],'e':['f'],'f':[]}, facade = False)
sage: H = P.hasse_diagram()
sage: P.cover_relations()
[[e, f], [c, d], [a, b], [b, d], [d, f]]
sage: H.edges()
[(a, b, None), (c, d, None), (b, d, None), (e, f, None), (d, f, None)]

sage: P = Poset((divisors(15), attrcall("divides")), facade = False)
sage: H = P.hasse_diagram()
sage: H.vertices()
[1, 3, 5, 15]
sage: H.edges()
[(1, 3, None), (1, 5, None), (3, 15, None), (5, 15, None)]
sage: H.set_latex_options(format="dot2tex")   # optional - dot2tex
sage: view(H, tight_page=True)  # optional - dot2tex, not tested (opens external window)
```

height()

Return the height (number of elements in a longest chain) of the poset.

EXAMPLES:

```
sage: P = Poset({0:[1],2:[3,4],4:[5,6]})
sage: P.height()
3
sage: Posets.PentagonPoset().height()
4
```

TESTS:

```
sage: Poset().height()
0
```

incidence_algebra(*R*, *prefix='I'*)

Return the incidence algebra of `self` over `R`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: P.incidence_algebra(QQ)
Incidence algebra of Finite lattice containing 16 elements
 over Rational Field
```

incomparability_graph()

Returns the incomparability graph of `self`.

This is the complement of the comparability graph.

See also:

`comparability_graph()`, `sage.graphs.comparability`

EXAMPLES:

```
sage: p = posets.ChainPoset(4)
sage: p.incomparability_graph().size()
0

sage: p = posets.DiamondPoset(5)
sage: g = p.incomparability_graph(); g
Incomparability graph on 5 vertices
sage: g.size()
3
```

interval(*x*, *y*)

Return a list of the elements z such that $x \leq z \leq y$.

INPUT:

- x – any element of the poset
- y – any element of the poset

EXAMPLES:

```
sage: uc = [[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]]
sage: dag = DiGraph(dict(zip(range(len(uc)),uc)))
sage: P = Poset(dag)
sage: I = set(map(P,[2,5,6,4,7]))
sage: I == set(P.interval(2,7))
True

sage: dg = DiGraph({"a":["b","c"], "b":["d"], "c":["d"]})
sage: P = Poset(dg, facade = False)
sage: P.interval("a","d")
[a, b, c, d]
```

intervals(**args*, ***kwds*)

Deprecated: Use `relations()` instead. See trac ticket #19360 for details.

intervals_iterator(**args*, ***kwds*)

Deprecated: Use `relations_iterator()` instead. See trac ticket #19360 for details.

intervals_number()

Return the number of relations in the poset.

A relation is a pair of elements x and y such that $x \leq y$ in the poset.

Relations are also often called intervals. The number of intervals is the dimension of the incidence algebra.

See also:

`relations_iterator()`, `relations()`

EXAMPLES:

```
sage: P = Posets.PentagonPoset()
sage: P.relations_number()
13

sage: from sage.combinat.tamari_lattices import TamariLattice
sage: TamariLattice(4).relations_number()
68
```

TESTS:

```
sage: Poset().relations_number()
0
```

is_EL_labelling(*f*, *return_raising_chains=False*)

Returns `True` if `f` is an EL labelling of `self`.

A labelling f of the edges of the Hasse diagram of a poset is called an EL labelling (edge lexicographic labelling) if for any two elements u and v with $u \leq v$,

- there is a unique f-raising chain from u to v in the Hasse diagram, and this chain is lexicographically first among all chains from u to v.

For more details, see [Bj1980].

INPUT:

- `f` – a function taking two elements `a` and `b` in `self` such that `b` covers `a` and returning elements in a totally ordered set.
- `return_raising_chains` (optional; default:`False`) if `True`, returns the set of all raising chains in `self`, if possible.

EXAMPLES:

Let us consider a Boolean poset:

```
sage: P = Poset([[(0,0),(0,1),(1,0),(1,1)],[[(0,0),(0,1)],[(0,0),(1,0)],[(0,1),(1,1)],[(1,0)
sage: label = lambda a,b: min( i for i in [0,1] if a[i] != b[i] )
sage: P.is_EL_labelling(label)
True
sage: P.is_EL_labelling(label,return_raising_chains=True)
{((0, 0), (0, 1)): [1],
 ((0, 0), (1, 0)): [0],
 ((0, 0), (1, 1)): [0, 1],
 ((0, 1), (1, 1)): [0],
 ((1, 0), (1, 1)): [1]}
```

REFERENCES:

is_bounded()

Return `True` if the poset is bounded, and `False` otherwise.

A poset is bounded if it contains both a unique maximal element and a unique minimal element.

EXAMPLES:

```
sage: P = Poset({0:[3], 1:[3], 2:[3], 3:[4, 5], 4:[], 5:[]})
sage: P.is_bounded()
False
sage: Q = Posets.DiamondPoset(5)
sage: Q.is_bounded()
True
```

See also:

`has_bottom()`, `has_top()`

TESTS:

```
sage: Poset().is_bounded()  # Test empty poset
False
sage: Poset({1:[]}).is_bounded()  # Here top == bottom
True
```

```
sage: ( len([P for P in Posets(5) if P.is_bounded()]) ==
....: Posets(3).cardinality() )
True
```

is_chain()

Return `True` if the poset is totally ordered ("chain"), and `False` otherwise.

EXAMPLES:

```
sage: I = Poset({0:[1], 1:[2], 2:[3], 3:[4]})
sage: I.is_chain()
True

sage: II = Poset({0:[1], 2:[3]})
sage: II.is_chain()
False

sage: V = Poset({0:[1, 2]})
sage: V.is_chain()
False
```

TESTS:

```
sage: [len([P for P in Posets(n) if P.is_chain()]) for n in range(5)]
[1, 1, 1, 1, 1]
```

is_chain_of_poset(*elms*, *ordered=False*)

Return `True` if $elms$ is a chain of the poset, and `False` otherwise.

INPUT:

- `elms` – a list or other iterable containing some elements of the poset
- `ordered` – a Boolean. If `True`, then return `True` only if elements in $elms$ are strictly increasing in the poset; this makes no sense if $elms$ is a set. If `False` (the default), then elements can be repeated and be in any order.

EXAMPLES:

```
sage: P = Poset((divisors(12), attrcall("divides")))
sage: sorted(P.list())
[1, 2, 3, 4, 6, 12]
sage: P.is_chain_of_poset([12, 3])
True
sage: P.is_chain_of_poset({3, 4, 12})
False
sage: P.is_chain_of_poset([12, 3], ordered=True)
False
sage: P.is_chain_of_poset((1, 1, 3))
True
sage: P.is_chain_of_poset((1, 1, 3), ordered=True)
False
sage: P.is_chain_of_poset((1, 3), ordered=True)
True
```

TESTS:

```
sage: P = Posets.BooleanLattice(4)
sage: P.is_chain_of_poset([])
True
sage: P.is_chain_of_poset((1,3,7,15,14))
```

```
False
sage: P.is_chain_of_poset({10})
True
sage: P.is_chain_of_poset([32])
Traceback (most recent call last):
...
ValueError: element (=32) not in poset
```

is_connected()

Return `True` if the poset is connected, and `False` otherwise.

A poset is connected if it's Hasse diagram is connected.

If a poset is not connected, then it can be divided to parts S_1 and S_2 so that every element of S_1 is incomparable to every element of S_2.

EXAMPLES:

```
sage: P = Poset({1:[2, 3], 3:[4, 5]})
sage: P.is_connected()
True

sage: P = Poset({1:[2, 3], 3:[4, 5], 6:[7, 8]})
sage: P.is_connected()
False
```

See also:

`connected_components()`

TESTS:

```
sage: Poset().is_connected()  # Test empty poset
True
```

is_eulerian()

Return `True` if the poset is Eulerian, and `False` otherwise.

The poset is expected to be graded and bounded.

A poset is Eulerian if every non-trivial interval has the same number of elements of even rank as of odd rank.

See Wikipedia article Eulerian_poset.

EXAMPLES:

```
sage: P = Poset({0:[1, 2, 3], 1:[4, 5], 2:[4, 6], 3:[5, 6],
....: 4:[7, 8], 5:[7, 8], 6:[7, 8], 7:[9], 8:[9]})
sage: P.is_eulerian()
True
sage: P = Poset({0:[1, 2, 3], 1:[4, 5, 6], 2:[4, 6], 3:[5,6],
....: 4:[7], 5:[7], 6:[7]})
sage: P.is_eulerian()
False
```

Canonical examples of Eulerian posets are the face lattices of convex polytopes:

```
sage: P = polytopes.cube().face_lattice()
sage: P.is_eulerian()
True
```

TESTS:

```
sage: Poset().is_eulerian()
Traceback (most recent call last):
...
TypeError: the poset is not bounded
sage: Posets.PentagonPoset().is_eulerian()
Traceback (most recent call last):
...
TypeError: the poset is not graded
sage: Poset({1: []}).is_eulerian()
True
```

is_gequal(*x*, *y*)

Return `True` if x is greater than or equal to y in the poset, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_gequal(3, 1)
True
sage: P.is_gequal(2, 2)
True
sage: P.is_gequal(0, 1)
False
```

See also:

`is_greater_than()`, `is_lequal()`.

is_graded()

Return `True` if the poset is graded, and `False` otherwise.

A poset is graded if all its maximal chains have the same length.

There are various competing definitions for graded posets (see Wikipedia article Graded_poset). This definition is from section 3.1 of Richard Stanley's *Enumerative Combinatorics, Vol. 1* [EnumComb1].

Every graded poset is ranked. The converse is true for bounded posets, including lattices.

EXAMPLES:

```
sage: P = Posets.PentagonPoset()  # Not even ranked
sage: P.is_graded()
False

sage: P = Poset({1:[2, 3], 3:[4]})  # Ranked, but not graded
sage: P.is_graded()
False

sage: P = Poset({1:[3, 4], 2:[3, 4], 5:[6]})
sage: P.is_graded()
True

sage: P = Poset([[1], [2], [], [4], []])
sage: P.is_graded()
False
```

See also:

`is_ranked()`, `level_sets()`

TESTS:

```
sage: Poset().is_graded()  # Test empty poset
True
```

is_greater_than(*x*, *y*)

Return `True` if x is greater than but not equal to y in the poset, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_greater_than(3, 1)
True
sage: P.is_greater_than(1, 2)
False
sage: P.is_greater_than(3, 3)
False
sage: P.is_greater_than(0, 1)
False
```

For non-facade posets also > works:

```
sage: P = Poset({3: [1, 2]}, facade=False)
sage: P(2) > P(3)
True
```

See also:

`is_gequal()`, `is_less_than()`.

is_incomparable_chain_free(*m*, *n=None*)

Return `True` if the poset is $(m+n)$-free, and `False` otherwise.

A poset is $(m+n)$-free if there is no incomparable chains of lengths m and n. Three cases have special name:

- ''interval order'' is $(2+2)$-free
- ''semiorder'' (or ''unit interval order'') is $(1+3)$-free and $(2+2)$-free
- ''weak order'' is $(1+2)$-free.

INPUT:

- `m`, `n` - positive integers

It is also possible to give a list of integer pairs as argument. See below for an example.

EXAMPLES:

```
sage: B3 = Posets.BooleanLattice(3)
sage: B3.is_incomparable_chain_free(1, 3)
True
sage: B3.is_incomparable_chain_free(2, 2)
False

sage: IP6 = Posets.IntegerPartitions(6)
sage: IP6.is_incomparable_chain_free(1, 3)
False
sage: IP6.is_incomparable_chain_free(2, 2)
True
```

A list of pairs as an argument:

```
sage: B3.is_incomparable_chain_free([[1, 3], [2, 2]])
False
```

We show how to get an incomparable chain pair:

```
sage: P = Posets.PentagonPoset()
sage: chains_1_2 = Poset({0:[], 1:[2]})
sage: incomps = P.isomorphic_subposets(chains_1_2)[0]
sage: sorted(incomps.list()), incomps.cover_relations()
([1, 2, 3], [[2, 3]])
```

TESTS:

```
sage: Poset().is_incomparable_chain_free(1,1)  # Test empty poset
True

sage: [len([p for p in Posets(n) if p.is_incomparable_chain_free(((3, 1), (2, 2)))]) for n i
[1, 1, 2, 5, 14, 42]

sage: Q = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: Q.is_incomparable_chain_free(2, 20/10)
True
sage: Q.is_incomparable_chain_free(2, pi)
Traceback (most recent call last):
...
TypeError: 2 and pi must be integers.
sage: Q.is_incomparable_chain_free(2, -1)
Traceback (most recent call last):
...
ValueError: 2 and -1 must be positive integers.
sage: P = Poset(((0, 1, 2, 3, 4), ((0, 1), (1, 2), (0, 3), (4, 2))))
sage: P.is_incomparable_chain_free((3, 1))
Traceback (most recent call last):
...
TypeError: (3, 1) is not a tuple of tuples.
sage: P.is_incomparable_chain_free([3, 1], [2, 2])
Traceback (most recent call last):
...
TypeError: [3, 1] and [2, 2] must be integers.
sage: P.is_incomparable_chain_free([[3, 1], [2, 2]])
True
sage: P.is_incomparable_chain_free(([3, 1], [2, 2]))
True
sage: P.is_incomparable_chain_free([3, 1], 2)
Traceback (most recent call last):
...
TypeError: [3, 1] and 2 must be integers.
sage: P.is_incomparable_chain_free(([3, 1], [2, 2, 2]))
Traceback (most recent call last):
...
ValueError: '([3, 1], [2, 2, 2])' is not a tuple of length-2 tuples.
```

AUTHOR:

- Eric Rowland (2013-05-28)

REFERENCES:

is_induced_subposet(*other*)

Return `True` if the poset is an induced subposet of `other`, and `False` otherwise.

A poset P is an induced subposet of Q if every element of P is an element of Q, and $x \leq_P y$ iff $x \leq_Q y$. Note that "induced" here has somewhat different meaning compared to that of graphs.

INPUT:

- `other`, a poset.

Note: This method does not check whether the poset is a *isomorphic* (i.e., up to relabeling) subposet of `other`, but only if `other` directly contains the poset as an induced subposet. For isomorphic subposets see `has_isomorphic_subposet()`.

EXAMPLES:

```
sage: P = Poset({1:[2, 3]})
sage: Q = Poset({1:[2, 4], 2:[3]})
sage: P.is_induced_subposet(Q)
False
sage: R = Poset({0:[1], 1:[3, 4], 3:[5], 4:[2]})
sage: P.is_induced_subposet(R)
True
```

TESTS:

```
sage: P = Poset({2:[1]})
sage: Poset().is_induced_subposet(P)
True
sage: Poset().is_induced_subposet(Poset())
True
sage: P.is_induced_subposet(Poset())
False
```

Bad input:

```
sage: Poset().is_induced_subposet('junk')
Traceback (most recent call last):
...
AttributeError: 'str' object has no attribute 'subposet'
```

is_isomorphic(*other*)

Returns True if both posets are isomorphic.

EXAMPLES:

```
sage: P = Poset(([1,2,3],[[1,3],[2,3]]))
sage: Q = Poset(([4,5,6],[[4,6],[5,6]]))
sage: P.is_isomorphic( Q )
True
```

is_join_semilattice()

Return `True` if the poset has a join operation, and `False` otherwise.

A join is the least upper bound for given elements, if it exists.

EXAMPLES:

```
sage: P = Poset([[1,3,2], [4], [4,5,6], [6], [7], [7], [7], []])
sage: P.is_join_semilattice()
True
```

Elements 3 and 4 have no common upper bound at all; elements 1 and 2 have upper bounds 3 and 4, but no least upper bound:

```
sage: P = Poset({1:[3, 4], 2:[3, 4]})
sage: P.is_join_semilattice()
False
```

See also:

is_meet_semilattice(), is_lattice()

TESTS:

```
sage: Poset().is_join_semilattice()  # Test empty lattice
True
sage: len([P for P in Posets(4) if P.is_join_semilattice()])
5
```

is_lequal(x, y)

Return `True` if x is less than or equal to y in the poset, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_lequal(2, 4)
True
sage: P.is_lequal(2, 2)
True
sage: P.is_lequal(0, 1)
False
sage: P.is_lequal(3, 2)
False
```

See also:

is_less_than(), is_gequal().

is_less_than(x, y)

Return `True` if x is less than but not equal to y in the poset, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_less_than(1, 3)
True
sage: P.is_less_than(0, 1)
False
sage: P.is_less_than(2, 2)
False
```

For non-facade posets also < works:

```
sage: P = Poset({3: [1, 2]}, facade=False)
sage: P(1) < P(2)
False
```

See also:

is_lequal(), is_greater_than().

is_linear_extension(l)

Returns whether `l` is a linear extension of `self`

INPUT:

- `l` – a list (or iterable) containing all of the elements of `self` exactly once

See also:

linear_extension(), linear_extensions()

EXAMPLES:

```
sage: P = Poset((divisors(12), attrcall("divides")), facade=True, linear_extension=True)
sage: P.list()
[1, 2, 3, 4, 6, 12]
sage: P.is_linear_extension([1, 2, 4, 3, 6, 12])
True
sage: P.is_linear_extension([1, 2, 4, 6, 3, 12])
False

sage: [p for p in Permutations(list(P)) if P.is_linear_extension(p)]
[[1, 2, 3, 4, 6, 12],
 [1, 2, 3, 6, 4, 12],
 [1, 2, 4, 3, 6, 12],
 [1, 3, 2, 4, 6, 12],
 [1, 3, 2, 6, 4, 12]]
sage: list(P.linear_extensions())
[[1, 2, 3, 4, 6, 12],
 [1, 2, 3, 6, 4, 12],
 [1, 2, 4, 3, 6, 12],
 [1, 3, 2, 4, 6, 12],
 [1, 3, 2, 6, 4, 12]]
```

Note: This is used and systematically tested in LinearExtensionsOfPosets

TESTS:

Check that trac ticket #15313 is fixed:

```
sage: P = Poset((divisors(12), attrcall("divides")), facade=True, linear_extension=True)
sage: P.is_linear_extension([1,2,4,3,6,12,1337])
False
sage: P.is_linear_extension([1,2,4,3,6,666,12,1337])
False
sage: P = Poset(DiGraph(5))
sage: P.is_linear_extension(['David', 'McNeil', 'La', 'Lamentable', 'Aventure', 'de', 'Simor
False
```

is_meet_semilattice()

Return True if the poset has a meet operation, and False otherwise.

A meet is the greatest lower bound for given elements, if it exists.

EXAMPLES:

```
sage: P = Poset({1:[2, 3, 4], 2:[5, 6], 3:[6], 4:[6, 7]})
sage: P.is_meet_semilattice()
True

sage: Q = P.dual()
sage: Q.is_meet_semilattice()
False
```

See also:

is_join_semilattice(), is_lattice()

TESTS:

```
sage: Poset().is_meet_semilattice()  # Test empty lattice
True
sage: len([P for P in Posets(4) if P.is_meet_semilattice()])
5
```

is_parent_of(*x*)

Returns True if x is an element of the poset.

TESTS:

```
sage: from sage.combinat.posets.posets import FinitePoset
sage: P5 = FinitePoset(DiGraph({(5,):[(4,1),(3,2)], \
        (4,1):[(3,1,1),(2,2,1)], \
        (3,2):[(3,1,1),(2,2,1)], \
        (3,1,1):[(2,1,1,1)], \
        (2,2,1):[(2,1,1,1)], \
        (2,1,1,1):[(1,1,1,1,1)], \
        (1,1,1,1,1):[]}))
sage: x = P5.list()[3]
sage: x in P5
True
```

For the sake of speed, an element with the right class and parent is assumed to be in this parent. This can possibly be counterfeited by feeding garbage to the constructor:

```
sage: x = P5.element_class(P5, "a", 5)
sage: x in P5
True
```

is_rank_symmetric()

Return `True` if the poset is rank symmetric, and `False` otherwise.

The poset is expected to be graded and connected.

A poset of rank h (maximal chains have $h+1$ elements) is rank symmetric if the number of elements are equal in ranks i and $h-i$ for every i in $0, 1, \ldots, h$.

EXAMPLES:

```
sage: P = Poset({1:[3, 4, 5], 2:[3, 4, 5], 3:[6], 4:[7], 5:[7]})
sage: P.is_rank_symmetric()
True
sage: P = Poset({1:[2], 2:[3, 4], 3:[5], 4:[5]})
sage: P.is_rank_symmetric()
False
```

TESTS:

```
sage: Poset().is_rank_symmetric()  # Test empty poset
True
```

is_ranked()

Return `True` if the poset is ranked, and `False` otherwise.

A poset is ranked if there is a function r from poset elements to integers so that $r(x) = r(y) + 1$ when x covers y.

Informally said a ranked poset can be "levelized": every element is on a "level", and every cover relation goes only one level up.

EXAMPLES:

```
sage: P = Poset( ([1, 2, 3, 4], [[1, 2], [2, 4], [3, 4]] ))
sage: P.is_ranked()
True

sage: P = Poset([[1, 5], [2, 6], [3], [4],[], [6, 3], [4]])
sage: P.is_ranked()
False
```

See also:

`rank_function()`, `rank()`, `is_graded()`

TESTS:

```
sage: Poset().is_ranked()  # Test empty poset
True
```

is_slender()

Return `True` if the poset is slender, and `False` otherwise.

A finite graded poset is called slender if every rank 2 interval contains three or four elements, as defined in [Stan2009]. (This notion of "slender" is unrelated to the eponymous notion defined by Graetzer and Kelly in "The Free $\mathfrak{m}$-Lattice on the Poset H", Order 1 (1984), 47–65.)

This function *does not* check if the poset is graded or not. Instead it just returns `True` if the poset does not contain 5 distinct elements x, y, a, b and c such that $x \lessdot a, b, c \lessdot y$ where $\lessdot$ is the covering relation.

EXAMPLES:

```
sage: P = Poset(([1, 2, 3, 4], [[1, 2], [1, 3], [2, 4], [3, 4]]))
sage: P.is_slender()
True
sage: P = Poset(([1,2,3,4,5],[[1,2],[1,3],[1,4],[2,5],[3,5],[4,5]]))
sage: P.is_slender()
False

sage: W = WeylGroup(['A', 2])
sage: G = W.bruhat_poset()
sage: G.is_slender()
True
sage: W = WeylGroup(['A', 3])
sage: G = W.bruhat_poset()
sage: G.is_slender()
True
```

TESTS:

```
sage: Poset().is_slender()  # Test empty poset
True
```

isomorphic_subposets(*other*)

Return a list of subposets of $self$ isomorphic to $other$.

By subposet we mean `self.subposet(X)` which is isomorphic to `other` and where `X` is a subset of elements of `self`.

INPUT:

- `other` – a finite poset

EXAMPLES:

```
sage: C2=Poset({0:[1]})
sage: C3=Poset({'a':['b'], 'b':['c']})
sage: for x in C3.isomorphic_subposets(C2): print x.cover_relations()
[['b', 'c']]
[['a', 'c']]
[['a', 'b']]
sage: D = Poset({1:[2,3], 2:[4], 3:[4]})
sage: N5 = Posets.PentagonPoset()
sage: len(N5.isomorphic_subposets(D))
2
```

Note: If this function takes too much time, try using `isomorphic_subposets_iterator()`.

isomorphic_subposets_iterator(*other*)

Return an iterator over the subposets of $self$ isomorphic to $other$.

By subposet we mean `self.subposet(X)` which is isomorphic to `other` and where `X` is a subset of elements of `self`.

INPUT:

- `other` – a finite poset

EXAMPLES:

```
sage: D = Poset({1:[2,3], 2:[4], 3:[4]})
sage: N5 = Posets.PentagonPoset()
sage: for P in N5.isomorphic_subposets_iterator(D):
....:     print P.cover_relations()
[[0, 1], [0, 2], [1, 4], [2, 4]]
[[0, 1], [0, 3], [1, 4], [3, 4]]
[[0, 1], [0, 2], [1, 4], [2, 4]]
[[0, 1], [0, 3], [1, 4], [3, 4]]
```

Warning: This function will return same subposet as many times as there are automorphism on it. This is due to `subgraph_search_iterator()` returning labelled subgraphs. On the other hand, this function does not eat memory like `isomorphic_subposets()` does.

join_matrix()

Deprecated as a function of posets, moved to lattices.

Convert a poset P to join-semilattice and use it like `JoinSemilattice(P).join_matrix()`.

kazhdan_lusztig_polynomial(*x=None*, *y=None*, *q=None*, *canonical_labels=None*)

Return the Kazhdan-Lusztig polynomial $P_{x,y}(q)$ of `self`.

We follow the definition given in [EPW14]. Let G denote a graded poset with unique minimal and maximal elements and χ_G denote the characteristic polynomial of G. Let I_x and F^x denote the principal order ideal and filter of x respectively. Define the *Kazhdan-Lusztig polynomial* of G as the unique polynomial $P_G(q)$ satisfying the following:

1. If $\operatorname{rank} G = 0$, then $P_G(q) = 1$.
2. If $\operatorname{rank} G > 0$, then $\deg P_G(q) < \frac{1}{2} \operatorname{rank} G$.
3. We have

$$q^{\operatorname{rank} G} P_G(q^{-1}) = \sum_{x \in G} \chi_{I_x}(q) P_{F^x}(q).$$

We then extend this to $P_{x,y}(q)$ by considering the subposet corresponding to the (closed) interval $[x, y]$. We also define $P_{\emptyset}(q) = 0$ (so if $x \not\leq y$, then $P_{x,y}(q) = 0$).

INPUT:

- `q` – (default: $q \in \mathbf{Z}[q]$) the indeterminate q
- `x` – (default: the minimal element) the element x
- `y` – (default: the maximal element) the element y
- `canonical_labels` – (optional) for subposets, use the canonical labeling (this can limit recursive calls for posets with large amounts of symmetry, but producing the labeling takes time); if not specified, this is `True` if `x` and `y` are both not specified and `False` otherwise

EXAMPLES:

```
sage: L = posets.BooleanLattice(3)
sage: L.kazhdan_lusztig_polynomial()
1

sage: L = posets.SymmetricGroupWeakOrderPoset(4)
sage: L.kazhdan_lusztig_polynomial()
1
sage: x = '2314'
sage: y = '3421'
sage: L.kazhdan_lusztig_polynomial(x, y)
-q + 1
sage: L.kazhdan_lusztig_polynomial(x, y, var('t'))
-t + 1
```

REFERENCES:

AUTHORS:

- Travis Scrimshaw (27-12-2014)

le(*x*, *y*)

Return `True` if x is less than or equal to y in the poset, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_lequal(2, 4)
True
sage: P.is_lequal(2, 2)
True
sage: P.is_lequal(0, 1)
False
sage: P.is_lequal(3, 2)
False
```

See also:

`is_less_than()`, `is_gequal()`.

lequal_matrix(*ring=Integer Ring*, *sparse=False*)

Computes the matrix whose `(i,j)` entry is 1 if `self.linear_extension()[i] < self.linear_extension()[j]` and 0 otherwise.

INPUT:

- `ring` – the ring of coefficients (default: `ZZ`)
- `sparse` – whether the returned matrix is sparse or not (default: `True`)

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]], facade = False)
sage: LEQM = P.lequal_matrix(); LEQM
[1 1 1 1 1 1 1 1]
[0 1 0 1 0 0 0 1]
[0 0 1 1 1 0 1 1]
[0 0 0 1 0 0 0 1]
[0 0 0 0 1 0 0 1]
[0 0 0 0 0 1 1 1]
[0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 0 1]
sage: LEQM[1,3]
1
sage: P.linear_extension()[1] < P.linear_extension()[3]
True
sage: LEQM[2,5]
0
sage: P.linear_extension()[2] < P.linear_extension()[5]
False
```

We now demonstrate the usage of the optional parameters:

```
sage: P.lequal_matrix(ring=QQ, sparse=False).parent()
Full MatrixSpace of 8 by 8 dense matrices over Rational Field
```

level_sets()

Return a list `l` such that `l[i]` is the set of minimal elements of the poset obtained from `self` by removing the elements in `l[0], l[1], ..., l[i-1]`. (In particular, `l[0]` is the set of minimal elements of `self`.)

EXAMPLES:

```
sage: P = Poset({0:[1,2],1:[3],2:[3],3:[]})
sage: [len(x) for x in P.level_sets()]
[1, 2, 1]

sage: Q = Poset({0:[1,2], 1:[3], 2:[4], 3:[4]})
sage: [len(x) for x in Q.level_sets()]
[1, 2, 1, 1]
```

linear_extension(*linear_extension=None*, *check=True*)

Return a linear extension of this poset.

A linear extension of a finite poset P of size n is a total ordering $\pi := \pi_0\pi_1\ldots\pi_{n-1}$ of its elements such that $i < j$ whenever $\pi_i < \pi_j$ in the poset P.

INPUT:

- `linear_extension` – (default: `None`) a list of the elements of `self`
- `check` – a boolean (default: True); whether to check that `linear_extension` is indeed a linear extension of `self`.

See also:

`is_linear_extension()`, `linear_extensions()`

EXAMPLES:

```
sage: P = Poset((divisors(15), attrcall("divides")), facade=True)
```

Without optional argument, the default linear extension of the poset is returned, as a plain list:

```
sage: P.linear_extension()
[1, 3, 5, 15]
```

Otherwise, a full-featured linear extension is constructed as an element of `P.linear_extensions()`:

```
sage: l = P.linear_extension([1,5,3,15]); l
[1, 5, 3, 15]
sage: type(l)
<class 'sage.combinat.posets.linear_extensions.LinearExtensionsOfPoset_with_category.element
sage: l.parent()
The set of all linear extensions of Finite poset containing 4 elements
```

By default, the linear extension is checked for correctness:

```
sage: l = P.linear_extension([1,3,15,5])
Traceback (most recent call last):
...
ValueError: [1, 3, 15, 5] is not a linear extension of Finite poset containing 4 elements
```

This can be disabled (at your own risks!) with:

```
sage: P.linear_extension([1,3,15,5], check=False)
[1, 3, 15, 5]
```

Todo

- Is it acceptable to have those two features for a single method?
- In particular, we miss a short idiom to get the default linear extension

---

`linear_extensions` (*facade=False*)

Returns the enumerated set of all the linear extensions of this poset

INPUT:

- `facade` – a boolean (default: `False`); whether to return the linear extensions as plain lists

> **Warning:** The `facade` option is not yet fully functional:
>
> ```
> sage: P = Poset((divisors(12), attrcall("divides")), linear_extension=True)
> sage: L = P.linear_extensions(facade=True); L
> The set of all linear extensions of Finite poset containing 6 elements with distinguish
> sage: L([1, 2, 3, 4, 6, 12])
> Traceback (most recent call last):
> ...
> TypeError: Cannot convert list to sage.structure.element.Element
> ```

See also:

`linear_extension()`, `is_linear_extension()`

EXAMPLES:

```
sage: P = Poset((divisors(12), attrcall("divides")), linear_extension=True)
sage: P.list()
[1, 2, 3, 4, 6, 12]
sage: L = P.linear_extensions(); L
The set of all linear extensions of Finite poset containing 6 elements with distinguished li
```

```
sage: l = L.an_element(); l
[1, 2, 3, 4, 6, 12]
sage: L.cardinality()
5
sage: L.list()
[[1, 2, 3, 4, 6, 12],
 [1, 2, 3, 6, 4, 12],
 [1, 2, 4, 3, 6, 12],
 [1, 3, 2, 4, 6, 12],
 [1, 3, 2, 6, 4, 12]]
```

Each element is aware that it is a linear extension of P:

```
sage: type(l.parent())
<class 'sage.combinat.posets.linear_extensions.LinearExtensionsOfPoset_with_category'>
```

With `facade=True`, the elements of `L` are plain lists instead:

```
sage: L = P.linear_extensions(facade=True)
sage: l = L.an_element()
sage: type(l)
<type 'list'>
```

Warning: In Sage <= 4.8, this function used to return a plain list of lists. To recover the previous functionality, please use:

```
sage: L = list(P.linear_extensions(facade=True)); L
[[1, 2, 3, 4, 6, 12],
 [1, 2, 3, 6, 4, 12],
 [1, 2, 4, 3, 6, 12],
 [1, 3, 2, 4, 6, 12],
 [1, 3, 2, 6, 4, 12]]
sage: type(L[0])
<type 'list'>
```

TESTS:

```
sage: D = Poset({ 0:[1,2], 1:[3], 2:[3,4] })
sage: list(D.linear_extensions())
[[0, 1, 2, 3, 4], [0, 1, 2, 4, 3], [0, 2, 1, 3, 4], [0, 2, 1, 4, 3], [0, 2, 4, 1, 3]]
```

linear_extensions_graph()

Return the linear extensions graph of the poset.

Vertices of the graph are linear extensions of the poset. Two vertices are connected by an edge if the linear extensions differ by only one adjacent transposition.

EXAMPLES:

```
sage: P = Poset({1:[3,4],2:[4]})
sage: G = P.linear_extensions_graph(); G
Graph on 5 vertices
sage: G.degree_sequence()
[3, 2, 2, 2, 1]

sage: chevron = Poset({1:[2,6], 2:[3], 4:[3,5], 6:[5]})
sage: G = chevron.linear_extensions_graph(); G
Graph on 22 vertices
```

```
sage: G.size()
36
```

TESTS:

```
sage: Poset().linear_extensions_graph()
Graph on 1 vertex

sage: A4 = Posets.AntichainPoset(4)
sage: G = A4.linear_extensions_graph()
sage: G.is_regular()
True
```

list()

List the elements of the poset. This just returns the result of `linear_extension()`.

EXAMPLES:

```
sage: D = Poset({ 0:[1,2], 1:[3], 2:[3,4] }, facade = False)
sage: D.list()
[0, 1, 2, 3, 4]
sage: type(D.list()[0])
<class 'sage.combinat.posets.elements.FinitePoset_with_category.element_class'>
```

lower_covers(x)

Return the list of lower covers of the element `x`.

A lower cover of x is an element y such that $y < x$ and there is no element z so that $y < z < x$.

EXAMPLES:

```
sage: P = Poset([[1,5], [2,6], [3], [4], [], [6,3], [4]])
sage: P.lower_covers(3)
[2, 5]
sage: P.lower_covers(0)
[]
```

See also:

`upper_covers()`

lower_covers_iterator(x)

Return an iterator over the lower covers of the element `x`.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[]})
sage: l0 = P.lower_covers_iterator(3)
sage: type(l0)
<type 'generator'>
sage: l0.next()
2
```

lt(x, y)

Return `True` if x is less than but not equal to y in the poset, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.is_less_than(1, 3)
True
sage: P.is_less_than(0, 1)
```

```
False
sage: P.is_less_than(2, 2)
False
```

For non-facade posets also < works:

```
sage: P = Poset({3: [1, 2]}, facade=False)
sage: P(1) < P(2)
False
```

See also:

is_lequal(), is_greater_than().

maximal_antichains()

Return the maximal antichains of the poset.

An antichain a of poset P is *maximal* if there is no element $e \in P \setminus a$ such that $a \cup \{e\}$ is an antichain.

EXAMPLES:

```
sage: P=Poset({'a':['b', 'c'], 'b':['d','e']})
sage: P.maximal_antichains()
[['a'], ['b', 'c'], ['c', 'd', 'e']]

sage: Posets.PentagonPoset().maximal_antichains()
[[0], [1, 2], [1, 3], [4]]
```

See also:

antichains(), maximal_chains()

maximal_chains(*partial=None*)

Return all maximal chains of this poset.

Each chain is listed in increasing order.

INPUT:

- partial – list (optional); if present, find all maximal chains starting with the elements in partial

Returns list of the maximal chains of this poset.

This is used in constructing the order complex for the poset.

EXAMPLES:

```
sage: P = Posets.BooleanLattice(3)
sage: P.maximal_chains()
[[0, 1, 3, 7], [0, 1, 5, 7], [0, 2, 3, 7], [0, 2, 6, 7], [0, 4, 5, 7], [0, 4, 6, 7]]
sage: P.maximal_chains(partial=[0,2])
[[0, 2, 3, 7], [0, 2, 6, 7]]
sage: Q = Posets.ChainPoset(6)
sage: Q.maximal_chains()
[[0, 1, 2, 3, 4, 5]]
```

See also:

maximal_antichains(), chains()

maximal_elements()

Return the list of the maximal elements of the poset.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.maximal_elements()
[4]
```

See also:

minimal_elements().

meet_matrix()
Deprecated as a function of posets, moved to lattices.

Convert a poset P to meet-semilattice and use it like MeetSemilattice(P).join_matrix().

minimal_elements()
Return the list of the minimal elements of the poset.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P(0) in P.minimal_elements()
True
sage: P(1) in P.minimal_elements()
True
sage: P(2) in P.minimal_elements()
True
```

See also:

maximal_elements().

mobius_function(**args*, ***kwds*)
Deprecated: Use moebius_function() instead. See trac ticket #19855 for details.

mobius_function_matrix(**args*, ***kwds*)
Deprecated: Use moebius_function_matrix() instead. See trac ticket #19855 for details.

moebius_function(*x*, *y*)
Returns the value of the Möbius function of the poset on the elements x and y.

EXAMPLES:

```
sage: P = Poset([[1,2,3],[4],[4],[4],[]])
sage: P.moebius_function(P(0),P(4))
2
sage: sum([P.moebius_function(P(0),v) for v in P])
0
sage: sum([abs(P.moebius_function(P(0),v)) \
....:       for v in P])
6
sage: for u,v in P.cover_relations_iterator():
....:     if P.moebius_function(u,v) != -1:
....:         print "Bug in moebius_function!"

sage: Q = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: Q.moebius_function(Q(0),Q(7))
0
sage: Q.moebius_function(Q(0),Q(5))
0
sage: Q.moebius_function(Q(2),Q(7))
2
sage: Q.moebius_function(Q(3),Q(3))
1
```

```
sage: sum([Q.moebius_function(Q(0),v) for v in Q])
0
```

moebius_function_matrix(*ring=Integer Ring*, *sparse=False*)

Returns a matrix whose `(i,j)` entry is the value of the Möbius function evaluated at `self.linear_extension()[i]` and `self.linear_extension()[j]`.

INPUT:

- `ring` – the ring of coefficients (default: `ZZ`)
- `sparse` – whether the returned matrix is sparse or not (default: `True`)

EXAMPLES:

```
sage: P = Poset([[4,2,3],[],[1],[1],[1]])
sage: x,y = (P.linear_extension()[0],P.linear_extension()[1])
sage: P.moebius_function(x,y)
-1
sage: M = P.moebius_function_matrix(); M
[ 1 -1 -1 -1  2]
[ 0  1  0  0 -1]
[ 0  0  1  0 -1]
[ 0  0  0  1 -1]
[ 0  0  0  0  1]
sage: M[0,4]
2
sage: M[0,1]
-1
```

We now demonstrate the usage of the optional parameters:

```
sage: P.moebius_function_matrix(ring=QQ, sparse=False).parent()
Full MatrixSpace of 5 by 5 dense matrices over Rational Field
```

open_interval(*x*, *y*)

Return the list of elements z such that $x < z < y$ in the poset.

EXAMPLES:

```
sage: P = Poset((divisors(1000), attrcall("divides")))
sage: P.open_interval(2, 100)
[4, 10, 20, 50]
```

See also:

`closed_interval()`

TESTS:

```
sage: C = Posets.ChainPoset(10)
sage: C.open_interval(3, 3)
[]
sage: C.open_interval(3, 4)
[]
sage: C.open_interval(7, 3)
[]
sage: A = Posets.AntichainPoset(10)
sage: A.open_interval(3, 7)
[]
```

order_complex(*on_ints=False*)

Return the order complex associated to this poset.

The order complex is the simplicial complex with vertices equal to the elements of the poset, and faces given by the chains.

INPUT:

- `on_ints` – a boolean (default: False)

EXAMPLES:

```
sage: P = Posets.BooleanLattice(3)
sage: S = P.order_complex(); S
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5, 6, 7) and 6 facets
sage: S.f_vector()
[1, 8, 19, 18, 6]
sage: S.homology()      # S is contractible
{0: 0, 1: 0, 2: 0, 3: 0}
sage: Q = P.subposet([1,2,3,4,5,6])
sage: Q.order_complex().homology()    # a circle
{0: 0, 1: Z}

sage: P = Poset((divisors(15), attrcall("divides")), facade = True)
sage: P.order_complex()
Simplicial complex with vertex set (1, 3, 5, 15) and facets {(1, 3, 15), (1, 5, 15)}
```

If `on_ints`, then the elements of the poset are labelled $0, 1, \ldots$ in the chain complex:

```
sage: P.order_complex(on_ints=True)
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 2, 3), (0, 1, 3)}
```

order_filter(*elements*)

Return the order filter generated by the elements of an iterable `elements`.

I is an order filter if, for any x in I and y such that $y \geq x$, then y is in I. This is also called upper set or upset.

EXAMPLES:

```
sage: P = Poset((divisors(1000), attrcall("divides")))
sage: P.order_filter([20, 25])
[20, 40, 25, 50, 100, 200, 125, 250, 500, 1000]
```

See also:

`order_ideal()`, `principal_order_filter()`.

TESTS:

```
sage: P = Poset()  # Test empty poset
sage: P.order_filter([])
[]
sage: C = Posets.ChainPoset(5)
sage: C.order_filter([])
[]
```

order_ideal(*elements*)

Return the order ideal generated by the elements of an iterable `elements`.

I is an order ideal if, for any x in I and y such that $y \leq x$, then y is in I. This is also called lower set or downset.

EXAMPLES:

```
sage: P = Poset((divisors(1000), attrcall("divides")))
sage: P.order_ideal([20, 25])
[1, 2, 4, 5, 10, 20, 25]
```

See also:

order_filter(),principal_order_ideal().

TESTS:

```
sage: P = Poset()  # Test empty poset
sage: P.order_ideal([])
[]
sage: C = Posets.ChainPoset(5)
sage: C.order_ideal([])
[]
```

order_polynomial()

Return the order polynomial of self.

The order polynomial $\Omega_P(q)$ of a poset P is defined as the unique polynomial S such that for each integer $m \geq 1$, $S(m)$ is the number of order-preserving maps from P to $\{1, \ldots, m\}$.

See sections 3.12 and 3.15 of [EnumComb1], and also [St1986].

See also:

order_polytope()

EXAMPLES:

```
sage: P = Posets.AntichainPoset(3)
sage: P.order_polynomial()
q^3

sage: P = Posets.ChainPoset(3)
sage: f = P.order_polynomial(); f
1/6*q^3 + 1/2*q^2 + 1/3*q
sage: [f(i) for i in range(4)]
[0, 1, 4, 10]
```

order_polytope()

Return the order polytope of the poset self.

The order polytope of a finite poset P is defined as the subset of $\mathbf{R}^P$ consisting of all maps $x : P \to \mathbf{R}$ satisfying

$$0 \leq x(p) \leq 1 \text{ for all } p \in P,$$

and

$$x(p) \leq x(q) \text{ for all } p, q \in P \text{ satisfying } p < q.$$

This polytope was defined and studied in [St1986].

EXAMPLES:

```
sage: P = posets.AntichainPoset(3)
sage: Q = P.order_polytope();Q
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 8 vertices
sage: P = posets.PentagonPoset()
sage: Q = P.order_polytope();Q
```

```
A 5-dimensional polyhedron in QQ^5 defined as the convex hull of 8 vertices

sage: P = Poset([[1,2,3],[[1,2],[1,3]]])
sage: Q = P.order_polytope()
sage: Q.contains((1,0,0))
False
sage: Q.contains((0,1,1))
True
```

REFERENCES:

ordinal_product(*other*, *labels='pairs'*)

Return the ordinal product of `self` and `other`.

The ordinal product of two posets P and Q is a partial order on the Cartesian product of the underlying sets of P and Q, defined as follows (see [EnumComb1], p. 284).

In the ordinal product, $(p, q) \leq (p', q')$ if either $p \leq p'$ or $p = p'$ and $q \leq q'$.

This construction is not symmetric in P and Q.

INPUT:

- `other` – a poset
- `labels` – either `'integers'` or `'pairs'` (default); how the resulting poset will be labeled

See also:

`product()`, `ordinal_sum()`

EXAMPLES:

```
sage: P1 = Poset((['a', 'b'], [['a', 'b']]))
sage: P2 = Poset((['c', 'd'], [['c', 'd']]))
sage: P = P1.ordinal_product(P2); P
Finite poset containing 4 elements
sage: sorted(P.cover_relations())
[[('a', 'c'), ('a', 'd')], [('a', 'd'), ('b', 'c')],
[('b', 'c'), ('b', 'd')]]
```

TESTS:

```
sage: P1.ordinal_product(24)
Traceback (most recent call last):
...
ValueError: the input is not a finite poset
sage: P1.ordinal_product(P2, labels='camembert')
Traceback (most recent call last):
...
ValueError: labels must be either 'pairs' or 'integers'
```

ordinal_sum(*other*, *labels='pairs'*)

Return a poset or (semi)lattice isomorphic to ordinal sum of the poset with `other`.

The ordinal sum of P and Q is a poset that contains every element and relation from both P and Q, and where every element of P is smaller than every element of Q.

Mathematically, it is only defined when P and Q have no common element; here we force that by giving them different names in the resulting poset.

The ordinal sum on lattices is lattice; resp. for meet- and join-semilattices. Hence we check if we can return (semi)lattice instead of plain poset.

INPUT:

- `other`, a poset.
- `labels` - (defaults to 'pairs') If set to 'pairs', each element `v` in this poset will be named `(0,v)` and each element `u` in `other` will be named `(1,u)` in the result. If set to 'integers', the elements of the result will be relabeled with consecutive integers.

EXAMPLES:

```
sage: P1 = Poset( ([1, 2, 3, 4], [[1, 2], [1, 3], [1, 4]]) )
sage: P2 = Poset( ([1, 2, 3,], [[2,1], [3,1]]) )
sage: P3 = P1.ordinal_sum(P2); P3
Finite poset containing 7 elements
sage: len(P1.maximal_elements())*len(P2.minimal_elements())
6
sage: len(P1.cover_relations()+P2.cover_relations())
5
sage: len(P3.cover_relations()) # Every element of P2 is greater than elements of P1.
11
sage: P3.list()  # random
[(0, 1), (0, 2), (0, 4), (0, 3), (1, 2), (1, 3), (1, 1)]
sage: P4 = P1.ordinal_sum(P2, labels='integers')
sage: P4.list()  # random
[0, 1, 2, 3, 5, 6, 4]
```

Return type depends on input types:

```
sage: P = Poset({1:[2]}); P
Finite poset containing 2 elements
sage: JL = JoinSemilattice({1:[2]}); JL
Finite join-semilattice containing 2 elements
sage: L = LatticePoset({1:[2]}); L
Finite lattice containing 2 elements
sage: P.ordinal_sum(L)
Finite poset containing 4 elements
sage: L.ordinal_sum(JL)
Finite join-semilattice containing 4 elements
sage: L.ordinal_sum(L)
Finite lattice containing 4 elements
```

See also:

`ordinal_summands()`, `disjoint_union()`, `ordinal_product()`

ordinal_summands()

Return the ordinal summands of the poset as subposets.

The ordinal summands of a poset P is the longest list of non-empty subposets $P_1, \ldots, P_n$ whose ordinal sum is P. This decomposition is unique.

See also:

`ordinal_sum()`

EXAMPLES:

```
sage: P = Poset({'a': ['c', 'd'], 'b': ['d'], 'c': ['x', 'y'],
....: 'd': ['x', 'y']})
sage: parts = P.ordinal_summands(); parts
[Finite poset containing 4 elements, Finite poset containing 2 elements]
sage: sorted(parts[0])
['a', 'b', 'c', 'd']
```

```
sage: Q = parts[0].ordinal_sum(parts[1])
sage: Q.is_isomorphic(P)
True
```

ALGORITHM:

Suppose that a poset P is the ordinal sum of posets L and U. Then P contains maximal antichains l and u such that every element of u covers every element of l; they correspond to maximal elements of L and minimal elements of U.

We consider a linear extension $x_1, \ldots, x_n$ of the poset's elements.

We keep track of the maximal elements of subposet induced by elements $0, \ldots, x_i$ and minimal elements of subposet induced by elements $x_{i+1}, \ldots, x_n$, incrementing i one by one. We then check if l and u fit the previous description.

TESTS:

```
sage: Poset().ordinal_summands()
[Finite poset containing 0 elements]
sage: Poset({1: []}).ordinal_summands()
[Finite poset containing 1 elements]
```

p_partition_enumerator(*tup*, *R*, *check=False*)

Return a P-partition enumerator of `self`.

Given a total order $\prec$ on the elements of a finite poset P (the order of P and the total order $\prec$ can be unrelated; in particular, the latter does not have to extend the former), a P-partition enumerator is the quasisymmetric function $\sum_f \prod_{p \in P} x_{f(p)}$, where the first sum is taken over all P-partitions f.

A P-partition is a function $f : P \to \{1, 2, 3, ...\}$ satisfying the following properties for any two elements i and j of P satisfying $i <_P j$:

- if $i \prec j$ then $f(i) \leq f(j)$,
- if $j \prec i$ then $f(i) < f(j)$.

INPUT:

- `tup` – the tuple containing all elements of P (each of them exactly once), in the order dictated by the total order $\prec$
- `R` – a commutative ring

OUTPUT:

The P-partition enumerator of `self` according to `tup` in the algebra $QSym$ of quasisymmetric functions over the base ring R.

EXAMPLES:

```
sage: P = Poset([[1,2,3,4],[[1,4],[2,4],[4,3]]])
sage: FP = P.p_partition_enumerator((3,1,2,4), QQ, check=True); FP
2*M[1, 1, 1, 1] + 2*M[1, 2, 1] + M[2, 1, 1] + M[3, 1]

sage: expansion = FP.expand(5)
sage: xs = expansion.parent().gens()
sage: expansion == sum([xs[a]*xs[b]*xs[c]*xs[d] for a in range(5) for b in range(5) for c in
True

sage: P = Poset([[],[]])
sage: FP = P.p_partition_enumerator((), QQ, check=True); FP
M[]
```

plot(*label_elements=True*, *element_labels=None*, *layout='acyclic'*, *cover_labels=None*, ***kwds*)

Return a Graphic object for the Hasse diagram of the poset.

If the poset is ranked, the plot uses the rank function for the heights of the vertices.

INPUT:

- `label_elements` (default: `True`) - whether to display element labels
- `element_labels` (default: `None`) - a dictionary of element labels
- `cover_labels` - a dictionary, list or function representing labels of the covers of `self`. When set to `None` (default) no label is displayed on the edges of the Hasse Diagram.
- `layout` – the type of layout used to display the Diagram. Set to `'acyclic'` by default (see `GenericGraph.plot` for more information).

Note: All options of `GenericGraph.plot` are also available through this function.

EXAMPLES:

```
sage: D = Poset({ 1:[2,3], 2:[4], 3:[4,5] })
sage: D.plot(label_elements=False)
Graphics object consisting of 6 graphics primitives
sage: D.plot()
Graphics object consisting of 11 graphics primitives
sage: type(D.plot())
<class 'sage.plot.graphics.Graphics'>
sage: elm_labs = {1:'a', 2:'b', 3:'c', 4:'d', 5:'e'}
sage: D.plot(element_labels=elm_labs)
Graphics object consisting of 11 graphics primitives
```

Plot of a ranked poset:

```
sage: P = Poset(DiGraph('E@ACA@?'))
sage: P.is_ranked()
True
sage: P.plot()
Graphics object consisting of 12 graphics primitives
```

The keyword `cover_labels` can be used to decorate edges:

```
sage: P = posets.ChainPoset(3)
sage: P.plot(cover_labels=lambda a, b: a + b)
Graphics object consisting of 8 graphics primitives
sage: P = Poset({0: [1,2]})
sage: P.plot(cover_labels={(0,1): 'here', (0,2): 'there'})
Graphics object consisting of 8 graphics primitives
sage: P = Poset({2: [1], 0: [1]})
sage: P.plot(cover_labels=[(2,1,'da'), (0,1,'niet')])
Graphics object consisting of 8 graphics primitives
```

TESTS:

We check that `label_elements` and `element_labels` are honored:

```
sage: def get_plot_labels(P): return sorted(t.string for t in P if isinstance(t, sage.plot.t
sage: P1 = Poset({ 0:[1,2], 1:[3], 2:[3,4] })
sage: P2 = Poset({ 0:[1,2], 1:[3], 2:[3,4] }, facade=True)
sage: get_plot_labels(P1.plot(label_elements=False))
```

```
[]
sage: get_plot_labels(P1.plot(label_elements=True))
['0', '1', '2', '3', '4']
sage: element_labels = {0:'a', 1:'b', 2:'c', 3:'d', 4:'e'}
sage: get_plot_labels(P1.plot(element_labels=element_labels))
['a', 'b', 'c', 'd', 'e']
sage: get_plot_labels(P2.plot(element_labels=element_labels))
['a', 'b', 'c', 'd', 'e']
```

The following checks that trac ticket #18936 has been fixed and labels still work:

```
sage: P = Poset({0: [1,2], 1:[3]})
sage: heights = {1 : [0], 2 : [1], 3 : [2,3]}
sage: P.plot(heights=heights)
Graphics object consisting of 8 graphics primitives
sage: elem_labels = {0 : 'a', 1 : 'b', 2 : 'c', 3 : 'd'}
sage: P.plot(element_labels=elem_labels, heights=heights)
Graphics object consisting of 8 graphics primitives
```

Plot of the empy poset:

```
sage: P = Poset({})
sage: P.plot()
Graphics object consisting of 0 graphics primitives
```

product (*other*)

Return the Cartesian product of the poset with `other`.

The Cartesian (or 'direct') product of P and Q is defined by $(p, q) \leq (p', q')$ iff $p \leq p'$ in P and $q \leq q'$ in Q.

Product of (semi)lattices are returned as a (semi)lattice.

EXAMPLES:

```
sage: P = Posets.ChainPoset(3)
sage: Q = Posets.ChainPoset(4)
sage: PQ = P.product(Q) ; PQ
Finite lattice containing 12 elements
sage: len(PQ.cover_relations())
17
sage: Q.product(P).is_isomorphic(PQ)
True

sage: P = Posets.BooleanLattice(2)
sage: Q = P.product(P)
sage: Q.is_isomorphic(Posets.BooleanLattice(4))
True
```

TESTS:

```
sage: Poset({0:[1]}).product(Poset())  # Product with empty poset
Finite poset containing 0 elements
```

We check that trac ticket #19113 is fixed:

```
sage: L = LatticePoset({1:[]})
sage: type(L) == type(L.product(L))
True
```

promotion(*i=1*)

Compute the (extended) promotion on the linear extension of the poset `self`.

INPUT:

- `i` – an integer between 1 and n (default: 1)

OUTPUT:

- an isomorphic poset, with the same default linear extension

The extended promotion is defined on a poset `self` of size n by applying the promotion operator $\tau_i\tau_{i+1}\cdots\tau_{n-1}$ to the default linear extension π of `self` (see `promotion()`), and relabeling `self` accordingly. For more details see [Stan2009].

When the vertices of the poset `self` are labelled by $\{1, 2, \ldots, n\}$, the linear extension is the identity, and $i = 1$, the above algorithm corresponds to the promotion operator on posets defined by Schützenberger as follows. Remove 1 from `self` and replace it by the minimum j of all labels covering 1 in the poset. Then, remove j and replace it by the minimum of all labels covering j, and so on. This process ends when a label is a local maximum. Place the label $n + 1$ at this vertex. Finally, decrease all labels by 1.

EXAMPLES:

```
sage: P = Poset(([1,2], [[1,2]]), linear_extension=True, facade=False)
sage: P.promotion()
Finite poset containing 2 elements with distinguished linear extension
sage: P == P.promotion()
True

sage: P = Poset(([1,2,3,4,5,6,7], [[1,2],[1,4],[2,3],[2,5],[3,6],[4,7],[5,6]]))
sage: P.list()
[1, 2, 3, 5, 6, 4, 7]
sage: Q = P.promotion(4); Q
Finite poset containing 7 elements with distinguished linear extension
sage: Q.cover_relations()
[[1, 2], [1, 6], [2, 3], [2, 5], [3, 7], [5, 7], [6, 4]]
```

Note that if one wants to obtain the promotion defined by Schützenberger's algorithm directly on the poset, one needs to make sure the linear extension is the identity:

```
sage: P = P.with_linear_extension([1,2,3,4,5,6,7])
sage: P.list()
[1, 2, 3, 4, 5, 6, 7]
sage: Q = P.promotion(4); Q
Finite poset containing 7 elements with distinguished linear extension
sage: Q.cover_relations()
[[1, 2], [1, 6], [2, 3], [2, 4], [3, 5], [4, 5], [6, 7]]
sage: Q = P.promotion()
sage: Q.cover_relations()
[[1, 2], [1, 3], [2, 4], [2, 5], [3, 6], [4, 7], [5, 7]]
```

Here is an example for a poset not labelled by $\{1, 2, \ldots, n\}$:

```
sage: P = Poset((divisors(30), attrcall("divides")), linear_extension=True)
sage: P.list()
[1, 2, 3, 5, 6, 10, 15, 30]
sage: P.cover_relations()
[[1, 2], [1, 3], [1, 5], [2, 6], [2, 10], [3, 6], [3, 15],
 [5, 10], [5, 15], [6, 30], [10, 30], [15, 30]]
sage: Q = P.promotion(4); Q
Finite poset containing 8 elements with distinguished linear extension
sage: Q.cover_relations()
```

```
[[1, 2], [1, 3], [1, 6], [2, 5], [2, 15], [3, 5], [3, 10],
 [5, 30], [6, 10], [6, 15], [10, 30], [15, 30]]
```

See also:

- linear_extension()
- with_linear_extension() and the linear_extension option of Poset()
- promotion()
- evacuation()

AUTHOR:

- Anne Schilling (2012-02-18)

random_order_ideal(*direction='down'*)

Return a random order ideal with uniform probability.

INPUT:

- direction – 'up','down' or 'antichain' (default: 'down')

OUTPUT:

A randomly selected order ideal (or order filter if direction='up', or antichain if direction='antichain') where all order ideals have equal probability of occurring.

ALGORITHM:

Uses the coupling from the past algorithm described in [Propp1997].

EXAMPLES:

```
sage: P = Posets.BooleanLattice(3)
sage: P.random_order_ideal()
[0, 1, 2, 3, 4, 5, 6]
sage: P.random_order_ideal(direction='up')
[6, 7]
sage: P.random_order_ideal(direction='antichain')
[1, 2]

sage: P = posets.TamariLattice(5)
sage: a = P.random_order_ideal('antichain')
sage: P.is_antichain_of_poset(a)
True
sage: a = P.random_order_ideal('up')
sage: P.is_order_filter(a)
True
sage: a = P.random_order_ideal('down')
sage: P.is_order_ideal(a)
True
```

REFERENCES:

random_subposet(*p*)

Return a random subposet that contains each element with probability p.

EXAMPLES:

```
sage: P = Posets.BooleanLattice(3)
sage: set_random_seed(0)
sage: Q = P.random_subposet(0.5)
sage: Q.cover_relations()
[[0, 2], [0, 5], [2, 3], [3, 7], [5, 7]]
```

rank(*element=None*)

Return the rank of an element `element` in the poset `self`, or the rank of the poset if `element` is `None`.

(The rank of a poset is the length of the longest chain of elements of the poset.)

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]], facade = False)
sage: P.rank(5)
2
sage: P.rank()
3
sage: Q = Poset([[1,2],[3],[],[]])

sage: P = Posets.SymmetricGroupBruhatOrderPoset(4)
sage: [(v,P.rank(v)) for v in P]
[('1234', 0),
 ('1243', 1),
...
 ('4312', 5),
 ('4321', 6)]
```

rank_function()

Return the (normalized) rank function of the poset, if it exists.

A *rank function* of a poset P is a function r that maps elements of P to integers and satisfies: $r(x) = r(y) + 1$ if x covers y. The function r is normalized such that its minimum value on every connected component of the Hasse diagram of P is 0. This determines the function r uniquely (when it exists).

OUTPUT:

- a lambda function, if the poset admits a rank function
- `None`, if the poset does not admit a rank function

EXAMPLES:

```
sage: P = Poset(([1,2,3,4],[[1,4],[2,3],[3,4]]), facade=True)
sage: P.rank_function() is not None
True
sage: P = Poset(([1,2,3,4,5],[[1,2],[2,3],[3,4],[1,5],[5,4]]), facade=True)
sage: P.rank_function() is not None
False
sage: P = Poset(([1,2,3,4,5,6,7,8],[[1,4],[2,3],[3,4],[5,7],[6,7]]), facade=True)
sage: f = P.rank_function(); f is not None
True
sage: f(5)
0
sage: f(2)
0
```

TESTS:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: r = P.rank_function()
```

```
sage: for u,v in P.cover_relations_iterator():
....:     if r(v) != r(u) + 1:
....:         print "Bug in rank_function!"

sage: Q = Poset([[1,2],[4],[3],[4],[]])
sage: Q.rank_function() is None
True
```

relabel(*relabeling*)

Return a copy of this poset with its elements relabelled.

INPUT:

- `relabeling` – a function or dictionary

This function should map each (non-wrapped) element of `self` to some distinct object.

EXAMPLES:

```
sage: P = Poset((divisors(12), attrcall("divides")), linear_extension=True, facade = False)
sage: P.list()
[1, 2, 3, 4, 6, 12]
sage: P.cover_relations()
[[1, 2], [1, 3], [2, 4], [2, 6], [3, 6], [4, 12], [6, 12]]
sage: Q = P.relabel(lambda x: 12/x)
sage: Q.list()
[12, 6, 4, 3, 2, 1]
sage: Q.cover_relations()
[[12, 6], [12, 4], [6, 3], [6, 2], [4, 2], [3, 1], [2, 1]]
```

Here we relabel the elements of a poset by $\{0, 1, 2, ...\}$, using a dictionary:

```
sage: P = Poset((divisors(12), attrcall("divides")), linear_extension=True, facade=False)
sage: relabeling = {c.element:i for (i,c) in enumerate(P)}; relabeling
{1: 0, 2: 1, 3: 2, 4: 3, 6: 4, 12: 5}
sage: Q = P.relabel(relabeling)
sage: Q.list()
[0, 1, 2, 3, 4, 5]
sage: Q.cover_relations()
[[0, 1], [0, 2], [1, 3], [1, 4], [2, 4], [3, 5], [4, 5]]
```

Mind the `c.element`; this is because the relabeling is applied to the elements of the poset without the wrapping. Thanks to this convention, the same relabeling function can be used both for facade or non facade posets:

```
sage: P = Poset((divisors(12), attrcall("divides")), facade = True, linear_extension=True)
sage: P.list()
[1, 2, 3, 4, 6, 12]
sage: Q = P.relabel(lambda x: 12/x)
sage: Q.list()
[12, 6, 4, 3, 2, 1]
sage: Q.cover_relations()
[[12, 6], [12, 4], [6, 3], [6, 2], [4, 2], [3, 1], [2, 1]]
```

Relabeling a (semi)lattice gives a (semi)lattice:

sage: P=JoinSemilattice({0:[1]}) sage: type(P.relabel(lambda n: n+1)) <class 'sage.combinat.posets.lattices.FiniteJoinSemilattice_with_category'>

Note: As can be seen in the above examples, the default linear extension of `Q` is that of `P` after relabeling.

In particular, P and Q share the same internal Hasse diagram.

TESTS:

The following checks that trac ticket #14019 has been fixed:

```
sage: d = DiGraph({2:[1],3:[1]})
sage: p1 = Poset(d)
sage: p2 = p1.relabel({1:1,2:3,3:2})
sage: p1.hasse_diagram() == p2.hasse_diagram()
True
sage: p1 == p2
True

sage: d = DiGraph({2:[1],3:[1]})
sage: p1 = Poset(d)
sage: p2 = p1.relabel({1:2,2:3,3:1})
sage: p3 = p2.relabel({2:1,1:2,3:3})
sage: p1.hasse_diagram() == p3.hasse_diagram()
True
sage: p1 == p3
True
```

relations()

Return the list of all relations of the poset.

A relation is a pair of elements x and y such that $x \leq y$ in the poset.

The number of relations is the dimension of the incidence algebra.

OUTPUT:

A list of pairs (each pair is a list), where the first element of the pair is less than or equal to the second element.

See also:

relations_number(), relations_iterator()

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: P.relations()
[[1, 1], [1, 2], [1, 3], [1, 4], [0, 0], [0, 2], [0, 3],
[0, 4], [2, 2], [2, 3], [2, 4], [3, 3], [3, 4], [4, 4]]
```

TESTS:

```
sage: P = Poset()  # Test empty poset
sage: P.relations()
[]
```

AUTHOR:

- Rob Beezer (2011-05-04)

relations_iterator(*strict=False*)

Return an iterator for all the relations of the poset.

A relation is a pair of elements x and y such that $x \leq y$ in the poset.

INPUT:

- `strict` – a boolean (default `False`) if `True`, returns an iterator over relations $x < y$, excluding all relations $x \leq x$.

OUTPUT:

A generator that produces pairs (each pair is a list), where the first element of the pair is less than or equal to the second element.

See also:

`relations_number()`, `relations()`.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: it = P.relations_iterator()
sage: type(it)
<type 'generator'>
sage: it.next(), it.next()
([1, 1], [1, 2])

sage: P = posets.PentagonPoset()
sage: list(P.relations_iterator(strict=True))
[[0, 1], [0, 2], [0, 4], [0, 3], [1, 4], [2, 3], [2, 4], [3, 4]]
```

AUTHOR:

- Rob Beezer (2011-05-04)

relations_number()

Return the number of relations in the poset.

A relation is a pair of elements x and y such that $x \leq y$ in the poset.

Relations are also often called intervals. The number of intervals is the dimension of the incidence algebra.

See also:

`relations_iterator()`, `relations()`

EXAMPLES:

```
sage: P = Posets.PentagonPoset()
sage: P.relations_number()
13

sage: from sage.combinat.tamari_lattices import TamariLattice
sage: TamariLattice(4).relations_number()
68
```

TESTS:

```
sage: Poset().relations_number()
0
```

show(*label_elements=True*, *element_labels=None*, *cover_labels=None*, ***kwds*)

Displays the Hasse diagram of the poset.

INPUT:

- `label_elements` (default: `True`) - whether to display element labels
- `element_labels` (default: `None`) - a dictionary of element labels

- `cover_labels` - a dictionary, list or function representing labels of the covers of `self`. When set to `None` (default) no label is displayed on the edges of the Hasse Diagram.

Note: This method also accepts:

- All options of `GenericGraph.plot`
- All options of `Graphics.show`

EXAMPLES:

```
sage: D = Poset({ 0:[1,2], 1:[3], 2:[3,4] })
sage: D.plot(label_elements=False)
Graphics object consisting of 6 graphics primitives
sage: D.show()
sage: elm_labs = {0:'a', 1:'b', 2:'c', 3:'d', 4:'e'}
sage: D.show(element_labels=elm_labs)
```

One more example with cover labels:

```
sage: P = posets.PentagonPoset()
sage: P.show(cover_labels=lambda a, b: a - b)
```

subposet(*elements*)

Return the poset containing elements with partial order induced by that of `self`.

EXAMPLES:

```
sage: P = Poset({"a":["c","d"], "b":["d","e"], "c":["f"], "d":["f"], "e":["f"]}, facade = Fa
sage: Q = P.subposet(["a","b","f"]); Q
Finite poset containing 3 elements
sage: Q.cover_relations()
[[b, f], [a, f]]
```

A subposet of a facade poset is again a facade poset:

```
sage: P = Poset({"a":["c","d"], "b":["d","e"], "c":["f"], "d":["f"], "e":["f"]}, facade=True
sage: Q = P.subposet(["a","b","f"]); Q
Finite poset containing 3 elements
sage: Q.cover_relations()
[['b', 'f'], ['a', 'f']]
```

One may specified wrapped elements or not:

```
sage: P = Poset({"a":["c","d"], "b":["d","e"], "c":["f"], "d":["f"], "e":["f"]}, facade = Fa
sage: Q = P.subposet([P("a"),P("b"),P("f")]); Q
Finite poset containing 3 elements
sage: Q.cover_relations()
[[b, f], [a, f]]

sage: B = posets.BooleanLattice(2)
sage: above = B.principal_order_filter(0)
sage: Q = B.subposet(above)
sage: above_new = Q.principal_order_filter(Q.list()[0])
sage: Q.subposet(above_new)
Finite poset containing 4 elements
```

TESTS:

```
sage: P.subposet(("a","b","f"))
Finite poset containing 3 elements
```

```
sage: P.subposet(["a","b","x"])
Traceback (most recent call last):
...
ValueError: <type 'str'> is not an element of this poset
sage: P.subposet(3)
Traceback (most recent call last):
...
TypeError: 'sage.rings.integer.Integer' object is not iterable
```

to_graph(**args*, ***kwds*)

Deprecated: Use `cover_relations_graph()` instead. See trac ticket #17449 for details.

top()

Return the unique maximal element of the poset, if it exists.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4,5],4:[],5:[]})
sage: P.top() is None
True
sage: Q = Poset({0:[1],1:[]})
sage: Q.top()
1
```

See also:

`has_top()`, `bottom()`

TESTS:

```
sage: R = Poset([[0],[]])
sage: R.list()
[0]
sage: R.top() #Trac #10776
0
```

unwrap(*element*)

Return the element `element` of the poset `self` in unwrapped form.

INPUT:

- `element` – an element of `self`

EXAMPLES:

```
sage: P = Poset((divisors(15), attrcall("divides")), facade = False)
sage: x = P.an_element(); x
1
sage: x.parent()
Finite poset containing 4 elements
sage: P.unwrap(x)
1
sage: P.unwrap(x).parent()
Integer Ring
```

For a non facade poset, this is equivalent to using the `.element` attribute:

```
sage: P.unwrap(x) is x.element
True
```

For a facade poset, this does nothing:

```
sage: P = Poset((divisors(15), attrcall("divides")), facade=True)
sage: x = P.an_element()
sage: P.unwrap(x) is x
True
```

This method is useful in code where we don't know if `P` is a facade poset or not.

upper_covers(*x*)
Return the list of upper covers of the element `x`.

An upper cover of x is an element y such that $x < y$ and there is no element z so that $x < z < y$.

EXAMPLES:

```
sage: P = Poset([[1,5], [2,6], [3], [4], [], [6,3], [4]])
sage: P.upper_covers(1)
[2, 6]
```

See also:

`lower_covers()`

upper_covers_iterator(*x*)
Return an iterator over the upper covers of the element `x`.

EXAMPLES:

```
sage: P = Poset({0:[2], 1:[2], 2:[3], 3:[]})
sage: type(P.upper_covers_iterator(0))
<type 'generator'>
```

width()
Return the width of the poset (the size of its longest antichain).

It is computed through a matching in a bipartite graph; see Wikipedia article Dilworth's_theorem for more information. The width is also called Dilworth number.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: P.width()
6
```

TESTS:

```
sage: Poset().width()
0
```

with_bounds(*labels=('bottom', 'top')*)
Return the poset with bottom and top elements adjoined.

This functions always adds two new elements to the poset, i.e. it does not check if the poset already has a bottom or a top element.

For lattices and semilattices this function returns a lattice.

INPUT:

- `labels` – A pair of elements to use as a bottom and top element of the poset. Default is strings `'bottom'` and `'top'`.

EXAMPLES:

```
sage: V = Poset({0: [1, 2]})
sage: trafficsign = V.with_bounds(); trafficsign
Finite poset containing 5 elements
sage: trafficsign.list()
['bottom', 0, 1, 2, 'top']
sage: trafficsign = V.with_bounds(labels=(-1, -2))
sage: trafficsign.cover_relations()
[[-1, 0], [0, 1], [0, 2], [1, -2], [2, -2]]

sage: P = Posets.PentagonPoset()  # A lattice
sage: P.with_bounds()
Finite lattice containing 7 elements
```

TESTS:

```
sage: P = Poset().with_bounds()
sage: P.cover_relations()
[['bottom', 'top']]

sage: LatticePoset({}).with_bounds()
Finite lattice containing 2 elements

sage: Posets.PentagonPoset().with_bounds(labels=(4, 5))
Traceback (most recent call last):
...
ValueError: the poset already has element 4
```

with_linear_extension(*linear_extension*)

Return a copy of `self` with a different default linear extension.

EXAMPLES:

```
sage: P = Poset((divisors(12), attrcall("divides")), linear_extension=True)
sage: P.cover_relations()
[[1, 2], [1, 3], [2, 4], [2, 6], [3, 6], [4, 12], [6, 12]]
sage: list(P)
[1, 2, 3, 4, 6, 12]
sage: Q = P.with_linear_extension([1,3,2,6,4,12])
sage: list(Q)
[1, 3, 2, 6, 4, 12]
sage: Q.cover_relations()
[[1, 3], [1, 2], [3, 6], [2, 6], [2, 4], [6, 12], [4, 12]]
```

TESTS:

We check that we can pass in a list of elements of `P` instead:

```
sage: Q = P.with_linear_extension([P(_) for _ in [1,3,2,6,4,12]])
sage: list(Q)
[1, 3, 2, 6, 4, 12]
sage: Q.cover_relations()
[[1, 3], [1, 2], [3, 6], [2, 6], [2, 4], [6, 12], [4, 12]]
```

We check that this works for facade posets too:

```
sage: P = Poset((divisors(12), attrcall("divides")), facade=True)
sage: Q = P.with_linear_extension([1,3,2,6,4,12])
sage: list(Q)
[1, 3, 2, 6, 4, 12]
sage: Q.cover_relations()
```

```
[[1, 3], [1, 2], [3, 6], [2, 6], [2, 4], [6, 12], [4, 12]]
sage: sorted(Q.cover_relations()) == sorted(P.cover_relations())
True
```

Note: With the current implementation, this requires relabeling the internal `DiGraph` which is $O(n+m)$, where n is the number of elements and m the number of cover relations.

---

zeta_polynomial()

Return the zeta polynomial of the poset `self`.

The zeta polynomial of a poset is the unique polynomial $Z(q)$ such that for every integer $m > 1$, $Z(m)$ is the number of weakly increasing sequences $x_1 \leq x_2 \leq \cdots \leq x_{m-1}$ of elements of the poset.

The polynomial $Z(q)$ is integral-valued, but generally doesn't have integer coefficients. It can be computed as

$$Z(q) = \sum_{k \geq 1} \binom{q-2}{k-1} c_k,$$

where c_k is the number of all chains of length k in the poset.

For more information, see section 3.12 of [EnumComb1].

In particular, $Z(2)$ is the number of vertices and $Z(3)$ is the number of intervals.

EXAMPLES:

```
sage: Posets.ChainPoset(2).zeta_polynomial()
q
sage: Posets.ChainPoset(3).zeta_polynomial()
1/2*q^2 + 1/2*q
sage: P = posets.PentagonPoset()
sage: P.zeta_polynomial()
1/6*q^3 + q^2 - 1/6*q
sage: P = Posets.DiamondPoset(5)
sage: P.zeta_polynomial()
3/2*q^2 - 1/2*q
```

TESTS:

Checking the simplest cases:

```
sage: Poset({}).zeta_polynomial()
0
sage: Poset({1: []}).zeta_polynomial()
1
sage: Poset({1: [], 2: []}).zeta_polynomial()
2
```

class sage.combinat.posets.posets.**FinitePosets_n**(n)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The finite enumerated set of all posets on n vertices, up to an isomorphism.

EXAMPLES:

```
sage: P = Posets(3)
sage: P.cardinality()
5
sage: for p in P: print p.cover_relations()
```

```
[]
[[1, 2]]
[[0, 1], [0, 2]]
[[0, 1], [1, 2]]
[[1, 2], [0, 2]]
```

cardinality(*from_iterator=False*)

Return the cardinality of this object.

Note: By default, this returns pre-computed values obtained from the On-Line Encyclopedia of Integer Sequences (OEIS sequence A000112). To override this, pass the argument `from_iterator=True`.

EXAMPLES:

```
sage: P = Posets(3)
sage: P.cardinality()
5
sage: P.cardinality(from_iterator=True)
5
```

sage.combinat.posets.posets.**Poset**(*data=None*, *element_labels=None*, *cover_relations=False*, *linear_extension=False*, *category=None*, *facade=None*, *key=None*)

Construct a finite poset from various forms of input data.

INPUT:

- `data` – different input are accepted by this constructor:

  1. A two-element list or tuple `(E, R)`, where `E` is a collection of elements of the poset and `R` is a collection of relations `x <= y`, each represented as a two-element list/tuple/iterable such as `[x, y]`. The poset is then the transitive closure of the provided relations. If `cover_relations=True`, then `R` is assumed to contain exactly the cover relations of the poset. If `E` is empty, then `E` is taken to be the set of elements appearing in the relations `R`.

  2. A two-element list or tuple `(E, f)`, where `E` is the set of elements of the poset and `f` is a function such that, for any pair `x, y` of elements of `E`, `f(x, y)` returns whether `x <= y`. If `cover_relations=True`, then `f(x, y)` should instead return whether `x` is covered by `y`.

  3. A dictionary, list or tuple of upper covers: `data[x]` is a list of the elements that cover the element x in the poset.

     Warning: If data is a list or tuple of length 2, then it is handled by the above case..

  4. An acyclic, loop-free and multi-edge free `DiGraph`. If `cover_relations` is `True`, then the edges of the digraph are assumed to correspond to the cover relations of the poset. Otherwise, the cover relations are computed.

  5. A previously constructed poset (the poset itself is returned).

- `element_labels` – (default: `None`); an optional list or dictionary of objects that label the poset elements.

- `cover_relations` – a boolean (default: `False`); whether the data can be assumed to describe a directed acyclic graph whose arrows are cover relations; otherwise, the cover relations are first computed.

- `linear_extension` – a boolean (default: `False`); whether to use the provided list of elements as default linear extension for the poset; otherwise a linear extension is computed. If the data is given as the pair `(E, f)`, then `E` is taken to be the linear extension.

- facade – a boolean or None (default); whether the Poset() 's elements should be wrapped to make them aware of the Poset they belong to.
  - If facade = True, the Poset() 's elements are exactly those given as input.
  - If facade = False, the Poset() 's elements will become PosetElement objects.
  - If facade = None (default) the expected behaviour is the behaviour of facade = True, unless the opposite can be deduced from the context (i.e. for instance if a Poset() is built from another Poset(), itself built with facade = False)

OUTPUT:

FinitePoset – an instance of the FinitePoset class.

If category is specified, then the poset is created in this category instead of FinitePosets.

See also:

Posets, Posets, FinitePosets

EXAMPLES:

1. Elements and cover relations:

```
sage: elms = [1,2,3,4,5,6,7]
sage: rels = [[1,2],[3,4],[4,5],[2,5]]
sage: Poset((elms, rels), cover_relations = True, facade = False)
Finite poset containing 7 elements
```

Elements and non-cover relations:

```
sage: elms = [1,2,3,4]
sage: rels = [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
sage: P = Poset( [elms,rels] ,cover_relations=False); P
Finite poset containing 4 elements
sage: P.cover_relations()
[[1, 2], [2, 3], [3, 4]]
```

2. Elements and function: the standard permutations of [1, 2, 3, 4] with the Bruhat order:

```
sage: elms = Permutations(4)
sage: fcn = lambda p,q : p.bruhat_lequal(q)
sage: Poset((elms, fcn))
Finite poset containing 24 elements
```

With a function that identifies the cover relations: the set partitions of $\{1, 2, 3\}$ ordered by refinement:

```
sage: elms = SetPartitions(3)
sage: def fcn(A, B):
....:     if len(A) != len(B)+1:
....:         return False
....:     for a in A:
....:         if not any(set(a).issubset(b) for b in B):
....:             return False
....:     return True
sage: Poset((elms, fcn), cover_relations=True)
Finite poset containing 5 elements
```

3.A dictionary of upper covers:

```
sage: Poset({'a':['b','c'], 'b':['d'], 'c':['d'], 'd':[]})
Finite poset containing 4 elements
```

A list of upper covers:

```
sage: Poset([[1,2],[4],[3],[4],[]])
Finite poset containing 5 elements
```

A list of upper covers and a dictionary of labels:

```
sage: elm_labs = {0:"a",1:"b",2:"c",3:"d",4:"e"}
sage: P = Poset([[1,2],[4],[3],[4],[]], elm_labs, facade = False)
sage: P.list()
[a, b, c, d, e]
```

Warning: The special case where the argument data is a list or tuple of length 2 is handled by the above cases. So you cannot use this method to input a 2-element poset.

4.An acyclic DiGraph.

```
sage: dag = DiGraph({0:[2,3], 1:[3,4], 2:[5], 3:[5], 4:[5]})
sage: Poset(dag)
Finite poset containing 6 elements
```

Any directed acyclic graph without loops or multiple edges, as long as `cover_relations=False`:

```
sage: dig = DiGraph({0:[2,3], 1:[3,4,5], 2:[5], 3:[5], 4:[5]})
sage: dig.allows_multiple_edges()
False
sage: dig.allows_loops()
False
sage: dig.transitive_reduction() == dig
False
sage: Poset(dig, cover_relations=False)
Finite poset containing 6 elements
sage: Poset(dig, cover_relations=True)
Traceback (most recent call last):
...
ValueError: Hasse diagram is not transitively reduced.
```

Default Linear extension

Every poset P obtained with `Poset` comes equipped with a default linear extension, which is also used for enumerating its elements. By default, this linear extension is computed, and has no particular significance:

```
sage: P = Poset((divisors(12), attrcall("divides")))
sage: P.list()
[1, 2, 4, 3, 6, 12]
sage: P.linear_extension()
[1, 2, 4, 3, 6, 12]
```

You may enforce a specific linear extension using the `linear_extension` option:

```
sage: P = Poset((divisors(12), attrcall("divides")), linear_extension=True)
sage: P.list()
[1, 2, 3, 4, 6, 12]
sage: P.linear_extension()
[1, 2, 3, 4, 6, 12]
```

Depending on popular request, `Poset` might eventually get modified to always use the provided list of elements as default linear extension, when it is one.

See also:

`FinitePoset.linear_extensions()`

Facade posets

When `facade = False`, the elements of a poset are wrapped so as to make them aware that they belong to that poset:

```
sage: P = Poset(DiGraph({'d':['c','b'],'c':['a'],'b':['a']}), facade = False)
sage: d,c,b,a = list(P)
sage: a.parent() is P
True
```

This allows for comparing elements according to P:

```
sage: c < a
True
```

However, this may have surprising effects:

```
sage: my_elements = ['a','b','c','d']
sage: any(x in my_elements for x in P)
False
```

and can be annoying when one wants to manipulate the elements of the poset:

```
sage: a + b
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'FinitePoset_with_category.element_class' and 'Fin
sage: a.element + b.element
'ab'
```

By default, facade posets are constructed instead:

```
sage: P = Poset(DiGraph({'d':['c','b'],'c':['a'],'b':['a']}))
```

In this example, the elements of the poset remain plain strings:

```
sage: d,c,b,a = list(P)
sage: type(a)
<type 'str'>
```

Of course, those strings are not aware of P. So to compare two such strings, one needs to query P:

```
sage: a < b
True
sage: P.lt(a,b)
False
```

which models the usual mathematical notation $a <_P b$.

Most operations seem to still work, but at this point there is no guarantee whatsoever:

```
sage: P.list()
['d', 'c', 'b', 'a']
sage: P.principal_order_ideal('a')
['d', 'c', 'b', 'a']
sage: P.principal_order_ideal('b')
['d', 'b']
sage: P.principal_order_ideal('d')
['d']
sage: TestSuite(P).run()
```

Warning: `DiGraph` is used to construct the poset, and the vertices of a `DiGraph` are converted to plain Python `int`'s if they are `Integer`'s:

```
sage: G = DiGraph({0:[2,3], 1:[3,4], 2:[5], 3:[5], 4:[5]})
sage: type(G.vertices()[0])
<type 'int'>
```

This is worked around by systematically converting back the vertices of a poset to `Integer`'s if they are `int`'s:

```
sage: P = Poset((divisors(15), attrcall("divides")), facade = False)
sage: type(P.an_element().element)
<type 'sage.rings.integer.Integer'>

sage: P = Poset((divisors(15), attrcall("divides")), facade=True)
sage: type(P.an_element())
<type 'sage.rings.integer.Integer'>
```

This may be abusive:

```
sage: P = Poset((range(5), operator.le), facade = True)
sage: P.an_element().parent()
Integer Ring
```

Unique representation

As most parents, `Poset` have unique representation (see `UniqueRepresentation`). Namely if two posets are created from two equal data, then they are not only equal but actually identical:

```
sage: data1 = [[1,2],[3],[3]]
sage: data2 = [[1,2],[3],[3]]
sage: P1 = Poset(data1)
sage: P2 = Poset(data2)
sage: P1 == P2
True
```

```
sage: P1 is P2
True
```

In situations where this behaviour is not desired, one can use the `key` option:

```
sage: P1 = Poset(data1, key = "foo")
sage: P2 = Poset(data2, key = "bar")
sage: P1 is P2
False
sage: P1 == P2
False
```

`key` can be any hashable value and is passed down to `UniqueRepresentation`. It is otherwise ignored by the poset constructor.

TESTS:

```
sage: P = Poset([[1,2],[3],[3]])
sage: type(hash(P))
<type 'int'>
```

Bad input:

```
sage: Poset([1,2,3], lambda x,y : x<y)
Traceback (most recent call last):
...
ValueError: element_labels should be a dict or a list if different
from None. (Did you intend data to be equal to a pair ?)
```

Another kind of bad input, digraphs with oriented cycles:

```
sage: Poset(DiGraph([[1,2],[2,3],[3,4],[4,1]]))
Traceback (most recent call last):
...
ValueError: The graph is not directed acyclic
```

sage.combinat.posets.posets.**is_poset**(*dig*)

Return `True` if a directed graph is acyclic and transitively reduced, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.posets import is_poset
sage: dig = DiGraph({0:[2, 3], 1:[3, 4, 5], 2:[5], 3:[5], 4:[5]})
sage: is_poset(dig)
False
sage: is_poset(dig.transitive_reduction())
True
```

5.1.160 q-Analogues

sage.combinat.q_analogues.**gaussian_binomial**(*n*, *k*, *q=None*, *algorithm='auto'*)

This is an alias of `q_binomial()`.

See `q_binomial()` for the full documentation.

EXAMPLES:

```
sage: gaussian_binomial(4,2)
q^4 + q^3 + 2*q^2 + q + 1
```

sage.combinat.q_analogues.**gaussian_multinomial**(*seq*, *q=None*, *binomial_algorithm='auto'*)

Return the q-multinomial coefficient.

This is also known as the Gaussian multinomial coefficient, and is defined by

$$\binom{n}{k_1, k_2, \ldots, k_m}_q = \frac{[n]_q!}{[k_1]_q![k_2]_q! \cdots [k_m]_q!}$$

where $n = k_1 + k_2 + \cdots + k_m$.

If q is unspecified, then the variable is the generator q for a univariate polynomial ring over the integers.

INPUT:

- `seq` – an iterable of the values k_1 to k_m defined above
- `q` – (default: `None`) the variable q; if `None`, then use a default variable in $\mathbf{Z}[q]$
- `binomial_algorithm` – (default: `'auto'`) the algorithm to use in `q_binomial()`; see possible values there

ALGORITHM:

We use the equivalent formula

$$\binom{k_1 + \cdots + k_m}{k_1, \ldots, k_m}_q = \prod_{i=1}^{m} \binom{\sum_{j=1}^{i} k_j}{k_i}_q.$$

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_multinomial
sage: q_multinomial([1,2,1])
q^5 + 2*q^4 + 3*q^3 + 3*q^2 + 2*q + 1
sage: q_multinomial([1,2,1], q=1) == multinomial([1,2,1])
True
sage: q_multinomial((3,2)) == q_binomial(5,3)
True
sage: q_multinomial([])
1
```

sage.combinat.q_analogues.**q_binomial**(*n*, *k*, *q=None*, *algorithm='auto'*)

Return the q-binomial coefficient.

This is also known as the Gaussian binomial coefficient, and is defined by

$$\binom{n}{k}_q = \frac{(1-q^n)(1-q^{n-1}) \cdots (1-q^{n-k+1})}{(1-q)(1-q^2) \cdots (1-q^k)}.$$

See Wikipedia article Gaussian_binomial_coefficient.

If q is unspecified, then the variable is the generator q for a univariate polynomial ring over the integers.

INPUT:

- `n, k` – the values n and k defined above
- `q` – (default: `None`) the variable q; if `None`, then use a default variable in $\mathbf{Z}[q]$
- `algorithm` – (default: `'auto'`) the algorithm to use and can be one of the following:
  - `'auto'` – automatically choose the algorithm; see the algorithm section below
  - `'naive'` – use the naive algorithm

– 'cyclotomic' – use cyclotomic algorithm

ALGORITHM:

The naive algorithm uses the product formula. The cyclotomic algorithm uses a product of cyclotomic polynomials (cf. [CH2006]).

When the algorithm is set to 'auto', we choose according to the following rules:

- If q is a polynomial:

  When n is small or k is small with respect to n, one uses the naive algorithm. When both n and k are big, one uses the cyclotomic algorithm.

- If q is in the symbolic ring, one uses the cyclotomic algorithm.

- Otherwise one uses the naive algorithm, unless q is a root of unity, then one uses the cyclotomic algorithm.

EXAMPLES:

By default, the variable is the generator of $\mathbf{Z}[q]$:

```
sage: from sage.combinat.q_analogues import q_binomial
sage: g = q_binomial(5,1) ; g
q^4 + q^3 + q^2 + q + 1
sage: g.parent()
Univariate Polynomial Ring in q over Integer Ring
```

The q-binomial coefficient vanishes unless $0 \leq k \leq n$:

```
sage: q_binomial(4,5)
0
sage: q_binomial(5,-1)
0
```

Other variables can be used, given as third parameter:

```
sage: p = ZZ['p'].gen()
sage: q_binomial(4,2,p)
p^4 + p^3 + 2*p^2 + p + 1
```

The third parameter can also be arbitrary values:

```
sage: q_binomial(5,1,2) == g.subs(q=2)
True
sage: q_binomial(5,1,1)
5
sage: q_binomial(4,2,-1)
2
sage: q_binomial(4,2,3.14)
152.030056160000
sage: R = GF(25, 't')
sage: t = R.gen(0)
sage: q_binomial(6, 3, t)
2*t + 3
```

We can also do this for more complicated objects such as matrices or symmetric functions:

```
sage: q_binomial(4,2,matrix([[2,1],[-1,3]]))
[ -6  84]
[-84  78]
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: q_binomial(4,1, s[2]+s[1])
```

```
s[] + s[1] + s[1, 1] + s[1, 1, 1] + 2*s[2] + 4*s[2, 1] + 3*s[2, 1, 1]
 + 4*s[2, 2] + 3*s[2, 2, 1] + s[2, 2, 2] + 3*s[3] + 7*s[3, 1] + 3*s[3, 1, 1]
 + 6*s[3, 2] + 2*s[3, 2, 1] + s[3, 3] + 4*s[4] + 6*s[4, 1] + s[4, 1, 1]
 + 3*s[4, 2] + 3*s[5] + 2*s[5, 1] + s[6]
```

TESTS:

One checks that the first two arguments are integers:

```
sage: q_binomial(1/2,1)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

One checks that n is nonnegative:

```
sage: q_binomial(-4,1)
Traceback (most recent call last):
...
ValueError: n must be nonnegative
```

This also works for variables in the symbolic ring:

```
sage: z = var('z')
sage: factor(q_binomial(4,2,z))
(z^2 + z + 1)*(z^2 + 1)
```

This also works for complex roots of unity:

```
sage: q_binomial(6, 1, QQbar(I))
I + 1
```

Note that the symbolic computation works (see trac ticket #14982):

```
sage: q_binomial(6, 1, I)
I + 1
```

Check that the algorithm does not matter:

```
sage: q_binomial(6,3, algorithm='naive') == q_binomial(6,3, algorithm='cyclotomic')
True
```

One more test:

```
sage: q_binomial(4, 2, Zmod(6)(2), algorithm='naive')
5
```

REFERENCES:

AUTHORS:

- Frederic Chapoton, David Joyner and William Stein

sage.combinat.q_analogues.**q_catalan_number**(*n*, *q=None*)

Returns the q-Catalan number of index n.

If q is unspecified, then it defaults to using the generator q for a univariate polynomial ring over the integers.

There are several q-Catalan numbers. This procedure returns the one which can be written using the q-binomial coefficients.

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_catalan_number
sage: q_catalan_number(4)
q^12 + q^10 + q^9 + 2*q^8 + q^7 + 2*q^6 + q^5 + 2*q^4 + q^3 + q^2 + 1
sage: p = ZZ['p'].0
sage: q_catalan_number(4,p)
p^12 + p^10 + p^9 + 2*p^8 + p^7 + 2*p^6 + p^5 + 2*p^4 + p^3 + p^2 + 1
```

The q-Catalan number of index n is only defined for n a nonnegative integer (trac ticket #11411):

```
sage: q_catalan_number(-2)
Traceback (most recent call last):
...
ValueError: Argument (-2) must be a nonnegative integer.
```

sage.combinat.q_analogues.**q_factorial**(*n*, *q=None*)

Returns the q-analogue of the factorial $n!$.

If q is unspecified, then it defaults to using the generator q for a univariate polynomial ring over the integers.

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_factorial
sage: q_factorial(3)
q^3 + 2*q^2 + 2*q + 1
sage: p = ZZ['p'].0
sage: q_factorial(3, p)
p^3 + 2*p^2 + 2*p + 1
```

The q-analogue of $n!$ is only defined for n a non-negative integer (trac ticket #11411):

```
sage: q_factorial(-2)
Traceback (most recent call last):
...
ValueError: Argument (-2) must be a nonnegative integer.
```

sage.combinat.q_analogues.**q_int**(*n*, *q=None*)

Return the q-analogue of the integer n.

The q-analogue of the integer n is given by

$$[n]_q = \begin{cases} 1 + q + \cdots + q^{n-1}, & \text{if } n \geq 0, \\ -q^{-n}[-n]_q, & \text{if } n \leq 0. \end{cases}$$

Consequently, if $q = 1$ then $[n]_1 = n$ and if $q \neq 1$ then $[n]_q = (q^n - 1)/(q - 1)$.

If the argument q is not specified then it defaults to the generator q of the univariate polynomial ring over the integers.

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_int
sage: q_int(3)
q^2 + q + 1
sage: q_int(-3)
(-q^2 - q - 1)/q^3
sage: p = ZZ['p'].0
sage: q_int(3,p)
p^2 + p + 1
sage: q_int(3/2)
Traceback (most recent call last):
```

```
...
ValueError: 3/2 must be an integer
```

TESTS:

We check that trac ticket #15805 is fixed:

```
sage: from sage.combinat.q_analogues import q_int
sage: q_int(0).parent()
Univariate Polynomial Ring in q over Integer Ring
```

sage.combinat.q_analogues.**q_jordan**(*t*, *q*)

INPUT:

- t – a partition of an integer
- q – an integer or an indeterminate

OUTPUT:

If q is the power of a prime number, the output is the number of complete flags in F_q^N (where N is the size of t) stable under a linear nilpotent endomorphism f whose Jordan type is given by t, i.e. such that for all i:

$$\dim(\ker f^i) = t[0] + \cdots + t[i-1]$$

If q is an indeterminate, the output is a polynomial whose values at powers of prime numbers are the previous numbers.

The result is cached.

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_jordan
sage: [q_jordan(mu,2) for mu in Partitions(5)]
[9765, 1029, 213, 93, 29, 9, 1]
sage: [q_jordan(mu,2) for mu in Partitions(6)]
[615195, 40635, 5643, 2331, 1491, 515, 147, 87, 47, 11, 1]

sage: q=PolynomialRing(ZZ,'q').gen()
sage: q_jordan(Partition([3,2,1]),q)
16*q^4 + 24*q^3 + 14*q^2 + 5*q + 1
```

If the partition is trivial (i.e. has only one part), we get the q-factorial (in this case, the nilpotent endomorphism is necessarily 0):

```
sage: from sage.combinat.q_analogues import q_factorial
sage: q_jordan(Partition([5]),3) == q_factorial(5,3)
True
sage: q_jordan(Partition([11]),5) == q_factorial(11,5)
True
```

TESTS:

```
sage: q_jordan(Partition([4,3,1]),1)
Traceback (most recent call last):
...
ValueError: q must not be equal to 1
```

AUTHOR:

- Xavier Caruso (2012-06-29)

sage.combinat.q_analogues.**q_multinomial**(*seq*, *q=None*, *binomial_algorithm='auto'*)

Return the q-multinomial coefficient.

This is also known as the Gaussian multinomial coefficient, and is defined by

$$\binom{n}{k_1, k_2, \ldots, k_m}_q = \frac{[n]_q!}{[k_1]_q![k_2]_q! \cdots [k_m]_q!}$$

where $n = k_1 + k_2 + \cdots + k_m$.

If q is unspecified, then the variable is the generator q for a univariate polynomial ring over the integers.

INPUT:

- `seq` – an iterable of the values k_1 to k_m defined above
- `q` – (default: `None`) the variable q; if `None`, then use a default variable in $\mathbf{Z}[q]$
- `binomial_algorithm` – (default: `'auto'`) the algorithm to use in `q_binomial()`; see possible values there

ALGORITHM:

We use the equivalent formula

$$\binom{k_1 + \cdots + k_m}{k_1, \ldots, k_m}_q = \prod_{i=1}^{m} \binom{\sum_{j=1}^{i} k_j}{k_i}_q.$$

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_multinomial
sage: q_multinomial([1,2,1])
q^5 + 2*q^4 + 3*q^3 + 3*q^2 + 2*q + 1
sage: q_multinomial([1,2,1], q=1) == multinomial([1,2,1])
True
sage: q_multinomial((3,2)) == q_binomial(5,3)
True
sage: q_multinomial([])
1
```

sage.combinat.q_analogues.**q_subgroups_of_abelian_group**(*la*, *mu*, *q=None*, *algorithm='birkhoff'*)

Return the q-number of subgroups of type `mu` in a finite abelian group of type `la`.

INPUT:

- `la` – type of the ambient group as a `Partition`
- `mu` – type of the subgroup as a `Partition`
- `q` – (default: `None`) an indeterminat or a prime number; if `None`, this defaults to $q \in \mathbf{Z}[q]$
- `algorithm` – (default: `'birkhoff'`) the algorithm to use can be one of the following:
  - `'birkhoff` – use the Birkhoff formula from [Bu87]
  - `'delsarte'` – use the formula from [Delsarte48]

OUTPUT:

The number of subgroups of type `mu` in a group of type `la` as a polynomial in `q`.

ALGORITHM:

Let q be a prime number and $\lambda = (\lambda_1, \ldots, \lambda_l)$ be a partition. A finite abelian q-group is of type λ if it is isomorphic to

$$\mathbf{Z}/q^{\lambda_1}\mathbf{Z} \times \cdots \times \mathbf{Z}/q^{\lambda_l}\mathbf{Z}.$$

The formula from [Bu87] works as follows: Let λ and μ be partitions. Let λ' and μ' denote the conjugate partitions to λ and μ, respectively. The number of subgroups of type μ in a group of type λ is given by

$$\prod_{i=1}^{\mu_1} q^{\mu'_{i+1}(\lambda'_i - \mu'_i)} \binom{\lambda'_i - \mu'_{i+1}}{\mu'_i - \mu'_{i+1}}_q$$

The formula from [Delsarte48] works as follows: Let λ and μ be partitions. Let $(s_1, s_2, \ldots, s_l)$ and $(r_1, r_2, \ldots, r_k)$ denote the parts of the partitions conjugate to λ and μ respectively. Let

$$\mathfrak{F}(\xi_1, \ldots, \xi_k) = \xi_1^{r_2}\xi_2^{r_3}\cdots\xi_{k-1}^{r_k} \prod_{i_1=r_2}^{r_1-1}(\xi_1 - q^{i_1}) \prod_{i_2=r_3}^{r_2-1}(\xi_2 - q^{i_2}) \cdots \prod_{i_k=0}^{r_k-1}(\xi_k - q^{-i_k}).$$

Then the number of subgroups of type μ in a group of type λ is given by

$$\frac{\mathfrak{F}(q^{s_1}, q^{s_2}, \ldots, q^{s_k})}{\mathfrak{F}(q^{r_1}, q^{r_2}, \ldots, q^{r_k})}.$$

EXAMPLES:

```
sage: from sage.combinat.q_analogues import q_subgroups_of_abelian_group
sage: q_subgroups_of_abelian_group([1,1], [1])
q + 1
sage: q_subgroups_of_abelian_group([3,3,2,1], [2,1])
q^6 + 2*q^5 + 3*q^4 + 2*q^3 + q^2
sage: R.<t> = QQ[]
sage: q_subgroups_of_abelian_group([5,3,1], [3,1], t)
t^4 + 2*t^3 + t^2
sage: q_subgroups_of_abelian_group([5,3,1], [3,1], 3)
144
sage: q_subgroups_of_abelian_group([1,1,1], [1]) == q_subgroups_of_abelian_group([1,1,1], [1,1])
True
sage: q_subgroups_of_abelian_group([5], [3])
1
sage: q_subgroups_of_abelian_group([1], [2])
0
sage: q_subgroups_of_abelian_group([2], [1,1])
0
```

TESTS:

Check the same examples with `algorithm='delsarte'`:

```
sage: q_subgroups_of_abelian_group([1,1], [1], algorithm='delsarte')
q + 1
sage: q_subgroups_of_abelian_group([3,3,2,1], [2,1], algorithm='delsarte')
q^6 + 2*q^5 + 3*q^4 + 2*q^3 + q^2
sage: q_subgroups_of_abelian_group([5,3,1], [3,1], t, algorithm='delsarte')
t^4 + 2*t^3 + t^2
sage: q_subgroups_of_abelian_group([5,3,1], [3,1], 3, algorithm='delsarte')
144
sage: q_subgroups_of_abelian_group([1,1,1], [1], algorithm='delsarte') == q_subgroups_of_abelian
True
sage: q_subgroups_of_abelian_group([5], [3], algorithm='delsarte')
1
```

```
sage: q_subgroups_of_abelian_group([1], [2], algorithm='delsarte')
0
sage: q_subgroups_of_abelian_group([2], [1,1], algorithm='delsarte')
0
```

REFERENCES:

AUTHORS:

- Amritanshu Prasad (2013-06-07): Implemented the Delsarte algorithm
- Tomer Bauer (2013-09-26): Implemented the Birkhoff algorithm

sage.combinat.q_analogues.**qt_catalan_number**(*n*)

Returns the q,t-Catalan number of index n.

EXAMPLES:

```
sage: from sage.combinat.q_analogues import qt_catalan_number
sage: qt_catalan_number(1)
1
sage: qt_catalan_number(2)
q + t
sage: qt_catalan_number(3)
q^3 + q^2*t + q*t^2 + t^3 + q*t
sage: qt_catalan_number(4)
q^6 + q^5*t + q^4*t^2 + q^3*t^3 + q^2*t^4 + q*t^5 + t^6 + q^4*t + q^3*t^2 + q^2*t^3 + q*t^4 + q^
```

The q,t-Catalan number of index n is only defined for n a nonnegative integer (trac ticket #11411):

```
sage: qt_catalan_number(-2)
Traceback (most recent call last):
...
ValueError: Argument (-2) must be a nonnegative integer.
```

5.1.161 q-Bernoulli Numbers and Polynomials

sage.combinat.q_bernoulli.**q_bernoulli**(*m*, *p=None*)

Compute Carlitz's q-analogue of the Bernoulli numbers.

For every nonnegative integer m, the q-Bernoulli number β_m is a rational function of the indeterminate q whose value at $q = 1$ is the usual Bernoulli number B_m.

INPUT:

- m – a nonnegative integer
- p (default: `None`) – an optional value for q

OUTPUT:

A rational function of the indeterminate q (if p is `None`)

Otherwise, the rational function is evaluated at p.

EXAMPLES:

```
sage: from sage.combinat.q_bernoulli import q_bernoulli
sage: q_bernoulli(0)
1
sage: q_bernoulli(1)
-1/(q + 1)
```

```
sage: q_bernoulli(2)
q/(q^3 + 2*q^2 + 2*q + 1)
sage: all(q_bernoulli(i)(q=1) == bernoulli(i) for i in range(12))
True
```

One can evaluate the rational function by giving a second argument:

```
sage: x = PolynomialRing(GF(2),'x').gen()
sage: q_bernoulli(5,x)
x/(x^6 + x^5 + x + 1)
```

The function does not accept negative arguments:

```
sage: q_bernoulli(-1)
Traceback (most recent call last):
...
ValueError: the argument must be a nonnegative integer
```

REFERENCES:

sage.combinat.q_bernoulli.**q_bernoulli_polynomial**(*m*)

Compute Carlitz's q-analogue of the Bernoulli polynomials.

For every nonnegative integer m, the q-Bernoulli polynomial is a polynomial in one variable x with coefficients in $\mathbf{Q}(q)$ whose value at $q = 1$ is the usual Bernoulli polynomial $B_m(x)$.

The original q-Bernoulli polynomials introduced by Carlitz were polynomials in q^y with coefficients in $\mathbf{Q}(q)$. This function returns these polynomials but expressed in the variable $x = (q^y - 1)/(q - 1)$. This allows to let $q = 1$ to recover the classical Bernoulli polynomials.

INPUT:

- m – a nonnegative integer

OUTPUT:

A polynomial in one variable x.

EXAMPLES:

```
sage: from sage.combinat.q_bernoulli import q_bernoulli_polynomial, q_bernoulli
sage: q_bernoulli_polynomial(0)
1
sage: q_bernoulli_polynomial(1)
(2/(q + 1))*x - 1/(q + 1)
sage: x = q_bernoulli_polynomial(1).parent().gen()
sage: all(q_bernoulli_polynomial(i)(q=1)==bernoulli_polynomial(x,i) for i in range(12))
True
sage: all(q_bernoulli_polynomial(i)(x=0)==q_bernoulli(i) for i in range(12))
True
```

The function does not accept negative arguments:

```
sage: q_bernoulli_polynomial(-1)
Traceback (most recent call last):
...
ValueError: the argument must be a nonnegative integer
```

REFERENCES: [Ca1948], [Ca1954]

5.1.162 Combinatorics quickref

Integer Sequences:

```
sage: s = oeis([1,3,19,211]); s                    # optional - internet
0: A000275: Coefficients of a Bessel function (reciprocal of J_0(z)); also pairs of permutations with
sage: s[0].programs() # optional - internet
0: (PARI) {a(n) = if( n<0, 0, n!^2 * 4^n * polcoeff( 1 / besselj(0, x + x * O(x^(2*n))), 2*n))}; /* _
```

Combinatorial objects:

```
sage: S = Subsets([1,2,3,4]); S.list(); S.<tab>        # not tested
sage: P = Partitions(10000); P.cardinality()
3616...31565042208186860588795256875406642059231055605290691643514 4
sage: Combinations([1,3,7]).random_element()           # random
sage: Compositions(5, max_part = 3).unrank(3)
[2, 2, 1]

sage: DyckWord([1,0,1,0,1,1,0,0]).to_binary_tree()
[., [., [[., .], .]]]
sage: Permutation([3,1,4,2]).robinson_schensted()
[[[1, 2], [3, 4]], [[1, 3], [2, 4]]]
sage: StandardTableau([[1, 4], [2, 5], [3]]).schuetzenberger_involution()
[[1, 3], [2, 4], [5]]
```

Constructions and Species:

```
sage: for (p, s) in cartesian_product([P,S]): print p, s # not tested
sage: DisjointUnionEnumeratedSets(Family(lambda n: IntegerVectors(n, 3), NonNegativeIntegers))  # not
```

Words:

```
sage: Words('abc', 4).list()
[word: aaaa, ..., word: cccc]

sage: Word('aabcacbaa').is_palindrome()
True
sage: WordMorphism('a->ab,b->a').fixed_point('a')
word: abaababaabaababaababaabaababaabaababaaba...
```

Polytopes:

```
sage: points = random_matrix(ZZ, 6, 3, x=7).rows()
sage: L = LatticePolytope(points)
sage: L.npoints(); L.plot3d()                          # random
```

Root systems, Coxeter and Weyl groups:

```
sage: WeylGroup(["B",3]).bruhat_poset()
Finite poset containing 48 elements
sage: RootSystem(["A",2,1]).weight_lattice().plot()    # not tested
```

Crystals:

```
sage: CrystalOfTableaux(["A",3], shape = [3,2]).some_flashy_feature() # not tested
```

Symmetric functions and combinatorial Hopf algebras:

```
sage: Sym = SymmetricFunctions(QQ); Sym.inject_shorthands()
...
doctest:...: RuntimeWarning: redefining global value `e`
sage: m( ( h[2,1] * (1 + 3 * p[2,1]) ) + s[2](s[3]) )
3*m[1, 1, 1] + ... + 10*m[5, 1] + 4*m[6]
```

Discrete groups, Permutation groups:

```
sage: S = SymmetricGroup(4)
sage: M = PolynomialRing(QQ, 'x0,x1,x2,x3')
sage: M.an_element() * S.an_element()
x1
```

Graph theory, posets, lattices (*Graph Theory*, *Posets*):

```
sage: Poset({1: [2,3], 2: [4], 3: [4]}).linear_extensions().cardinality()
2
```

5.1.163 Rankers

sage.combinat.ranker.**from_list**(*l*)

Returns a ranker from the list l.

INPUT:

- `l` - a list

OUTPUT:

- `[rank, unrank]` - functions

EXAMPLES:

```
sage: import sage.combinat.ranker as ranker
sage: l = [1,2,3]
sage: r,u = ranker.from_list(l)
sage: r(1)
0
sage: r(3)
2
sage: u(2)
3
sage: u(0)
1
```

sage.combinat.ranker.**on_fly**()

Returns a pair of enumeration functions rank / unrank.

rank assigns on the fly an integer, starting from 0, to any object passed as argument. The object should be hashable. unrank is the inverse function; it returns None for indices that have not yet been assigned.

EXAMPLES:

```
sage: [rank, unrank] = sage.combinat.ranker.on_fly()
sage: rank('a')
0
sage: rank('b')
1
```

```
sage: rank('c')
2
sage: rank('a')
0
sage: unrank(2)
'c'
sage: unrank(3)
sage: rank('d')
3
sage: unrank(3)
'd'
```

Todo

add tests as in combinat::rankers

---

sage.combinat.ranker.**rank_from_list**(*l*)

Return a rank function for the elements of `l`.

INPUT:

- `l` – a duplicate free list (or iterable) of hashable objects

OUTPUT:

- a function from the elements of `l` to `0,...,len(l)`

EXAMPLES:

```
sage: import sage.combinat.ranker as ranker
sage: l = ['a', 'b', 'c']
sage: r = ranker.rank_from_list(l)
sage: r('a')
0
sage: r('c')
2
```

For non elements a `ValueError` is raised, as with the usual `index` method of lists:

```
sage: r('blah')
Traceback (most recent call last):
...
ValueError: 'blah' is not in dict
```

Currently, the rank function is a `CallableDict`; but this is an implementation detail:

```
sage: type(r)
<type 'sage.misc.callable_dict.CallableDict'>
sage: r
{'a': 0, 'c': 2, 'b': 1}
```

With the current implementation, no error is issued in case of duplicate value in `l`. Instead, the rank function returns the position of some of the duplicates:

```
sage: r = ranker.rank_from_list(['a', 'b', 'a', 'c'])
sage: r('a')
2
```

Constructing the rank function itself is of complexity `O(len(l))`. Then, each call to the rank function consists of an essentially constant time dictionary lookup.

TESTS:

```
sage: TestSuite(r).run()
```

sage.combinat.ranker.**unrank**(*L*, *i*)

Return the i-th element of L.

INPUT:

- `L` – a list, tuple, finite enumerated set, ...
- `i` – an int or `Integer`

The purpose of this utility is to give a uniform idiom to recover the i-th element of an object `L`, whether `L` is a list, tuple (or more generally a `collections.Sequence`), an enumerated set, some old parent of Sage still implementing unranking in the method `__getitem__`, or an iterable (see `collections.Iterable`). See trac ticket #15919.

EXAMPLES:

Lists, tuples, and other `sequences`:

```
sage: from sage.combinat.ranker import unrank
sage: unrank(['a','b','c'], 2)
'c'
sage: unrank(('a','b','c'), 1)
'b'
sage: unrank(xrange(3,13,2), 1)
5
```

Enumerated sets:

```
sage: unrank(GF(7), 2)
2
sage: unrank(IntegerModRing(29), 10)
10
```

An old parent with unranking implemented in `__getitem__`:

```
sage: M = MatrixSpace(GF(3), 2, 2)
sage: hasattr(M, "unrank")
False
sage: M[42]
[1 0]
[2 1]
sage: unrank(M, 42)
[1 0]
[2 1]
```

An iterable:

```
sage: unrank(NN,4)
4
```

An iterator:

```
sage: unrank(('a{}'.format(i) for i in range(20)), 0)
'a0'
sage: unrank(('a{}'.format(i) for i in range(20)), 2)
'a2'
```

Warning: When unranking an iterator, it returns the `i`-th element beyond where it is currently at:

```
sage: from sage.combinat.ranker import unrank
sage: it = iter(range(20))
sage: unrank(it, 2)
2
sage: unrank(it, 2)
5
```

TESTS:

```
sage: from sage.combinat.ranker import unrank
sage: unrank(range(3), 10)
Traceback (most recent call last):
...
IndexError: list index out of range

sage: unrank(('a{}'.format(i) for i in range(20)), 22)
Traceback (most recent call last):
...
IndexError: index out of range

sage: M[100]
Traceback (most recent call last):
...
IndexError: list index out of range
```

sage.combinat.ranker.**unrank_from_list**(*l*)

Returns an unrank function from a list.

EXAMPLES:

```
sage: import sage.combinat.ranker as ranker
sage: l = [1,2,3]
sage: u = ranker.unrank_from_list(l)
sage: u(2)
3
sage: u(0)
1
```

5.1.164 Restricted growth arrays

class sage.combinat.restricted_growth.**RestrictedGrowthArrays**(*n*)

Bases: sage.combinat.combinat.CombinatorialClass

EXAMPLES:

```
sage: from sage.combinat.restricted_growth import RestrictedGrowthArrays
sage: R = RestrictedGrowthArrays(3)
sage: R == loads(dumps(R))
True
```

cardinality()

EXAMPLES:

```
sage: from sage.combinat.restricted_growth import RestrictedGrowthArrays
sage: R = RestrictedGrowthArrays(6)
```

```
sage: R.cardinality()
203
```

5.1.165 Ribbons

5.1.166 Ribbon Shaped Tableaux

class sage.combinat.ribbon_shaped_tableau.**RibbonShapedTableau**(*parent*, *t*)

Bases: sage.combinat.skew_tableau.SkewTableau

A ribbon shaped tableau.

For the purposes of this class, a ribbon shaped tableau is a skew tableau whose shape is a skew partition which:

- has at least one cell in row 1;
- has at least one cell in column 1;
- has exactly one cell in each of q consecutive diagonals, for some nonnegative integer q.

A ribbon is given by a list of the rows from top to bottom.

EXAMPLES:

```
sage: x = RibbonShapedTableau([[None, None, None, 2, 3], [None, 1, 4, 5], [3, 2]]); x
[[None, None, None, 2, 3], [None, 1, 4, 5], [3, 2]]
sage: x.pp()
  .  .  .  2  3
  .  1  4  5
  3  2
sage: x.shape()
[5, 4, 2] / [3, 1]
```

The entries labeled by `None` correspond to the inner partition. Using `None` is optional; the entries will be shifted accordingly.

```
sage: x = RibbonShapedTableau([[2,3],[1,4,5],[3,2]]); x.pp()
  .  .  .  2  3
  .  1  4  5
  3  2
```

TESTS:

```
sage: r = RibbonShapedTableau([[1], [2,3], [4, 5, 6]])
sage: r.to_permutation()
[4, 5, 6, 2, 3, 1]

sage: RibbonShapedTableau([[1,2],[3,4]]).evaluation()
[1, 1, 1, 1]
```

height()

Return the height of `self`.

The height is given by the number of rows in the outer partition.

EXAMPLES:

```
sage: RibbonShapedTableau([[2,3],[1,4,5]]).height()
2
```

spin()
Return the spin of `self`.

EXAMPLES:

```
sage: RibbonShapedTableau([[2,3],[1,4,5]]).spin()
1/2
```

width()
Return the width of the ribbon.

This is given by the length of the longest row in the outer partition.

EXAMPLES:

```
sage: RibbonShapedTableau([[2,3],[1,4,5]]).width()
4
sage: RibbonShapedTableau([]).width()
0
```

class sage.combinat.ribbon_shaped_tableau.**RibbonShapedTableaux**(*category=None*)
Bases: sage.combinat.skew_tableau.SkewTableaux

The set of all ribbon shaped tableaux.

Element
alias of RibbonShapedTableau

from_shape_and_word(*shape*, *word*)
Return the ribbon corresponding to the given ribbon shape and word.

EXAMPLES:

```
sage: RibbonShapedTableaux().from_shape_and_word([1,3],[1,3,3,7])
[[None, None, 1], [3, 3, 7]]
```

global_options(**get_value*, ***set_value*)
Sets the global options for elements of the tableau, skew_tableau, and tableau tuple classes. The defaults are for tableau to be displayed as a list, latexed as a Young diagram using the English convention.

OPTIONS:

- `ascii_art` – (default: `repr`) Controls the ascii art output for tableaux
  - `compact` – minimal length ascii art
  - `repr` – display using the diagram string representation
  - `table` – display as a table
- `convention` – (default: `English`) Sets the convention used for displaying tableaux and partitions
  - `English` – use the English convention
  - `French` – use the French convention
- `display` – (default: `list`) Controls the way in which tableaux are printed
  - `array` – alias for `diagram`
  - `compact` – minimal length string representation
  - `diagram` – display as Young diagram (similar to `pp()`
  - `ferrers_diagram` – alias for `diagram`
  - `list` – print tableaux as lists

  - –young_diagram – alias for diagram
- latex – (default: diagram) Controls the way in which tableaux are latexed
  - –array – alias for diagram
  - –diagram – as a Young diagram
  - –ferrers_diagram – alias for diagram
  - –list – as a list
  - –young_diagram – alias for diagram
- notation – alternative name for convention

Note: Changing the convention for tableaux also changes the convention for partitions.

If no parameters are set, then the function returns a copy of the options dictionary.

EXAMPLES:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T
[[1, 2, 3], [4, 5]]
sage: Tableaux.global_options(display="array")
sage: T
  1  2  3
  4  5
sage: Tableaux.global_options(convention="french")
sage: T
  4  5
  1  2  3
```

Changing the convention for tableaux also changes the convention for partitions and vice versa:

```
sage: P = Partition([3,3,1])
sage: print P.ferrers_diagram()
*
***
***
sage: Partitions.global_options(convention="english")
sage: print P.ferrers_diagram()
***
***
*
sage: T
  1  2  3
  4  5
```

The ASCII art can also be changed:

```
sage: t = Tableau([[1,2,3],[4,5]])
sage: ascii_art(t)
  1  2  3
  4  5
sage: Tableaux.global_options(ascii_art="table")
sage: ascii_art(t)
+---+---+
| 4 | 5 |
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
```

```
sage: Tableaux.global_options(ascii_art="compact")
sage: ascii_art(t)
|4|5|
|1|2|3|
sage: Tableaux.global_options.reset()
```

See `GlobalOptions` for more features of these options.

class sage.combinat.ribbon_shaped_tableau.**Ribbon_class**(*parent*, *t*)
Bases: sage.combinat.ribbon_shaped_tableau.RibbonShapedTableau

This exists solely for unpickling `Ribbon_class` objects.

class sage.combinat.ribbon_shaped_tableau.**StandardRibbonShapedTableaux**(*category=None*)
Bases: sage.combinat.skew_tableau.StandardSkewTableaux

The set of all standard ribbon shaped tableaux.

INPUT:

- `shape` – (optional) the composition shape of the rows

Element
alias of RibbonShapedTableau

from_permutation(*p*)
Return a standard ribbon of size `len(p)` from a permutation `p`. The lengths of each row are given by the distance between the descents of the permutation `p`.

EXAMPLES:

```
sage: import sage.combinat.ribbon_shaped_tableau as rst
sage: [StandardRibbonShapedTableaux().from_permutation(p) for p in Permutations(3)]
[[[1, 2, 3]],
 [[None, 2], [1, 3]],
 [[1, 3], [2]],
 [[None, 1], [2, 3]],
 [[1, 2], [3]],
 [[1], [2], [3]]]
```

from_shape_and_word(*shape*, *word*)
Return the ribbon corresponding to the given ribbon shape and word.

EXAMPLES:

```
sage: StandardRibbonShapedTableaux().from_shape_and_word([2,3],[1,2,3,4,5])
[[None, None, 1, 2], [3, 4, 5]]
```

global_options(**get_value*, ***set_value*)
Sets the global options for elements of the tableau, skew_tableau, and tableau tuple classes. The defaults are for tableau to be displayed as a list, latexed as a Young diagram using the English convention.

OPTIONS:

- `ascii_art` – (default: `repr`) Controls the ascii art output for tableaux
  - `compact` – minimal length ascii art
  - `repr` – display using the diagram string representation
  - `table` – display as a table
- `convention` – (default: `English`) Sets the convention used for displaying tableaux and partitions

– `English` – use the English convention

– `French` – use the French convention

- `display` – (default: `list`) Controls the way in which tableaux are printed
  – `array` – alias for `diagram`
  – `compact` – minimal length string representation
  – `diagram` – display as Young diagram (similar to `pp()`
  – `ferrers_diagram` – alias for `diagram`
  – `list` – print tableaux as lists
  – `young_diagram` – alias for `diagram`
- `latex` – (default: `diagram`) Controls the way in which tableaux are latexed
  – `array` – alias for `diagram`
  – `diagram` – as a Young diagram
  – `ferrers_diagram` – alias for `diagram`
  – `list` – as a list
  – `young_diagram` – alias for `diagram`
- `notation` – alternative name for `convention`

Note: Changing the `convention` for tableaux also changes the `convention` for partitions.

If no parameters are set, then the function returns a copy of the options dictionary.

EXAMPLES:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T
[[1, 2, 3], [4, 5]]
sage: Tableaux.global_options(display="array")
sage: T
  1  2  3
  4  5
sage: Tableaux.global_options(convention="french")
sage: T
  4  5
  1  2  3
```

Changing the `convention` for tableaux also changes the `convention` for partitions and vice versa:

```
sage: P = Partition([3,3,1])
sage: print P.ferrers_diagram()
*
***
***
sage: Partitions.global_options(convention="english")
sage: print P.ferrers_diagram()
***
***
*
sage: T
  1  2  3
  4  5
```

The ASCII art can also be changed:

```
sage: t = Tableau([[1,2,3],[4,5]])
sage: ascii_art(t)
  1  2  3
  4  5
sage: Tableaux.global_options(ascii_art="table")
sage: ascii_art(t)
+---+---+
| 4 | 5 |
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
sage: Tableaux.global_options(ascii_art="compact")
sage: ascii_art(t)
|4|5|
|1|2|3|
sage: Tableaux.global_options.reset()
```

See `GlobalOptions` for more features of these options.

class sage.combinat.ribbon_shaped_tableau.**StandardRibbonShapedTableaux_shape**(*shape*)

Bases: sage.combinat.ribbon_shaped_tableau.StandardRibbonShapedTableaux

Class of standard ribbon shaped tableaux of ribbon shape `shape`.

EXAMPLES:

```
sage: StandardRibbonShapedTableaux([2,2])
Standard ribbon shaped tableaux of shape [2, 2]
sage: StandardRibbonShapedTableaux([2,2]).first()
[[None, 2, 4], [1, 3]]
sage: StandardRibbonShapedTableaux([2,2]).last()
[[None, 1, 2], [3, 4]]
sage: StandardRibbonShapedTableaux([2,2]).cardinality()
5
sage: StandardRibbonShapedTableaux([2,2]).list()
[[[None, 2, 4], [1, 3]],
 [[None, 2, 3], [1, 4]],
 [[None, 1, 4], [2, 3]],
 [[None, 1, 3], [2, 4]],
 [[None, 1, 2], [3, 4]]]
sage: StandardRibbonShapedTableaux([3,2,2]).cardinality()
155
```

first()

Return the first standard ribbon of `self`.

EXAMPLES:

```
sage: StandardRibbonShapedTableaux([2,2]).first()
[[None, 2, 4], [1, 3]]
```

last()

Return the last standard ribbon of `self`.

EXAMPLES:

```
sage: StandardRibbonShapedTableaux([2,2]).last()
[[None, 1, 2], [3, 4]]
```

5.1.167 Ribbon Tableaux

class sage.combinat.ribbon_tableau.**MultiSkewTableau**(*parent*, **args*, ***kwds*)
Bases: sage.combinat.combinat.CombinatorialElement

A multi skew tableau which is a tuple of skew tableaux.

EXAMPLES:

```
sage: s = MultiSkewTableau([ [[None,1],[2,3]], [[1,2],[2]] ])
sage: s.size()
6
sage: s.weight()
[2, 3, 1]
sage: s.shape()
[[2, 2] / [1], [2, 1] / []]
```

TESTS:

```
sage: mst = MultiSkewTableau([ [[None,1],[2,3]], [[1,2],[2]] ])
sage: TestSuite(mst).run()
```

inversion_pairs()
Return a list of the inversion pairs of self.

EXAMPLES:

```
sage: s = MultiSkewTableau([ [[2,3],[5,5]], [[1,1],[3,3]], [[2],[6]] ])
sage: s.inversion_pairs()
[((0, (0, 0)), (1, (0, 0))),
 ((0, (1, 0)), (1, (0, 1))),
 ((0, (1, 1)), (1, (0, 0))),
 ((0, (1, 1)), (1, (1, 1))),
 ((0, (1, 1)), (2, (0, 0))),
 ((1, (0, 1)), (2, (0, 0))),
 ((1, (1, 1)), (2, (0, 0)))]
```

inversions()
Return the number of inversion pairs of self.

EXAMPLES:

```
sage: t1 = SkewTableau([[1]])
sage: t2 = SkewTableau([[2]])
sage: MultiSkewTableau([t1,t1]).inversions()
0
sage: MultiSkewTableau([t1,t2]).inversions()
0
sage: MultiSkewTableau([t2,t2]).inversions()
0
sage: MultiSkewTableau([t2,t1]).inversions()
1
sage: s = MultiSkewTableau([ [[2,3],[5,5]], [[1,1],[3,3]], [[2],[6]] ])
sage: s.inversions()
7
```

shape()
Return the shape of self.

EXAMPLES:

```
sage: s = SemistandardSkewTableaux([[2,2],[1]]).list()
sage: a = MultiSkewTableau([s[0],s[1],s[2]])
sage: a.shape()
[[2, 2] / [1], [2, 2] / [1], [2, 2] / [1]]
```

size()

Return the size of `self`, which is the sum of the sizes of the skew tableaux in `self`.

EXAMPLES:

```
sage: s = SemistandardSkewTableaux([[2,2],[1]]).list()
sage: a = MultiSkewTableau([s[0],s[1],s[2]])
sage: a.size()
9
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: s = SemistandardSkewTableaux([[2,2],[1]]).list()
sage: a = MultiSkewTableau([s[0],s[1],s[2]])
sage: a.weight()
[5, 3, 1]
```

class sage.combinat.ribbon_tableau.**MultiSkewTableaux**(*category=None*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Multiskew tableaux.

Element

alias of MultiSkewTableau

class sage.combinat.ribbon_tableau.**RibbonTableau**(*parent*, *st*)

Bases: sage.combinat.skew_tableau.SkewTableau

A ribbon tableau.

A ribbon is a connected skew shape which does not contain any 2×2 boxes. A ribbon tableau is a skew tableau whose shape is partitioned into ribbons, each of which is filled with identical entries.

EXAMPLES:

```
sage: rt = RibbonTableau([[None, 1],[2,3]]); rt
[[None, 1], [2, 3]]
sage: rt.inner_shape()
[1]
sage: rt.outer_shape()
[2, 2]

sage: rt = RibbonTableau([[None, None, 0, 0, 0], [None, 0, 0, 2], [1, 0, 1]]); rt.pp()
  .  .  0  0  0
  .  0  0  2
  1  0  1
```

In the previous example, each ribbon is uniquely determined by a non-zero entry. The 0 entries are used to fill in the rest of the skew shape.

Note: Sanity checks are not performed; lists can contain any object.

```
sage: RibbonTableau(expr=[[1,1],[[5],[3,4],[1,2]]])
[[None, 1, 2], [None, 3, 4], [5]]
```

TESTS:

```
sage: RibbonTableau([[0, 0, 3, 0], [1, 1, 0], [2, 0, 4]]).evaluation()
[2, 1, 1, 1]
```

length()

Return the length of the ribbons into a ribbon tableau.

EXAMPLES:

```
sage: RibbonTableau([[None, 1],[2,3]]).length()
1
sage: RibbonTableau([[1,0],[2,0]]).length()
2
```

to_word()

Return a word obtained from a row reading of `self`.

Warning: Unlike the `to_word` method on skew tableaux (which are a superclass of this), this method does not filter out `None` entries.

EXAMPLES:

```
sage: R = RibbonTableau([[0, 0, 3, 0], [1, 1, 0], [2, 0, 4]])
sage: R.to_word()
word: 2041100030
```

class sage.combinat.ribbon_tableau.**RibbonTableau_class**(*parent*, *st*)

Bases: sage.combinat.ribbon_tableau.RibbonTableau

This exists solely for unpickling `RibbonTableau_class` objects.

class sage.combinat.ribbon_tableau.**RibbonTableaux**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Ribbon tableaux.

A ribbon tableau is a skew tableau whose skew shape `shape` is tiled by ribbons of length `length`. The weight `weight` is calculated from the labels on the ribbons.

Note: Here we impose the condition that the ribbon tableaux are semistandard.

INPUT(Optional):

- `shape` – skew shape as a list of lists or an object of type SkewPartition
- `length` – integer, `shape` is partitioned into ribbons of length `length`
- `weight` – list of integers, computed from the values of non-zero entries labeling the ribbons

EXAMPLES:

```
sage: RibbonTableaux([[2,1],[]], [1,1,1], 1)
Ribbon tableaux of shape [2, 1] / [] and weight [1, 1, 1] with 1-ribbons
```

```
sage: R = RibbonTableaux([[5,4,3],[2,1]], [2,1], 3)
sage: for i in R: i.pp(); print
```

```
.  .  0  0  0
.  0  0  2
1  0  1

.  .  1  0  0
.  0  0  0
1  0  2

.  .  0  0  0
.  1  0  1
2  0  0
```

REFRENCES:

Element
alias of RibbonTableau

from_expr(*l*)
Return a RibbonTableau from a MuPAD-Combinat expr for a skew tableau. The first list in expr is the inner shape of the skew tableau. The second list are the entries in the rows of the skew tableau from bottom to top.

Provided primarily for compatibility with MuPAD-Combinat.

EXAMPLES:

```
sage: RibbonTableaux().from_expr([[1,1],[[5],[3,4],[1,2]]])
[[None, 1, 2], [None, 3, 4], [5]]
```

global_options(**get_value*, ***set_value*)
Sets the global options for elements of the tableau, skew_tableau, and tableau tuple classes. The defaults are for tableau to be displayed as a list, latexed as a Young diagram using the English convention.

OPTIONS:

- ascii_art – (default: repr) Controls the ascii art output for tableaux
  - compact – minimal length ascii art
  - repr – display using the diagram string representation
  - table – display as a table
- convention – (default: English) Sets the convention used for displaying tableaux and partitions
  - English – use the English convention
  - French – use the French convention
- display – (default: list) Controls the way in which tableaux are printed
  - array – alias for diagram
  - compact – minimal length string representation
  - diagram – display as Young diagram (similar to pp())
  - ferrers_diagram – alias for diagram
  - list – print tableaux as lists
  - young_diagram – alias for diagram
- latex – (default: diagram) Controls the way in which tableaux are latexed

- `array` – alias for `diagram`
- `diagram` – as a Young diagram
- `ferrers_diagram` – alias for `diagram`
- `list` – as a list
- `young_diagram` – alias for `diagram`

• `notation` – alternative name for `convention`

Note: Changing the `convention` for tableaux also changes the `convention` for partitions.

If no parameters are set, then the function returns a copy of the options dictionary.

EXAMPLES:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T
[[1, 2, 3], [4, 5]]
sage: Tableaux.global_options(display="array")
sage: T
  1  2  3
  4  5
sage: Tableaux.global_options(convention="french")
sage: T
  4  5
  1  2  3
```

Changing the `convention` for tableaux also changes the `convention` for partitions and vice versa:

```
sage: P = Partition([3,3,1])
sage: print P.ferrers_diagram()
*
***
***
sage: Partitions.global_options(convention="english")
sage: print P.ferrers_diagram()
***
***
*
sage: T
  1  2  3
  4  5
```

The ASCII art can also be changed:

```
sage: t = Tableau([[1,2,3],[4,5]])
sage: ascii_art(t)
  1  2  3
  4  5
sage: Tableaux.global_options(ascii_art="table")
sage: ascii_art(t)
+---+---+
| 4 | 5 |
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
sage: Tableaux.global_options(ascii_art="compact")
sage: ascii_art(t)
|4|5|
```

```
|1|2|3|
sage: Tableaux.global_options.reset()
```

See `GlobalOptions` for more features of these options.

class sage.combinat.ribbon_tableau.**RibbonTableaux_shape_weight_length**(*shape*, *weight*, *length*)

Bases: sage.combinat.ribbon_tableau.RibbonTableaux

Ribbon tableaux of a given shape, weight, and length.

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: RibbonTableaux([[2,1],[]],[1,1,1],1).cardinality()
2
sage: RibbonTableaux([[2,2],[]],[1,1],2).cardinality()
2
sage: RibbonTableaux([[4,3,3],[]],[2,1,1,1],2).cardinality()
5
```

TESTS:

```
sage: RibbonTableaux([6,6,6], [4,2], 3).cardinality()
6
sage: RibbonTableaux([3,3,3,2,1], [3,1], 3).cardinality()
1
sage: RibbonTableaux([3,3,3,2,1], [2,2], 3).cardinality()
2
sage: RibbonTableaux([3,3,3,2,1], [2,1,1], 3).cardinality()
5
sage: RibbonTableaux([3,3,3,2,1], [1,1,1,1], 3).cardinality()
12
sage: RibbonTableaux([5,4,3,2,1], [2,2,1], 3).cardinality()
10

sage: RibbonTableaux([8,7,6,5,1,1], [3,2,2,1], 3).cardinality()
85
sage: RibbonTableaux([5,4,3,2,1,1,1], [2,2,1], 3).cardinality()
10

sage: RibbonTableaux([7,7,7,2,1,1], [3,2,0,1,1], 3).cardinality()
25
```

Weights with some zeros in the middle and end:

```
sage: RibbonTableaux([3,3,3], [0,1,0,2,0], 3).cardinality()
3
sage: RibbonTableaux([3,3,3], [1,0,1,0,1,0,0,0], 3).cardinality()
6
```

class sage.combinat.ribbon_tableau.**SemistandardMultiSkewTableaux**(*shape*, *weight*)

Bases: sage.combinat.ribbon_tableau.MultiSkewTableaux

Semistandard multi skew tableaux.

A multi skew tableau is a k-tuple of skew tableaux of given shape with a specified total weight.

EXAMPLES:

```
sage: S = SemistandardMultiSkewTableaux([ [[2,1],[]], [[2,2],[1]] ], [2,2,2]); S
Semistandard multi skew tableaux of shape [[2, 1] / [], [2, 2] / [1]] and weight [2, 2, 2]
sage: S.list()
[[[[1, 1], [2]], [[None, 2], [3, 3]]],
 [[[1, 2], [2]], [[None, 1], [3, 3]]],
 [[[1, 3], [2]], [[None, 2], [1, 3]]],
 [[[1, 3], [2]], [[None, 1], [2, 3]]],
 [[[1, 1], [3]], [[None, 2], [2, 3]]],
 [[[1, 2], [3]], [[None, 2], [1, 3]]],
 [[[1, 2], [3]], [[None, 1], [2, 3]]],
 [[[2, 2], [3]], [[None, 1], [1, 3]]],
 [[[1, 3], [3]], [[None, 1], [2, 2]]],
 [[[2, 3], [3]], [[None, 1], [1, 2]]]]
```

sage.combinat.ribbon_tableau.**cospin_polynomial**(*part*, *weight*, *length*)

Return the cospin polynomial associated to `part`, `weight`, and `length`.

EXAMPLES:

```
sage: from sage.combinat.ribbon_tableau import cospin_polynomial
sage: cospin_polynomial([6,6,6],[4,2],3)
t^4 + t^3 + 2*t^2 + t + 1
sage: cospin_polynomial([3,3,3,2,1], [3,1], 3)
1
sage: cospin_polynomial([3,3,3,2,1], [2,2], 3)
t + 1
sage: cospin_polynomial([3,3,3,2,1], [2,1,1], 3)
t^2 + 2*t + 2
sage: cospin_polynomial([3,3,3,2,1], [1,1,1,1], 3)
t^3 + 3*t^2 + 5*t + 3
sage: cospin_polynomial([5,4,3,2,1,1,1], [2,2,1], 3)
2*t^2 + 6*t + 2
sage: cospin_polynomial([[6]*6, [3,3]], [4,4,2], 3)
3*t^4 + 6*t^3 + 9*t^2 + 5*t + 3
```

sage.combinat.ribbon_tableau.**count_rec**(*nexts*, *current*, *part*, *weight*, *length*)

INPUT:

- `nexts, current, part` – skew partitions
- `weight` – non-negative integer list
- `length` – integer

TESTS:

```
sage: from sage.combinat.ribbon_tableau import count_rec
sage: count_rec([], [], [[2, 1, 1], []], [2], 2)
[0]
sage: count_rec([[0], [1]], [[[2, 1, 1], [0, 0, 2, 0]], [[4], [2, 0, 0, 0]]], [[4, 1, 1], []], [
[1]
sage: count_rec([], [[[], [2, 2]]], [[2, 2], []], [2], 2)
[1]
sage: count_rec([], [[[], [2, 0, 2, 0]]], [[4], []], [2], 2)
[1]
sage: count_rec([[1], [1]], [[[2, 2], [0, 0, 2, 0]], [[4], [2, 0, 0, 0]]], [[4, 2], []], [2, 1],
[2]
sage: count_rec([[1], [1], [2]], [[[2, 2, 2], [0, 0, 2, 0]], [[4, 1, 1], [0, 2, 0, 0]], [[4, 2],
[4]
sage: count_rec([[4], [1]], [[[4, 2, 2], [0, 0, 2, 0]], [[4, 3, 1], [0, 2, 0, 0]]], [[4, 3, 3],
[5]
```

sage.combinat.ribbon_tableau.**graph_implementation_rec**(*skp*, *weight*, *length*, *function*)

TESTS:

```
sage: from sage.combinat.ribbon_tableau import graph_implementation_rec, list_rec
sage: graph_implementation_rec(SkewPartition([[1], []]), [1], 1, list_rec)
[[[], [[1]]]]
sage: graph_implementation_rec(SkewPartition([[2, 1], []]), [1, 2], 1, list_rec)
[[[], [[2], [1, 2]]]]
sage: graph_implementation_rec(SkewPartition([[], []]), [0], 1, list_rec)
[[[], []]]
```

sage.combinat.ribbon_tableau.**insertion_tableau**(*skp*, *perm*, *evaluation*, *tableau*, *length*)

INPUT:

- `skp` – skew partitions
- `perm, evaluation` – non-negative integers
- `tableau` – skew tableau
- `length` – integer

TESTS:

```
sage: from sage.combinat.ribbon_tableau import insertion_tableau
sage: insertion_tableau([[1], []], [1], 1, [[], []], 1)
[[], [[1]]]
sage: insertion_tableau([[2, 1], []], [1, 1], 2, [[], [[1]]], 1)
[[], [[2], [1, 2]]]
sage: insertion_tableau([[2, 1], []], [0, 0], 3, [[], [[2], [1, 2]]], 1)
[[], [[2], [1, 2]]]
sage: insertion_tableau([[1, 1], []], [1], 2, [[], [[1]]], 1)
[[], [[2], [1]]]
sage: insertion_tableau([[2], []], [0, 1], 2, [[], [[1]]], 1)
[[], [[1, 2]]]
sage: insertion_tableau([[2, 1], []], [0, 1], 3, [[], [[2], [1]]], 1)
[[], [[2], [1, 3]]]
sage: insertion_tableau([[1, 1], []], [2], 1, [[], []], 2)
[[], [[1], [0]]]
sage: insertion_tableau([[2], []], [2, 0], 1, [[], []], 2)
[[], [[1, 0]]]
sage: insertion_tableau([[2, 2], []], [0, 2], 2, [[], [[1], [0]]], 2)
[[], [[1, 2], [0, 0]]]
sage: insertion_tableau([[2, 2], []], [2, 0], 2, [[], [[1, 0]]], 2)
[[], [[2, 0], [1, 0]]]
sage: insertion_tableau([[2, 2], [1]], [3, 0], 1, [[], []], 3)
[[1], [[1, 0], [0]]]
```

sage.combinat.ribbon_tableau.**list_rec**(*nexts*, *current*, *part*, *weight*, *length*)

INPUT:

- `nexts, current, part` – skew partitions
- `weight` – non-negative integer list
- `length` – integer

TESTS:

```
sage: from sage.combinat.ribbon_tableau import list_rec
sage: list_rec([], [[[], [1]]], [[1], []], [1], 1)
```

```
[[[], [[1]]]]
sage: list_rec([[[[]], [[1]]]]], [[[1], [1, 1]]], [[2, 1], []], [1, 2], 1)
[[[], [[2], [1, 2]]]]
sage: list_rec([], [[[1], [3, 0]]], [[2, 2], [1]], [1], 3)
[[[1], [[1, 0], [0]]]]
sage: list_rec([[[[]], [[2]]]]], [[[1], [1, 1]]], [[2, 1], []], [0, 1, 2], 1)
[[[], [[3], [2, 3]]]]
sage: list_rec([], [[[], [2]]], [[1, 1], []], [1], 2)
[[[], [[1], [0]]]]
sage: list_rec([], [[[], [2, 0]]], [[2], []], [1], 2)
[[[], [[1, 0]]]]
sage: list_rec([[[[]], [[1], [0]]]], [[[], [[1, 0]]]]], [[[1, 1], [0, 2]], [[2], [2, 0]]], [[2, 2
[[[], [[1, 2], [0, 0]]], [[], [[2, 0], [1, 0]]]]
sage: list_rec([], [[[], [2, 2]]], [[2, 2], []], [2], 2)
[[[], [[1, 1], [0, 0]]]]
sage: list_rec([], [[[], [1, 1]]], [[2], []], [2], 1)
[[[], [[1, 1]]]]
sage: list_rec([[[[]], [[1, 1]]]]], [[[2], [1, 1]]], [[2, 2], []], [2, 2], 1)
[[[], [[2, 2], [1, 1]]]]
```

sage.combinat.ribbon_tableau.**spin_polynomial**(*part*, *weight*, *length*)

Returns the spin polynomial associated to `part`, `weight`, and `length`.

EXAMPLES:

```
sage: from sage.combinat.ribbon_tableau import spin_polynomial
sage: spin_polynomial([6,6,6],[4,2],3)
t^6 + t^5 + 2*t^4 + t^3 + t^2
sage: spin_polynomial([6,6,6],[4,1,1],3)
t^6 + 2*t^5 + 3*t^4 + 2*t^3 + t^2
sage: spin_polynomial([3,3,3,2,1], [2,2], 3)
t^(7/2) + t^(5/2)
sage: spin_polynomial([3,3,3,2,1], [2,1,1], 3)
2*t^(7/2) + 2*t^(5/2) + t^(3/2)
sage: spin_polynomial([3,3,3,2,1], [1,1,1,1], 3)
3*t^(7/2) + 5*t^(5/2) + 3*t^(3/2) + sqrt(t)
sage: spin_polynomial([5,4,3,2,1,1,1], [2,2,1], 3)
2*t^(9/2) + 6*t^(7/2) + 2*t^(5/2)
sage: spin_polynomial([[6]*6, [3,3]], [4,4,2], 3)
3*t^9 + 5*t^8 + 9*t^7 + 6*t^6 + 3*t^5
```

sage.combinat.ribbon_tableau.**spin_polynomial_square**(*part*, *weight*, *length*)

Returns the spin polynomial associated with `part`, `weight`, and `length`, with the substitution $t \to t^2$ made.

EXAMPLES:

```
sage: from sage.combinat.ribbon_tableau import spin_polynomial_square
sage: spin_polynomial_square([6,6,6],[4,2],3)
t^12 + t^10 + 2*t^8 + t^6 + t^4
sage: spin_polynomial_square([6,6,6],[4,1,1],3)
t^12 + 2*t^10 + 3*t^8 + 2*t^6 + t^4
sage: spin_polynomial_square([3,3,3,2,1], [2,2], 3)
t^7 + t^5
sage: spin_polynomial_square([3,3,3,2,1], [2,1,1], 3)
2*t^7 + 2*t^5 + t^3
sage: spin_polynomial_square([3,3,3,2,1], [1,1,1,1], 3)
3*t^7 + 5*t^5 + 3*t^3 + t
sage: spin_polynomial_square([5,4,3,2,1,1,1], [2,2,1], 3)
2*t^9 + 6*t^7 + 2*t^5
```

```
sage: spin_polynomial_square([[6]*6, [3,3]], [4,4,2], 3)
3*t^18 + 5*t^16 + 9*t^14 + 6*t^12 + 3*t^10
```

sage.combinat.ribbon_tableau.**spin_rec**(*t*, *nexts*, *current*, *part*, *weight*, *length*)

Routine used for constructing the spin polynomial.

INPUT:

- `weight` – list of non-negative integers
- `length` – the length of the ribbons we're tiling with
- `t` – the variable

EXAMPLES:

```
sage: from sage.combinat.ribbon_tableau import spin_rec
sage: sp = SkewPartition
sage: t = ZZ['t'].gen()
sage: spin_rec(t, [], [[[], [3, 3]]], sp([[2, 2, 2], []]), [2], 3)
[t^4]
sage: spin_rec(t, [[0], [t^4]], [[[2, 1, 1, 1, 1], [0, 3]], [[2, 2, 2], [3, 0]]], sp([[2, 2, 2,
[t^5]
sage: spin_rec(t, [], [[[], [3, 3, 0]]], sp([[3, 3], []]), [2], 3)
[t^2]
sage: spin_rec(t, [[t^4], [t^3], [t^2]], [[[2, 2, 2], [0, 0, 3]], [[3, 2, 1], [0, 3, 0]], [[3, 3
[t^6 + t^4 + t^2]
sage: spin_rec(t, [[t^5], [t^4], [t^6 + t^4 + t^2]], [[[2, 2, 2, 2, 1], [0, 0, 3]], [[3, 3, 1, 1
[2*t^7 + 2*t^5 + t^3]
```

5.1.168 Rigged Configurations

Todo

Proofread / point to the main classes rather than the modules?

- *Crystal of Rigged Configurations*
- *Rigged Configurations of*
- *Rigged Configurations*
- *Rigged Configuration Elements*
- *Tensor Product of Kirillov-Reshetikhin Tableaux*
- *Tensor Product of Kirillov-Reshetikhin Tableaux Elements*
- *Kirillov-Reshetikhin Tableaux*
- *Kleber Trees*
- *Rigged Partitions*

Bijections

- *Bijection between rigged configurations and KR tableaux*
- *Abstract classes for the rigged configuration bijections*

- *Bijection classes for type*
- *Bijection classes for type .*
- *Bijection classes for type .*
- *Bijection classes for type*
- *Bijection classes for type .*
- *Bijection classes for type .*
- *Bijection classes for type .*
- *Bijection classes for type .*
- *Bijection classes for type .*
- *Bijection between rigged configurations for and marginally large tableaux*

5.1.169 Features that are imported by default in the interpreter namespace

5.1.170 Abstract classes for the rigged configuration bijections

This file contains two sets of classes, one for the bijection from KR tableaux to rigged configurations and the other for the reverse bijection. We do this for two reasons, one is because we can store a state in the bijection locally, so we do not have to constantly pass it around between functions. The other is because it makes the code easier to read in the *_element.py files.

These classes are not meant to be used by the user and are only supposed to be used internally to perform the bijections between `TensorProductOfKirillovReshetikhinTableaux` and `RiggedConfigurations`.

AUTHORS:

- Travis Scrimshaw (2011-04-15): Initial version

class `sage.combinat.rigged_configurations.bij_abstract_class.`**`KRTToRCBijectionAbstract`**(*tp_krt*)
Root abstract class for the bijection from KR tableaux to rigged configurations.

This class holds the state of the bijection and generates the next state. This class should never be created directly.

`next_state`(*val*)
Build the next state in the bijection.

INPUT:

- `val` – The value we are adding

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 4, 1], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_A import KRTToRCBijectionTypeA
sage: bijection = KRTToRCBijectionTypeA(KRT(pathlist=[[5,3]]))
sage: bijection.cur_path.insert(0, [])
sage: bijection.cur_dims.insert(0, [0, 1])
sage: bijection.cur_path[0].insert(0, [3])
sage: bijection.next_state(3)
sage: bijection.ret_rig_con

-1[ ]-1

-1[ ]-1
```

```
(/)

(/)
```

run(*verbose=False*)

Run the bijection from a tensor product of KR tableaux to a rigged configuration.

INPUT:

- `tp_krt` – A tensor product of KR tableaux
- `verbose` – (Default: `False`) Display each step in the bijection

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 4, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_A import KRTToRCBijectionTypeA
sage: KRTToRCBijectionTypeA(KRT(pathlist=[[5,2]])).run()

-1[ ]-1

1[ ]1

0[ ]0

-1[ ]-1
```

class `sage.combinat.rigged_configurations.bij_abstract_class.`**RCToKRTBijectionAbstract**(*RC_eleme*

Root abstract class for the bijection from rigged configurations to tensor product of Kirillov-Reshetikhin tableaux.

This class holds the state of the bijection and generates the next state. This class should never be created directly.

next_state(*height*)

Build the next state in the bijection.

TESTS:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_A import RCToKRTBijectionTypeA
sage: bijection = RCToKRTBijectionTypeA(RC(partition_list=[[1],[1],[1],[1]]))
sage: bijection.next_state(0)
5
sage: bijection.cur_partitions
[(/)
 , (/)
 , (/)
 , (/)
 ]
```

run(*verbose=False*, *build_graph=False*)

Run the bijection from rigged configurations to tensor product of KR tableaux.

INPUT:

- `verbose` – (default: `False`) display each step in the bijection
- `build_graph` – (default: `False`) build the graph of each step of the bijection

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 1]])
sage: x = RC(partition_list=[[1],[1],[1],[1]])
sage: from sage.combinat.rigged_configurations.bij_type_A import RCToKRTBijectionTypeA
sage: RCToKRTBijectionTypeA(x).run()
[[2], [5]]
sage: bij = RCToKRTBijectionTypeA(x)
sage: bij.run(build_graph=True)
[[2], [5]]
sage: bij._graph
Digraph on 3 vertices
```

5.1.171 Bijection between rigged configurations for $B(\infty)$ and marginally large tableaux

AUTHORS:

- Travis Scrimshaw (2015-07-01): Initial version

REFERENCES:

class sage.combinat.rigged_configurations.bij_infinity.**FromRCIsomorphism**
Bases: sage.categories.morphism.Morphism

Crystal isomorphism of $B(\infty)$ in the rigged configuration model to the tableau model.

class sage.combinat.rigged_configurations.bij_infinity.**FromTableauIsomorphism**
Bases: sage.categories.morphism.Morphism

Crystal isomorphism of $B(\infty)$ in the tableau model to the rigged configuration model.

class sage.combinat.rigged_configurations.bij_infinity.**MLTToRCBijectionTypeB**(*tp_krt*)
Bases: sage.combinat.rigged_configurations.bij_type_B.KRTToRCBijectionTypeB

Initialize the bijection by obtaining the important information from the KR tableaux.

INPUT:

- `parent` – The parent of tensor product of KR tableaux

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 4, 1], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_A import KRTToRCBijectionTypeA
sage: bijection = KRTToRCBijectionTypeA(KRT(pathlist=[[3,1]]))
sage: TestSuite(bijection).run()
```

run()
Run the bijection from a marginally large tableaux to a rigged configuration.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['B',4])
sage: T = crystals.infinity.Tableaux(['B',4])
sage: Psi = T.crystal_morphism({T.module_generators[0]: RC.module_generators[0]})
sage: TS = [x.value for x in T.subcrystal(max_depth=4)]
sage: all(Psi(b) == RC(b) for b in TS) # long time # indirect doctest
True
```

class sage.combinat.rigged_configurations.bij_infinity.**MLTToRCBijectionTypeD**(*tp_krt*)
Bases: sage.combinat.rigged_configurations.bij_type_D.KRTToRCBijectionTypeD

Initialize the bijection by obtaining the important information from the KR tableaux.

INPUT:

- `parent` – The parent of tensor product of KR tableaux

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 4, 1], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_A import KRTToRCBijectionTypeA
sage: bijection = KRTToRCBijectionTypeA(KRT(pathlist=[[3,1]]))
sage: TestSuite(bijection).run()
```

run()

Run the bijection from a marginally large tableaux to a rigged configuration.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['D',4])
sage: T = crystals.infinity.Tableaux(['D',4])
sage: Psi = T.crystal_morphism({T.module_generators[0]: RC.module_generators[0]})
sage: TS = [x.value for x in T.subcrystal(max_depth=4)]
sage: all(Psi(b) == RC(b) for b in TS) # long time # indirect doctest
True
```

class sage.combinat.rigged_configurations.bij_infinity.**RCToMLTBijectionTypeB**(*RC_element*)

Bases: sage.combinat.rigged_configurations.bij_type_B.RCToKRTBijectionTypeB

Initialize the bijection helper.

INPUT:

- `RC_element` – The rigged configuration

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_abstract_class import RCToKRTBijectionAbstrac
sage: bijection = RCToKRTBijectionAbstract(RC(partition_list=[[1],[1],[1],[1]]))
sage: TestSuite(bijection).run()
```

run()

Run the bijection from rigged configurations to a marginally large tableau.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['B',4])
sage: T = crystals.infinity.Tableaux(['B',4])
sage: Psi = RC.crystal_morphism({RC.module_generators[0]: T.module_generators[0]})
sage: RCS = [x.value for x in RC.subcrystal(max_depth=4)]
sage: all(Psi(nu) == T(nu) for nu in RCS) # long time # indirect doctest
True
```

class sage.combinat.rigged_configurations.bij_infinity.**RCToMLTBijectionTypeD**(*RC_element*)

Bases: sage.combinat.rigged_configurations.bij_type_D.RCToKRTBijectionTypeD

Initialize the bijection helper.

INPUT:

- `RC_element` – The rigged configuration

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_abstract_class import RCToKRTBijectionAbstrac
sage: bijection = RCToKRTBijectionAbstract(RC(partition_list=[[1],[1],[1],[1]]))
sage: TestSuite(bijection).run()
```

run()

Run the bijection from rigged configurations to a marginally large tableau.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['D',4])
sage: T = crystals.infinity.Tableaux(['D',4])
sage: Psi = RC.crystal_morphism({RC.module_generators[0]: T.module_generators[0]})
sage: RCS = [x.value for x in RC.subcrystal(max_depth=4)]
sage: all(Psi(nu) == T(nu) for nu in RCS) # long time # indirect doctest
True
```

5.1.172 Bijection classes for type $A_n^{(1)}$

Part of the (internal) classes which run the bijection between rigged configurations and tensor products of Kirillov-Reshetikhin tableaux of type $A_n^{(1)}$.

AUTHORS:

- Travis Scrimshaw (2011-04-15): Initial version

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 4, 1], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_A import KRTToRCBijectionTypeA
sage: bijection = KRTToRCBijectionTypeA(KRT(pathlist=[[5,2]]))
sage: TestSuite(bijection).run()
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_A import RCToKRTBijectionTypeA
sage: bijection = RCToKRTBijectionTypeA(RC(partition_list=[[],[],[],[]]))
sage: TestSuite(bijection).run()
```

class sage.combinat.rigged_configurations.bij_type_A.**KRTToRCBijectionTypeA**(*tp_krt*)

Bases: sage.combinat.rigged_configurations.bij_abstract_class.KRTToRCBijectionAbstract

Specific implementation of the bijection from KR tableaux to rigged configurations for type $A_n^{(1)}$.

next_state(*val*)

Build the next state for type $A_n^{(1)}$.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 4, 1], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_A import KRTToRCBijectionTypeA
sage: bijection = KRTToRCBijectionTypeA(KRT(pathlist=[[4,3]]))
sage: bijection.cur_path.insert(0, [])
sage: bijection.cur_dims.insert(0, [0, 1])
sage: bijection.cur_path[0].insert(0, [3])
sage: bijection.next_state(3)
```

class sage.combinat.rigged_configurations.bij_type_A.**RCToKRTBijectionTypeA**(*RC_element*)

Bases: sage.combinat.rigged_configurations.bij_abstract_class.RCToKRTBijectionAbstract

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $A_n^{(1)}$.

next_state(*height*)

Build the next state for type $A_n^{(1)}$.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_A import RCToKRTBijectionTypeA
sage: bijection = RCToKRTBijectionTypeA(RC(partition_list=[[1],[1],[1],[1]]))
sage: bijection.next_state(0)
5
```

5.1.173 Bijection classes for type $A_{2n}^{(2)\dagger}$.

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $A_{2n}^{(2)\dagger}$.

AUTHORS:

- Travis Scrimshaw (2012-12-21): Initial version

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(CartanType(['A', 4, 2]).dual(), [[2,
sage: from sage.combinat.rigged_configurations.bij_type_A2_dual import KRTToRCBijectionTypeA2Dual
sage: bijection = KRTToRCBijectionTypeA2Dual(KRT(pathlist=[[2,1]]))
sage: TestSuite(bijection).run()
sage: RC = RiggedConfigurations(CartanType(['A', 4, 2]).dual(), [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_A2_dual import RCToKRTBijectionTypeA2Dual
sage: bijection = RCToKRTBijectionTypeA2Dual(RC(partition_list=[[],[]]))
sage: TestSuite(bijection).run()
```

class sage.combinat.rigged_configurations.bij_type_A2_dual.**KRTToRCBijectionTypeA2Dual**(*tp_krt*)

Bases: sage.combinat.rigged_configurations.bij_type_C.KRTToRCBijectionTypeC

Specific implementation of the bijection from KR tableaux to rigged configurations for type $A_{2n}^{(2)\dagger}$.

This inherits from type $C_n^{(1)}$ because we use the same methods in some places.

next_state(*val*)

Build the next state for type $A_{2n}^{(2)\dagger}$.

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(CartanType(['A', 4, 2]).dual
sage: from sage.combinat.rigged_configurations.bij_type_A2_dual import KRTToRCBijectionTypeA
sage: bijection = KRTToRCBijectionTypeA2Dual(KRT(pathlist=[[-1,2]]))
sage: bijection.cur_path.insert(0, [])
sage: bijection.cur_dims.insert(0, [0, 1])
sage: bijection.cur_path[0].insert(0, [2])
sage: bijection.next_state(2)
```

class sage.combinat.rigged_configurations.bij_type_A2_dual.**RCToKRTBijectionTypeA2Dual**(*RC_eleme*

Bases: sage.combinat.rigged_configurations.bij_type_C.RCToKRTBijectionTypeC

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $A_{2n}^{(2)\dagger}$.

next_state(*height*)

Build the next state for type $A_{2n}^{(2)\dagger}$.

TESTS:

```
sage: RC = RiggedConfigurations(CartanType(['A', 4, 2]).dual(), [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_A2_dual import RCToKRTBijectionTypeA
sage: bijection = RCToKRTBijectionTypeA2Dual(RC(partition_list=[[2],[2,2]]))
sage: bijection.next_state(1)
-1
```

5.1.174 Bijection classes for type $A_{2n}^{(2)}$.

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $A_{2n}^{(2)}$.

AUTHORS:

- Travis Scrimshaw (2012-12-21): Initial version

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 4, 2], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_A2_even import KRTToRCBijectionTypeA2Even
sage: bijection = KRTToRCBijectionTypeA2Even(KRT(pathlist=[[-1,2]]))
sage: TestSuite(bijection).run()
sage: RC = RiggedConfigurations(['A', 4, 2], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_A2_even import RCToKRTBijectionTypeA2Even
sage: bijection = RCToKRTBijectionTypeA2Even(RC(partition_list=[[],[]]))
sage: TestSuite(bijection).run()
```

class sage.combinat.rigged_configurations.bij_type_A2_even.**KRTToRCBijectionTypeA2Even**(*tp_krt*)

Bases: sage.combinat.rigged_configurations.bij_type_C.KRTToRCBijectionTypeC

Specific implementation of the bijection from KR tableaux to rigged configurations for type $A_{2n}^{(2)}$.

This inherits from type $C_n^{(1)}$ because we use the same methods in some places.

next_state(*val*)

Build the next state for type $A_{2n}^{(2)}$.

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 4, 2], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_A2_even import KRTToRCBijectionTypeA
sage: bijection = KRTToRCBijectionTypeA2Even(KRT(pathlist=[[-1,-2]]))
sage: bijection.cur_path.insert(0, [])
sage: bijection.cur_dims.insert(0, [0, 1])
sage: bijection.cur_path[0].insert(0, [-2])
sage: bijection.next_state(-2)
```

class sage.combinat.rigged_configurations.bij_type_A2_even.**RCToKRTBijectionTypeA2Even**(*RC_eleme*

Bases: sage.combinat.rigged_configurations.bij_type_C.RCToKRTBijectionTypeC

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $A_{2n}^{(2)}$.

next_state(*height*)

Build the next state for type $A_{2n}^{(2)}$.

TESTS:

```
sage: RC = RiggedConfigurations(['A', 4, 2], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_A2_even import RCToKRTBijectionTypeA
sage: bijection = RCToKRTBijectionTypeA2Even(RC(partition_list=[[2],[2,2]]))
sage: bijection.next_state(1)
-1
```

5.1.175 Bijection classes for type $A^{(2)}_{2n-1}$.

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $A^{(2)}_{2n-1}$.

AUTHORS:

- Travis Scrimshaw (2012-12-21): Initial version

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 5, 2], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_A2_odd import KRTToRCBijectionTypeA2Odd
sage: bijection = KRTToRCBijectionTypeA2Odd(KRT(pathlist=[[-1,2]]))
sage: TestSuite(bijection).run()
sage: RC = RiggedConfigurations(['A', 5, 2], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_A2_odd import RCToKRTBijectionTypeA2Odd
sage: bijection = RCToKRTBijectionTypeA2Odd(RC(partition_list=[[],[],[]]))
sage: TestSuite(bijection).run()
```

class sage.combinat.rigged_configurations.bij_type_A2_odd.**KRTToRCBijectionTypeA2Odd**(*tp_krt*)

Bases: sage.combinat.rigged_configurations.bij_type_A.KRTToRCBijectionTypeA

Specific implementation of the bijection from KR tableaux to rigged configurations for type $A^{(2)}_{2n-1}$.

This inherits from type $A^{(1)}_n$ because we use the same methods in some places.

next_state(*val*)

Build the next state for type $A^{(2)}_{2n-1}$.

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 5, 2], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_A2_odd import KRTToRCBijectionTypeA2
sage: bijection = KRTToRCBijectionTypeA2Odd(KRT(pathlist=[[-2,3]]))
sage: bijection.cur_path.insert(0, [])
sage: bijection.cur_dims.insert(0, [0, 1])
sage: bijection.cur_path[0].insert(0, [3])
sage: bijection.next_state(3)
```

class sage.combinat.rigged_configurations.bij_type_A2_odd.**RCToKRTBijectionTypeA2Odd**(*RC_element*)

Bases: sage.combinat.rigged_configurations.bij_type_A.RCToKRTBijectionTypeA

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $A^{(2)}_{2n-1}$.

next_state(*height*)

Build the next state for type $A^{(2)}_{2n-1}$.

TESTS:

```
sage: RC = RiggedConfigurations(['A', 5, 2], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_A2_odd import RCToKRTBijectionTypeA2
sage: bijection = RCToKRTBijectionTypeA2Odd(RC(partition_list=[[1],[2,1],[2]]))
```

```
sage: bijection.next_state(0)
-2
```

5.1.176 Bijection classes for type $B_n^{(1)}$.

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $B_n^{(1)}$.

AUTHORS:

- Travis Scrimshaw (2012-12-21): Initial version

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['B', 3, 1], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_B import KRTToRCBijectionTypeB
sage: bijection = KRTToRCBijectionTypeB(KRT(pathlist=[[-1,2]]))
sage: TestSuite(bijection).run()
sage: RC = RiggedConfigurations(['B', 3, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_B import RCToKRTBijectionTypeB
sage: bijection = RCToKRTBijectionTypeB(RC(partition_list=[[],[],[]]))
sage: TestSuite(bijection).run()
```

class sage.combinat.rigged_configurations.bij_type_B.**KRTToRCBijectionTypeB**(*tp_krt*)

Bases: sage.combinat.rigged_configurations.bij_type_C.KRTToRCBijectionTypeC

Specific implementation of the bijection from KR tableaux to rigged configurations for type $B_n^{(1)}$.

next_state(*val*)

Build the next state for type $B_n^{(1)}$.

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['B', 3, 1], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_B import KRTToRCBijectionTypeB
sage: bijection = KRTToRCBijectionTypeB(KRT(pathlist=[[-1,2]]))
sage: bijection.cur_path.insert(0, [])
sage: bijection.cur_dims.insert(0, [0, 1])
sage: bijection.cur_path[0].insert(0, [3])
sage: bijection.next_state(3)
```

other_outcome(*rc*, *pos_val*, *width_n*)

Do the other case (QS) possibility.

This arises from the ambiguity when we found a singular string at the max width in $\nu^{(n)}$. We had first attempted case (S), and if that resulted in an invalid rigged configuration, we now finish the bijection using case (QS).

EXAMPLES:

```
sage: RC = RiggedConfigurations(['B',3,1], [[2,1],[1,2]])
sage: rc = RC(partition_list=[[2,1], [2,1,1], [5,1]])
sage: t = rc.to_tensor_product_of_kirillov_reshetikhin_tableaux()
sage: t.to_rigged_configuration() == rc # indirect doctest
True
```

run(*verbose=False*)

Run the bijection from a tensor product of KR tableaux to a rigged configuration.

INPUT:

- tp_krt – A tensor product of KR tableaux
- verbose – (Default: False) Display each step in the bijection

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.bij_type_B import KRTToRCBijectionTypeB
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['B', 3, 1], [[2, 1]])
sage: KRTToRCBijectionTypeB(KRT(pathlist=[[0,3]])).run()

0[ ]0

-1[ ]-1
-1[ ]-1

0[]0

sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['B', 3, 1], [[3, 1]])
sage: KRTToRCBijectionTypeB(KRT(pathlist=[[-2,3,1]])).run()

(/)

-1[ ]-1

0[]0
```

TESTS:

Check that trac ticket #19384 is fixed:

```
sage: RC = RiggedConfigurations(['B',3,1], [[3,1],[3,1]])
sage: RC._test_bijection()
sage: RC = RiggedConfigurations(['B',3,1], [[1,1],[3,1],[1,1]])
sage: RC._test_bijection()
```

class sage.combinat.rigged_configurations.bij_type_B.**RCToKRTBijectionTypeB**(*RC_element*)

Bases: sage.combinat.rigged_configurations.bij_type_C.RCToKRTBijectionTypeC

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $B_n^{(1)}$.

next_state(*height*)

Build the next state for type $B_n^{(1)}$.

TESTS:

```
sage: RC = RiggedConfigurations(['B', 3, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_B import RCToKRTBijectionTypeB
sage: bijection = RCToKRTBijectionTypeB(RC(partition_list=[[1],[1,1],[1]]))
sage: bijection.next_state(0)
0
```

run(*verbose=False*, *build_graph=False*)

Run the bijection from rigged configurations to tensor product of KR tableaux for type $B_n^{(1)}$.

INPUT:

- verbose – (default: False) display each step in the bijection
- build_graph – (default: False) build the graph of each step of the bijection

EXAMPLES:

```
sage: RC = RiggedConfigurations(['B', 3, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_B import RCToKRTBijectionTypeB
sage: RCToKRTBijectionTypeB(RC(partition_list=[[1],[1,1],[1]])).run()
[[3], [0]]

sage: RC = RiggedConfigurations(['B', 3, 1], [[3, 1]])
sage: x = RC(partition_list=[[],[1],[1]])
sage: RCToKRTBijectionTypeB(x).run()
[[1], [3], [-2]]
sage: bij = RCToKRTBijectionTypeB(x)
sage: bij.run(build_graph=True)
[[1], [3], [-2]]
sage: bij._graph
Digraph on 6 vertices
```

5.1.177 Bijection classes for type $C_n^{(1)}$.

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $C_n^{(1)}$.

AUTHORS:

- Travis Scrimshaw (2012-12-21): Initial version

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['C', 3, 1], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_C import KRTToRCBijectionTypeC
sage: bijection = KRTToRCBijectionTypeC(KRT(pathlist=[[-1,2]]))
sage: TestSuite(bijection).run()
sage: RC = RiggedConfigurations(['C', 3, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_C import RCToKRTBijectionTypeC
sage: bijection = RCToKRTBijectionTypeC(RC(partition_list=[[],[],[]]))
sage: TestSuite(bijection).run()
```

class sage.combinat.rigged_configurations.bij_type_C.**KRTToRCBijectionTypeC**(*tp_krt*)

Bases: sage.combinat.rigged_configurations.bij_type_A.KRTToRCBijectionTypeA

Specific implementation of the bijection from KR tableaux to rigged configurations for type $C_n^{(1)}$.

This inherits from type $A_n^{(1)}$ because we use the same methods in some places.

next_state(*val*)

Build the next state for type $C_n^{(1)}$.

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['C', 3, 1], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_C import KRTToRCBijectionTypeC
sage: bijection = KRTToRCBijectionTypeC(KRT(pathlist=[[-1,2]]))
sage: bijection.cur_path.insert(0, [])
sage: bijection.cur_dims.insert(0, [0, 1])
sage: bijection.cur_path[0].insert(0, [2])
sage: bijection.next_state(2)
```

class sage.combinat.rigged_configurations.bij_type_C.**RCToKRTBijectionTypeC**(*RC_element*)

Bases: sage.combinat.rigged_configurations.bij_type_A.RCToKRTBijectionTypeA

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $C_n^{(1)}$.

next_state(*height*)

Build the next state for type $C_n^{(1)}$.

TESTS:

```
sage: RC = RiggedConfigurations(['C', 3, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_C import RCToKRTBijectionTypeC
sage: bijection = RCToKRTBijectionTypeC(RC(partition_list=[[2],[2],[1]]))
sage: bijection.next_state(0)
-1
```

5.1.178 Bijection classes for type $D_n^{(1)}$

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $D_n^{(1)}$.

AUTHORS:

- Travis Scrimshaw (2011-04-15): Initial version

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_D import KRTToRCBijectionTypeD
sage: bijection = KRTToRCBijectionTypeD(KRT(pathlist=[[3, 2]]))
sage: TestSuite(bijection).run()
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_D import RCToKRTBijectionTypeD
sage: bijection = RCToKRTBijectionTypeD(RC(partition_list=[[],[],[],[]]))
sage: TestSuite(bijection).run()
```

class sage.combinat.rigged_configurations.bij_type_D.**KRTToRCBijectionTypeD**(*tp_krt*)

Bases: sage.combinat.rigged_configurations.bij_type_A.KRTToRCBijectionTypeA

Specific implementation of the bijection from KR tableaux to rigged configurations for type $D_n^{(1)}$.

This inherits from type $A_n^{(1)}$ because we use the same methods in some places.

doubling_map()

Perform the doubling map of the rigged configuration at the current state of the bijection.

This is the map $B(\Lambda) \hookrightarrow B(2\Lambda)$ which doubles each of the rigged partitions and updates the vacancy numbers accordingly.

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1], [[4,1]])
sage: from sage.combinat.rigged_configurations.bij_type_D import KRTToRCBijectionTypeD
sage: bijection = KRTToRCBijectionTypeD(KRT(pathlist=[[-1,4,3,2]]))
sage: bijection.cur_path.insert(0, [])
sage: bijection.cur_dims.insert(0, [0, 1])
sage: bijection.cur_path[0].insert(0, [2])
sage: bijection.next_state(2)
sage: bijection.ret_rig_con

-2[ ]-2

(/)

(/)
```

```
(/)

sage: bijection.cur_dims
[[0, 1]]
sage: bijection.doubling_map()
sage: bijection.ret_rig_con

-4[ ][ ]-4

(/)

(/)

(/)

sage: bijection.cur_dims
[[0, 2]]
```

halving_map()

Perform the halving map of the rigged configuration at the current state of the bijection.

This is the inverse map to $B(\Lambda) \hookrightarrow B(2\Lambda)$ which halves each of the rigged partitions and updates the vacancy numbers accordingly.

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1], [[4,1]])
sage: from sage.combinat.rigged_configurations.bij_type_D import KRTToRCBijectionTypeD
sage: bijection = KRTToRCBijectionTypeD(KRT(pathlist=[[-1,4,3,2]]))
sage: bijection.cur_path.insert(0, [])
sage: bijection.cur_dims.insert(0, [0, 1])
sage: bijection.cur_path[0].insert(0, [2])
sage: bijection.next_state(2)
sage: test = bijection.ret_rig_con
sage: bijection.doubling_map()
sage: bijection.halving_map()
sage: test == bijection.ret_rig_con
True
```

next_state(*val*)

Build the next state for type $D_n^{(1)}$.

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_D import KRTToRCBijectionTypeD
sage: bijection = KRTToRCBijectionTypeD(KRT(pathlist=[[5,3]]))
sage: bijection.cur_path.insert(0, [])
sage: bijection.cur_dims.insert(0, [0, 1])
sage: bijection.cur_path[0].insert(0, [3])
sage: bijection.next_state(3)
```

run(*verbose=False*)

Run the bijection from a tensor product of KR tableaux to a rigged configuration for type $D_n^{(1)}$.

INPUT:

- `tp_krt` – A tensor product of KR tableaux
- `verbose` – (Default: `False`) Display each step in the bijection

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_D import KRTToRCBijectionTypeD
sage: KRTToRCBijectionTypeD(KRT(pathlist=[[-3,2]])).run()

-1[ ]-1

2[ ]2

-1[ ]-1

-1[ ]-1
```

class sage.combinat.rigged_configurations.bij_type_D.**RCToKRTBijectionTypeD**(*RC_element*)

Bases: sage.combinat.rigged_configurations.bij_type_A.RCToKRTBijectionTypeA

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $D_n^{(1)}$.

doubling_map()

Perform the doubling map of the rigged configuration at the current state of the bijection.

This is the map $B(\Lambda) \hookrightarrow B(2\Lambda)$ which doubles each of the rigged partitions and updates the vacancy numbers accordingly.

TESTS:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[4, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_D import RCToKRTBijectionTypeD
sage: bijection = RCToKRTBijectionTypeD(RC(partition_list=[[],[],[],[1]]))
sage: bijection.cur_partitions
[(/)
 , (/)
 , (/)
 , -1[ ]-1
 ]
sage: bijection.doubling_map()
sage: bijection.cur_partitions
[(/)
 , (/)
 , (/)
 , -2[ ][ ]-2
 ]
```

halving_map()

Perform the halving map of the rigged configuration at the current state of the bijection.

This is the inverse map to $B(\Lambda) \hookrightarrow B(2\Lambda)$ which halves each of the rigged partitions and updates the vacancy numbers accordingly.

TESTS:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[4, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_D import RCToKRTBijectionTypeD
sage: bijection = RCToKRTBijectionTypeD(RC(partition_list=[[],[],[],[1]]))
sage: test = bijection.cur_partitions
sage: bijection.doubling_map()
sage: bijection.halving_map()
sage: test == bijection.cur_partitions
True
```

next_state(*height*)

Build the next state for type $D_n^{(1)}$.

TESTS:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_D import RCToKRTBijectionTypeD
sage: bijection = RCToKRTBijectionTypeD(RC(partition_list=[[],[1,1],[1],[1]]))
sage: bijection.next_state(0)
1
```

run(*verbose=False*, *build_graph=False*)

Run the bijection from rigged configurations to tensor product of KR tableaux for type $D_n^{(1)}$.

INPUT:

- `verbose` – (default: `False`) display each step in the bijection
- `build_graph` – (default: `False`) build the graph of each step of the bijection

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 1]])
sage: x = RC(partition_list=[[1],[1],[1],[1]])
sage: from sage.combinat.rigged_configurations.bij_type_D import RCToKRTBijectionTypeD
sage: RCToKRTBijectionTypeD(x).run()
[[2], [-3]]
sage: bij = RCToKRTBijectionTypeD(x)
sage: bij.run(build_graph=True)
[[2], [-3]]
sage: bij._graph
Digraph on 3 vertices
```

5.1.179 Bijection classes for type $D_{n+1}^{(2)}$.

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $D_{n+1}^{(2)}$.

AUTHORS:

- Travis Scrimshaw (2011-04-15): Initial version

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 2], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_D_twisted import KRTToRCBijectionTypeDTwisted
sage: bijection = KRTToRCBijectionTypeDTwisted(KRT(pathlist=[[-1,2]]))
sage: TestSuite(bijection).run()
sage: RC = RiggedConfigurations(['D', 4, 2], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_D_twisted import RCToKRTBijectionTypeDTwisted
sage: bijection = RCToKRTBijectionTypeDTwisted(RC())
sage: TestSuite(bijection).run()
```

class sage.combinat.rigged_configurations.bij_type_D_twisted.**KRTToRCBijectionTypeDTwisted**(*tp_k*

Bases: sage.combinat.rigged_configurations.bij_type_D.KRTToRCBijectionTypeD, sage.combinat.rigged_configurations.bij_type_A2_even.KRTToRCBijectionTypeA2Even

Specific implementation of the bijection from KR tableaux to rigged configurations for type $D_{n+1}^{(2)}$.

This inherits from type $C_n^{(1)}$ and $D_n^{(1)}$ because we use the same methods in some places.

next_state(*val*)

Build the next state for type $D_{n+1}^{(2)}$.

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 2], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_D_twisted import KRTToRCBijectionTyp
sage: bijection = KRTToRCBijectionTypeDTwisted(KRT(pathlist=[[-1,2]]))
sage: bijection.cur_path.insert(0, [])
sage: bijection.cur_dims.insert(0, [0, 1])
sage: bijection.cur_path[0].insert(0, [2])
sage: bijection.next_state(2)
```

run(*verbose=False*)

Run the bijection from a tensor product of KR tableaux to a rigged configuration for type $D_{n+1}^{(2)}$.

INPUT:

- `tp_krt` – A tensor product of KR tableaux
- `verbose` – (Default: `False`) Display each step in the bijection

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 2], [[3,1]])
sage: from sage.combinat.rigged_configurations.bij_type_D_twisted import KRTToRCBijectionTyp
sage: KRTToRCBijectionTypeDTwisted(KRT(pathlist=[[-1,3,2]])).run()

-1[ ]-1

0[ ]0

1[ ]1
```

class sage.combinat.rigged_configurations.bij_type_D_twisted.**RCToKRTBijectionTypeDTwisted**(*RC_*

Bases: sage.combinat.rigged_configurations.bij_type_D.RCToKRTBijectionTypeD, sage.combinat.rigged_configurations.bij_type_A2_even.RCToKRTBijectionTypeA2Even

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $D_{n+1}^{(2)}$.

next_state(*height*)

Build the next state for type $D_{n+1}^{(2)}$.

TESTS:

```
sage: RC = RiggedConfigurations(['D', 4, 2], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_D_twisted import RCToKRTBijectionTyp
sage: bijection = RCToKRTBijectionTypeDTwisted(RC(partition_list=[[2],[2,2],[2,2]]))
sage: bijection.next_state(0)
-1
```

run(*verbose=False*, *build_graph=False*)

Run the bijection from rigged configurations to tensor product of KR tableaux for type $D_{n+1}^{(2)}$.

INPUT:

- `verbose` – (default: `False`) display each step in the bijection
- `build_graph` – (default: `False`) build the graph of each step of the bijection

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D', 4, 2], [[3, 1]])
sage: x = RC(partition_list=[[],[1],[1]])
sage: from sage.combinat.rigged_configurations.bij_type_D_twisted import RCToKRTBijectionTyp
sage: RCToKRTBijectionTypeDTwisted(x).run()
[[1], [3], [-2]]
sage: bij = RCToKRTBijectionTypeDTwisted(x)
sage: bij.run(build_graph=True)
[[1], [3], [-2]]
sage: bij._graph
Digraph on 6 vertices
```

5.1.180 Bijection classes for type $D_4^{(3)}$.

Part of the (internal) classes which runs the bijection between rigged configurations and KR tableaux of type $D_4^{(3)}$.

AUTHORS:

- Travis Scrimshaw (2014-09-10): Initial version

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 3], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_D_tri import KRTToRCBijectionTypeDTri
sage: bijection = KRTToRCBijectionTypeDTri(KRT(pathlist=[[-1,2]]))
sage: TestSuite(bijection).run()
sage: RC = RiggedConfigurations(['D', 4, 3], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_D_tri import RCToKRTBijectionTypeDTri
sage: bijection = RCToKRTBijectionTypeDTri(RC(partition_list=[[],[]]))
sage: TestSuite(bijection).run()
```

class sage.combinat.rigged_configurations.bij_type_D_tri.**KRTToRCBijectionTypeDTri**(*tp_krt*)

Bases: sage.combinat.rigged_configurations.bij_type_A.KRTToRCBijectionTypeA

Specific implementation of the bijection from KR tableaux to rigged configurations for type $D_4^{(3)}$.

This inherits from type $A_n^{(1)}$ because we use the same methods in some places.

next_state(*val*)

Build the next state for type $D_4^{(3)}$.

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 3], [[2,1]])
sage: from sage.combinat.rigged_configurations.bij_type_D_tri import KRTToRCBijectionTypeDTr
sage: bijection = KRTToRCBijectionTypeDTri(KRT(pathlist=[[-1,2]]))
sage: bijection.cur_path.insert(0, [])
sage: bijection.cur_dims.insert(0, [0, 1])
sage: bijection.cur_path[0].insert(0, [2])
sage: bijection.next_state(2)
```

class sage.combinat.rigged_configurations.bij_type_D_tri.**RCToKRTBijectionTypeDTri**(*RC_element*)

Bases: sage.combinat.rigged_configurations.bij_type_A.RCToKRTBijectionTypeA

Specific implementation of the bijection from rigged configurations to tensor products of KR tableaux for type $D_4^{(3)}$.

next_state(*height*)

Build the next state for type $D_4^{(3)}$.

TESTS:

```
sage: RC = RiggedConfigurations(['D', 4, 3], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bij_type_D_tri import RCToKRTBijectionTypeDTr
sage: bijection = RCToKRTBijectionTypeDTri(RC(partition_list=[[3],[2]]))
sage: bijection.next_state(1)
-3
```

5.1.181 Bijection between rigged configurations and KR tableaux

Functions which are big switch statements to create the bijection class of the correct type.

AUTHORS:

- Travis Scrimshaw (2011-04-15): Initial version
- Travis Scrimshaw (2012-12-21): Added all non-exceptional bijection types
- Travis Scrimshaw (2014-09-10): Added type $D_4^{(3)}$

sage.combinat.rigged_configurations.bijection.**KRTToRCBijection**(*tp_krt*)
Return the correct KR tableaux to rigged configuration bijection helper class.

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 4, 1], [[2,1]])
sage: from sage.combinat.rigged_configurations.bijection import KRTToRCBijection
sage: bijection = KRTToRCBijection(KRT(pathlist=[[5,2]]))
```

sage.combinat.rigged_configurations.bijection.**RCToKRTBijection**(*rigged_configuration_elt*)
Return the correct rigged configuration to KR tableaux bijection helper class.

TESTS:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 1]])
sage: from sage.combinat.rigged_configurations.bijection import RCToKRTBijection
sage: bijection = RCToKRTBijection(RC(partition_list=[[1],[1],[1],[1]]))
```

5.1.182 Kleber Trees

A Kleber tree is a tree of weights generated by Kleber's algorithm [Kleber1]. The nodes correspond to the weights in the positive Weyl chamber obtained by subtracting a (non-zero) positive root. The edges are labeled by the coefficients of the roots of the difference.

AUTHORS:

- Travis Scrimshaw (2011-05-03): Initial version
- Travis Scrimshaw (2013-02-13): Added support for virtual trees and improved LaTeX output

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KleberTree(['A', 3, 1], [[3,2], [2,1], [1,1], [1,1]])
Kleber tree of Cartan type ['A', 3, 1] and B = ((3, 2), (2, 1), (1, 1), (1, 1))
sage: KleberTree(['D', 4, 1], [[2,2]])
Kleber tree of Cartan type ['D', 4, 1] and B = ((2, 2),)
```

TESTS:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['A', 3, 1], [[3,2], [2,1], [1,1], [1,1]])
sage: sorted((x.weight.to_vector(), x.up_root.to_vector()) for x in KT.list())
[((0, 0, 2), (1, 1, 0)),
 ((0, 0, 2), (2, 2, 1)),
 ((0, 1, 0), (0, 0, 1)),
 ((0, 1, 0), (1, 1, 1)),
 ((0, 2, 2), (1, 0, 0)),
 ((1, 0, 3), (1, 1, 0)),
 ((1, 1, 1), (1, 1, 1)),
 ((2, 0, 0), (0, 1, 1)),
 ((2, 1, 2), (0, 0, 0)),
 ((3, 0, 1), (0, 1, 1))]

sage: KT = KleberTree(['A', 7, 1], [[3,2], [2,1], [1,1]])
sage: KT
Kleber tree of Cartan type ['A', 7, 1] and B = ((3, 2), (2, 1), (1, 1))
sage: sorted((x.weight.to_vector(), x.up_root.to_vector()) for x in KT.list())
[((0, 0, 0, 1, 1, 0, 0), (1, 1, 1, 0, 0, 0, 0)),
 ((0, 0, 1, 0, 0, 1, 0), (2, 3, 3, 2, 1, 0, 0)),
 ((0, 0, 3, 0, 0, 0, 0), (1, 1, 0, 0, 0, 0, 0)),
 ((0, 1, 1, 1, 0, 0, 0), (1, 1, 1, 0, 0, 0, 0)),
 ((1, 0, 0, 2, 0, 0, 0), (0, 1, 1, 0, 0, 0, 0)),
 ((1, 0, 1, 0, 1, 0, 0), (1, 2, 2, 1, 0, 0, 0)),
 ((1, 1, 2, 0, 0, 0, 0), (0, 0, 0, 0, 0, 0, 0)),
 ((2, 0, 1, 1, 0, 0, 0), (0, 1, 1, 0, 0, 0, 0))]
```

class sage.combinat.rigged_configurations.kleber_tree.**KleberTree**(*cartan_type*, *B*, *classical_ct*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The tree that is generated by Kleber's algorithm.

A Kleber tree is a tree of weights generated by Kleber's algorithm [Kleber1]. It is used to generate the set of all admissible rigged configurations for the simply-laced affine types $A_n^{(1)}$, $D_n^{(1)}$, $E_6^{(1)}$, $E_7^{(1)}$, and $E_8^{(1)}$.

See also:

There is a modified version for non-simply-laced affine types at VirtualKleberTree.

The nodes correspond to the weights in the positive Weyl chamber obtained by subtracting a (non-zero) positive root. The edges are labeled by the coefficients of the roots, and X is a child of Y if Y is the root else if the edge label of Y to its parent Z is greater (in every component) than the label from X to Y.

For a Kleber tree, one needs to specify an affine (simply-laced) Cartan type and a sequence of pairs (r, s), where s is any positive integer and r is a node in the Dynkin diagram. Each (r, s) can be viewed as a rectangle of width s and height r.

INPUT:

- cartan_type – an affine simply-laced Cartan type
- B – a list of dimensions of rectangles by $[r, c]$ where r is the number of rows and c is the number of columns

REFERENCES:

EXAMPLES:

Simply-laced example:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['A', 3, 1], [[3,2], [1,1]])
sage: KT.list()
[Kleber tree node with weight [1, 0, 2] and upwards edge root [0, 0, 0],
 Kleber tree node with weight [0, 0, 1] and upwards edge root [1, 1, 1]]
sage: KT = KleberTree(['A', 3, 1], [[3,2], [2,1], [1,1], [1,1]])
sage: KT.cardinality()
10
sage: KT = KleberTree(['D', 4, 1], [[2,2]])
sage: KT.cardinality()
3
sage: KT = KleberTree(['D', 4, 1], [[4,5]])
sage: KT.cardinality()
1
```

From [Kleber2]:

```
sage: KT = KleberTree(['E', 6, 1], [[4, 2]])  # long time (9s on sage.math, 2012)
sage: KT.cardinality()  # long time
12
```

We check that relabelled types work (trac ticket #16876):

```
sage: ct = CartanType(['A',3,1]).relabel(lambda x: x+2)
sage: kt = KleberTree(ct, [[3,1],[5,1]])
sage: list(kt)
[Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 0, 0],
 Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 1, 1]]
sage: kt = KleberTree(['A',3,1], [[1,1],[3,1]])
sage: list(kt)
[Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 0, 0],
 Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 1, 1]]
```

Element
alias of KleberTreeNode

breadth_first_iter()
Iterate over all nodes in the tree following a breadth-first traversal.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['A', 3, 1], [[2, 2], [2, 3]])
sage: for x in KT.breadth_first_iter(): x
Kleber tree node with weight [0, 5, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 3, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 3, 0] and upwards edge root [1, 2, 1]
Kleber tree node with weight [2, 1, 2] and upwards edge root [0, 1, 0]
Kleber tree node with weight [1, 1, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 1, 0] and upwards edge root [1, 2, 1]
```

cartan_type()
Return the Cartan type of this Kleber tree.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['A', 3, 1], [[1,1]])
sage: KT.cartan_type()
['A', 3, 1]
```

depth_first_iter()

Iterate (recursively) over the nodes in the tree following a depth-first traversal.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['A', 3, 1], [[2, 2], [2, 3]])
sage: for x in KT.depth_first_iter(): x
Kleber tree node with weight [0, 5, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 3, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [2, 1, 2] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 3, 0] and upwards edge root [1, 2, 1]
Kleber tree node with weight [1, 1, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 1, 0] and upwards edge root [1, 2, 1]
```

digraph()

Return a DiGraph representation of this Kleber tree.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['D', 4, 1], [[2, 2]])
sage: KT.digraph()  # optional - dot2tex graphviz
Digraph on 3 vertices
```

latex_options(***options*)

Return the current latex options if no arguments are passed, otherwise set the corresponding latex option.

OPTIONS:

- `hspace` – (default: 2.5) the horizontal spacing of the tree nodes
- `vspace` – (default: `x`) the vertical spacing of the tree nodes, here `x` is the minimum of -2.5 or $-.75n$ where n is the rank of the classical type
- `edge_labels` – (default: `True`) display edge labels
- `use_vector_notation` – (default: `False`) display edge labels using vector notation instead of a linear combination

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['D', 3, 1], [[2,1], [2,1]])
sage: KT.latex_options(vspace=-4, use_vector_notation=True)
sage: sorted(KT.latex_options().items())
[('edge_labels', True), ('hspace', 2.5), ('use_vector_notation', True), ('vspace', -4)]
```

plot(***options*)

Return the plot of self as a directed graph.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['D', 4, 1], [[2, 2]])
sage: print(KT.plot())  # optional - dot2tex graphviz
Graphics object consisting of 8 graphics primitives
```

class sage.combinat.rigged_configurations.kleber_tree.**KleberTreeNode**(*parent_obj*, *node_weight*, *dominant_root*, *parent_node=None*)

Bases: sage.structure.element.Element

A node in the Kleber tree.

This class is meant to be used internally by the Kleber tree class and should not be created directly by the user.

For more on the Kleber tree and the nodes, see KleberTree.

The dominating root is the up_root which is the difference between the parent node's weight and this node's weight.

INPUT:

- parent_obj – The parent object of this element
- node_weight – The weight of this node
- dominant_root – The dominating root
- parent_node – (default:None) The parent node of this node

depth()

Return the depth of this node in the tree.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: RS = RootSystem(['A', 2])
sage: WS = RS.weight_space()
sage: R = RS.root_space()
sage: KT = KleberTree(['A', 2, 1], [[1,1]])
sage: n = KT(WS.sum_of_terms([(1,5), (2,2)]), R.zero())
sage: n.depth
0
sage: n2 = KT(WS.sum_of_terms([(1,5), (2,2)]), R.zero(), n)
sage: n2.depth
1
```

multiplicity()

Return the multiplicity of self.

The multiplicity of a node x of depth d weight λ in a simply-laced Kleber tree is equal to:

$$\prod_{i>0} \prod_{a \in \overline{I}} \binom{p_i^{(a)} + m_i^{(a)}}{p_i^{(a)}}$$

Recall that

$$m_i^{(a)} = \left(\lambda^{(i-1)} - 2\lambda^{(i)} + \lambda^{(i+1)} \mid \overline{\Lambda}_a \right),$$
$$p_i^{(a)} = \left(\alpha_a \mid \lambda^{(i)} \right) - \sum_{j>i} (j-i) L_j^{(a)},$$

where $\lambda^{(i)}$ is the weight node at depth i in the path to x from the root and we set $\lambda^{(j)} = \lambda$ for all $j \geq d$.

Note that $m_i^{(a)} = 0$ for all $i > d$.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import KleberTree
sage: KT = KleberTree(['A',3,1], [[3,2],[2,1],[1,1],[1,1]])
sage: for x in KT: x, x.multiplicity()
(Kleber tree node with weight [2, 1, 2] and upwards edge root [0, 0, 0], 1)
(Kleber tree node with weight [0, 2, 2] and upwards edge root [1, 0, 0], 1)
(Kleber tree node with weight [1, 0, 3] and upwards edge root [1, 1, 0], 2)
(Kleber tree node with weight [1, 1, 1] and upwards edge root [1, 1, 1], 4)
(Kleber tree node with weight [0, 0, 2] and upwards edge root [2, 2, 1], 2)
(Kleber tree node with weight [3, 0, 1] and upwards edge root [0, 1, 1], 1)
(Kleber tree node with weight [2, 0, 0] and upwards edge root [0, 1, 1], 2)
(Kleber tree node with weight [0, 0, 2] and upwards edge root [1, 1, 0], 1)
(Kleber tree node with weight [0, 1, 0] and upwards edge root [1, 1, 1], 2)
(Kleber tree node with weight [0, 1, 0] and upwards edge root [0, 0, 1], 1)
```

TESTS:

We check that trac ticket #16057 is fixed:

```
sage: RC = RiggedConfigurations(['D',4,1], [[1,3],[3,3],[4,3]])
sage: sum(x.multiplicity() for x in RC.kleber_tree()) == len(RC.module_generators)
True
```

class sage.combinat.rigged_configurations.kleber_tree.**KleberTreeTypeA2Even**(*cartan_type*, *B*)

Bases: sage.combinat.rigged_configurations.kleber_tree.VirtualKleberTree

Kleber tree for types $A_{2n}^{(2)}$ and $A_{2n}^{(2)\dagger}$.

Note that here for $A_{2n}^{(2)}$ we use $\tilde{\gamma}_a$ in place of γ_a in constructing the virtual Kleber tree, and so we end up selecting all nodes since $\tilde{\gamma}_a = 1$ for all $a \in \overline{I}$. For type $A_{2n}^{(2)\dagger}$, we have $\gamma_a = 1$ for all $a \in \overline{I}$.

See also:

VirtualKleberTree

breadth_first_iter(*all_nodes=False*)

Iterate over all nodes in the tree following a breadth-first traversal.

INPUT:

- `all_nodes` – (default: `False`) if `True`, output all nodes in the tree

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import VirtualKleberTree
sage: KT = VirtualKleberTree(['A', 4, 2], [[2,1]])
sage: for x in KT.breadth_first_iter(): x
Kleber tree node with weight [0, 2, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 2, 1]
sage: for x in KT.breadth_first_iter(True): x
Kleber tree node with weight [0, 2, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 2, 1]
```

depth_first_iter(*all_nodes=False*)

Iterate (recursively) over the nodes in the tree following a depth-first traversal.

INPUT:

- `all_nodes` – (default: `False`) if `True`, output all nodes in the tree

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import VirtualKleberTree
sage: KT = VirtualKleberTree(['A', 4, 2], [[2,1]])
sage: for x in KT.depth_first_iter(): x
Kleber tree node with weight [0, 2, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 2, 1]
sage: for x in KT.depth_first_iter(True): x
Kleber tree node with weight [0, 2, 0] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
Kleber tree node with weight [0, 0, 0] and upwards edge root [1, 2, 1]
```

class sage.combinat.rigged_configurations.kleber_tree.**VirtualKleberTree**(*cartan_type*, *B*)

Bases: sage.combinat.rigged_configurations.kleber_tree.KleberTree

A virtual Kleber tree.

We can use a modified version of the Kleber algorithm called the virtual Kleber algorithm [OSS03] to compute all admissible rigged configurations for non-simply-laced types. This uses the following embeddings into the simply-laced types:

$$\begin{aligned} C_n^{(1)}, A_{2n}^{(2)}, A_{2n}^{(2)\dagger}, D_{n+1}^{(2)} &\hookrightarrow A_{2n-1}^{(1)} \\ A_{2n-1}^{(2)}, B_n^{(1)} &\hookrightarrow D_{n+1}^{(1)} \\ E_6^{(2)}, F_4^{(1)} &\hookrightarrow E_6^{(1)} \\ D_4^{(3)}, G_2^{(1)} &\hookrightarrow D_4^{(1)} \end{aligned}$$

One then selects the subset of admissible nodes which are translates of the virtual requirements. In the graph, the selected nodes are indicated by brackets [].

Note: Because these are virtual nodes, all information is given in the corresponding simply-laced type.

See also:

For more on the Kleber algorithm, see KleberTree.

REFERENCES:

INPUT:

- cartan_type – an affine non-simply-laced Cartan type
- B – a list of dimensions of rectangles by $[r, c]$ where r is the number of rows and c is the number of columns

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import VirtualKleberTree
sage: KT = VirtualKleberTree(['C', 4, 1], [[2,2]])
sage: KT.cardinality()
3
sage: KT.base_tree().cardinality()
6
sage: KT = VirtualKleberTree(['C', 4, 1], [[4,5]])
sage: KT.cardinality()
1
sage: KT = VirtualKleberTree(['D', 5, 2], [[2,1], [1,1]])
sage: KT.cardinality()
8
sage: KT = VirtualKleberTree(CartanType(['A', 4, 2]).dual(), [[1,1], [2,2]])
```

```
sage: KT.cardinality()
15
```

base_tree()

Return the underlying virtual Kleber tree associated to `self`.

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import VirtualKleberTree
sage: KT = VirtualKleberTree(['C', 4, 1], [[2,2]])
sage: KT.base_tree()
Kleber tree of Cartan type ['A', 7, 1] and B = ((2, 2), (6, 2))
```

breadth_first_iter(*all_nodes=False*)

Iterate over all nodes in the tree following a breadth-first traversal.

INPUT:

- `all_nodes` – (default: `False`) if `True`, output all nodes in the tree

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import VirtualKleberTree
sage: KT = VirtualKleberTree(['C', 2, 1], [[1,1], [2,1]])
sage: for x in KT.breadth_first_iter(): x
Kleber tree node with weight [1, 2, 1] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
sage: for x in KT.breadth_first_iter(True): x
Kleber tree node with weight [1, 2, 1] and upwards edge root [0, 0, 0]
Kleber tree node with weight [0, 2, 0] and upwards edge root [1, 1, 1]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
```

depth_first_iter(*all_nodes=False*)

Iterate (recursively) over the nodes in the tree following a depth-first traversal.

INPUT:

- `all_nodes` – (default: `False`) if `True`, output all nodes in the tree

EXAMPLES:

```
sage: from sage.combinat.rigged_configurations.kleber_tree import VirtualKleberTree
sage: KT = VirtualKleberTree(['C', 2, 1], [[1,1], [2,1]])
sage: for x in KT.depth_first_iter(): x
Kleber tree node with weight [1, 2, 1] and upwards edge root [0, 0, 0]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
sage: for x in KT.depth_first_iter(True): x
Kleber tree node with weight [1, 2, 1] and upwards edge root [0, 0, 0]
Kleber tree node with weight [0, 2, 0] and upwards edge root [1, 1, 1]
Kleber tree node with weight [1, 0, 1] and upwards edge root [0, 1, 0]
```

5.1.183 Kirillov-Reshetikhin Tableaux

Kirillov-Reshetikhin tableaux are rectangular tableaux with r rows and s columns that naturally arise under the bijection between rigged configurations and tableaux [RigConBijection]. They are in bijection with the elements of the Kirillov-Reshetikhin crystal $B^{r,s}$ under the (inverse) filling map [OSS13] [SchScr]. They do not have to satisfy the semistandard row or column restrictions. These tensor products are the result from the bijection from rigged configurations [RigConBijection].

For more information, see `KirillovReshetikhinTableaux` and `TensorProductOfKirillovReshetikhinTableaux`.

AUTHORS:

- Travis Scrimshaw (2012-01-03): Initial version
- Travis Scrimshaw (2012-11-14): Added bijection to KR crystals

REFERENCES:

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxBn**(*cartan_type*, *r*, *s*)

Bases: sage.combinat.rigged_configurations.kr_tableaux.KRTableauxTypeHorizonal

Kirillov-Reshetkhin tableaux $B^{n,s}$ of type $B_n^{(1)}$.

TESTS:

```
sage: KRT = crystals.KirillovReshetikhin(['B', 2, 1], 2, 3, model='KR')
sage: TestSuite(KRT).run()
```

Element

alias of KRTableauxSpinElement

from_kirillov_reshetikhin_crystal(*krc*)

Construct an element of `self` from the Kirillov-Reshetikhin crystal element `krc`.

EXAMPLES:

```
sage: KR = crystals.KirillovReshetikhin(['B',3,1], 3, 3, model='KR')
sage: C = crystals.KirillovReshetikhin(['B',3,1], 3, 3, model='KN')
sage: krc = C.module_generators[1].f_string([3,2,3,1,3,3]); krc
[++-, [[2], [0], [-3]]]
sage: KR.from_kirillov_reshetikhin_crystal(krc)
[[1, 1, 2], [2, 2, -3], [-3, -3, -1]]
```

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxDTwistedSpin**(*cartan_type*, *r*, *s*)

Bases: sage.combinat.rigged_configurations.kr_tableaux.KRTableauxRectangle

Kirillov-Reshetikhin tableaux $B^{r,s}$ of type $D_n^{(2)}$ with $r = n$.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 2], 1, 1, model='KR')
sage: KRT.cardinality()
8
sage: KRC = crystals.KirillovReshetikhin(['D', 4, 2], 1, 1, model='KN')
sage: KRT.cardinality() == KRC.cardinality()
True
```

Element

alias of KRTableauxSpinElement

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxRectangle**(*cartan_type*, *r*, *s*)

Bases: sage.combinat.rigged_configurations.kr_tableaux.KirillovReshetikhinTableaux

Kirillov-Reshetkhin tableaux $B^{r,s}$ whose module generator is a single $r \times s$ rectangle.

These are Kirillov-Reshetkhin tableaux $B^{r,s}$ of type:

- $A_n^{(1)}$ for all $1 \leq r \leq n$,
- $C_n^{(1)}$ when $r = n$.

TESTS:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 3, 1], 2, 2, model='KR')
sage: TestSuite(KRT).run()
sage: KRT = crystals.KirillovReshetikhin(['C', 3, 1], 3, 2, model='KR')
sage: TestSuite(KRT).run() # long time
```

from_kirillov_reshetikhin_crystal(*krc*)

Construct a KirillovReshetikhinTableauxElement.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 1, model='KR')
sage: C = crystals.KirillovReshetikhin(['A',4,1], 2, 1, model='KN')
sage: krc = C(4,3); krc
[[3], [4]]
sage: KRT.from_kirillov_reshetikhin_crystal(krc)
[[3], [4]]
```

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxSpin**(*cartan_type*, *r*, *s*)

Bases: sage.combinat.rigged_configurations.kr_tableaux.KRTableauxRectangle

Kirillov-Reshetikhin tableaux $B^{r,s}$ of type $D_n^{(1)}$ with $r = n, n-1$.

TESTS:

```
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 1], 3, 2, model='KR')
sage: TestSuite(KRT).run()
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 1], 4, 2, model='KR')
sage: TestSuite(KRT).run()
```

Element

alias of KRTableauxSpinElement

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxSpinElement**(*parent*, *list*, ***options*)

Bases: sage.combinat.rigged_configurations.kr_tableaux.KirillovReshetikhinTableauxElement

Kirillov-Reshetikhin tableau for spinors.

Here we are in the embedding $B(\Lambda_n) \hookrightarrow B(2\Lambda_n)$, so e_i and f_i act by e_i^2 and f_i^2 respectively for all $i \neq 0$. We do this so our columns are full width (as opposed to half width and/or uses a $\pm$ representation).

e(*i*)

Calculate the action of e_i on self.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 4, 1, model='KR')
sage: KRT(-1, -4, 3, 2).e(1)
[[1], [3], [-4], [-2]]
sage: KRT(-1, -4, 3, 2).e(3)
```

epsilon(*i*)

Compute ε_i of self.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 4, 1, model='KR')
sage: KRT(-1, -4, 3, 2).epsilon(1)
1
```

```
sage: KRT(-1, -4, 3, 2).epsilon(3)
0
```

f(*i*)

Calculate the action of f_i on `self`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 4, 1, model='KR')
sage: KRT(-1, -4, 3, 2).f(1)
sage: KRT(-1, -4, 3, 2).f(3)
[[2], [4], [-3], [-1]]
```

left_split()

Return the image of `self` under the left column splitting map.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 1], 4, 3, model='KR')
sage: elt = KRT(-3,-4,2,1,-3,-4,2,1,-2,-4,3,1); elt.pp()
  1  1  1
  2  2  3
 -4 -4 -4
 -3 -3 -2
sage: elt.left_split().pp()
  1 (X)    1  1
  2        2  3
 -4       -4 -4
 -3       -3 -2
```

phi(*i*)

Compute φ_i of `self`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 4, 1, model='KR')
sage: KRT(-1, -4, 3, 2).phi(1)
0
sage: KRT(-1, -4, 3, 2).phi(3)
1
```

to_array(*rows=True*)

Return a 2-dimensional array representation of this Kirillov-Reshetikhin element.

If the output is in rows, then it outputs the top row first (in the English convention) from left to right.

For example: if the reading word is $[2, 1, 4, 3]$, so as a 2×2 tableau:

```
1 3
2 4
```

we output `[[1, 3], [2, 4]]`.

If the output is in columns, then it outputs the leftmost column first with the bottom element first. In other words this parses the reading word into its columns.

Continuing with the previous example, the output would be `[[2, 1], [4, 3]]`.

INPUT:

- `rows` – (Default: `True`) Set to `True` if the resulting array is by row, otherwise it is by column.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 1], 4, 3, model='KR')
sage: elt = KRT(-3,-4,2,1,-3,-4,2,1,-2,-4,3,1)
sage: elt.to_array()
[[1, 1, 1], [2, 2, 3], [-4, -4, -4], [-3, -3, -2]]
sage: elt.to_array(False)
[[-3, -4, 2, 1], [-3, -4, 2, 1], [-2, -4, 3, 1]]
```

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxTypeBox**(*cartan_type*, *r*, *s*)

Bases: sage.combinat.rigged_configurations.kr_tableaux.KRTableauxTypeVertical

Kirillov-Reshetikhin tableaux $B^{r,s}$ of type:

- $A_{2n}^{(2)}$ for all $r \leq n$,
- $D_{n+1}^{(2)}$ for all $r < n$,
- $D_4^{(3)}$ for $r = 1$.

TESTS:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 2], 2, 2, model='KR')
sage: TestSuite(KRT).run()
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 2], 2, 2, model='KR')
sage: TestSuite(KRT).run() # long time
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 3], 1, 2, model='KR')
sage: TestSuite(KRT).run() # long time
```

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxTypeDTri2**(*cartan_type*, *r*, *s*)

Bases: sage.combinat.rigged_configurations.kr_tableaux.KirillovReshetikhinTableaux

Kirillov-Reshetikhin tableaux $B^{2,s}$ of type $D_4^{(3)}$.

Warning: The Kashiwara-Nakashima version is not implemented due to the non-trivial multiplicities of classical components, so `classical_decomposition()` does not work.

Element
alias of KRTableauxTypeDTri2Element

module_generators()
The module generators of `self`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,3], 2, 1, model='KR')
sage: KRT.module_generators
([[1], [2]], [[1], [0]], [[1], [E]], [[E], [E]])
```

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxTypeDTri2Element**(*parent*, *list*, ***options*)

Bases: sage.combinat.rigged_configurations.kr_tableaux.KirillovReshetikhinTableauxElement

A Kirillov-Reshetikhin tableau in $B^{2,s}$ of type $D_4^{(3)}$.

e(*i*)
Perform the action of e_i on `self`.

Todo

Implement a direct action of e_0 without moving to rigged configurations.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,3], 2, 1, model='KR')
sage: KRT.module_generators[0].e(0)
[[2], [E]]
```

epsilon(*i*)

Compute ε_i of `self`.

Todo

Implement a direct action of ϵ_0 without moving to KR crystals.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,3], 2, 2, model='KR')
sage: KRT.module_generators[0].epsilon(0)
6
```

f(*i*)

Perform the action of f_i on `self`.

Todo

Implement a direct action of f_0 without moving to rigged configurations.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,3], 2, 1, model='KR')
sage: KRT.module_generators[0].f(0)
sage: KRT.module_generators[3].f(0)
[[1], [0]]
```

phi(*i*)

Compute φ_i of `self`.

Todo

Compute ϕ_0 without moving to KR crystals.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,3], 2, 2, model='KR')
sage: KRT.module_generators[0].phi(0)
0
```

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxTypeHorizonal**(*cartan_type*, *r*, *s*)

Bases: sage.combinat.rigged_configurations.kr_tableaux.KirillovReshetikhinTableaux

Kirillov-Reshetikhin tableaux $B^{r,s}$ of type:

- $C_n^{(1)}$ for $1 \leq r < n$,
- $A_{2n}^{(2)\dagger}$ for $1 \leq r \leq n$.

TESTS:

```
sage: KRT = crystals.KirillovReshetikhin(['C', 3, 1], 2, 2, model='KR')
sage: TestSuite(KRT).run() # long time
sage: KRT = crystals.KirillovReshetikhin(CartanType(['A', 4, 2]).dual(), 2, 2, model='KR')
sage: TestSuite(KRT).run()
```

from_kirillov_reshetikhin_crystal(*krc*)

Construct an element of `self` from the Kirillov-Reshetikhin crystal element `krc`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['C',4,1], 2, 3, model='KR')
sage: C = crystals.KirillovReshetikhin(['C',4,1], 2, 3, model='KN')
sage: krc = C(4,3); krc
[[3], [4]]
sage: KRT.from_kirillov_reshetikhin_crystal(krc)
[[3, -2, 1], [4, -1, 2]]
```

class sage.combinat.rigged_configurations.kr_tableaux.**KRTableauxTypeVertical**(*cartan_type*, *r*, *s*)

Bases: sage.combinat.rigged_configurations.kr_tableaux.KirillovReshetikhinTableaux

Kirillov-Reshetkihn tableaux $B^{r,s}$ of type:

- $D_n^{(1)}$ for all $1 \leq r < n-1$,
- $B_n^{(1)}$ for all $1 \leq r < n$,
- $A_{2n-1}^{(2)}$ for all $1 \leq r \leq n$.

TESTS:

```
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 1], 1, 1, model='KR')
sage: TestSuite(KRT).run()
sage: KRT = crystals.KirillovReshetikhin(['B', 3, 1], 2, 2, model='KR')
sage: TestSuite(KRT).run() # long time
sage: KRT = crystals.KirillovReshetikhin(['A', 5, 2], 2, 2, model='KR')
sage: TestSuite(KRT).run() # long time
```

from_kirillov_reshetikhin_crystal(*krc*)

Construct an element of `self` from the Kirillov-Reshetikhin crystal element `krc`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 3, model='KR')
sage: C = crystals.KirillovReshetikhin(['D',4,1], 2, 3, model='KN')
sage: krc = C(4,3); krc
[[3], [4]]
sage: KRT.from_kirillov_reshetikhin_crystal(krc)
[[3, -2, 1], [4, -1, 2]]
```

class sage.combinat.rigged_configurations.kr_tableaux.**KirillovReshetikhinTableaux**(*cartan_type*, *r*, *s*)

Bases: sage.combinat.crystals.tensor_product.CrystalOfWords

Kirillov-Reshetikhin tableaux.

Kirillov-Reshetikhin tableaux are rectangular tableaux with r rows and s columns that naturally arise under the bijection between rigged configurations and tableaux [RigConBijection]. They are in bijection with the elements of the Kirillov-Reshetikhin crystal $B^{r,s}$ under the (inverse) filling map.

Whenever $B^{r,s} \cong B(s\Lambda_r)$ as a classical crystal (which is the case for $B^{r,s}$ in type $A_n^{(1)}$, $B^{n,s}$ in type $C_n^{(1)}$ and $D_{n+1}^{(2)}$, $B^{n,s}$ and $B^{n-1,s}$ in type $D_n^{(1)}$) then the filling map is trivial.

For $B^{r,s}$ in:

- type $D_n^{(1)}$ when $r \leq n-2$,
- type $B_n^{(1)}$ when $r < n$,
- type $A_{2n-1}^{(2)}$ for all r,

the filling map is defined in [OSS2011].

For the spinor cases in type $D_n^{(1)}$, the crystal $B^{k,s}$ where $k = n-1, n$, is isomorphic as a classical crystal to $B(s\Lambda_k)$, and here we consider the Kirillov-Reshetikhin tableaux as living in $B(2s\Lambda_k)$ under the natural doubling map. In this case, the crystal operators e_i and f_i act as e_i^2 and f_i^2 respectively. See [BijectionDn].

For the spinor case in type $B_n^{(1)}$, the crystal $B^{n,s}$, we consider the images under the natural doubling map into $B^{n,2s}$. The classical components of this crystal are now given by removing 2×2 boxes. The filling map is the same as below (see the non-spin type $C_n^{(1)}$).

For $B^{r,s}$ in:

- type $C_n^{(1)}$ when $r < n$,
- type $A_{2n}^{(2)\dagger}$ for all r,

the filling map is given as follows. Suppose we are considering the (classically) highest weight element in the classical component $B(\lambda)$. Then we fill it in with the horizontal dominoes $[\overline{\imath}, i]$ in the i-th row from the top (in English notation) and reordering the columns so that they are increasing. Recall from above that $B^{n,s} \cong B(s\Lambda_n)$ in type $C_n^{(1)}$.

For $B^{r,s}$ in:

- type $A_{2n}^{(2)}$ for all r,
- type $D_{n+1}^{(2)}$ when $r < n$,
- type $D_4^{(3)}$ when $r = 1$,

the filling map is the same as given in [OSS2011] except for the rightmost column which is given by the column $[1, 2, \ldots, k, \emptyset, \ldots \emptyset]$ where $k = (r + x - 1)/2$ in Step 3 of [OSS2011].

For the spinor case in type $D_{n+1}^{(2)}$, the crystal $B^{n,s}$, we define the filling map in the same way as in type $D_n^{(1)}$.

Note: The filling map and classical decompositions in non-spinor cases can be classified by how the special node 0 connects with the corresponding classical diagram.

The classical crystal stucture is given by the usual Kashiwara-Nakashima tableaux rules. That is to embed this into $B(\Lambda_1)^{\otimes ns}$ by using the reading word and then applying the classical crystal operator. The affine crystal stucture is given by converting to the corresponding KR crystal element, performing the affine crystal operator, and pulling back to a KR tableau.

For more information about the bijection between rigged configurations and tensor products of Kirillov-Reshetikhin tableaux, see `TensorProductOfKirillovReshetikhinTableaux`.

Note: The tableaux for all non-simply-laced types are provably correct if the bijection with `rigged configurations` holds. Therefore this is currently only proven for $B^{r,1}$ or $B^{1,s}$ and in general for types $A_n^{(1)}$ and $D_n^{(1)}$.

INPUT:

- `cartan_type` – the Cartan type
- `r` – the Dynkin diagram index (typically the number of rows)
- `s` – the number of columns

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 1, model='KR')
sage: elt = KRT(4, 3); elt
[[3], [4]]

sage: KRT = crystals.KirillovReshetikhin(['D', 4, 1], 2, 1, model='KR')
sage: elt = KRT(-1, 1); elt
[[1], [-1]]
```

We can create highest weight crystals from a given shape or weight:

```
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 1], 2, 2, model='KR')
sage: KRT.module_generator(shape=[1,1])
[[1, 1], [2, -1]]
sage: KRT.module_generator(column_shape=[2])
[[1, 1], [2, -1]]
sage: WS = RootSystem(['D',4,1]).weight_space()
sage: KRT.module_generator(weight=WS.sum_of_terms([[0,-2],[2,1]]))
[[1, 1], [2, -1]]
sage: WSC = RootSystem(['D',4]).weight_space()
sage: KRT.module_generator(classical_weight=WSC.fundamental_weight(2))
[[1, 1], [2, -1]]
```

We can go between `KashiwaraNakashimaTableaux()` and `KirillovReshetikhinTableaux` elements:

```
sage: KRCrys = crystals.KirillovReshetikhin(['D', 4, 1], 2, 2, model='KN')
sage: KRTab = crystals.KirillovReshetikhin(['D', 4, 1], 2, 2, model='KR')
sage: elt = KRCrys(3, 2); elt
[[2], [3]]
sage: k = KRTab(elt); k
[[2, 1], [3, -1]]
sage: KRCrys(k)
[[2], [3]]
```

We check that the classical weights in the classical decompositions agree in a few different type:

```
sage: KRCrys = crystals.KirillovReshetikhin(['D', 4, 1], 2, 2, model='KN')
sage: KRTab = crystals.KirillovReshetikhin(['D', 4, 1], 2, 2, model='KR')
sage: all(t.classical_weight() == KRCrys(t).classical_weight() for t in KRTab)
True
sage: KRCrys = crystals.KirillovReshetikhin(['B', 3, 1], 2, 2, model='KN')
sage: KRTab = crystals.KirillovReshetikhin(['B', 3, 1], 2, 2, model='KR')
sage: all(t.classical_weight() == KRCrys(t).classical_weight() for t in KRTab)
True
sage: KRCrys = crystals.KirillovReshetikhin(['C', 3, 1], 2, 2, model='KN')
sage: KRTab = crystals.KirillovReshetikhin(['C', 3, 1], 2, 2, model='KR')
sage: all(t.classical_weight() == KRCrys(t).classical_weight() for t in KRTab)
True
sage: KRCrys = crystals.KirillovReshetikhin(['D', 4, 2], 2, 2, model='KN')
sage: KRTab = crystals.KirillovReshetikhin(['D', 4, 2], 2, 2, model='KR')
sage: all(t.classical_weight() == KRCrys(t).classical_weight() for t in KRTab)
True
sage: KRCrys = crystals.KirillovReshetikhin(['A', 4, 2], 2, 2, model='KN')
sage: KRTab = crystals.KirillovReshetikhin(['A', 4, 2], 2, 2, model='KR')
```

```
sage: all(t.classical_weight() == KRCrys(t).classical_weight() for t in KRTab)
True
```

Element
alias of KirillovReshetikhinTableauxElement

affinization()
Return the corresponding affinization crystal of self.

EXAMPLES:
```
sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1, model='KR')
sage: K.affinization()
Affinization of Kirillov-Reshetikhin tableaux of type ['A', 2, 1] and shape (1, 1)
```

classical_decomposition()
Return the classical crystal decomposition of self.

EXAMPLES:
```
sage: crystals.KirillovReshetikhin(['D', 4, 1], 2, 2, model='KR').classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[], [1, 1], [2, 2]]
```

from_kirillov_reshetikhin_crystal(*krc*)
Construct an element of self from the Kirillov-Reshetikhin crystal element krc.

EXAMPLES:
```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 1, model='KR')
sage: C = crystals.KirillovReshetikhin(['A',4,1], 2, 1, model='KN')
sage: krc = C(4,3); krc
[[3], [4]]
sage: KRT.from_kirillov_reshetikhin_crystal(krc)
[[3], [4]]
```

kirillov_reshetikhin_crystal()
Return the corresponding KR crystal in the Kashiwara-Nakashima model.

EXAMPLES:
```
sage: crystals.KirillovReshetikhin(['A', 4, 1], 2, 1, model='KR').kirillov_reshetikhin_cryst
Kirillov-Reshetikhin crystal of type ['A', 4, 1] with (r,s)=(2,1)
```

module_generator(*i=None*, ***options*)
Return the specified module generator.

INPUT:

- i – the index of the module generator

We can also get a module generator by using one of the following optional arguments:

- shape – the associated shape
- column_shape – the shape given as columns (a column of length k correspond to a classical weight ω_k)
- weight – the weight
- classical_weight – the classical weight

If no arugments are specified, then return the unique module generator of classical weight $s\Lambda_r$.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 1], 2, 2, model='KR')
sage: KRT.module_generator(1)
[[1, 1], [2, -1]]
sage: KRT.module_generator(shape=[1,1])
[[1, 1], [2, -1]]
sage: KRT.module_generator(column_shape=[2])
[[1, 1], [2, -1]]
sage: WS = RootSystem(['D',4,1]).weight_space()
sage: KRT.module_generator(weight=WS.sum_of_terms([[0,-2],[2,1]]))
[[1, 1], [2, -1]]
sage: WSC = RootSystem(['D',4]).weight_space()
sage: KRT.module_generator(classical_weight=WSC.fundamental_weight(2))
[[1, 1], [2, -1]]
sage: KRT.module_generator()
[[1, 1], [2, 2]]

sage: KRT = crystals.KirillovReshetikhin(['A', 3, 1], 2, 2, model='KR')
sage: KRT.module_generator()
[[1, 1], [2, 2]]
```

r()

Return the value r for this tableaux class which corresponds to the number of rows.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 1, model='KR')
sage: KRT.r()
2
```

s()

Return the value s for this tableaux class which corresponds to the number of columns.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 1, model='KR')
sage: KRT.s()
1
```

tensor(**crystals*, ***options*)

Return the tensor product of `self` with `crystals`.

If `crystals` is a list of (a tensor product of) KR tableaux, this returns a `TensorProductOfKirillovReshetikhinTableaux`.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A', 3, 1], 2, 2, model='KR')
sage: TP = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 3, 1], [[1,3],[3,1]])
sage: K.tensor(TP, K)
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1]
 and factor(s) ((2, 2), (1, 3), (3, 1), (2, 2))

sage: C = crystals.KirillovReshetikhin(['A',3,1], 3, 1, model='KN')
sage: K.tensor(K, C)
Full tensor product of the crystals
 [Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (2, 2),
  Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (2, 2),
  Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(3,1)]
```

class sage.combinat.rigged_configurations.kr_tableaux.**KirillovReshetikhinTableauxElement**(*parent*, *list*, ***options*)

Bases: sage.combinat.crystals.tensor_product.TensorProductOfRegularCrystalsElement

A Kirillov-Reshetikhin tableau.

For more information, see KirillovReshetikhinTableaux and TensorProductOfKirillovReshetikhinTableaux.

classical_weight()

Return the classical weight of self.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: elt = KRT(3,2,-1,1); elt
[[2, 1], [3, -1]]
sage: elt.classical_weight()
(0, 1, 1, 0)
```

e(i)

Perform the action of e_i on self.

Todo

Implement a direct action of e_0 without moving to KR crystals.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: KRT.module_generators[0].e(0)
[[-2, 1], [-1, -1]]
```

epsilon(i)

Compute ε_i of self.

Todo

Implement a direct action of ε_0 without moving to KR crystals.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: KRT.module_generators[0].epsilon(0)
2
```

f(i)

Perform the action of f_i on self.

Todo

Implement a direct action of f_0 without moving to KR crystals.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: KRT.module_generators[0].f(0)
[[1, 1], [2, -1]]
```

left_split()
Return the image of `self` under the left column splitting map.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 3, model='KR')
sage: mg = KRT.module_generators[1]; mg.pp()
  1 -2  1
  2 -1  2
sage: ls = mg.left_split(); ls.pp()
  1 (X)  -2  1
  2      -1  2
sage: ls.parent()
Tensor product of Kirillov-Reshetikhin tableaux of type ['D', 4, 1] and factor(s) ((2, 1), (
```

lusztig_involution()
Return the result of the classical Lusztig involution on `self`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 3, model='KR')
sage: mg = KRT.module_generators[1]
sage: mg.lusztig_involution()
[[-2, -2, 1], [-1, -1, 2]]
sage: elt = mg.f_string([2,1,3,2]); elt
[[3, -2, 1], [4, -1, 2]]
sage: elt.lusztig_involution()
[[-4, -2, 1], [-3, -1, 2]]
```

phi(i)
Compute φ_i of `self`.

Todo

Compute φ_0 without moving to KR crystals.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: KRT.module_generators[0].phi(0)
2
```

pp()
Pretty print `self`.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 2, model='KR')
sage: elt = KRT(2, 1, 4, 3); elt
[[1, 3], [2, 4]]
sage: elt.pp()
1  3
2  4
```

right_split()
Return the image of `self` under the right column splitting map.

Let $*$ denote the `Lusztig involution`, and ls as the `left splitting map`. The right splitting map is defined as $\mathrm{rs} := * \circ \mathrm{ls} \circ *$.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 3, model='KR')
sage: mg = KRT.module_generators[1]; mg.pp()
  1 -2  1
  2 -1  2
sage: ls = mg.right_split(); ls.pp()
 -2  1 (X)   1
 -1  2       2
sage: ls.parent()
Tensor product of Kirillov-Reshetikhin tableaux of type ['D', 4, 1] and factor(s) ((2, 2), (
```

to_array(*rows=True*)

Return a 2-dimensional array representation of this Kirillov-Reshetikhin element.

If the output is in rows, then it outputs the top row first (in the English convention) from left to right.

For example: if the reading word is $[2, 1, 4, 3]$, so as a 2×2 tableau:

```
1 3
2 4
```

we output `[[1, 3], [2, 4]]`.

If the output is in columns, then it outputs the leftmost column first with the bottom element first. In other words this parses the reading word into its columns.

Continuing with the previous example, the output would be `[[2, 1], [4, 3]]`.

INPUT:

- `rows` – (Default: `True`) Set to `True` if the resulting array is by row, otherwise it is by column.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 2, model='KR')
sage: elt = KRT(2, 1, 4, 3)
sage: elt.to_array()
[[1, 3], [2, 4]]
sage: elt.to_array(False)
[[2, 1], [4, 3]]
```

to_classical_highest_weight(*index_set=None*)

Return the classical highest weight element corresponding to `self`.

INPUT:

- `index_set` – (Default: `None`) Return the highest weight with respect to the index set. If `None` is passed in, then this uses the classical index set.

OUTPUT:

A pair `[H, f_str]` where `H` is the highest weight element and `f_str` is a list of a_i of f_{a_i} needed to reach `H`.

EXAMPLES:

```
sage: KRTab = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: elt = KRTab(3,2,-1,1); elt
[[2, 1], [3, -1]]
sage: elt.to_classical_highest_weight()
[[[1, 1], [2, -1]], [1, 2]]
```

to_kirillov_reshetikhin_crystal()

Construct a `KashiwaraNakashimaTableaux()` element from `self`.

We construct the Kirillov-Reshetikhin crystal element as follows:

1.Determine the shape λ of the KR crystal from the weight.

2.Determine a path $e_{i_1} e_{i_2} \cdots e_{i_k}$ to the highest weight.

3.Apply $f_{i_k} \cdots f_{i_2} f_{i_1}$ to a highest weight KR crystal of shape λ.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: elt = KRT(3,2,-1,1); elt
[[2, 1], [3, -1]]
sage: elt.to_kirillov_reshetikhin_crystal()
[[2], [3]]
```

TESTS:

Spinor tests:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 4, 3, model='KR')
sage: KRC = crystals.KirillovReshetikhin(['D',4,1], 4, 3, model='KN')
sage: elt = KRT(-3,-4,2,1,-3,-4,2,1,-2,-4,3,1); elt
[[1, 1, 1], [2, 2, 3], [-4, -4, -4], [-3, -3, -2]]
sage: ret = elt.to_kirillov_reshetikhin_crystal(); ret
[++--, [[1], [3], [-4], [-3]]]
sage: test = KRT(ret); test
[[1, 1, 1], [2, 2, 3], [-4, -4, -4], [-3, -3, -2]]
sage: test == elt
True
```

to_tableau()

Return a Tableau object of self.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['A', 4, 1], 2, 2, model='KR')
sage: elt = KRT(2, 1, 4, 3); elt
[[1, 3], [2, 4]]
sage: t = elt.to_tableau(); t
[[1, 3], [2, 4]]
sage: type(t)
<class 'sage.combinat.tableau.Tableaux_all_with_category.element_class'>
```

weight()

Return the weight of self.

EXAMPLES:

```
sage: KR = crystals.KirillovReshetikhin(['D',4,1], 2, 2, model='KR')
sage: KR.module_generators[1].weight()
-2*Lambda[0] + Lambda[2]
```

5.1.184 Crystal of Rigged Configurations

AUTHORS:

- Travis Scrimshaw (2010-09-26): Initial version

We only consider the highest weight crystal structure, not the Kirillov-Reshetikhin structure, and we extend this to symmetrizable types.

class sage.combinat.rigged_configurations.rc_crystal.**CrystalOfNonSimplyLacedRC**(*vct*, *wt*, *WLR*)

Bases: sage.combinat.rigged_configurations.rc_crystal.CrystalOfRiggedConfigurations

Highest weight crystal of rigged configurations in non-simply-laced type.

Element
alias of RCHWNonSimplyLacedElement

from_virtual(*vrc*)
Convert vrc in the virtual crystal into a rigged configution of the original Cartan type.

INPUT:

- vrc – a virtual rigged configuration

EXAMPLES:

```
sage: La = RootSystem(['C', 3]).weight_lattice().fundamental_weights()
sage: RC = crystals.RiggedConfigurations(La[2])
sage: elt = RC(partition_list=[[0], [1], [1]])
sage: elt == RC.from_virtual(RC.to_virtual(elt))
True
```

to_virtual(*rc*)
Convert rc into a rigged configuration in the virtual crystal.

INPUT:

- rc – a rigged configuration element

EXAMPLES:

```
sage: La = RootSystem(['C', 3]).weight_lattice().fundamental_weights()
sage: RC = crystals.RiggedConfigurations(La[2])
sage: elt = RC(partition_list=[[], [1], [1]]); elt

(/)

0[ ]0

-1[ ]-1

sage: RC.to_virtual(elt)

(/)

0[ ]0

-2[ ][ ]-2

0[ ]0

(/)
```

virtual()
Return the corresponding virtual crystal.

EXAMPLES:

```
sage: La = RootSystem(['C', 2, 1]).weight_lattice().fundamental_weights()
sage: RC = crystals.RiggedConfigurations(La[0])
```

```
sage: RC
Crystal of rigged configurations of type ['C', 2, 1] and weight Lambda[0]
sage: RC.virtual
Crystal of rigged configurations of type ['A', 3, 1] and weight 2*Lambda[0]
```

class sage.combinat.rigged_configurations.rc_crystal.**CrystalOfRiggedConfigurations**(*wt*, *WLR*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

A highest weight crystal of rigged configurations.

The crystal structure for finite simply-laced types is given in [CrysStructSchilling06]. These were then shown to be the crystal operators in all finite types in [SchScr] and all simply-laced and a large class of foldings of simply-laced types in [SalScr].

INPUT:

- cartan_type – (optional) a Cartan type
- wt – the highest weight vector in the weight lattice

EXAMPLES:

For simplicity, we display the rigged configurations horizontally:

```
sage: RiggedConfigurations.global_options(display='horizontal')
```

We start with a simply-laced finite type:

```
sage: La = RootSystem(['A', 2]).weight_lattice().fundamental_weights()
sage: RC = crystals.RiggedConfigurations(La[1] + La[2])
sage: mg = RC.highest_weight_vector()
sage: mg.f_string([1,2])
0[ ]0   0[ ]-1
sage: mg.f_string([1,2,2])
0[ ]0   -2[ ][ ]-2
sage: mg.f_string([1,2,2,2])
sage: mg.f_string([2,1,1,2])
-1[ ][ ]-1   -1[ ][ ]-1
sage: RC.cardinality()
8
sage: T = crystals.Tableaux(['A', 2], shape=[2,1])
sage: RC.digraph().is_isomorphic(T.digraph(), edge_labels=True)
True
```

We reset the global options:

```
sage: RiggedConfigurations.global_options.reset()
```

REFERENCES:

Element
alias of RCHighestWeightElement

global_options(**get_value*, ***set_value*)
Sets and displays the global options for rigged configurations. If no parameters are set, then the function returns a copy of the options dictionary.

The options to partitions can be accessed as the method RiggedConfigurations.global_options of RiggedConfigurations.

OPTIONS:

- convention – (default: English) Sets the convention used for displaying tableaux and partitions
  - English – use the English convention
  - French – use the French convention
- display – (default: vertical) Specifies how rigged configurations should be printed
  - horizontal – displayed horizontally
  - vertical – displayed vertically
- element_ascii_art – (default: True) display using the repr option element_ascii_art
- half_width_boxes_type_b – (default: True) display the last rigged partition in affine type B as half width boxes
- notation – alternative name for convention

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',3,1], [[2,2],[1,1],[1,1]])
sage: elt = RC(partition_list=[[3,1], [3], [1]])
sage: elt

-3[ ][ ][ ]-3
-1[ ]-1

1[ ][ ][ ]1

-1[ ]-1

sage: RiggedConfigurations.global_options(display="horizontal", convention="french")
sage: elt
-1[ ]-1          1[ ][ ][ ]1    -1[ ]-1
-3[ ][ ][ ]-3
```

Changing the convention for rigged configurations also changes the convention option for tableaux and vice versa:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T.pp()
  4  5
  1  2  3
sage: Tableaux.global_options(convention="english")
sage: elt
-3[ ][ ][ ]-3    1[ ][ ][ ]1    -1[ ]-1
-1[ ]-1
sage: T.pp()
  1  2  3
  4  5
sage: RiggedConfigurations.global_options.reset()
```

See GlobalOptions for more features of these options.

weight_lattice_realization()

Return the weight lattice realization used to express the weights of elements in self.

EXAMPLES:

```
sage: La = RootSystem(['A', 2, 1]).weight_lattice(extended=True).fundamental_weights()
sage: RC = crystals.RiggedConfigurations(La[0])
sage: RC.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]
```

5.1.185 Rigged Configurations of $\mathcal{B}(\infty)$

AUTHORS:

- Travis Scrimshaw (2013-04-16): Initial version

class sage.combinat.rigged_configurations.rc_infinity.**InfinityCrystalOfNonSimplyLacedRC**(*vct*)
Bases: sage.combinat.rigged_configurations.rc_infinity.InfinityCrystalOfRiggedConfigurati

Rigged configurations for $\mathcal{B}(\infty)$ in non-simply-laced types.

class Element(*parent*, *rigged_partitions*=[ ], ***options*)
Bases: sage.combinat.rigged_configurations.rigged_configuration_element.RCNonSimplyLa

A rigged configuration in $\mathcal{B}(\infty)$ in non-simply-laced types.

TESTS:

```
sage: RC = crystals.infinity.RiggedConfigurations(['C', 3])
sage: elt = RC(partition_list=[[1],[1,1],[1]])
sage: TestSuite(elt).run()
```

weight()
Return the weight of self.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['C', 3])
sage: elt = RC(partition_list=[[1],[1,1],[1]], rigging_list=[[0],[-1,-1],[0]])
sage: elt.weight()
(-1, -1, 0)

sage: RC = crystals.infinity.RiggedConfigurations(['F', 4, 1])
sage: mg = RC.highest_weight_vector()
sage: elt = mg.f_string([1,0,3,4,2,2]); ascii_art(elt)
-1[ ]-1  0[ ]1  -2[ ][ ]-2  0[ ]1  -1[ ]-1
sage: wt = elt.weight(); wt
-Lambda[0] + Lambda[1] - 2*Lambda[2] + 3*Lambda[3] - Lambda[4] - delta
sage: al = RC.weight_lattice_realization().simple_roots()
sage: wt == -(al[0] + al[1] + 2*al[2] + al[3] + al[4])
True
```

InfinityCrystalOfNonSimplyLacedRC.**from_virtual**(*vrc*)
Convert vrc in the virtual crystal into a rigged configution of the original Cartan type.

INPUT:

- vrc – a virtual rigged configuration

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['C', 2])
sage: elt = RC(partition_list=[[3],[2]], rigging_list=[[-2],[0]])
sage: vrc_elt = RC.to_virtual(elt)
sage: ret = RC.from_virtual(vrc_elt); ret

-3[ ][ ][ ]-2

-1[ ][ ]0
```

```
sage: ret == elt
True
```

InfinityCrystalOfNonSimplyLacedRC.**to_virtual**(*rc*)

Convert `rc` into a rigged configuration in the virtual crystal.

INPUT:

- `rc` – a rigged configuration element

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['C', 2])
sage: mg = RC.highest_weight_vector()
sage: elt = mg.f_string([1,2,2,1,1]); elt

-3[ ][ ][ ]-2

-1[ ][ ]0

sage: velt = RC.to_virtual(elt); velt

-3[ ][ ][ ]-2

-2[ ][ ][ ][ ]0

-3[ ][ ][ ]-2

sage: velt.parent()
The infinity crystal of rigged configurations of type ['A', 3]
```

InfinityCrystalOfNonSimplyLacedRC.**virtual**()

Return the corresponding virtual crystal.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['C', 3])
sage: RC
The infinity crystal of rigged configurations of type ['C', 3]
sage: RC.virtual
The infinity crystal of rigged configurations of type ['A', 5]
```

class sage.combinat.rigged_configurations.rc_infinity.**InfinityCrystalOfRiggedConfigurations**(*c*

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Rigged configuration model for $\mathcal{B}(\infty)$.

The crystal is generated by the empty rigged configuration with the same crystal structure given by the `highest weight model` except we remove the condition that the resulting rigged configuration needs to be valid when applying f_a.

INPUT:

- `cartan_type` – a Cartan type

EXAMPLES:

For simplicity, we display all of the rigged configurations horizontally:

```
sage: RiggedConfigurations.global_options(display='horizontal')
```

We begin with a simply-laced finite type:

```
sage: RC = crystals.infinity.RiggedConfigurations(['A', 3]); RC
The infinity crystal of rigged configurations of type ['A', 3]

sage: RC.global_options(display='horizontal')

sage: mg = RC.highest_weight_vector(); mg
(/)  (/)  (/)
sage: elt = mg.f_string([2,1,3,2]); elt
0[ ]0   -2[ ]-1   0[ ]0
        -2[ ]-1
sage: elt.e(1)
sage: elt.e(3)
sage: mg.f_string([2,1,3,2]).e(2)
-1[ ]-1  0[ ]1  -1[ ]-1
sage: mg.f_string([2,3,2,1,3,2])
0[ ]0  -3[ ][ ]-1  -1[ ][ ]-1
       -2[ ]-1
```

Next we consider a non-simply-laced finite type:

```
sage: RC = crystals.infinity.RiggedConfigurations(['C', 3])
sage: mg = RC.highest_weight_vector()
sage: mg.f_string([2,1,3,2])
0[ ]0   -2[ ]-1   0[ ]0
        -2[ ]-1
sage: mg.f_string([2,3,2,1,3,2])
-1[ ]-1  -2[ ][ ][ ]-1  -1[ ][ ]0
```

We can also construct $B(\infty)$ using rigged configurations in affine types:

```
sage: RC = crystals.infinity.RiggedConfigurations(['A', 3, 1])
sage: mg = RC.highest_weight_vector()
sage: mg.f_string([0,1,2,3,0,1,3])
-1[ ]0  -1[ ]-1  1[ ]1  -1[ ][ ]-1
-1[ ]0  -1[ ]-1

sage: RC = crystals.infinity.RiggedConfigurations(['C', 3, 1])
sage: mg = RC.highest_weight_vector()
sage: mg.f_string([1,2,3,0,1,2,3,3,0])
-2[ ][ ]-1  -1[ ]0  -1[ ]-1  -4[ ][ ][ ]-2
            -1[ ]0  -1[ ]-1

sage: RC = crystals.infinity.RiggedConfigurations(['A', 6, 2])
sage: mg = RC.highest_weight_vector()
sage: mg.f_string([1,2,3,0,1,2,3,3,0])
0[ ]-1  0[ ]1  -1[ ]-1  -4[ ][ ][ ]-2
0[ ]-1  0[ ]1  -1[ ]-1
```

We reset the global options:

```
sage: RiggedConfigurations.global_options.reset()
```

class **Element** (*parent*, *rigged_partitions*=[ ], ***options*)

Bases: sage.combinat.rigged_configurations.rigged_configuration_element.RiggedConfigu

A rigged configuration in $\mathcal{B}(\infty)$ in simply-laced types.

TESTS:

```
sage: RC = crystals.infinity.RiggedConfigurations(['A', 3, 1])
sage: elt = RC(partition_list=[[1,1]]*4, rigging_list=[[1,1], [0,0], [0,0], [-1,-1]])
sage: TestSuite(elt).run()
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['A', 3, 1])
sage: elt = RC(partition_list=[[1,1]]*4, rigging_list=[[1,1], [0,0], [0,0], [-1,-1]])
sage: elt.weight()
-2*delta
```

InfinityCrystalOfRiggedConfigurations.**global_options**(**get_value*, ***set_value*)

Sets and displays the global options for rigged configurations. If no parameters are set, then the function returns a copy of the options dictionary.

The options to partitions can be accessed as the method RiggedConfigurations.global_options of RiggedConfigurations.

OPTIONS:

- `convention` – (default: `English`) Sets the convention used for displaying tableaux and partitions
  - `English` – use the English convention
  - `French` – use the French convention
- `display` – (default: `vertical`) Specifies how rigged configurations should be printed
  - `horizontal` – displayed horizontally
  - `vertical` – displayed vertically
- `element_ascii_art` – (default: `True`) display using the repr option `element_ascii_art`
- `half_width_boxes_type_b` – (default: `True`) display the last rigged partition in affine type B as half width boxes
- `notation` – alternative name for `convention`

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',3,1], [[2,2],[1,1],[1,1]])
sage: elt = RC(partition_list=[[3,1], [3], [1]])
sage: elt

-3[ ][ ][ ]-3
-1[ ]-1

1[ ][ ][ ]1

-1[ ]-1

sage: RiggedConfigurations.global_options(display="horizontal", convention="french")
sage: elt
-1[ ]-1          1[ ][ ][ ]1    -1[ ]-1
-3[ ][ ][ ]-3
```

Changing the `convention` for rigged configurations also changes the `convention` option for tableaux and vice versa:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T.pp()
  4  5
  1  2  3
sage: Tableaux.global_options(convention="english")
sage: elt
-3[ ][ ][ ]-3   1[ ][ ][ ]1   -1[ ]-1
-1[ ]-1
sage: T.pp()
  1  2  3
  4  5
sage: RiggedConfigurations.global_options.reset()
```

See `GlobalOptions` for more features of these options.

`InfinityCrystalOfRiggedConfigurations.`**`weight_lattice_realization`**`()`

Return the weight lattice realization used to express the weights of elements in `self`.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['A', 2, 1])
sage: RC.weight_lattice_realization()
Extended weight lattice of the Root system of type ['A', 2, 1]
```

5.1.186 Rigged Configuration Elements

A rigged configuration element is a sequence of `RiggedPartition` objects.

AUTHORS:

- Travis Scrimshaw (2010-09-26): Initial version
- Travis Scrimshaw (2012-10-25): Added virtual rigged confingurations

class `sage.combinat.rigged_configurations.rigged_configuration_element.`**`KRRCNonSimplyLacedElem`**

Bases: `sage.combinat.rigged_configurations.rigged_configuration_element.KRRiggedConfigura`
`sage.combinat.rigged_configurations.rigged_configuration_element.RCNonSimplyLacedEleme`

$U'_q(\mathfrak{g})$ rigged configurations in non-simply-laced types.

TESTS:

```
sage: RC = RiggedConfigurations(['C',2,1], [[1,2],[1,1],[2,1]])
sage: elt = RC(partition_list=[[3],[2]]); elt

0[ ][ ][ ]0

0[ ][ ]0
sage: TestSuite(elt).run()
```

`cc``()`

Compute the cocharge statistic.

Computes the cocharge statistic [OSS03] on this rigged configuration (ν, J) by computing the cocharge as a virtual rigged configuration $(\hat{\nu}, \hat{J})$ and then using the identity $cc(\hat{\nu}, \hat{J}) = \gamma_0 cc(\nu, J)$.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C', 3, 1], [[2,1], [1,1]])
sage: RC(partition_list=[[1,1],[2,1],[1,1]]).cocharge()
1
```

cocharge()

Compute the cocharge statistic.

Computes the cocharge statistic [OSS03] on this rigged configuration (ν, J) by computing the cocharge as a virtual rigged configuration $(\hat{\nu}, \hat{J})$ and then using the identity $cc(\hat{\nu}, \hat{J}) = \gamma_0 cc(\nu, J)$.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C', 3, 1], [[2,1], [1,1]])
sage: RC(partition_list=[[1,1],[2,1],[1,1]]).cocharge()
1
```

e(a)

Return the action of e_a on `self`.

This works by lifting into the virtual configuration, then applying

$$e_a^v = \prod_{j \in \iota(a)} \hat{e}_j^{\gamma_j}$$

and pulling back.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',6,2], [[1,1]]*7)
sage: elt = RC(partition_list=[[1]*5,[2,1,1],[3,2]])
sage: elt.e(3)

0[ ]0
0[ ]0
0[ ]0
0[ ]0
0[ ]0

0[ ][ ]0
1[ ]1
1[ ]1

1[ ][ ]1
1[ ]0
```

f(a)

Return the action of f_a on `self`.

This works by lifting into the virtual configuration, then applying

$$f_a^v = \prod_{j \in \iota(a)} \hat{f}_j^{\gamma_j}$$

and pulling back.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',6,2], [[1,1]]*7)
sage: elt = RC(partition_list=[[1]*5,[2,1,1],[2,1]], rigging_list=[[0]*5,[0,1,1],[1,0]])
sage: elt.f(3)
```

```
0[ ]0
0[ ]0
0[ ]0
0[ ]0
0[ ]0

1[ ][ ]1
1[ ]1
1[ ]1

-1[ ][ ][ ]-1
 0[ ][ ]0
```

class sage.combinat.rigged_configurations.rigged_configuration_element.**KRRCSimplyLacedElement**

Bases: sage.combinat.rigged_configurations.rigged_configuration_element.KRRiggedConfigura

$U'_q(\mathfrak{g})$ rigged configurations in simply-laced types.

TESTS:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [2,1], [1,1]])
sage: elt = RC(partition_list=[[1], [1], []]); elt

0[ ]0

0[ ]0

(/)

sage: TestSuite(elt).run()
```

cc()

Compute the cocharge statistic of `self`.

Computes the cocharge statistic [CrysStructSchilling06] on this rigged configuration (ν, J). The cocharge statistic is defined as:

$$cc(\nu, J) = \frac{1}{2} \sum_{a,b \in I_0} \sum_{j,k>0} (\alpha_a \mid \alpha_b) \min(j,k) m_j^{(a)} m_k^{(b)} + \sum_{a \in I} \sum_{i>0} \left| J^{(a,i)} \right|.$$

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [2,1], [1,1]])
sage: RC(partition_list=[[1], [1], []]).cocharge()
1
```

charge()

Compute the charge statistic of `self`.

Let B denote a set of rigged configurations. The *charge* c of a rigged configuration b is computed as

$$c(b) = \max(cc(b) \mid b \in B) - cc(b).$$

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [2,1], [1,1]])
sage: RC(partition_list=[[],[],[]]).charge()
2
sage: RC(partition_list=[[1], [1], []]).charge()
1
```

cocharge()

Compute the cocharge statistic of `self`.

Computes the cocharge statistic [CrysStructSchilling06] on this rigged configuration (ν, J). The cocharge statistic is defined as:

$$cc(\nu, J) = \frac{1}{2} \sum_{a,b \in I_0} \sum_{j,k>0} (\alpha_a \mid \alpha_b) \min(j,k) m_j^{(a)} m_k^{(b)} + \sum_{a \in I} \sum_{i>0} \left| J^{(a,i)} \right|.$$

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [2,1], [1,1]])
sage: RC(partition_list=[[1], [1], []]).cocharge()
1
```

class sage.combinat.rigged_configurations.rigged_configuration_element.**KRRCTypeA2DualElement**

Bases: sage.combinat.rigged_configurations.rigged_configuration_element.KRRCNonSimplyLace

$U_q'(\mathfrak{g})$ rigged configurations in type $A_{2n}^{(2)\dagger}$.

cc()

Compute the cocharge statistic.

Computes the cocharge statistic [RigConBijection] on this rigged configuration (ν, J). The cocharge statistic is computed as:

$$cc(\nu, J) = \frac{1}{2} \sum_{a \in I_0} \sum_{i>0} t_a^\vee m_i^{(a)} \left(\sum_{j>0} \min(i,j) L_j^{(a)} - p_i^{(a)} \right) + \sum_{a \in I} t_a^\vee \sum_{i>0} \left| J^{(a,i)} \right|.$$

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[1,1],[2,2]])
sage: sc = RC.cartan_type().as_folding().scaling_factors()
sage: all(mg.cocharge() * sc[0] == mg.to_virtual_configuration().cocharge()
....:     for mg in RC.module_generators)
True
```

cocharge()

Compute the cocharge statistic.

Computes the cocharge statistic [RigConBijection] on this rigged configuration (ν, J). The cocharge statistic is computed as:

$$cc(\nu, J) = \frac{1}{2} \sum_{a \in I_0} \sum_{i>0} t_a^\vee m_i^{(a)} \left(\sum_{j>0} \min(i,j) L_j^{(a)} - p_i^{(a)} \right) + \sum_{a \in I} t_a^\vee \sum_{i>0} \left| J^{(a,i)} \right|.$$

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[1,1],[2,2]])
sage: sc = RC.cartan_type().as_folding().scaling_factors()
sage: all(mg.cocharge() * sc[0] == mg.to_virtual_configuration().cocharge()
....:     for mg in RC.module_generators)
True
```

epsilon(a)

Return the value of ε_a of `self`.

Here we need to modify the usual definition by $\varepsilon'_n := 2\varepsilon_n$.

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[1,1], [2,2]])
sage: def epsilon(x, i):
....:     x = x.e(i)
....:     eps = 0
....:     while x is not None:
....:         x = x.e(i)
....:         eps = eps + 1
....:     return eps
sage: all(epsilon(rc, 2) == rc.epsilon(2) for rc in RC)
True
```

phi(a)

Return the value of φ_a of `self`.

Here we need to modify the usual definition by $\varphi'_n := 2\varphi_n$.

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[1,1], [2,2]])
sage: def phi(x, i):
....:     x = x.f(i)
....:     ph = 0
....:     while x is not None:
....:         x = x.f(i)
....:         ph = ph + 1
....:     return ph
sage: all(phi(rc, 2) == rc.phi(2) for rc in RC)
True
```

class sage.combinat.rigged_configurations.rigged_configuration_element.**KRRiggedConfigurationE**

Bases: sage.combinat.rigged_configurations.rigged_configuration_element.RiggedConfigurati

$U'_q(\mathfrak{g})$ rigged configurations.

EXAMPLES:

We can go between rigged configurations and tensor products of tensor products of KR tableaux:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[1,1], [2,1]])
sage: rc_elt = RC(partition_list=[[1], [1,1], [1], [1]])
sage: tp_krtab = rc_elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(); tp_krtab
[[-2]] (X) [[1], [2]]
sage: tp_krcrys = rc_elt.to_tensor_product_of_kirillov_reshetikhin_crystals(); tp_krcrys
[[[-2]], [[1], [2]]]
```

```
sage: tp_krcrys == tp_krtab.to_tensor_product_of_kirillov_reshetikhin_crystals()
True
sage: RC(tp_krcrys) == rc_elt
True
sage: RC(tp_krtab) == rc_elt
True
sage: tp_krtab.to_rigged_configuration() == rc_elt
True
```

check()

Make sure all of the riggings are less than or equal to the vacancy number.

TESTS:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 1]])
sage: elt = RC(partition_list=[[1], [1], [], []])
sage: elt.check()
```

classical_weight()

Return the classical weight of `self`.

The classical weight Λ of a rigged configuration is

$$\Lambda = \sum_{a \in \overline{I}} \sum_{i>0} iL_i^{(a)} \Lambda_a - \sum_{a \in \overline{I}} \sum_{i>0} im_i^{(a)} \alpha_a.$$

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D',4,1], [[2,2]])
sage: elt = RC(partition_list=[[2],[2,1],[1],[1]])
sage: elt.classical_weight()
(0, 1, 1, 0)
```

This agrees with the corresponding classical weight as KR tableaux:

```
sage: krt = elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(); krt
[[2, 1], [3, -1]]
sage: krt.classical_weight() == elt.classical_weight()
True
```

TESTS:

We check the classical weights agree in an entire crystal:

```
sage: RC = RiggedConfigurations(['A',2,1], [[2,1], [1,1]])
sage: for x in RC:
....:     y = x.to_tensor_product_of_kirillov_reshetikhin_tableaux()
....:     assert x.classical_weight() == y.classical_weight()
```

complement_rigging(*reverse_factors=False*)

Apply the complement rigging morphism θ to `self`.

Consider a highest weight rigged configuration (ν, J), the complement rigging morphism $\theta : RC(L) \to RC(L)$ is given by sending $(\nu, J) \mapsto (\nu, J')$, where J' is obtained by taking the coriggings $x' = p_i^{(a)} - x$, and then extending as a crystal morphism. (The name comes from taking the complement partition for the riggings in a $m_i^{(a)} \times p_i^{(a)}$ box.)

INPUT:

- `reverse_factors` – (default: `False`) if `True`, then this returns an element in $RC(B')$ where B' is the tensor factors of `self` in reverse order

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D',4,1], [[1,1],[2,2]])
sage: mg = RC.module_generators[-1]
sage: ascii_art(mg)
1[ ][ ]1  0[ ][ ]0  0[ ][ ]0  0[ ][ ]0
          0[ ][ ]0
sage: ascii_art(mg.complement_rigging())
1[ ][ ]0  0[ ][ ]0  0[ ][ ]0  0[ ][ ]0
          0[ ][ ]0

sage: lw = mg.to_lowest_weight([1,2,3,4])[0]
sage: ascii_art(lw)
-1[ ][ ]-1  0[ ][ ]0  0[ ][ ]0  0[ ][ ]0
-1[ ]-1     0[ ][ ]0  0[ ]0     0[ ]0
-1[ ]-1     0[ ]0
            0[ ]0
sage: ascii_art(lw.complement_rigging())
-1[ ][ ][ ]-1  0[ ][ ][ ]0  0[ ][ ][ ]0  0[ ][ ][ ]0
-1[ ]-1        0[ ][ ][ ]0
sage: lw.complement_rigging() == mg.complement_rigging().to_lowest_weight([1,2,3,4])[0]
True

sage: mg.complement_rigging(True).parent()
Rigged configurations of type ['D', 4, 1] and factor(s) ((2, 2), (1, 1))
```

We check that the Lusztig involution (under the modification of also mapping to the highest weight element) intertwines with the complement map θ (that reverses the tensor factors) under the bijection Φ:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 2], [2, 1], [1, 2]])
sage: for mg in RC.module_generators: # long time
....:     y = mg.to_tensor_product_of_kirillov_reshetikhin_tableaux()
....:     hw = y.lusztig_involution().to_highest_weight([1,2,3,4])[0]
....:     c = mg.complement_rigging(True)
....:     hwc = c.to_tensor_product_of_kirillov_reshetikhin_tableaux()
....:     assert hw == hwc
```

TESTS:

We check that trac ticket #18898 is fixed:

```
sage: RC = RiggedConfigurations(['D',4,1], [[2,1], [2,1], [2,3]])
sage: x = RC(partition_list=[[1], [1,1], [1], [1]], rigging_list=[[0], [2,1], [0], [0]])
sage: ascii_art(x)
0[ ]0  2[ ]2  0[ ]0  0[ ]0
       2[ ]1
sage: ascii_art(x.complement_rigging())
0[ ]0  2[ ]1  0[ ]0  0[ ]0
       2[ ]0
```

delta(*return_b=False*)

Return the image of `self` under the left box removal map δ.

The map $\delta : RC(B^{r,1} \otimes B) \to RC(B^{r-1,1} \otimes B)$ (if $r = 1$, then we remove the left-most factor) is the basic map in the bijection Φ between rigged configurations and tensor products of Kirillov-Reshetikhin tableaux. For more information, see `to_tensor_product_of_kirillov_reshetikhin_tableaux()`. We can extend δ when the left-most factor is not a single column by precomposing with a `left_split()`.

Note: Due to the special nature of the bijection for the spinor cases in types $D_n^{(1)}$, $B_n^{(1)}$, and $A_{2n-1}^{(2)}$, this

map is not defined in these cases.

INPUT:

- `return_b` – (default: `False`) whether to return the resulting letter from δ

OUTPUT:

The resulting rigged configuration or if `return_b` is `True`, then a tuple of the resulting rigged configuration and the letter.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',4,1], [[3,2]])
sage: mg = RC.module_generators[-1]
sage: ascii_art(mg)
0[ ][ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
          0[ ][ ]0  0[ ][ ]0  0[ ]0
                    0[ ][ ]0  0[ ]0
sage: ascii_art(mg.left_box())
0[ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
       0[ ]0     0[ ][ ]0  0[ ]0
sage: x,b = mg.left_box(True)
sage: b
-1
sage: b.parent()
The crystal of letters for type ['C', 4]
```

e(a)

Return the action of the crystal operator e_a on `self`.

For the classical operators, this implements the method defined in [CrysStructSchilling06]. For e_0, this converts the class to a tensor product of KR tableaux and does the corresponding e_0 and pulls back.

Todo

Implement e_0 without appealing to tensor product of KR tableaux.

INPUT:

- `a` – the index of the partition to remove a box

OUTPUT:

The resulting rigged configuration element.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2,1]])
sage: elt = RC(partition_list=[[1], [1], [1], [1]])
sage: elt.e(3)
sage: elt.e(1)

(/)

0[ ]0

0[ ]0

-1[ ]-1
```

epsilon(*a*)

Return ε_a of `self`.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 2]])
sage: I = RC.index_set()
sage: matrix([[mg.epsilon(i) for i in I] for mg in RC.module_generators])
[4 0 0 0 0]
[3 0 0 0 0]
[2 0 0 0 0]
```

f(*a*)

Return the action of the crystal operator f_a on `self`.

For the classical operators, this implements the method defined in [CrysStructSchilling06]. For f_0, this converts the class to a tensor product of KR tableaux and does the corresponding f_0 and pulls back.

Todo

Implement f_0 without appealing to tensor product of KR tableaux.

INPUT:

- `a` – the index of the partition to add a box

OUTPUT:

The resulting rigged configuration element.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2,1]])
sage: elt = RC(partition_list=[[1], [1], [1], [1]])
sage: elt.f(1)
sage: elt.f(2)

0[ ]0

-1[ ]-1
-1[ ]-1

1[ ]1

-1[ ]-1
```

left_box(*return_b=False*)

Return the image of `self` under the left box removal map δ.

The map $\delta : RC(B^{r,1} \otimes B) \to RC(B^{r-1,1} \otimes B)$ (if $r = 1$, then we remove the left-most factor) is the basic map in the bijection Φ between rigged configurations and tensor products of Kirillov-Reshetikhin tableaux. For more information, see `to_tensor_product_of_kirillov_reshetikhin_tableaux()`. We can extend δ when the left-most factor is not a single column by precomposing with a `left_split()`.

Note: Due to the special nature of the bijection for the spinor cases in types $D_n^{(1)}$, $B_n^{(1)}$, and $A_{2n-1}^{(2)}$, this map is not defined in these cases.

INPUT:

- `return_b` – (default: `False`) whether to return the resulting letter from δ

OUTPUT:

The resulting rigged configuration or if `return_b` is `True`, then a tuple of the resulting rigged configuration and the letter.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',4,1], [[3,2]])
sage: mg = RC.module_generators[-1]
sage: ascii_art(mg)
0[ ][ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
          0[ ][ ]0  0[ ][ ]0  0[ ]0
                    0[ ][ ]0  0[ ]0
sage: ascii_art(mg.left_box())
0[ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
       0[ ]0     0[ ][ ]0  0[ ]0
sage: x,b = mg.left_box(True)
sage: b
-1
sage: b.parent()
The crystal of letters for type ['C', 4]
```

left_column_box()

Return the image of `self` under the left column box splitting map γ.

Consider the map $\gamma : RC(B^{r,1} \otimes B) \to RC(B^{1,1} \otimes B^{r-1,1} \otimes B)$ for $r > 1$, which is a natural strict classical crystal injection. On rigged configurations, the map γ adds a singular string of length 1 to $\nu^{(a)}$.

We can extend γ when the left-most factor is not a single column by precomposing with a `left_split()`.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',3,1], [[3,1], [2,1]])
sage: mg = RC.module_generators[-1]
sage: ascii_art(mg)
0[ ]0  0[ ][ ]0  0[ ]0
       0[ ]0     0[ ]0
sage: ascii_art(mg.left_column_box())
0[ ]0  0[ ][ ]0  0[ ]0
0[ ]0  0[ ]0     0[ ]0
       0[ ]0

sage: RC = RiggedConfigurations(['C',3,1], [[2,1], [1,1], [3,1]])
sage: mg = RC.module_generators[7]
sage: ascii_art(mg)
1[ ]0  0[ ][ ]0  0[ ]0
       0[ ]0     0[ ]0
sage: ascii_art(mg.left_column_box())
1[ ]1  0[ ][ ]0  0[ ]0
1[ ]0  0[ ]0     0[ ]0
```

left_split()

Return the image of `self` under the left column splitting map β.

Consider the map $\beta : RC(B^{r,s} \otimes B) \to RC(B^{r,1} \otimes B^{r,s-1} \otimes B)$ for $s > 1$ which is a natural classical crystal injection. On rigged configurations, the map β does nothing (except possibly changing the vacancy numbers).

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',4,1], [[3,3]])
sage: mg = RC.module_generators[-1]
sage: ascii_art(mg)
0[ ][ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
          0[ ][ ]0  0[ ][ ]0  0[ ]0
                    0[ ][ ]0  0[ ]0
sage: ascii_art(mg.left_split())
0[ ][ ]0  0[ ][ ]0  1[ ][ ]0  0[ ]0
          0[ ][ ]0  1[ ][ ]0  0[ ]0
                    1[ ][ ]0  0[ ]0
```

phi(a)

Return φ_a of `self`.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 2]])
sage: I = RC.index_set()
sage: matrix([[mg.phi(i) for i in I] for mg in RC.module_generators])
[0 0 2 0 0]
[1 0 1 0 0]
[2 0 0 0 0]
```

right_column_box()

Return the image of `self` under the right column box splitting map γ^*.

Consider the map $\gamma^* : RC(B \otimes B^{r,1}) \to RC(B \otimes B^{r-1,1} \otimes B^{1,1})$ for $r > 1$, which is a natural strict classical crystal injection. On rigged configurations, the map γ adds a string of length 1 with rigging 0 to $\nu^{(a)}$ for all $a < r$ to a classically highest weight element and extended as a classical crystal morphism.

We can extend γ^* when the right-most factor is not a single column by precomposing with a `right_split()`.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',3,1], [[2,1], [1,1], [3,1]])
sage: mg = RC.module_generators[7]
sage: ascii_art(mg)
1[ ]0  0[ ][ ]0  0[ ]0
       0[ ]0     0[ ]0
sage: ascii_art(mg.right_column_box())
1[ ]0  0[ ][ ]0  0[ ]0
1[ ]0  0[ ]0     0[ ]0
       0[ ]0
```

right_split()

Return the image of `self` under the right column splitting map β^*.

Let θ denote the `complement rigging map` which reverses the tensor factors and β denote the `left splitting map`, we define the right splitting map by $\beta^* := \theta \circ \beta \circ \theta$.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',4,1], [[3,3]])
sage: mg = RC.module_generators[-1]
sage: ascii_art(mg)
0[ ][ ]0  0[ ][ ]0  0[ ][ ]0  0[ ]0
          0[ ][ ]0  0[ ][ ]0  0[ ]0
                    0[ ][ ]0  0[ ]0
sage: ascii_art(mg.right_split())
0[ ][ ]0  0[ ][ ]0  1[ ][ ]1  0[ ]0
```

```
          0[ ][ ]0  1[ ][ ]1  0[ ]0
                    1[ ][ ]1  0[ ]0

sage: RC = RiggedConfigurations(['D',4,1], [[2,2],[1,2]])
sage: elt = RC(partition_list=[[3,1], [2,2,1], [2,1], [2]])
sage: ascii_art(elt)
-1[ ][ ][ ]-1  0[ ][ ]0  -1[ ][ ]-1  1[ ][ ]1
 0[ ]0         0[ ][ ]0  -1[ ]-1
               0[ ]0
sage: ascii_art(elt.right_split())
-1[ ][ ][ ]-1  0[ ][ ]0  -1[ ][ ]-1  1[ ][ ]1
 1[ ]0         0[ ][ ]0  -1[ ]-1
               0[ ]0
```

We check that the bijection commutes with the right spliting map:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[1,1], [2,2]])
sage: all(rc.right_split().to_tensor_product_of_kirillov_reshetikhin_tableaux()
....:     == rc.to_tensor_product_of_kirillov_reshetikhin_tableaux().right_split() for rc in
True
```

to_tensor_product_of_kirillov_reshetikhin_crystals(*display_steps=False*, *build_graph=False*)

Return the corresponding tensor product of Kirillov-Reshetikhin crystals.

This is a composition of the map to a tensor product of KR tableaux, and then to a tensor product of KR crystals.

INPUT:

- `display_steps` – (default: `False`) boolean which indicates if we want to print each step in the algorithm
- `build_graph` – (default: `False`) boolean which indicates if we want to construct and return a graph of the bijection whose vertices are rigged configurations obtained at each step and edges are labeled by either the return value of δ or the doubling/halving map

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 2]])
sage: elt = RC(partition_list=[[2], [2,2], [1], [1]])
sage: krc = elt.to_tensor_product_of_kirillov_reshetikhin_crystals(); krc
[[[2, 3], [3, -2]]]
```

We can recover the rigged configuration:

```
sage: ret = RC(krc); ret

0[ ][ ]0

-2[ ][ ]-2
-2[ ][ ]-2

0[ ]0

0[ ]0

sage: elt == ret
True
```

We can also construct and display a graph of the bijection as follows:

```
sage: ret, G = elt.to_tensor_product_of_kirillov_reshetikhin_crystals(build_graph=True)
sage: view(G, tightpage=True) # not tested
```

to_tensor_product_of_kirillov_reshetikhin_tableaux(*display_steps=False*, *build_graph=False*)

Perform the bijection from this rigged configuration to a tensor product of Kirillov-Reshetikhin tableaux given in [RigConBijection] for single boxes and with [BijectionLRT] and [BijectionDn] for multiple columns and rows.

Note: This is only proven to be a bijection in types $A_n^{(1)}$ and $D_n^{(1)}$, as well as $\bigotimes_i B^{r_i,1}$ and $\bigotimes_i B^{1,s_i}$ for general affine types.

INPUT:

- `display_steps` – (default: `False`) boolean which indicates if we want to print each step in the algorithm
- `build_graph` – (default: `False`) boolean which indicates if we want to construct and return a graph of the bijection whose vertices are rigged configurations obtained at each step and edges are labeled by either the return value of δ or the doubling/halving map

OUTPUT:

- The tensor product of KR tableaux element corresponding to this rigged configuration.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: RC(partition_list=[[2], [2,2], [2], [2]]).to_tensor_product_of_kirillov_reshetikhin_ta
[[3, 3], [5, 5]]
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 2]])
sage: elt = RC(partition_list=[[2], [2,2], [1], [1]])
sage: tp_krt = elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(); tp_krt
[[2, 3], [3, -2]]
```

This is invertible by calling `to_rigged_configuration()`:

```
sage: ret = tp_krt.to_rigged_configuration(); ret

0[ ][ ]0

-2[ ][ ]-2
-2[ ][ ]-2

0[ ]0

0[ ]0

sage: elt == ret
True
```

To view the steps of the bijection in the output, run with the `display_steps=True` option:

```
sage: elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(True)
====================
...
====================

0[ ]0
```

```
-2[ ][ ]-2
 0[ ]0

0[ ]0

0[ ]0

--------------------
[[3, 2]]
--------------------
...
[[2, 3], [3, -2]]
```

We can also construct and display a graph of the bijection as follows:

```
sage: ret, G = elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(build_graph=True)
sage: view(G, tightpage=True) # not tested
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['E', 6, 1], [[2,2]])
sage: map(lambda x: x.weight(), RC.module_generators)
[-4*Lambda[0] + 2*Lambda[2], -2*Lambda[0] + Lambda[2], 0]
sage: KR = crystals.KirillovReshetikhin(['E',6,1], 2,2)
sage: map(lambda x: x.weight(), KR.module_generators) # long time
[0, -2*Lambda[0] + Lambda[2], -4*Lambda[0] + 2*Lambda[2]]

sage: RC = RiggedConfigurations(['D', 6, 1], [[4,2]])
sage: map(lambda x: x.weight(), RC.module_generators)
[-4*Lambda[0] + 2*Lambda[4], -4*Lambda[0] + Lambda[2] + Lambda[4],
 -2*Lambda[0] + Lambda[4], -4*Lambda[0] + 2*Lambda[2],
 -2*Lambda[0] + Lambda[2], 0]
```

class sage.combinat.rigged_configurations.rigged_configuration_element.**RCHWNonSimplyLacedElem**

Bases: sage.combinat.rigged_configurations.rigged_configuration_element.RCNonSimplyLacedE

Rigged configurations in highest weight crystals.

TESTS:

```
sage: La = RootSystem(['C',2,1]).weight_lattice(extended=True).fundamental_weights()
sage: RC = crystals.RiggedConfigurations(['C',2,1], La[0])
sage: elt = RC(partition_list=[[1,1],[2],[2]]); ascii_art(elt)
-1[ ]-1  2[ ][ ]2  -2[ ][ ]-2
-1[ ]-1
sage: TestSuite(elt).run()
```

check()

Make sure all of the riggings are less than or equal to the vacancy number.

TESTS:

```
sage: La = RootSystem(['C',2,1]).weight_lattice(extended=True).fundamental_weights()
sage: RC = crystals.RiggedConfigurations(['C',2,1], La[0])
sage: elt = RC(partition_list=[[1,1],[2],[2]])
sage: elt.check()
```

f(*a*)

Return the action of f_a on `self`.

This works by lifting into the virtual configuration, then applying

$$f_a^v = \prod_{j \in \iota(a)} \hat{f}_j^{\gamma_j}$$

and pulling back.

EXAMPLES:

```
sage: La = RootSystem(['C',2,1]).weight_lattice(extended=True).fundamental_weights()
sage: RC = crystals.RiggedConfigurations(['C',2,1], La[0])
sage: elt = RC(partition_list=[[1,1],[2],[2]])
sage: elt.f(0)
sage: ascii_art(elt.f(1))
0[ ]0   0[ ][ ]0  -1[ ][ ]-1
0[ ]0  -1[ ]-1
sage: elt.f(2)
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: La = RootSystem(['C',2,1]).weight_lattice(extended=True).fundamental_weights()
sage: B = crystals.RiggedConfigurations(['C',2,1], La[0])
sage: mg = B.module_generators[0]
sage: mg.f_string([0,1,2]).weight()
2*Lambda[1] - Lambda[2] - delta
```

class sage.combinat.rigged_configurations.rigged_configuration_element.**RCHighestWeightElement**

Bases: sage.combinat.rigged_configurations.rigged_configuration_element.RiggedConfigurat

Rigged configurations in highest weight crystals.

TESTS:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_weights()
sage: RC = crystals.RiggedConfigurations(['A',2,1], La[0])
sage: elt = RC(partition_list=[[1,1],[1],[2]]); elt

-1[ ]-1
-1[ ]-1

1[ ]1

-1[ ][ ]-1

sage: TestSuite(elt).run()
```

check()

Make sure all of the riggings are less than or equal to the vacancy number.

TESTS:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_weights()
sage: RC = crystals.RiggedConfigurations(['A',2,1], La[0])
sage: elt = RC(partition_list=[[1,1],[1],[2]])
sage: elt.check()
```

f(a)

Return the action of the crystal operator f_a on `self`.

This implements the method defined in [CrysStructSchilling06] which finds the value k which is the length of the string with the smallest nonpositive rigging of largest length. Then it adds a box from a string of length k in the a-th rigged partition, keeping all colabels fixed and decreasing the new label by one. If no such string exists, then it adds a new string of length 1 with label -1. If any of the resulting vacancy numbers are larger than the labels (i.e. it is an invalid rigged configuration), then f_a is undefined.

INPUT:

- a – the index of the partition to add a box

OUTPUT:

The resulting rigged configuration element.

EXAMPLES:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_weights()
sage: RC = crystals.RiggedConfigurations(['A',2,1], La[0])
sage: elt = RC(partition_list=[[1,1],[1],[2]])
sage: elt.f(0)

-2[ ][ ]-2
-1[ ]-1

1[ ]1

0[ ][ ]0

sage: elt.f(1)

0[ ]0
0[ ]0

-1[ ]-1
-1[ ]-1

0[ ][ ]0

sage: elt.f(2)
```

weight()

Return the weight of `self`.

EXAMPLES:

```
sage: La = RootSystem(['A',2,1]).weight_lattice(extended=True).fundamental_weights()
sage: B = crystals.RiggedConfigurations(['A',2,1], La[0])
sage: mg = B.module_generators[0]
```

```
sage: mg.f_string([0,1,2,0]).weight()
-Lambda[0] + Lambda[1] + Lambda[2] - 2*delta
```

class sage.combinat.rigged_configurations.rigged_configuration_element.**RCNonSimplyLacedElemer**

Bases: sage.combinat.rigged_configurations.rigged_configuration_element.RiggedConfigurat

Rigged configuration elements for non-simply-laced types.

TESTS:

```
sage: RC = crystals.infinity.RiggedConfigurations(['C',2,1])
sage: elt = RC.module_generators[0].f_string([1,0,2,2,0,1]); elt

-2[ ][ ]-1

-2[ ]-1
-2[ ]-1

-2[ ][ ]-1

sage: TestSuite(elt).run()
```

e(a)

Return the action of e_a on `self`.

This works by lifting into the virtual configuration, then applying

$$e_a^v = \prod_{j \in \iota(a)} \hat{e}_j^{\gamma_j}$$

and pulling back.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['C',2,1])
sage: elt = RC(partition_list=[[2],[1,1],[2]], rigging_list=[[-1],[-1,-1],[-1]])
sage: ascii_art(elt.e(0))
0[ ]0  -2[ ]-1  -2[ ][ ]-1
       -2[ ]-1
sage: ascii_art(elt.e(1))
-3[ ][ ]-2  0[ ]1  -3[ ][ ]-2
sage: ascii_art(elt.e(2))
-2[ ][ ]-1  -2[ ]-1  0[ ]0
            -2[ ]-1
```

f(a)

Return the action of f_a on `self`.

This works by lifting into the virtual configuration, then applying

$$f_a^v = \prod_{j \in \iota(a)} \hat{f}_j^{\gamma_j}$$

and pulling back.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['C',2,1])
sage: elt = RC(partition_list=[[2],[1,1],[2]], rigging_list=[[-1],[-1,-1],[-1]])
sage: ascii_art(elt.f(0))
-4[ ][ ][ ]-2  -2[ ]-1  -2[ ][ ]-1
                  -2[ ]-1
sage: ascii_art(elt.f(1))
-1[ ][ ]0  -2[ ][ ]-2  -1[ ][ ]0
             -2[ ]-1
sage: ascii_art(elt.f(2))
-2[ ][ ]-1  -2[ ]-1  -4[ ][ ][ ]-2
             -2[ ]-1
```

to_virtual_configuration()

Return the corresponding rigged configuration in the virtual crystal.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',2,1], [[1,2],[1,1],[2,1]])
sage: elt = RC(partition_list=[[3],[2]]); elt

0[ ][ ][ ]0

0[ ][ ]0
sage: elt.to_virtual_configuration()

0[ ][ ][ ]0

0[ ][ ][ ][ ]0

0[ ][ ][ ]0
```

class sage.combinat.rigged_configurations.rigged_configuration_element.**RiggedConfigurationEle**

Bases: sage.structure.list_clone.ClonableArray

A rigged configuration for simply-laced types.

For more information on rigged configurations, see RiggedConfigurations. For rigged configurations for non-simply-laced types, use RCNonSimplyLacedElement.

Typically to create a specific rigged configuration, the user will pass in the optional argument partition_list and if the user wants to specify the rigging values, give the optional argument rigging_list as well. If rigging_list is not passed, the rigging values are set to the corresponding vacancy numbers.

INPUT:

- parent – the parent of this element
- rigged_partitions – a list of rigged partitions

There are two optional arguments to explicitly construct a rigged configuration. The first is partition_list which gives a list of partitions, and the second is rigging_list which is a list of corresponding lists of riggings. If only partition_list is specified, then it sets the rigging equal to the calculated vacancy numbers.

If we are constructing a rigged configuration from a rigged configuration (say of another type) and we don't want to recompute the vacancy numbers, we can use the use_vacancy_numbers to avoid the recomputation.

EXAMPLES:

Type $A_n^{(1)}$ examples:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: RC(partition_list=[[2], [2, 2], [2], [2]])

0[ ][ ]0

-2[ ][ ]-2
-2[ ][ ]-2

2[ ][ ]2

-2[ ][ ]-2


sage: RC = RiggedConfigurations(['A', 4, 1], [[1, 1], [1, 1]])
sage: RC(partition_list=[[], [], [], []])

(/)

(/)

(/)

(/)
```

Type $D_n^{(1)}$ examples:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 2]])
sage: RC(partition_list=[[3], [3,2], [4], [3]])

-1[ ][ ][ ]-1

1[ ][ ][ ]1
0[ ][ ]0

-3[ ][ ][ ][ ]-3

-1[ ][ ][ ]-1


sage: RC = RiggedConfigurations(['D', 4, 1], [[1, 1], [2, 1]])
sage: RC(partition_list=[[1], [1,1], [1], [1]])

1[ ]1

0[ ]0
0[ ]0

0[ ]0

0[ ]0

sage: elt = RC(partition_list=[[1], [1,1], [1], [1]], rigging_list=[[0], [0,0], [0], [0]]); elt

1[ ]0

0[ ]0
```

```
0[ ]0

0[ ]0

0[ ]0
```

```
sage: from sage.combinat.rigged_configurations.rigged_partition import RiggedPartition
sage: RC2 = RiggedConfigurations(['D', 5, 1], [[2, 1], [3, 1]])
sage: l = [RiggedPartition()] + list(elt)
sage: ascii_art(RC2(*l))
(/)  1[ ]0  0[ ]0  0[ ]0  0[ ]0
            0[ ]0
sage: ascii_art(RC2(*l, use_vacancy_numbers=True))
(/)  1[ ]0  0[ ]0  0[ ]0  0[ ]0
            0[ ]0
```

check()

Check the rigged configuration is properly defined.

There is nothing to check here.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['A', 4])
sage: b = RC.module_generators[0].f_string([1,2,1,1,2,4,2,3,3,2])
sage: b.check()
```

e(a)

Return the action of the crystal operator e_a on `self`.

This implements the method defined in [CrysStructSchilling06] which finds the value k which is the length of the string with the smallest negative rigging of smallest length. Then it removes a box from a string of length k in the a-th rigged partition, keeping all colabels fixed and increasing the new label by one. If no such string exists, then e_a is undefined.

INPUT:

- a – the index of the partition to remove a box

OUTPUT:

The resulting rigged configuration element.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2,1]])
sage: elt = RC(partition_list=[[1], [1], [1], [1]])
sage: elt.e(3)
sage: elt.e(1)

(/)

0[ ]0

0[ ]0

-1[ ]-1
```

epsilon(a)

Return ε_a of `self`.

Let x_ℓ be the smallest string of $\nu^{(a)}$ or 0 if $\nu^{(a)} = \emptyset$, then we have $\varepsilon_a = -\min(0, x_\ell)$.

EXAMPLES:

```
sage: La = RootSystem(['B',2]).weight_lattice().fundamental_weights()
sage: RC = crystals.RiggedConfigurations(La[1]+La[2])
sage: I = RC.index_set()
sage: matrix([[rc.epsilon(i) for i in I] for rc in RC[:4]])
[0 0]
[1 0]
[0 1]
[0 2]
```

f(*a*)

Return the action of the crystal operator f_a on `self`.

This implements the method defined in [CrysStructSchilling06] which finds the value k which is the length of the string with the smallest nonpositive rigging of largest length. Then it adds a box from a string of length k in the a-th rigged partition, keeping all colabels fixed and decreasing the new label by one. If no such string exists, then it adds a new string of length 1 with label -1. However we need to modify the definition to work for $B(\infty)$ by removing the condition that the resulting rigged configuration is valid.

INPUT:

- `a` – the index of the partition to add a box

OUTPUT:

The resulting rigged configuration element.

EXAMPLES:

```
sage: RC = crystals.infinity.RiggedConfigurations(['A', 3])
sage: nu0 = RC.module_generators[0]
sage: nu0.f(2)

(/)

-2[ ]-1

(/)
```

nu()

Return the list ν of rigged partitions of this rigged configuration element.

OUTPUT:

The ν array as a list.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: RC(partition_list=[[2], [2,2], [2], [2]]).nu()
[0[ ][ ]0
 , -2[ ][ ]-2
 -2[ ][ ]-2
 , 2[ ][ ]2
 , -2[ ][ ]-2
 ]
```

partition_rigging_lists()

Return the list of partitions and the associated list of riggings of `self`.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',3,1], [[1,2],[2,2]])
sage: rc = RC(partition_list=[[2],[1],[1]], rigging_list=[[-1],[0],[-1]]); rc

-1[ ][ ]-1

1[ ]0

-1[ ]-1

sage: rc.partition_rigging_lists()
[[[2], [1], [1]], [[-1], [0], [-1]]]
```

phi(a)

Return φ_a of `self`.

Let x_ℓ be the smallest string of $\nu^{(a)}$ or 0 if $\nu^{(a)} = \emptyset$, then we have $\varepsilon_a = p_\infty^{(a)} - \min(0, x_\ell)$.

EXAMPLES:

```
sage: La = RootSystem(['B',2]).weight_lattice().fundamental_weights()
sage: RC = crystals.RiggedConfigurations(La[1]+La[2])
sage: I = RC.index_set()
sage: matrix([[rc.phi(i) for i in I] for rc in RC[:4]])
[1 1]
[0 3]
[0 2]
[1 1]
```

vacancy_number(a, i)

Return the vacancy number $p_i^{(a)}$.

INPUT:

- `a` – the index of the rigged partition
- `i` – the row of the rigged partition

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: elt = RC(partition_list=[[1], [2,1], [1], []])
sage: elt.vacancy_number(2, 3)
-2
sage: elt.vacancy_number(2, 2)
-2
sage: elt.vacancy_number(2, 1)
-1

sage: RC = RiggedConfigurations(['D',4,1], [[2,1], [2,1]])
sage: x = RC(partition_list=[[3], [3,1,1], [2], [3,1]]); ascii_art(x)
-1[ ][ ][ ]-1  1[ ][ ][ ]1  0[ ][ ]0  -3[ ][ ][ ]-3
               0[ ]0                  -1[ ]-1
               0[ ]0
sage: x.vacancy_number(2,2)
1
```

5.1.187 Rigged Configurations

AUTHORS:

- Travis Scrimshaw (2010-09-26): Initial version

sage.combinat.rigged_configurations.rigged_configurations.**KirillovReshetikhinCrystal**(*cartan_t*, *r*, *s*)

Return the KR crystal $B^{r,s}$ using rigged configurations.

This is the rigged configuration $RC(B^{r,s})$ or $RC(L)$ with $L = (L_i^{(a)})$ and $L_i^{(a)} = \delta_{a,r}\delta_{i,s}$.

EXAMPLES:

```
sage: K1 = crystals.kirillov_reshetikhin.RiggedConfigurations(['A',6,2], 2, 1)
sage: K2 = crystals.kirillov_reshetikhin.LSPaths(['A',6,2], 2, 1)
sage: K1.digraph().is_isomorphic(K2.digraph(), edge_labels=True)
True
```

TESTS:

We explicitly import and check we get the same crystal:

```
sage: from sage.combinat.rigged_configurations.rigged_configurations import KirillovReshetikhinC
sage: K1 = crystals.kirillov_reshetikhin.RiggedConfigurations(['A',6,2], 2, 1)
sage: K1 is KirillovReshetikhinCrystal(['A',6,2], 2, 1)
True
```

class sage.combinat.rigged_configurations.rigged_configurations.**RCNonSimplyLaced**(*cartan_type*, *dims*)

Bases: sage.combinat.rigged_configurations.rigged_configurations.RiggedConfigurations

Rigged configurations in non-simply-laced types.

These are rigged configurations which lift to virtual rigged configurations in a simply-laced type.

For more on rigged configurations, see RiggedConfigurations.

Element
alias of KRRCNonSimplyLacedElement

from_virtual(*vrc*)
Convert vrc in the virtual crystal into a rigged configution of the original Cartan type.

INPUT:

- vrc – a virtual rigged configuration

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',2,1], [[1,2],[1,1],[2,1]])
sage: elt = RC(partition_list=[[3],[2]])
sage: vrc_elt = RC.to_virtual(elt)
sage: ret = RC.from_virtual(vrc_elt); ret

0[ ][ ][ ]0

0[ ][ ]0
sage: ret == elt
True
```

kleber_tree()
Return the underlying (virtual) Kleber tree used to generate all highest weight rigged configurations.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',3,1], [[1,1], [2,1]])
sage: RC.kleber_tree()
Virtual Kleber tree of Cartan type ['C', 3, 1] and B = ((1, 1), (2, 1))
```

module_generators()

Module generators for this set of rigged configurations.

Iterate over the highest weight rigged configurations by moving through the KleberTree and then setting appropriate values of the partitions.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C', 3, 1], [[1,2]])
sage: for x in RC.module_generators: x

(/)

(/)

(/)


0[ ][ ]0

0[ ][ ]0

0[ ]0


sage: RC = RiggedConfigurations(['D',4,3], [[1,1]])
sage: RC.module_generators
(

     0[ ]0
(/)  0[ ]0

(/)  0[ ]0
   ,
)
```

to_virtual(*rc*)

Convert rc into a rigged configuration in the virtual crystal.

INPUT:

- rc – a rigged configuration element

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',2,1], [[1,2],[1,1],[2,1]])
sage: elt = RC(partition_list=[[3],[2]]); elt

0[ ][ ][ ]0

0[ ][ ]0
sage: velt = RC.to_virtual(elt); velt

0[ ][ ][ ]0

0[ ][ ][ ][ ]0
```

```
0[ ][ ][ ]0
sage: velt.parent()
Rigged configurations of type ['A', 3, 1] and factor(s) ((1, 2), (3, 2), (1, 1), (3, 1), (2,
```

virtual()

Return the corresponding virtual crystal.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['C',2,1], [[1,2],[1,1],[2,1]])
sage: RC
Rigged configurations of type ['C', 2, 1] and factor(s) ((1, 2), (1, 1), (2, 1))
sage: RC.virtual
Rigged configurations of type ['A', 3, 1] and factor(s) ((1, 2), (3, 2), (1, 1), (3, 1), (2,
```

class sage.combinat.rigged_configurations.rigged_configurations.**RCTypeA2Dual**(*cartan_type*, *dims*)

Bases: sage.combinat.rigged_configurations.rigged_configurations.RCTypeA2Even

Rigged configurations of type $A_{2n}^{(2)\dagger}$.

For more on rigged configurations, see RiggedConfigurations.

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[1,2],[1,1],[2,1]])
sage: RC
Rigged configurations of type ['BC', 2, 2]^* and factor(s) ((1, 2), (1, 1), (2, 1))
sage: RC.cardinality()
750
sage: RC.virtual
Rigged configurations of type ['A', 3, 1] and factor(s) ((1, 2), (3, 2), (1, 1), (3, 1), (2, 1),
sage: RC = RiggedConfigurations(CartanType(['A',2,2]).dual(), [[1,1]])
sage: RC.cardinality()
3
sage: RC = RiggedConfigurations(CartanType(['A',2,2]).dual(), [[1,2],[1,1]])
sage: TestSuite(RC).run() # long time
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[2,1]])
sage: TestSuite(RC).run() # long time
```

Element

alias of KRRCTypeA2DualElement

from_virtual(*vrc*)

Convert vrc in the virtual crystal into a rigged configution of the original Cartan type.

INPUT:

- vrc – a virtual rigged configuration element

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[2,2]])
sage: elt = RC(partition_list=[[1],[1]])
sage: velt = RC.to_virtual(elt)
sage: ret = RC.from_virtual(velt); ret

-1[ ]-1

1[ ]1
```

```
sage: ret == elt
True
```

module_generators()

Module generators for rigged configurations of type $A_{2n}^{(2)\dagger}$.

Iterate over the highest weight rigged configurations by moving through the KleberTree and then setting appropriate values of the partitions. This also skips rigged configurations where $P_i^{(n)} < 1$ when i is odd.

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A', 4, 2]).dual(), [[1,1]])
sage: for x in RC.module_generators: x

(/)

(/)
```

to_virtual(*rc*)

Convert rc into a rigged configuration in the virtual crystal.

INPUT:

- rc – a rigged configuration element

EXAMPLES:

```
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[2,2]])
sage: elt = RC(partition_list=[[1],[1]]); elt

-1[ ]-1

1[ ]1

sage: velt = RC.to_virtual(elt); velt

-1[ ]-1

2[ ]2

-1[ ]-1

sage: velt.parent()
Rigged configurations of type ['A', 3, 1] and factor(s) ((2, 2), (2, 2))
```

class sage.combinat.rigged_configurations.rigged_configurations.**RCTypeA2Even**(*cartan_type*, *dims*)

Bases: sage.combinat.rigged_configurations.rigged_configurations.RCNonSimplyLaced

Rigged configurations for type $A_{2n}^{(2)}$.

For more on rigged configurations, see RiggedConfigurations.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',4,2], [[2,1], [1,2]])
sage: RC.cardinality()
150
sage: RC = RiggedConfigurations(['A',2,2], [[1,1]])
sage: RC.cardinality()
3
```

```
sage: RC = RiggedConfigurations(['A',2,2], [[1,2],[1,1]])
sage: TestSuite(RC).run() # long time
sage: RC = RiggedConfigurations(['A',4,2], [[2,1]])
sage: TestSuite(RC).run() # long time
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',4,2], [[1,1], [2,2]])
sage: RC.cardinality()
250
```

from_virtual(*vrc*)

Convert `vrc` in the virtual crystal into a rigged configution of the original Cartan type.

INPUT:

- `vrc` – a virtual rigged configuration element

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',4,2], [[2,2]])
sage: elt = RC(partition_list=[[1],[1]])
sage: velt = RC.to_virtual(elt)
sage: ret = RC.from_virtual(velt); ret

-1[ ]-1

1[ ]1

sage: ret == elt
True
```

to_virtual(*rc*)

Convert `rc` into a rigged configuration in the virtual crystal.

INPUT:

- `rc` – a rigged configuration element

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',4,2], [[2,2]])
sage: elt = RC(partition_list=[[1],[1]]); elt

-1[ ]-1

1[ ]1

sage: velt = RC.to_virtual(elt); velt

-1[ ]-1

2[ ]2

-1[ ]-1

sage: velt.parent()
Rigged configurations of type ['A', 3, 1] and factor(s) ((2, 2), (2, 2))
```

virtual()

Return the corresponding virtual crystal.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',4,2], [[1,2],[1,1],[2,1]])
sage: RC
Rigged configurations of type ['BC', 2, 2] and factor(s) ((1, 2), (1, 1), (2, 1))
sage: RC.virtual
Rigged configurations of type ['A', 3, 1] and factor(s) ((1, 2), (3, 2), (1, 1), (3, 1), (2,
```

class sage.combinat.rigged_configurations.rigged_configurations.**RiggedConfigurations**(*cartan_type*, *B*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Rigged configurations as $U_q'(\mathfrak{g})$-crystals.

Let $\overline{I}$ denote the classical index set associated to the Cartan type of the rigged configurations. A rigged configuration of multiplicity array $L_i^{(a)}$ and dominant weight Λ is a sequence of partitions $\{\nu^{(a)} \mid a \in \overline{I}\}$ such that

$$\sum_{\overline{I} \times \mathbb{Z}_{>0}} i m_i^{(a)} \alpha_a = \sum_{\overline{I} \times \mathbb{Z}_{>0}} i L_i^{(a)} \Lambda_a - \Lambda$$

where α_a is a simple root, Λ_a is a fundamental weight, and $m_i^{(a)}$ is the number of rows of length i in the partition $\nu^{(a)}$.

Each partition $\nu^{(a)}$, in the sequence also comes with a sequence of statistics $p_i^{(a)}$ called *vacancy numbers* and a weakly decreasing sequence $J_i^{(a)}$ of length $m_i^{(a)}$ called *riggings*. Vacancy numbers are computed based upon the partitions and $L_i^{(a)}$, and the riggings must satisfy $\max J_i^{(a)} \leq p_i^{(a)}$. We call such a partition a *rigged partition*. For more, see [RigConBijection] [CrysStructSchilling06] [BijectionLRT].

Rigged configurations form combinatorial objects first introduced by Kerov, Kirillov and Reshetikhin that arose from studies of statistical mechanical models using the Bethe Ansatz. They are sequences of rigged partitions. A rigged partition is a partition together with a label associated to each part that satisfy certain constraints. The labels are also called riggings.

Rigged configurations exist for all affine Kac-Moody Lie algebras. See for example [HKOTT2002]. In Sage they are specified by providing a Cartan type and a list of rectangular shapes B. The list of all (highest weight) rigged configurations for given B is computed via the (virtual) Kleber algorithm (see also KleberTree and VirtualKleberTree).

Rigged configurations in simply-laced types all admit a classical crystal structure [CrysStructSchilling06]. For non-simply-laced types, the crystal is given by using virtual rigged configurations [OSS03]. The highest weight rigged configurations are those where all riggings are nonnegative. The list of all rigged configurations is computed from the highest weight ones using the crystal operators.

Rigged configurations are conjecturally in bijection with TensorProductOfKirillovReshetikhinTableaux of non-exceptional affine types where the list B corresponds to the tensor factors $B^{r,s}$. The bijection has been proven in types $A_n^{(1)}$ and $D_n^{(1)}$ and when the only non-zero entries of $L_i^{(a)}$ are either only $L_1^{(a)}$ or only $L_i^{(1)}$ (corresponding to single columns or rows respectively) [RigConBijection], [BijectionLRT], [BijectionDn].

KR crystals are implemented in Sage, see KirillovReshetikhinCrystal(), however, in the bijection with rigged configurations a different realization of the elements in the crystal are obtained, which are coined KR tableaux, see KirillovReshetikhinTableaux. For more details see [OSS2011].

Note: All non-simply-laced rigged configurations have not been proven to give rise to aligned virtual crystals (i.e. have the correct crystal structure or ismorphic as affine crystals to the tensor product of KR tableaux).

INPUT:

- `cartan_type` – a Cartan type
- `B` – a list of positive integer tuples (r, s) corresponding to the tensor factors in the bijection with tensor product of Kirillov-Reshetikhin tableaux or equivalently the sequence of width s and height r rectangles

REFERENCES:

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [1, 2], [1, 1]])
sage: RC
Rigged configurations of type ['A', 3, 1] and factor(s) ((3, 2), (1, 2), (1, 1))

sage: RC = RiggedConfigurations(['A', 3, 1], [[2,1]]); RC
Rigged configurations of type ['A', 3, 1] and factor(s) ((2, 1),)
sage: RC.cardinality()
6
sage: len(RC.list()) == RC.cardinality()
True
sage: RC.list() # random
[

                                          0[ ]0
(/)  (/)      (/)      -1[ ]-1  -1[ ]-1
                                          -1[ ]-1
(/)  -1[ ]-1  0[ ]0    0[ ]0    1[ ]1    -1[ ]-1

(/)  (/)      -1[ ]-1  (/)      -1[ ]-1  0[ ]0
   ,        ,        ,        ,        ,
]
```

A rigged configuration element with all riggings equal to the vacancy numbers can be created as follows:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3,2], [2,1], [1,1], [1,1]]); RC
Rigged configurations of type ['A', 3, 1] and factor(s) ((3, 2), (2, 1), (1, 1), (1, 1))
sage: elt = RC(partition_list=[[1],[],[]]); elt

0[ ]0

(/)

(/)
```

If on the other hand we also want to specify the riggings, this can be achieved as follows:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [1, 2], [1, 1]])
sage: RC(partition_list=[[2],[2],[2]])

1[ ][ ]1

0[ ][ ]0

0[ ][ ]0
sage: RC(partition_list=[[2],[2],[2]], rigging_list=[[0],[0],[0]])

1[ ][ ]0

0[ ][ ]0
```

```
0[ ][ ]0
```

A larger example:

```
sage: RC = RiggedConfigurations(['D', 7, 1], [[3,3],[5,2],[4,3],[2,3],[4,4],[3,1],[1,4],[2,2]])
sage: elt = RC(partition_list=[[2],[3,2,1],[2,2,1,1],[2,2,1,1,1,1],[3,2,1,1,1,1],[2,1,1],[2,2]],
....:          rigging_list=[[2],[1,0,0],[4,1,2,1],[1,0,0,0,0,0],[0,1,0,0,0,0],[0,0,0],[0,0]])
sage: elt

3[ ][ ]2

1[ ][ ][ ]1
2[ ][ ]0
1[ ]0

4[ ][ ]4
4[ ][ ]1
3[ ]2
3[ ]1

2[ ][ ]1
2[ ][ ]0
0[ ]0
0[ ]0
0[ ]0
0[ ]0

0[ ][ ][ ]0
2[ ][ ]1
0[ ]0
0[ ]0
0[ ]0
0[ ]0

0[ ][ ]0
0[ ]0
0[ ]0

0[ ][ ]0
0[ ][ ]0
```

To obtain the KR tableau under the bijection between rigged configurations and KR tableaux, we can type the following. This example was checked against Reiho Sakamoto's Mathematica program on rigged configurations:

```
sage: output = elt.to_tensor_product_of_kirillov_reshetikhin_tableaux(); output
[[1, 1, 1], [2, 3, 3], [3, 4, -5]] (X) [[1, 1], [2, 2], [3, 3], [5, -6], [6, -5]] (X)
[[1, 1, 2], [2, 2, 3], [3, 3, 7], [4, 4, -7]] (X) [[1, 1, 1], [2, 2, 2]] (X)
[[1, 1, 1, 3], [2, 2, 3, 4], [3, 3, 4, 5], [4, 4, 5, 6]] (X) [[1], [2], [3]] (X) [[1, 1, 1, 1]]
sage: elt.to_tensor_product_of_kirillov_reshetikhin_tableaux().to_rigged_configuration() == elt
True
sage: output.to_rigged_configuration().to_tensor_product_of_kirillov_reshetikhin_tableaux() == o
True
```

We can also convert between rigged configurations and tensor products of KR crystals:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 1]])
sage: elt = RC(partition_list=[[1],[1,1],[1],[1]])
sage: tp_krc = elt.to_tensor_product_of_kirillov_reshetikhin_crystals(); tp_krc
```

```
[[]]
sage: ret = RC(tp_krc)
sage: ret == elt
True

sage: RC = RiggedConfigurations(['D', 4, 1], [[4,1], [3,3]])
sage: KR1 = crystals.KirillovReshetikhin(['D', 4, 1], 4, 1)
sage: KR2 = crystals.KirillovReshetikhin(['D', 4, 1], 3, 3)
sage: T = crystals.TensorProduct(KR1, KR2)
sage: t = T[1]; t
[[++++, []], [+++-, [[1], [2], [4], [-4]]]]
sage: ret = RC(t)
sage: ret.to_tensor_product_of_kirillov_reshetikhin_crystals()
[[++++, []], [+++-, [[1], [2], [4], [-4]]]]
```

TESTS:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3,2], [2,1], [1,1], [1,1]])
sage: len(RC.module_generators)
17
sage: RC = RiggedConfigurations(['D', 4, 1], [[1, 1]])
sage: RC.cardinality()
8

sage: RC = RiggedConfigurations(['D', 4, 1], [[2, 1]])
sage: c = RC.cardinality(); c
29
sage: K = crystals.KirillovReshetikhin(['D',4,1],2,1)
sage: K.cardinality() == c
True
```

Element
alias of KRRCSimplyLacedElement

cardinality()
Return the cardinality of self.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [1, 2]])
sage: RC.cardinality()
100
sage: len(RC.list())
100

sage: RC = RiggedConfigurations(['E', 7, 1], [[1,1]])
sage: RC.cardinality()
134
sage: len(RC.list())
134

sage: RC = RiggedConfigurations(['B', 3, 1], [[2,2],[1,2]])
sage: RC.cardinality()
5130
```

fermionic_formula(*q=None*, *only_highest_weight=False*, *weight=None*)
Return the fermoinic formula associated to self.

Given a set of rigged configurations $RC(\lambda, L)$, the fermonic formula is defined as:

$$M(\lambda, L; q) = \sum_{(\nu,J)} q^{cc(\nu,J)}$$

where we sum over all (classically highest weight) rigged configurations of weight λ where cc is the `cocharge statistic`. This is known to reduce to

$$M(\lambda, L; q) = \sum_{\nu} q^{cc(\nu)} \prod_{(a,i)\in I\times\mathbf{Z}} \begin{bmatrix} p_i^{(a)} + m_i^{(a)} \\ m_i^{(a)} \end{bmatrix}_q .$$

The generating function of $M(\lambda, L; q)$ in the weight algebra subsumes all fermionic formulas:

$$M(L; q) = \sum_{\lambda\in P} M(\lambda, L; q)\lambda.$$

This is conjecturally equal to the `one dimensional configuration sum` of the corresponding tensor product of Kirillov-Reshetikhin crystals, see [HKOTT2002]. This has been proven in general for type $A_n^{(1)}$ [BijectionLRT], single factors $B^{r,s}$ in type $D_n^{(1)}$ [OSS2011] with the result from [Sakamoto13], as well as for a tensor product of single columns [OSS2003], [BijectionDn] or a tensor product of single rows [OSS03] for all non-exceptional types.

INPUT:

- `q` – the variable q
- `only_highest_weight` – use only the classicaly highest weight rigged configurations
- `weight` – return the fermionic formula $M(\lambda, L; q)$ where λ is the classical weight `weight`

REFERENCES:

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 2, 1], [[1,1], [1,1]])
sage: RC.fermionic_formula()
B[-2*Lambda[1] + 2*Lambda[2]] + (q+1)*B[-Lambda[1]]
 + (q+1)*B[Lambda[1] - Lambda[2]] + B[2*Lambda[1]]
 + B[-2*Lambda[2]] + (q+1)*B[Lambda[2]]
sage: t = QQ['t'].gen(0)
sage: RC.fermionic_formula(t)
B[-2*Lambda[1] + 2*Lambda[2]] + (t+1)*B[-Lambda[1]]
 + (t+1)*B[Lambda[1] - Lambda[2]] + B[2*Lambda[1]]
 + B[-2*Lambda[2]] + (t+1)*B[Lambda[2]]
sage: La = RC.weight_lattice_realization().classical().fundamental_weights()
sage: RC.fermionic_formula(weight=La[2])
q + 1
sage: RC.fermionic_formula(only_highest_weight=True, weight=La[2])
q
```

Only using the highest weight elements on other types:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3,1], [2,2]])
sage: RC.fermionic_formula(only_highest_weight=True)
q*B[Lambda[1] + Lambda[2]] + B[2*Lambda[2] + Lambda[3]]
sage: RC = RiggedConfigurations(['D', 4, 1], [[3,1], [4,1], [2,1]])
sage: RC.fermionic_formula(only_highest_weight=True)
(q^4+q^3+q^2)*B[Lambda[1]] + (q^2+q)*B[Lambda[1] + Lambda[2]]
 + q*B[Lambda[1] + 2*Lambda[3]] + q*B[Lambda[1] + 2*Lambda[4]]
 + B[Lambda[2] + Lambda[3] + Lambda[4]] + (q^3+2*q^2+q)*B[Lambda[3] + Lambda[4]]
sage: RC = RiggedConfigurations(['E', 6, 1], [[2,2]])
```

```
sage: RC.fermionic_formula(only_highest_weight=True)
q^2*B[0] + q*B[Lambda[2]] + B[2*Lambda[2]]
sage: RC = RiggedConfigurations(['B', 3, 1], [[3,1], [2,2]])
sage: RC.fermionic_formula(only_highest_weight=True) # long time
q*B[Lambda[1] + Lambda[2] + Lambda[3]] + q^2*B[Lambda[1]
 + Lambda[3]] + (q^2+q)*B[Lambda[2] + Lambda[3]] + B[2*Lambda[2]
 + Lambda[3]] + (q^3+q^2)*B[Lambda[3]]
sage: RC = RiggedConfigurations(['C', 3, 1], [[3,1], [2,2]])
sage: RC.fermionic_formula(only_highest_weight=True) # long time
(q^3+q^2)*B[Lambda[1] + Lambda[2]] + q*B[Lambda[1] + 2*Lambda[2]]
 + (q^2+q)*B[2*Lambda[1] + Lambda[3]] + B[2*Lambda[2] + Lambda[3]]
 + (q^4+q^3+q^2)*B[Lambda[3]]
sage: RC = RiggedConfigurations(['D', 4, 2], [[3,1], [2,2]])
sage: RC.fermionic_formula(only_highest_weight=True) # long time
(q^2+q)*B[Lambda[1] + Lambda[2] + Lambda[3]] + (q^5+2*q^4+q^3)*B[Lambda[1]
 + Lambda[3]] + (q^3+q^2)*B[2*Lambda[1] + Lambda[3]] + (q^4+q^3+q^2)*B[Lambda[2]
 + Lambda[3]] + B[2*Lambda[2] + Lambda[3]] + (q^6+q^5+q^4)*B[Lambda[3]]
sage: RC = RiggedConfigurations(CartanType(['A',4,2]).dual(), [[1,1],[2,2]])
sage: RC.fermionic_formula(only_highest_weight=True)
(q^3+q^2)*B[Lambda[1]] + (q^2+q)*B[Lambda[1] + 2*Lambda[2]]
 + B[Lambda[1] + 4*Lambda[2]] + q*B[3*Lambda[1]] + q*B[4*Lambda[2]]
```

TESTS:

```
sage: RC = RiggedConfigurations(['A', 2, 1], [[1,1], [1,1]])
sage: KR = RC.tensor_product_of_kirillov_reshetikhin_crystals()
sage: RC.fermionic_formula() == KR.one_dimensional_configuration_sum()
True
sage: KT = RC.tensor_product_of_kirillov_reshetikhin_tableaux()
sage: RC.fermionic_formula() == KT.one_dimensional_configuration_sum()
True
sage: RC = RiggedConfigurations(['C', 2, 1], [[2,1], [2,1]])
sage: KR = RC.tensor_product_of_kirillov_reshetikhin_crystals()
sage: RC.fermionic_formula() == KR.one_dimensional_configuration_sum() # long time
True
sage: t = QQ['t'].gen(0)
sage: RC = RiggedConfigurations(['D', 4, 1], [[1,1], [2,1]])
sage: KR = RC.tensor_product_of_kirillov_reshetikhin_crystals()
sage: RC.fermionic_formula(t) == KR.one_dimensional_configuration_sum(t) # long time
True
```

global_options(**get_value*, ***set_value*)

Sets and displays the global options for rigged configurations. If no parameters are set, then the function returns a copy of the options dictionary.

The options to partitions can be accessed as the method RiggedConfigurations.global_options of RiggedConfigurations.

OPTIONS:

- convention – (default: English) Sets the convention used for displaying tableaux and partitions
  - English – use the English convention
  - French – use the French convention
- display – (default: vertical) Specifies how rigged configurations should be printed
  - horizontal – displayed horizontally
  - vertical – displayed vertically

- element_ascii_art – (default: True) display using the repr option element_ascii_art
- half_width_boxes_type_b – (default: True) display the last rigged partition in affine type B as half width boxes
- notation – alternative name for convention

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',3,1], [[2,2],[1,1],[1,1]])
sage: elt = RC(partition_list=[[3,1], [3], [1]])
sage: elt

-3[ ][ ][ ]-3
-1[ ]-1

1[ ][ ][ ]1

-1[ ]-1

sage: RiggedConfigurations.global_options(display="horizontal", convention="french")
sage: elt
-1[ ]-1          1[ ][ ][ ]1    -1[ ]-1
-3[ ][ ][ ]-3
```

Changing the convention for rigged configurations also changes the convention option for tableaux and vice versa:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T.pp()
  4  5
  1  2  3
sage: Tableaux.global_options(convention="english")
sage: elt
-3[ ][ ][ ]-3    1[ ][ ][ ]1    -1[ ]-1
-1[ ]-1
sage: T.pp()
  1  2  3
  4  5
sage: RiggedConfigurations.global_options.reset()
```

See GlobalOptions for more features of these options.

kleber_tree()

Return the underlying Kleber tree used to generate all highest weight rigged configurations.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A',3,1], [[1,1], [2,1]])
sage: RC.kleber_tree()
Kleber tree of Cartan type ['A', 3, 1] and B = ((1, 1), (2, 1))
```

module_generators()

Module generators for this set of rigged configurations.

Iterate over the highest weight rigged configurations by moving through the KleberTree and then setting appropriate values of the partitions.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['D', 4, 1], [[2,1]])
sage: for x in RC.module_generators: x
```

```
(/)

(/)

(/)

(/)


0[ ]0

0[ ]0
0[ ]0

0[ ]0

0[ ]0
```

TESTS:

We check that this works with relabelled Cartan types (trac ticket #16876):

```
sage: ct = CartanType(['A',3,1]).relabel(lambda x: x+2)
sage: RC = RiggedConfigurations(ct, [[4,1],[5,1]])
sage: len(RC.module_generators)
2
sage: ct = CartanType(['A',3,1]).relabel(lambda x: (x+2) % 4)
sage: RC = RiggedConfigurations(ct, [[0,1],[1,1]])
sage: len(RC.module_generators)
2
```

tensor(**crystals*, ***options*)

Return the tensor product of `self` with `crystals`.

If `crystals` is a list of rigged configurations of the same Cartan type, then this returns a new `RiggedConfigurations`.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[2,1],[1,3]])
sage: RC2 = RiggedConfigurations(['A', 3, 1], [[1,1], [3,3]])
sage: RC.tensor(RC2, RC2)
Rigged configurations of type ['A', 3, 1]
 and factor(s) ((2, 1), (1, 3), (1, 1), (3, 3), (1, 1), (3, 3))

sage: K = crystals.KirillovReshetikhin(['A', 3, 1], 2, 2, model='KR')
sage: RC.tensor(K)
Full tensor product of the crystals
 [Rigged configurations of type ['A', 3, 1] and factor(s) ((2, 1), (1, 3)),
  Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (2, 2)]
```

tensor_product_of_kirillov_reshetikhin_crystals()

Return the corresponding tensor product of Kirillov-Reshetikhin crystals.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3,1],[2,2]])
sage: RC.tensor_product_of_kirillov_reshetikhin_crystals()
Full tensor product of the crystals
[Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(3,1),
```

```
Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(2,2)]
```

tensor_product_of_kirillov_reshetikhin_tableaux()
Return the corresponding tensor product of Kirillov-Reshetikhin tableaux.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [1, 2]])
sage: RC.tensor_product_of_kirillov_reshetikhin_tableaux()
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and factor(s) ((3, 2), (
```

5.1.188 Rigged Partitions

Class and methods of the rigged partition which are used by the rigged configuration class. This is an internal class used by the rigged configurations and KR tableaux during the bijection, and is not to be used by the end-user.

We hold the partitions as an 1-dim array of positive integers where each value corresponds to the length of the row. This is the shape of the partition which can be accessed by the regular index.

The data for the vacancy number is also stored in a 1-dim array which each entry corresponds to the row of the tableau, and similarly for the partition values.

AUTHORS:

- Travis Scrimshaw (2010-09-26): Initial version

Todo

Convert this to using multiplicities m_i (perhaps with a dictionary?)?

---

class sage.combinat.rigged_configurations.rigged_partition.**RiggedPartition**(*shape=None*, *rigging_list=None*, *vacancy_nums=None*)

Bases: sage.combinat.combinat.CombinatorialObject

The RiggedPartition class which is the data structure of a rigged (i.e. marked or decorated) Young diagram of a partition.

Note that this class as a stand-alone object does not make sense since the vacancy numbers are calculated using the entire rigged configuration. For more, see RiggedConfigurations.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: RP = RC(partition_list=[[2],[2,2],[2,1],[2]])[2]
sage: RP
 0[ ][ ]0
-1[ ]-1

sage: type(RP)
<class 'sage.combinat.rigged_configurations.rigged_partition.RiggedPartition'>
```

get_num_cells_to_column(*end_column*, *t=1*)
Get the number of cells in all columns before the end_column.

INPUT:

- `end_column` – The index of the column to end at
- `t` – The scaling factor

OUTPUT:

- The number of cells

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: RP = RC(partition_list=[[2],[2,2],[2,1],[2]])[2]
sage: RP.get_num_cells_to_column(1)
2
sage: RP.get_num_cells_to_column(2)
3
sage: RP.get_num_cells_to_column(3)
3
sage: RP.get_num_cells_to_column(3, 2)
5
```

insert_cell(*max_width*)

Insert a cell given at a singular value as long as its less than the specified width.

Note that `insert_cell()` does not update riggings or vacancy numbers, but it does prepare the space for them. Returns the width of the row we inserted at.

INPUT:

- `max_width` – The maximum width (i.e. row length) that we can insert the cell at

OUTPUT:

- The width of the row we inserted at.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: RP = RC(partition_list=[[2],[2,2],[2,1],[2]])[2]
sage: RP.insert_cell(2)
2
sage: RP
 0[ ][ ][ ]None
-1[ ]-1
```

remove_cell(*row*, *num_cells=1*)

Removes a cell at the specified `row`.

Note that `remove_cell()` does not set/update the vacancy numbers or the riggings, but guarantees that the location has been allocated in the returned index.

INPUT:

- `row` – The row to remove the cell from
- `num_cells` – (Default: 1) The number of cells to remove

OUTPUT:

- The location of the newly constructed row or `None` if unable to remove row or if deleted a row.

EXAMPLES:

```
sage: RC = RiggedConfigurations(['A', 4, 1], [[2, 2]])
sage: RP = RC(partition_list=[[2],[2,2],[2,1],[2]])[2]
sage: RP.remove_cell(0)
```

```
0
sage: RP
 0[ ]0
-1[ ]-1
```

class sage.combinat.rigged_configurations.rigged_partition.**RiggedPartitionTypeB**(*arg0*, *arg1=None*, *arg2=None*)

Bases: sage.combinat.rigged_configurations.rigged_partition.RiggedPartition

Rigged partitions for type $B_n^{(1)}$ which has special printing rules which comes from the fact that the n-th partition can have columns of width $\frac{1}{2}$.

5.1.189 Tensor Product of Kirillov-Reshetikhin Tableaux

A tensor product of KirillovReshetikhinTableaux which are tableaux of r rows and s columns which naturally arise in the bijection between rigged configurations and tableaux and which are in bijection with the elements of the Kirillov-Reshetikhin crystal $B^{r,s}$, see KirillovReshetikhinCrystal().

AUTHORS:

- Travis Scrimshaw (2010-09-26): Initial version

EXAMPLES:

Type $A_n^{(1)}$ examples:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[3,1], [2,1]])
sage: KRT
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and factor(s) ((3, 1), (2, 1))
sage: KRT.cardinality()
24
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[1,1], [2,1], [3,1]])
sage: KRT
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and factor(s) ((1, 1), (2, 1), (
sage: len(KRT.module_generators)
5
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[1,1], [2,1], [3,1]])
sage: KRT.cardinality()
96
```

Type $D_n^{(1)}$ examples:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1], [[1, 1], [2, 1], [1, 1]]
sage: KRT
Tensor product of Kirillov-Reshetikhin tableaux of type ['D', 4, 1] and factor(s) ((1, 1), (2, 1), (
sage: T = KRT(pathlist=[[1], [-2, 2], [1]])
sage: T
[[1]] (X) [[2], [-2]] (X) [[1]]
sage: T2 = KRT(pathlist=[[1], [2, -2], [1]])
sage: T2
[[1]] (X) [[-2], [2]] (X) [[1]]
sage: T == T2
False
```

class sage.combinat.rigged_configurations.tensor_product_kr_tableaux.**HighestWeightTensorKRT**(*t*

Bases: sage.structure.unique_representation.UniqueRepresentation

Class so we do not have to build the module generators for TensorProductOfKirillovReshetikhinTableaux at initialization.

Warning: This class is for internal use only!

cardinality()

Return the cardinality of self which is the number of highest weight elements.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D',4,1], [[2,2]])
sage: from sage.combinat.rigged_configurations.tensor_product_kr_tableaux import HighestWeig
sage: HW = HighestWeightTensorKRT(KRT)
sage: HW.cardinality()
3
sage: len(HW)
3
sage: len(KRT.module_generators)
3
```

class sage.combinat.rigged_configurations.tensor_product_kr_tableaux.**TensorProductOfKirillovR**

Bases: sage.combinat.crystals.tensor_product.FullTensorProductOfRegularCrystals

A tensor product of KirillovReshetikhinTableaux.

Through the bijection with rigged configurations, the tableaux that are produced in all nonexceptional types are all of rectangular shapes and do not necessarily obey the usual strict increase in columns and weak increase in rows. The relation between the elements of the Kirillov-Reshetikhin crystal, given by the Kashiwara-Nakashima tableaux, and the Kirillov-Reshetikhin tableaux is given by a filling map.

Note: The tableaux for all non-simply-laced types are provably correct if the bijection with rigged configurations holds. Therefore this is currently only proven for $B^{r,1}$ or $B^{1,s}$ and in general for types $A_n^{(1)}$ and $D_n^{(1)}$.

For more information see [OSS2011] and KirillovReshetikhinTableaux.

For more information on KR crystals, see sage.combinat.crystals.kirillov_reshetikhin.

INPUT:

- cartan_type – a Cartan type
- B – an (ordered) list of pairs (r, s) which give the dimension of a rectangle with r rows and s columns and corresponds to a Kirillov-Reshetikhin tableaux factor of $B^{r,s}$.

REFERENCES:

EXAMPLES:

We can go between tensor products of KR crystals and rigged configurations:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[3,1],[2,2]])
sage: tp_krt = KRT(pathlist=[[3,2,1],[3,2,3,2]]); tp_krt
[[1], [2], [3]] (X) [[2, 2], [3, 3]]
sage: RC = RiggedConfigurations(['A',3,1], [[3,1],[2,2]])
sage: rc_elt = tp_krt.to_rigged_configuration(); rc_elt

-2[ ][ ]-2

0[ ][ ]0
```

```
(/)

sage: tp_krc = tp_krt.to_tensor_product_of_kirillov_reshetikhin_crystals(); tp_krc
[[[1], [2], [3]], [[2, 2], [3, 3]]]
sage: KRT(tp_krc) == tp_krt
True
sage: rc_elt == tp_krt.to_rigged_configuration()
True
sage: KR1 = crystals.KirillovReshetikhin(['A',3,1], 3,1)
sage: KR2 = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: T = crystals.TensorProduct(KR1, KR2)
sage: t = T(KR1(3,2,1), KR2(3,2,3,2))
sage: KRT(t) == tp_krt
True
sage: t == tp_krc
True
```

We can get the highest weight elements by using the attribute `module_generators`:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[3,1], [2,1]])
sage: list(KRT.module_generators)
[[[1], [2], [3]] (X) [[1], [2]], [[1], [3], [4]] (X) [[1], [2]]]
```

To create elements directly (i.e. not passing in KR tableaux elements), there is the **pathlist** option will receive a list of lists which contain the reversed far-eastern reading word of the tableau. That is to say, in English notation, the word obtain from reading bottom-to-top, left-to-right.

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[3,2], [1,2], [2,1]]
sage: elt = KRT(pathlist=[[3, 2, 1, 4, 2, 1], [1, 3], [3, 1]])
sage: elt.pp()
  1  1 (X)    1  3 (X)    1
  2  2                    3
  3  4
```

One can still create elements in the same way as tensor product of crystals:

```
sage: K1 = crystals.KirillovReshetikhin(['A',3,1], 3, 2, model='KR')
sage: K2 = crystals.KirillovReshetikhin(['A',3,1], 1, 2, model='KR')
sage: K3 = crystals.KirillovReshetikhin(['A',3,1], 2, 1, model='KR')
sage: eltlong = KRT(K1(3, 2, 1, 4, 2, 1), K2(1, 3), K3(3, 1))
sage: eltlong == elt
True
```

Element
alias of `TensorProductOfKirillovReshetikhinTableauxElement`

rigged_configurations()
Return the corresponding set of rigged configurations.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[1,3], [2,1]])
sage: KRT.rigged_configurations()
Rigged configurations of type ['A', 3, 1] and factor(s) ((1, 3), (2, 1))
```

tensor(**crystals*, ***options*)
Return the tensor product of `self` with `crystals`.

If `crystals` is a list of (a tensor product of) KR tableaux, this returns a `TensorProductOfKirillovReshetikhinTableaux`.

EXAMPLES:

```
sage: TP = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 3, 1], [[1,3],[3,1]])
sage: K = crystals.KirillovReshetikhin(['A', 3, 1], 2, 2, model='KR')
sage: TP.tensor(K, TP)
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1]
 and factor(s) ((1, 3), (3, 1), (2, 2), (1, 3), (3, 1))

sage: C = crystals.KirillovReshetikhin(['A',3,1], 3, 1, model='KN')
sage: TP.tensor(K, C)
Full tensor product of the crystals
 [Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (1, 3),
  Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (3, 1),
  Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and shape (2, 2),
  Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(3,1)]
```

tensor_product_of_kirillov_reshetikhin_crystals()

Return the corresponding tensor product of Kirillov-Reshetikhin crystals.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[3,1],[2,2]])
sage: KRT.tensor_product_of_kirillov_reshetikhin_crystals()
Full tensor product of the crystals [Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (
Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(2,2)]
```

TESTS:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1], [[4,1], [3,3]])
sage: KR1 = crystals.KirillovReshetikhin(['D', 4, 1], 4, 1)
sage: KR2 = crystals.KirillovReshetikhin(['D', 4, 1], 3, 3)
sage: T = crystals.TensorProduct(KR1, KR2)
sage: T == KRT.tensor_product_of_kirillov_reshetikhin_crystals()
True
sage: T is KRT.tensor_product_of_kirillov_reshetikhin_crystals()
True
```

5.1.190 Tensor Product of Kirillov-Reshetikhin Tableaux Elements

A tensor product of KirillovReshetikhinTableauxElement.

AUTHORS:

- Travis Scrimshaw (2010-09-26): Initial version

class sage.combinat.rigged_configurations.tensor_product_kr_tableaux_element.**TensorProductOfK**

Bases: sage.combinat.crystals.tensor_product.TensorProductOfRegularCrystalsElement

An element in a tensor product of Kirillov-Reshetikhin tableaux.

For more on tensor product of Kirillov-Reshetikhin tableaux, see TensorProductOfKirillovReshetikhinTableaux.

The most common way to construct an element is to specify the option pathlist which is a list of lists which will be used to generate the individual factors of KirillovReshetikhinTableauxElement.

EXAMPLES:

Type $A_n^{(1)}$ examples:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 3, 1], [[1,1], [2,1], [1,1
sage: T = KRT(pathlist=[[2], [4,1], [3], [4,2], [3,1], [2,1]])
sage: T
[[2]] (X) [[1], [4]] (X) [[3]] (X) [[2], [4]] (X) [[1], [3]] (X) [[1], [2]]
sage: T.to_rigged_configuration()

0[ ][ ]0
1[ ]1

1[ ][ ]0
1[ ]0
1[ ]0

0[ ][ ]0

sage: T = KRT(pathlist=[[1], [2,1], [1], [4,1], [3,1], [2,1]])
sage: T
[[1]] (X) [[1], [2]] (X) [[1]] (X) [[1], [4]] (X) [[1], [3]] (X) [[1], [2]]
sage: T.to_rigged_configuration()

(/)

1[ ]0
1[ ]0

0[ ]0
```

Type $D_n^{(1)}$ examples:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1], [[1,1], [1,1], [1,1
sage: T = KRT(pathlist=[[-1], [-1], [1], [1]])
sage: T
[[-1]] (X) [[-1]] (X) [[1]] (X) [[1]]
sage: T.to_rigged_configuration()

0[ ][ ]0
0[ ][ ]0

0[ ][ ]0
0[ ][ ]0

0[ ][ ]0

0[ ][ ]0

sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1], [[2,1], [1,1], [1,1
sage: T = KRT(pathlist=[[3,2], [1], [-1], [1]])
sage: T
[[2], [3]] (X) [[1]] (X) [[-1]] (X) [[1]]
sage: T.to_rigged_configuration()

0[ ]0
0[ ]0
0[ ]0

0[ ]0
0[ ]0
0[ ]0
```

```
1[ ]0

1[ ]0

sage: T.to_rigged_configuration().to_tensor_product_of_kirillov_reshetikhin_tableaux()
[[2], [3]] (X) [[1]] (X) [[-1]] (X) [[1]]
```

classical_weight()

Return the classical weight of `self`.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D',4,1], [[2,2]])
sage: elt = KRT(pathlist=[[3,2,-1,1]]); elt
[[2, 1], [3, -1]]
sage: elt.classical_weight()
(0, 1, 1, 0)
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[2,2],[1,3]])
sage: elt = KRT(pathlist=[[2,1,3,2],[1,4,4]]); elt
[[1, 2], [2, 3]] (X) [[1, 4, 4]]
sage: elt.classical_weight()
(2, 2, 1, 2)
```

left_split()

Return the image of `self` under the left column splitting map.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[2,2],[1,3]])
sage: elt = KRT(pathlist=[[2,1,3,2],[1,4,4]]); elt.pp()
  1  2 (X)   1  4  4
  2  3
sage: elt.left_split().pp()
  1 (X)   2 (X)   1  4  4
  2       3
```

lusztig_involution()

Return the result of the classical Lusztig involution on `self`.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[2,2],[1,3]])
sage: elt = KRT(pathlist=[[2,1,3,2],[1,4,4]])
sage: li = elt.lusztig_involution(); li
[[1, 1, 4]] (X) [[2, 3], [3, 4]]
sage: li.parent()
Tensor product of Kirillov-Reshetikhin tableaux of type ['A', 3, 1] and factor(s) ((1, 3), (
```

pp()

Pretty print `self`.

EXAMPLES:

```
sage: TPKRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',4,1], [[2,2],[3,1],[3
sage: TPKRT.module_generators[0].pp()
  1  1 (X)   1 (X)   1  1  1
  2  2       2       2  2  2
             3       3  3  3
```

right_split()

Return the image of `self` under the right column splitting map.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A',3,1], [[2,2],[1,3]])
sage: elt = KRT(pathlist=[[2,1,3,2],[1,4,4]]); elt.pp()
  1  2 (X)   1  4  4
  2  3
sage: elt.right_split().pp()
  1  2 (X)   1  4 (X)   4
  2  3
```

Let $*$ denote the `Lusztig involution`, we check that $* \circ \mathrm{ls} \circ * = \mathrm{rs}$:

```
sage: all(x.lusztig_involution().left_split().lusztig_involution() == x.right_split() for x
True
```

to_rigged_configuration (*display_steps=False*)

Perform the bijection from `self` to a `rigged configuration` which is described in [RigConBijection], [BijectionLRT], and [BijectionDn].

Note: This is only proven to be a bijection in types $A_n^{(1)}$ and $D_n^{(1)}$, as well as $\bigotimes_i B^{r_i,1}$ and $\bigotimes_i B^{1,s_i}$ for general affine types.

INPUT:

- `display_steps` – (default: `False`) Boolean which indicates if we want to output each step in the algorithm.

OUTPUT:

The rigged configuration corresponding to `self`.

EXAMPLES:

Type $A_n^{(1)}$ example:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['A', 3, 1], [[2,1], [2,1],
sage: T = KRT(pathlist=[[4, 2], [3, 1], [2, 1]])
sage: T
[[2], [4]] (X) [[1], [3]] (X) [[1], [2]]
sage: T.to_rigged_configuration()

0[ ]0

1[ ]1
1[ ]0

0[ ]0
```

Type $D_n^{(1)}$ example:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1], [[2,2]])
sage: T = KRT(pathlist=[[2,1,4,3]])
sage: T
[[1, 3], [2, 4]]
sage: T.to_rigged_configuration()

0[ ]0

-1[ ]-1
-1[ ]-1
```

```
0[ ]0

(/)
```

Type $D_n^{(1)}$ spinor example:

```
sage: CP = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 5, 1], [[5,1],[2,1],[1,
sage: elt = CP(pathlist=[[-2,-5,4,3,1],[-1,2],[1],[1],[1]])
sage: elt
[[1], [3], [4], [-5], [-2]] (X) [[2], [-1]] (X) [[1]] (X) [[1]] (X) [[1]]
sage: elt.to_rigged_configuration()

2[ ][ ]1

0[ ][ ]0
0[ ]0

0[ ][ ]0
0[ ]0

0[ ]0

0[ ][ ]0
```

This is invertible by calling `to_tensor_product_of_kirillov_reshetikhin_tableaux()`:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D', 4, 1], [[2,2]])
sage: T = KRT(pathlist=[[2,1,4,3]])
sage: rc = T.to_rigged_configuration()
sage: ret = rc.to_tensor_product_of_kirillov_reshetikhin_tableaux(); ret
[[1, 3], [2, 4]]
sage: ret == T
True
```

to_tensor_product_of_kirillov_reshetikhin_crystals()

Return a tensor product of Kirillov-Reshetikhin crystals corresponding to `self`.

This works by performing the filling map on each individual factor. For more on the filling map, see `KirillovReshetikhinTableaux`.

EXAMPLES:

```
sage: KRT = crystals.TensorProductOfKirillovReshetikhinTableaux(['D',4,1], [[1,1],[2,2]])
sage: elt = KRT(pathlist=[[-1],[-1,2,-1,1]]); elt
[[-1]] (X) [[2, 1], [-1, -1]]
sage: tp_krc = elt.to_tensor_product_of_kirillov_reshetikhin_crystals(); tp_krc
[[[-1]], [[2], [-1]]]
```

We can recover the original tensor product of KR tableaux:

```
sage: ret = KRT(tp_krc); ret
[[-1]] (X) [[2, 1], [-1, -1]]
sage: ret == elt
True
```

5.1.191 Root Systems

Quickref

- `T = CartanType(["A", 3]), T.is_finite()` – Cartan types
- `T.dynkin_diagram(), DynkinDiagram(["G",2])` – Dynkin diagrams
- `T.cartan_matrix(), CartanMatrix(["F",4])` – Cartan matrices
- `RootSystem(T).weight_lattice()` – Root systems
- `WeylGroup(["B", 6, 1]).simple_reflections()` – Affine Weyl groups
- `WeylCharacterRing(["D", 4])` – Weyl character rings

Introductory material

- *Root Systems* – This overview
- `CartanType` – An introduction to Cartan types
- `RootSystem` – An introduction to root systems
- *Tutorial: visualizing root systems* – A root system visualization tutorial
- The Lie Methods and Related Combinatorics thematic tutorial

Related material

- *Crystals* – Crystals

Cartan datum

- *Cartan types*
- *Dynkin diagrams*
- *Cartan matrices*
- *Coxeter Matrices*
- *Coxeter Types*

Root systems

- *Root systems*
- *Tutorial: visualizing root systems*
- *Root lattice realizations*
- *Group algebras of root lattice realizations*
- *Weight lattice realizations*
- *Root lattices and root spaces*
- *Weight lattices and weight spaces*
- *Ambient lattices and ambient spaces*

Coxeter groups

- *Coxeter Groups*
- *Weyl Groups*
- *Extended Affine Weyl Groups*
- *Fundamental Group of an Extended Affine Weyl Group*

See also:

The categories `CoxeterGroups` and `WeylGroups`

Representation theory

- *Weyl Character Rings*
- *Integrable Representations of Affine Lie Algebras*
- *Branching Rules*
- *Hecke algebra representations*
- *Nonsymmetric Macdonald polynomials*

Root system data and code for specific families of Cartan types

- *Root system data for affine Cartan types*
- *Root system data for dual Cartan types*
- *Root system data for folded Cartan types*
- *Root system data for reducible Cartan types*
- *Root system data for relabelled Cartan types*

Root system data and code for specific Cartan types

- *Root system data for type A*
- *Root system data for type B*
- *Root system data for type C*
- *Root system data for type D*
- *Root system data for type E*
- *Root system data for type F*
- *Root system data for type G*
- *Root system data for type H*
- *Root system data for type I*
- *Root system data for (untwisted) type A affine*
- *Root system data for (untwisted) type B affine*
- *Root system data for (untwisted) type C affine*

- *Root system data for (untwisted) type D affine*
- *Root system data for (untwisted) type E affine*
- *Root system data for (untwisted) type F affine*
- *Root system data for (untwisted) type G affine*
- *Root system data for type BC affine*

5.1.192 Root system features that are imported by default in the interpreter namespace

5.1.193 Ambient lattices and ambient spaces

class sage.combinat.root_system.ambient_space.**AmbientSpace**(*root_system*, *base_ring*)
Bases: sage.misc.cachefunc.ClearCacheOnPickle, sage.combinat.free_module.CombinatorialFree

Abstract class for ambient spaces

All subclasses should implement a class method `smallest_base_ring` taking a Cartan type as input, and a method `dimension` working on a partially initialized instance with just `root_system` as attribute. There is no safe default implementation for the later, so none is provided.

EXAMPLES:

```
sage: AL = RootSystem(['A',2]).ambient_lattice()
```

Note: This is only used so far for finite root systems.

Caveat: Most of the ambient spaces currently have a basis indexed by $0, \ldots, n$, unlike the usual mathematical convention:

```
sage: e = AL.basis()
sage: e[0], e[1], e[2]
((1, 0, 0), (0, 1, 0), (0, 0, 1))
```

This will be cleaned up!

See also:

- sage.combinat.root_system.type_A.AmbientSpace
- sage.combinat.root_system.type_B.AmbientSpace
- sage.combinat.root_system.type_C.AmbientSpace
- sage.combinat.root_system.type_D.AmbientSpace
- sage.combinat.root_system.type_E.AmbientSpace
- sage.combinat.root_system.type_F.AmbientSpace
- sage.combinat.root_system.type_G.AmbientSpace
- sage.combinat.root_system.type_dual.AmbientSpace
- sage.combinat.root_system.type_affine.AmbientSpace

TESTS:

```
sage: types = CartanType.samples(crystallographic = True)+[CartanType(["A",2],["C",5])]
sage: for e in [ct.root_system().ambient_space() for ct in types]:
...           TestSuite(e).run()
```

coroot_lattice()

EXAMPLES:

```
sage: e = RootSystem(["A", 3]).ambient_lattice()
sage: e.coroot_lattice()
Ambient lattice of the Root system of type ['A', 3]
```

dimension()

Returns the dimension of this ambient space.

EXAMPLES:

```
sage: from sage.combinat.root_system.ambient_space import AmbientSpace
sage: e = RootSystem(['F',4]).ambient_space()
sage: AmbientSpace.dimension(e)
Traceback (most recent call last):
...
NotImplementedError
```

from_vector_notation(*weight*, *style='lattice'*)

INPUT:

- `weight` - a vector or tuple representing a weight

Returns an element of self. If the weight lattice is not of full rank, it coerces it into the weight lattice, or its ambient space by orthogonal projection. This arises in two cases: for SL(r+1), the weight lattice is contained in a hyperplane of codimension one in the ambient, space, and for types E6 and E7, the weight lattice is contained in a subspace of codimensions 2 or 3, respectively.

If style="coroots" and the data is a tuple of integers, it is assumed that the data represent a linear combination of fundamental weights. If style="coroots", and the root lattice is not of full rank in the ambient space, it is projected into the subspace corresponding to the semisimple derived group. This arises with Cartan type A, E6 and E7.

EXAMPLES:

```
sage: RootSystem("A2").ambient_space().from_vector_notation((1,0,0))
(1, 0, 0)
sage: RootSystem("A2").ambient_space().from_vector_notation([1,0,0])
(1, 0, 0)
sage: RootSystem("A2").ambient_space().from_vector_notation((1,0),style="coroots")
(2/3, -1/3, -1/3)
```

fundamental_weight(*i*)

Returns the fundamental weight Λ_i in `self`

In several of the ambient spaces, it is more convenient to construct all fundamental weights at once. To support this, we provide this default implementation of `fundamental_weight` using the method `fundamental_weights`. Beware that this will cause a loop if neither `fundamental_weight` nor `fundamental_weights` is implemented.

EXAMPLES:

```
sage: e =  RootSystem(['F',4]).ambient_space()
sage: e.fundamental_weight(3)
(3/2, 1/2, 1/2, 1/2)
```

```
sage: e =  RootSystem(['G',2]).ambient_space()
sage: e.fundamental_weight(1)
(1, 0, -1)

sage: e =  RootSystem(['E',6]).ambient_space()
sage: e.fundamental_weight(3)
(-1/2, 1/2, 1/2, 1/2, 1/2, -5/6, -5/6, 5/6)
```

reflection(*root*, *coroot=None*)

EXAMPLES:

```
sage: e = RootSystem(["A", 3]).ambient_lattice()
sage: a = e.simple_root(0); a
(-1, 0, 0, 0)
sage: b = e.simple_root(1); b
(1, -1, 0, 0)
sage: s_a = e.reflection(a)
sage: s_a(b)
(0, -1, 0, 0)
```

simple_coroot(*i*)

Returns the i-th simple coroot, as an element of this space

EXAMPLES:

```
sage: R = RootSystem(["A",3])
sage: L = R.ambient_lattice()
sage: L.simple_coroot(1)
(1, -1, 0, 0)
sage: L.simple_coroot(2)
(0, 1, -1, 0)
sage: L.simple_coroot(3)
(0, 0, 1, -1)
```

classmethod smallest_base_ring(*cartan_type=None*)

Returns the smallest ground ring over which the ambient space can be realized.

This class method will get called with the Cartan type as input. This default implementation returns $\mathbf{Q}$; subclasses should override it as appropriate.

EXAMPLES:

```
sage: e = RootSystem(['F',4]).ambient_space()
sage: e.smallest_base_ring()
Rational Field
```

to_ambient_space_morphism()

Return the identity map on `self`.

This is present for uniformity of use; the corresponding method for abstract root and weight lattices/spaces, is not trivial.

EXAMPLES:

```
sage: P = RootSystem(['A',2]).ambient_space()
sage: f = P.to_ambient_space_morphism()
sage: p = P.an_element()
sage: p
(2, 2, 3)
sage: f(p)
(2, 2, 3)
```

```
sage: f(p)==p
True
```

class sage.combinat.root_system.ambient_space.**AmbientSpaceElement**(*M*, *x*)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

associated_coroot()

EXAMPLES:

```
sage: e = RootSystem(['F',4]).ambient_space()
sage: a = e.simple_root(0); a
(1/2, -1/2, -1/2, -1/2)
sage: a.associated_coroot()
(1, -1, -1, -1)
```

coerce_to_e6()

For type E7 or E8, orthogonally projects an element of the root lattice into the E6 root lattice. This operation on weights corresponds to intersection with the semisimple subgroup E6.

EXAMPLES:

```
sage: [b.coerce_to_e6() for b in RootSystem("E8").ambient_space().basis()]
[(1, 0, 0, 0, 0, 0, 0, 0), (0, 1, 0, 0, 0, 0, 0, 0), (0, 0, 1, 0, 0, 0, 0, 0),
(0, 0, 0, 1, 0, 0, 0, 0), (0, 0, 0, 0, 1, 0, 0, 0), (0, 0, 0, 0, 0, 1/3, 1/3, -1/3),
(0, 0, 0, 0, 0, 1/3, 1/3, -1/3), (0, 0, 0, 0, 0, -1/3, -1/3, 1/3)]
```

coerce_to_e7()

For type E8, this orthogonally projects the given element of the E8 root lattice into the E7 root lattice. This operation on weights corresponds to intersection with the semisimple subgroup E7.

EXAMPLES:

```
sage: [b.coerce_to_e7() for b in RootSystem("E8").ambient_space().basis()]
[(1, 0, 0, 0, 0, 0, 0, 0), (0, 1, 0, 0, 0, 0, 0, 0),
 (0, 0, 1, 0, 0, 0, 0, 0), (0, 0, 0, 1, 0, 0, 0, 0),
 (0, 0, 0, 0, 1, 0, 0, 0), (0, 0, 0, 0, 0, 1, 0, 0),
 (0, 0, 0, 0, 0, 0, 1/2, -1/2), (0, 0, 0, 0, 0, 0, -1/2, 1/2)]
```

coerce_to_sl()

For type ['A',r], this coerces the element of the ambient space into the root space by orthogonal projection. The root space has codimension one and corresponds to the Lie algebra of SL(r+1,CC), whereas the full weight space corresponds to the Lie algebra of GL(r+1,CC). So this operation corresponds to multiplication by a (possibly fractional) power of the determinant to give a weight determinant one.

EXAMPLES:

```
sage: [fw.coerce_to_sl() for fw in RootSystem("A2").ambient_space().fundamental_weights()]
[(2/3, -1/3, -1/3), (1/3, 1/3, -2/3)]
sage: L = RootSystem("A2xA3").ambient_space()
```

```
sage: L([1,2,3,4,5,0,0]).coerce_to_sl()
(-1, 0, 1, 7/4, 11/4, -9/4, -9/4)
```

dot_product (*lambdacheck*)

The scalar product with elements of the coroot lattice embedded in the ambient space.

EXAMPLES:

```
sage: e = RootSystem(['A',2]).ambient_space()
sage: a = e.simple_root(0); a
(-1, 0, 0)
sage: a.inner_product(a)
2
```

inner_product (*lambdacheck*)

The scalar product with elements of the coroot lattice embedded in the ambient space.

EXAMPLES:

```
sage: e = RootSystem(['A',2]).ambient_space()
sage: a = e.simple_root(0); a
(-1, 0, 0)
sage: a.inner_product(a)
2
```

is_positive_root()

EXAMPLES:

```
sage: R = RootSystem(['A',3]).ambient_space()
sage: r=R.simple_root(1)+R.simple_root(2)
sage: r.is_positive_root()
True
sage: r=R.simple_root(1)-R.simple_root(2)
sage: r.is_positive_root()
False
```

scalar (*lambdacheck*)

The scalar product with elements of the coroot lattice embedded in the ambient space.

EXAMPLES:

```
sage: e = RootSystem(['A',2]).ambient_space()
sage: a = e.simple_root(0); a
(-1, 0, 0)
sage: a.inner_product(a)
2
```

to_ambient()

Map `self` to the ambient space.

This exists for uniformity. Its analogue for root and weight lattice realizations, is not trivial.

EXAMPLES:

```
sage: v = CartanType(['C',3]).root_system().ambient_space().an_element(); v
(2, 2, 3)
sage: v.to_ambient()
(2, 2, 3)
```

5.1.194 Associahedron

Todo

- fix adjacency matrix
- edit graph method to get proper vertex labellings
- UniqueRepresentation?

AUTHORS:

- Christian Stump

class sage.combinat.root_system.associahedron.**Associahedra**(*base_ring*, *ambient_dim*)
Bases: sage.geometry.polyhedron.parent.Polyhedra_QQ_ppl

Parent of Associahedra of specified dimension

EXAMPLES:

```
sage: from sage.combinat.root_system.associahedron import Associahedra
sage: parent = Associahedra(QQ,2);  parent
Polyhedra in QQ^2
sage: type(parent)
<class 'sage.combinat.root_system.associahedron.Associahedra_with_category'>
sage: parent(['A',2])
Generalized associahedron of type ['A', 2] with 5 vertices
```

Importantly, the parent knows the dimension of the ambient space. If you try to construct an associahedron of a different dimension, a ValueError is raised:

```
sage: parent(['A',3])
Traceback (most recent call last):
...
ValueError: V-representation data requires a list of length ambient_dim
```

Element
alias of Associahedron_class

sage.combinat.root_system.associahedron.**Associahedron**(*cartan_type*)
Construct an associahedron.

The generalized associahedron is a polytopal complex with vertices in one-to-one correspondence with clusters in the cluster complex, and with edges between two vertices if and only if the associated two clusters intersect in codimension 1.

The associahedron of type A_n is one way to realize the classical associahedron as defined in the Wikipedia article Associahedron.

A polytopal realization of the associahedron can be found in [CFZ]. The implementation is based on [CFZ], Theorem 1.5, Remark 1.6, and Corollary 1.9.

EXAMPLES:

```
sage: Asso = polytopes.associahedron(['A',2]); Asso
Generalized associahedron of type ['A', 2] with 5 vertices

sage: sorted(Asso.Hrepresentation(), key=repr)
[An inequality (-1, 0) x + 1 >= 0,
 An inequality (0, -1) x + 1 >= 0,
 An inequality (0, 1) x + 1 >= 0,
```

```
 An inequality (1, 0) x + 1 >= 0,
 An inequality (1, 1) x + 1 >= 0]

sage: Asso.Vrepresentation()
(A vertex at (1, -1), A vertex at (1, 1), A vertex at (-1, 1),
 A vertex at (-1, 0), A vertex at (0, -1))

sage: polytopes.associahedron(['B',2])
Generalized associahedron of type ['B', 2] with 6 vertices
```

The two pictures of [CFZ] can be recovered with:

```
sage: Asso = polytopes.associahedron(['A',3]); Asso
Generalized associahedron of type ['A', 3] with 14 vertices
sage: Asso.plot()
Graphics3d Object

sage: Asso = polytopes.associahedron(['B',3]); Asso
Generalized associahedron of type ['B', 3] with 20 vertices
sage: Asso.plot()
Graphics3d Object
```

TESTS:

```
sage: sorted(polytopes.associahedron(['A',3]).vertices())
[A vertex at (-3/2, 0, -1/2), A vertex at (-3/2, 0, 3/2),
 A vertex at (-3/2, 1, -3/2), A vertex at (-3/2, 2, -3/2),
 A vertex at (-3/2, 2, 3/2), A vertex at (-1/2, -1, -1/2),
 A vertex at (-1/2, 0, -3/2), A vertex at (1/2, -2, 1/2),
 A vertex at (1/2, -2, 3/2), A vertex at (3/2, -2, 1/2),
 A vertex at (3/2, -2, 3/2), A vertex at (3/2, 0, -3/2),
 A vertex at (3/2, 2, -3/2), A vertex at (3/2, 2, 3/2)]

sage: sorted(polytopes.associahedron(['B',3]).vertices())
[A vertex at (-3, 0, 0), A vertex at (-3, 0, 3),
 A vertex at (-3, 2, -2), A vertex at (-3, 4, -3),
 A vertex at (-3, 5, -3), A vertex at (-3, 5, 3),
 A vertex at (-2, 1, -2), A vertex at (-2, 3, -3),
 A vertex at (-1, -2, 0), A vertex at (-1, -1, -1),
 A vertex at (1, -4, 1), A vertex at (1, -3, 0),
 A vertex at (2, -5, 2), A vertex at (2, -5, 3),
 A vertex at (3, -5, 2), A vertex at (3, -5, 3),
 A vertex at (3, -3, 0), A vertex at (3, 3, -3),
 A vertex at (3, 5, -3), A vertex at (3, 5, 3)]

sage: polytopes.associahedron(['A',4]).f_vector()
(1, 42, 84, 56, 14, 1)
sage: polytopes.associahedron(['B',4]).f_vector()
(1, 70, 140, 90, 20, 1)
```

class sage.combinat.root_system.associahedron.**Associahedron_class**(*parent*, *Vrep*, *Hrep*, ***kwds*)

Bases: sage.geometry.polyhedron.backend_ppl.Polyhedron_QQ_ppl

The Python class of an associahedron

You should use the Associahedron() convenience function to construct associahedra from the Cartan type.

TESTS:

```
sage: Asso = polytopes.associahedron(['A',2]); Asso
Generalized associahedron of type ['A', 2] with 5 vertices
sage: TestSuite(Asso).run()
```

cartan_type()
Return the Cartan type of `self`.

EXAMPLES:

```
sage: polytopes.associahedron(['A',3]).cartan_type()
['A', 3]
```

vertices_in_root_space()
Return the vertices of `self` as elements in the root space.

EXAMPLES:

```
sage: Asso = polytopes.associahedron(['A',2])
sage: Asso.vertices()
(A vertex at (1, -1), A vertex at (1, 1),
 A vertex at (-1, 1), A vertex at (-1, 0),
 A vertex at (0, -1))

sage: Asso.vertices_in_root_space()
(alpha[1] - alpha[2], alpha[1] + alpha[2], -alpha[1] + alpha[2],
-alpha[1], -alpha[2])
```

5.1.195 Branching Rules

class sage.combinat.root_system.branching_rules.**BranchingRule**(*R*, *S*, *f*, *name='default'*, *intermediate_types=[]*, *intermediate_names=[]*)

Bases: sage.structure.sage_object.SageObject

A class for branching rules.

Rtype()
In a branching rule R => S, returns the Cartan Type of the ambient group R.

EXAMPLES:

```
sage: branching_rule("A3","A2","levi").Rtype()
['A', 3]
```

Stype()
In a branching rule R => S, returns the Cartan Type of the subgroup S.

EXAMPLES:

```
sage: branching_rule("A3","A2","levi").Stype()
['A', 2]
```

branch(*chi*, *style=None*)
INPUT:

- `chi` – A character of the WeylCharacterRing with Cartan type self.Rtype().

Returns the branched character.

EXAMPLES:

```
sage: G2=WeylCharacterRing("G2",style="coroots")
sage: chi=G2(1,1); chi.degree()
64
sage: b=G2.maximal_subgroup("A2"); b
extended branching rule G2 => A2
sage: b.branch(chi)
A2(0,1) + A2(1,0) + A2(0,2) + 2*A2(1,1) + A2(2,0) + A2(1,2) + A2(2,1)
sage: A2=WeylCharacterRing("A2",style="coroots"); A2
The Weyl Character Ring of Type A2 with Integer Ring coefficients
sage: chi.branch(A2,rule=b)
A2(0,1) + A2(1,0) + A2(0,2) + 2*A2(1,1) + A2(2,0) + A2(1,2) + A2(2,1)
```

describe(*verbose=False*, *debug=False*, *no_r=False*)

Describes how extended roots restrict under self.

EXAMPLES:

```
sage: branching_rule("G2","A2","extended").describe()

3
O=<=O---O
1   2   0
G2~

root restrictions G2 => A2:

O---O
1   2
A2

0 => 2
2 => 1

For more detailed information use verbose=True
```

In this example, 0 is the affine root, that is, the negative of the highest root, for "$G2$". If i => j is printed, this means that the i-th simple (or affine) root of the ambient group restricts to the j-th simple root of the subgroup. For reference the Dynkin diagrams are also printed. The extended Dynkin diagram of the ambient group is printed if the affine root restricts to a simple root. More information is printed if the parameter *verbose* is true.

sage.combinat.root_system.branching_rules.**branch_weyl_character**(*chi*, *R*, *S*, *rule='default'*)

A branching rule describes the restriction of representations from a Lie group or algebra G to a subgroup H. See for example, R. C. King, Branching rules for classical Lie groups using tensor and spinor methods. J. Phys. A 8 (1975), 429-449, Howe, Tan and Willenbring, Stable branching rules for classical symmetric pairs, Trans. Amer. Math. Soc. 357 (2005), no. 4, 1601-1626, McKay and Patera, Tables of Dimensions, Indices and Branching Rules for Representations of Simple Lie Algebras (Marcel Dekker, 1981), and Fauser, Jarvis, King and Wybourne, New branching rules induced by plethysm. J. Phys. A 39 (2006), no. 11, 2611–2655. If $H \subset G$ we will write $G \Rightarrow H$ to denote the branching rule, which is a homomorphism of WeylCharacterRings.

INPUT:

- `chi` – a character of G
- `R` – the Weyl Character Ring of G

- S – the Weyl Character Ring of H
- rule – an element of the BranchingRule class or one (most usually) a keyword such as:
  - "levi"
  - "automorphic"
  - "symmetric"
  - "extended"
  - "orthogonal_sum"
  - "tensor"
  - "triality"
  - "miscellaneous"

The BranchingRule class is a wrapper for functions from the weight lattice of G to the weight lattice of H. An instance of this class encodes an embedding of H into G. The usual way to specify an embedding is to supply a keyword, which tells Sage to use one of the built-in rules. We will discuss these first.

To explain the predefined rules, we survey the most important branching rules. These may be classified into several cases, and once this is understood, the detailed classification can be read off from the Dynkin diagrams. Dynkin classified the maximal subgroups of Lie groups in Mat. Sbornik N.S. 30(72):349-462 (1952).

We will list give predefined rules that cover most cases where the branching rule is to a maximal subgroup. For convenience, we also give some branching rules to subgroups that are not maximal. For example, a Levi subgroup may or may not be maximal.

For example, there is a "levi" branching rule defined from $SL(5)$ (with Cartan type A_4) to $SL(4)$ (with Cartan type A_3), so we may compute the branching rule as follows:

EXAMPLES:

```
sage: A3=WeylCharacterRing("A3",style="coroots")
sage: A2=WeylCharacterRing("A2",style="coroots")
sage: [A3(fw).branch(A2,rule="levi") for fw in A3.fundamental_weights()]
[A2(0,0) + A2(1,0), A2(0,1) + A2(1,0), A2(0,0) + A2(0,1)]
```

In this case the Levi branching rule is the default branching rule so we may omit the specification rule="levi".

If a subgroup is not maximal, you may specify a branching rule by finding a chain of intermediate subgroups. For this purpose, branching rules may be multiplied as in the following example.

EXAMPLES:

```
sage: A4=WeylCharacterRing("A4",style="coroots")
sage: A2=WeylCharacterRing("A2",style="coroots")
sage: br=branching_rule("A4","A3")*branching_rule("A3","A2")
sage: A4(1,0,0,0).branch(A2,rule=br)
2*A2(0,0) + A2(1,0)
```

You may try omitting the rule if it is "obvious". Default rules are provided for the following cases:

$$A_{2s} \Rightarrow B_s,$$
$$A_{2s-1} \Rightarrow C_s,$$
$$A_{2*s-1} \Rightarrow D_s.$$

The above default rules correspond to embedding the group $SO(2s+1)$, $Sp(2s)$ or $SO(2s)$ into the corresponding general or special linear group by the standard representation. Default rules are also specified for the

following cases:

$$B_{s+1} \Rightarrow D_s,$$
$$D_s \Rightarrow B_s.$$

These correspond to the embedding of $O(n)$ into $O(n+1)$ where $n = 2s$ or $2s+1$. Finally, the branching rule for the embedding of a Levi subgroup is also implemented as a default rule.

EXAMPLES:

```
sage: A1 = WeylCharacterRing("A1", style="coroots")
sage: A2 = WeylCharacterRing("A2", style="coroots")
sage: D4 = WeylCharacterRing("D4", style="coroots")
sage: B3 = WeylCharacterRing("B3", style="coroots")
sage: B4 = WeylCharacterRing("B4", style="coroots")
sage: A6 = WeylCharacterRing("A6", style="coroots")
sage: A7 = WeylCharacterRing("A7", style="coroots")
sage: def try_default_rule(R,S): return [R(f).branch(S) for f in R.fundamental_weights()]
sage: try_default_rule(A2,A1)
[A1(0) + A1(1), A1(0) + A1(1)]
sage: try_default_rule(D4,B3)
[B3(0,0,0) + B3(1,0,0), B3(1,0,0) + B3(0,1,0), B3(0,0,1), B3(0,0,1)]
sage: try_default_rule(B4,D4)
[D4(0,0,0,0) + D4(1,0,0,0), D4(1,0,0,0) + D4(0,1,0,0),
D4(0,1,0,0) + D4(0,0,1,1), D4(0,0,1,0) + D4(0,0,0,1)]
sage: try_default_rule(A7,D4)
[D4(1,0,0,0), D4(0,1,0,0), D4(0,0,1,1), D4(0,0,2,0) + D4(0,0,0,2),
D4(0,0,1,1),
D4(0,1,0,0),
D4(1,0,0,0)]
sage: try_default_rule(A6,B3)
[B3(1,0,0), B3(0,1,0), B3(0,0,2), B3(0,0,2), B3(0,1,0), B3(1,0,0)]
```

If a default rule is not known, you may cue Sage as to what the Lie group embedding is by supplying a rule from the list of predefined rules. We will treat these next.

Levi Type

These can be read off from the Dynkin diagram. If removing a node from the Dynkin diagram produces another Dynkin diagram, there is a branching rule. A Levi subgroup may or may not be maximal. If it is maximal, there may or may not be a built-in branching rule for but you may obtain the Levi branching rule by first branching to

a suitable maximal subgroup. For these rules use the option `rule="levi"`:

$$
\begin{aligned}
A_r &\Rightarrow A_{r-1} \\
B_r &\Rightarrow A_{r-1} \\
B_r &\Rightarrow B_{r-1} \\
C_r &\Rightarrow A_{r-1} \\
C_r &\Rightarrow C_{r-1} \\
D_r &\Rightarrow A_{r-1} \\
D_r &\Rightarrow D_{r-1} \\
E_r &\Rightarrow A_{r-1} \quad r = 7, 8 \\
E_r &\Rightarrow D_{r-1} \quad r = 6, 7, 8 \\
E_r &\Rightarrow E_{r-1} \\
F_4 &\Rightarrow B_3 \\
F_4 &\Rightarrow C_3 \\
G_2 &\Rightarrow A_1 (\text{short root})
\end{aligned}
$$

Not all Levi subgroups are maximal subgroups. If the Levi is not maximal there may or may not be a pre-programmed `rule="levi"` for it. If there is not, the branching rule may still be obtained by going through an intermediate subgroup that is maximal using rule="extended". Thus the other Levi branching rule from $G_2 \Rightarrow A_1$ corresponding to the long root is available by first branching $G_2 \Rightarrow A_2$ then $A_2 \Rightarrow A_1$. Similarly the branching rules to the Levi subgroup:

$$E_r \Rightarrow A_{r-1} \quad r = 6, 7, 8$$

may be obtained by first branching $E_6 \Rightarrow A_5 \times A_1$, $E_7 \Rightarrow A_7$ or $E_8 \Rightarrow A_8$.

EXAMPLES:

```
sage: A1 = WeylCharacterRing("A1")
sage: A2 = WeylCharacterRing("A2")
sage: A3 = WeylCharacterRing("A3")
sage: A4 = WeylCharacterRing("A4")
sage: A5 = WeylCharacterRing("A5")
sage: B2 = WeylCharacterRing("B2")
sage: B3 = WeylCharacterRing("B3")
sage: B4 = WeylCharacterRing("B4")
sage: C2 = WeylCharacterRing("C2")
sage: C3 = WeylCharacterRing("C3")
sage: D3 = WeylCharacterRing("D3")
sage: D4 = WeylCharacterRing("D4")
sage: G2 = WeylCharacterRing("G2")
sage: F4 = WeylCharacterRing("F4",style="coroots")
sage: E6=WeylCharacterRing("E6",style="coroots")
sage: E7=WeylCharacterRing("E7",style="coroots")
sage: D5=WeylCharacterRing("D5",style="coroots")
sage: D6=WeylCharacterRing("D6",style="coroots")
sage: [B3(w).branch(A2,rule="levi") for w in B3.fundamental_weights()]
[A2(0,0,0) + A2(1,0,0) + A2(0,0,-1),
A2(0,0,0) + A2(1,0,0) + A2(1,1,0) + A2(1,0,-1) + A2(0,-1,-1) + A2(0,0,-1),
A2(-1/2,-1/2,-1/2) + A2(1/2,-1/2,-1/2) + A2(1/2,1/2,-1/2) + A2(1/2,1/2,1/2)]
```

The last example must be understood as follows. The representation of B_3 being branched is spin, which is not a representation of $SO(7)$ but of its double cover $\mathrm{spin}(7)$. The group A_2 is really GL(3) and the double cover of $SO(7)$ induces a cover of $GL(3)$ that is trivial over $SL(3)$ but not over the center of $GL(3)$. The weight lattice for this $GL(3)$ consists of triples (a, b, c) of half integers such that $a - b$ and $b - c$ are in $\mathbf{Z}$, and this is reflected in the last decomposition.

```
sage: [C3(w).branch(A2,rule="levi") for w in C3.fundamental_weights()]
[A2(1,0,0) + A2(0,0,-1),
A2(1,1,0) + A2(1,0,-1) + A2(0,-1,-1),
A2(-1,-1,-1) + A2(1,-1,-1) + A2(1,1,-1) + A2(1,1,1)]
sage: [D4(w).branch(A3,rule="levi") for w in D4.fundamental_weights()]
[A3(1,0,0,0) + A3(0,0,0,-1),
A3(0,0,0,0) + A3(1,1,0,0) + A3(1,0,0,-1) + A3(0,0,-1,-1),
A3(1/2,-1/2,-1/2,-1/2) + A3(1/2,1/2,1/2,-1/2),
A3(-1/2,-1/2,-1/2,-1/2) + A3(1/2,1/2,-1/2,-1/2) + A3(1/2,1/2,1/2,1/2)]
sage: [B3(w).branch(B2,rule="levi") for w in B3.fundamental_weights()]
[2*B2(0,0) + B2(1,0), B2(0,0) + 2*B2(1,0) + B2(1,1), 2*B2(1/2,1/2)]
sage: C3 = WeylCharacterRing(['C',3])
sage: [C3(w).branch(C2,rule="levi") for w in C3.fundamental_weights()]
[2*C2(0,0) + C2(1,0),
 C2(0,0) + 2*C2(1,0) + C2(1,1),
 C2(1,0) + 2*C2(1,1)]
sage: [D5(w).branch(D4,rule="levi") for w in D5.fundamental_weights()]
[2*D4(0,0,0,0) + D4(1,0,0,0),
 D4(0,0,0,0) + 2*D4(1,0,0,0) + D4(1,1,0,0),
 D4(1,0,0,0) + 2*D4(1,1,0,0) + D4(1,1,1,0),
 D4(1/2,1/2,1/2,-1/2) + D4(1/2,1/2,1/2,1/2),
 D4(1/2,1/2,1/2,-1/2) + D4(1/2,1/2,1/2,1/2)]
sage: G2(1,0,-1).branch(A1,rule="levi")
A1(1,0) + A1(1,-1) + A1(0,-1)
sage: E6=WeylCharacterRing("E6",style="coroots")
sage: D5=WeylCharacterRing("D5",style="coroots")
sage: fw = E6.fundamental_weights()
sage: [E6(fw[i]).branch(D5,rule="levi") for i in [1,2,6]]
[D5(0,0,0,0,0) + D5(0,0,0,0,1) + D5(1,0,0,0,0),
 D5(0,0,0,0,0) + D5(0,0,0,1,0) + D5(0,0,0,0,1) + D5(0,1,0,0,0),
 D5(0,0,0,0,0) + D5(0,0,0,1,0) + D5(1,0,0,0,0)]
 sage: E7=WeylCharacterRing("E7",style="coroots")
 sage: A3xA3xA1=WeylCharacterRing("A3xA3xA1",style="coroots")
 sage: E7(1,0,0,0,0,0,0).branch(A3xA3xA1,rule="extended") # long time (0.7s)
 A3xA3xA1(0,0,1,0,0,1,1) + A3xA3xA1(0,1,0,0,1,0,0) + A3xA3xA1(1,0,0,1,0,0,1) +
  A3xA3xA1(1,0,1,0,0,0,0) + A3xA3xA1(0,0,0,1,0,1,0) + A3xA3xA1(0,0,0,0,0,0,2)
sage: fw = E7.fundamental_weights()
sage: [E7(fw[i]).branch(D6,rule="levi") for i in [1,2,7]] # long time (0.3s)
[3*D6(0,0,0,0,0,0) + 2*D6(0,0,0,0,1,0) + D6(0,1,0,0,0,0),
 3*D6(0,0,0,0,0,1) + 2*D6(1,0,0,0,0,0) + 2*D6(0,0,1,0,0,0) + D6(1,0,0,0,1,0),
 D6(0,0,0,0,0,1) + 2*D6(1,0,0,0,0,0)]
sage: D7=WeylCharacterRing("D7",style="coroots")
sage: E8=WeylCharacterRing("E8",style="coroots")
sage: D7=WeylCharacterRing("D7",style="coroots")
sage: E8(1,0,0,0,0,0,0,0).branch(D7,rule="levi") # long time (7s)
 3*D7(0,0,0,0,0,0,0) + 2*D7(0,0,0,0,0,1,0) + 2*D7(0,0,0,0,0,0,1) + 2*D7(1,0,0,0,0,0,0)
 + D7(0,1,0,0,0,0,0) + 2*D7(0,0,1,0,0,0,0) + D7(0,0,0,1,0,0,0) + D7(1,0,0,0,0,1,0) + D7(1,0,0,0,
sage: E8(0,0,0,0,0,0,0,1).branch(D7,rule="levi") # long time (0.6s)
 D7(0,0,0,0,0,0,0) + D7(0,0,0,0,0,1,0) + D7(0,0,0,0,0,0,1) + 2*D7(1,0,0,0,0,0,0) + D7(0,1,0,0,0,
sage: [F4(fw).branch(B3,rule="levi") for fw in F4.fundamental_weights()] # long time (1s)
 [B3(0,0,0) + 2*B3(1/2,1/2,1/2) + 2*B3(1,0,0) + B3(1,1,0),
 B3(0,0,0) + 6*B3(1/2,1/2,1/2) + 5*B3(1,0,0) + 7*B3(1,1,0) + 3*B3(1,1,1)
 + 6*B3(3/2,1/2,1/2) + 2*B3(3/2,3/2,1/2) + B3(2,0,0) + 2*B3(2,1,0) + B3(2,1,1),
 3*B3(0,0,0) + 6*B3(1/2,1/2,1/2) + 4*B3(1,0,0) + 3*B3(1,1,0) + B3(1,1,1) + 2*B3(3/2,1/2,1/2),
 3*B3(0,0,0) + 2*B3(1/2,1/2,1/2) + B3(1,0,0)]
sage: [F4(fw).branch(C3,rule="levi") for fw in F4.fundamental_weights()] # long time (1s)
 [3*C3(0,0,0) + 2*C3(1,1,1) + C3(2,0,0),
 3*C3(0,0,0) + 6*C3(1,1,1) + 4*C3(2,0,0) + 2*C3(2,1,0) + 3*C3(2,2,0) + C3(2,2,2) + C3(3,1,0) + 2
```

```
 2*C3(1,0,0) + 3*C3(1,1,0) + C3(2,0,0) + 2*C3(2,1,0) + C3(2,1,1),
 2*C3(1,0,0) + C3(1,1,0)]
sage: A1xA1 = WeylCharacterRing("A1xA1")
sage: [A3(hwv).branch(A1xA1,rule="levi") for hwv in A3.fundamental_weights()]
[A1xA1(1,0,0,0) + A1xA1(0,0,1,0),
A1xA1(1,1,0,0) + A1xA1(1,0,1,0) + A1xA1(0,0,1,1),
A1xA1(1,1,1,0) + A1xA1(1,0,1,1)]
sage: A1xB1=WeylCharacterRing("A1xB1",style="coroots")
sage: [B3(x).branch(A1xB1,rule="levi") for x in B3.fundamental_weights()]
[2*A1xB1(1,0) + A1xB1(0,2),
3*A1xB1(0,0) + 2*A1xB1(1,2) + A1xB1(2,0) + A1xB1(0,2),
A1xB1(1,1) + 2*A1xB1(0,1)]
```

Automorphic Type

If the Dynkin diagram has a symmetry, then there is an automorphism that is a special case of a branching rule. There is also an exotic "triality" automorphism of D_4 having order 3. Use `rule="automorphic"` (or for D_4 `rule="triality"`):

$$A_r \Rightarrow A_r$$
$$D_r \Rightarrow D_r$$
$$E_6 \Rightarrow E_6$$

EXAMPLES:

```
sage: [A3(chi).branch(A3,rule="automorphic") for chi in A3.fundamental_weights()]
[A3(0,0,0,-1), A3(0,0,-1,-1), A3(0,-1,-1,-1)]
sage: [D4(chi).branch(D4,rule="automorphic") for chi in D4.fundamental_weights()]
[D4(1,0,0,0), D4(1,1,0,0), D4(1/2,1/2,1/2,1/2), D4(1/2,1/2,1/2,-1/2)]
```

Here is an example with D_4 triality:

```
sage: [D4(chi).branch(D4,rule="triality") for chi in D4.fundamental_weights()]
[D4(1/2,1/2,1/2,-1/2), D4(1,1,0,0), D4(1/2,1/2,1/2,1/2), D4(1,0,0,0)]
```

Symmetric Type

Related to the automorphic type, when G admits an outer automorphism (usually of degree 2) we may consider the branching rule to the isotropy subgroup H. Outer automorphisms correspond to symmetries of the Dynkin diagram. For such isotropy subgroups use `rule="symmetric"`. We may thus obtain the following branching rules.

$$A_{2r} \Rightarrow B_r$$
$$A_{2r-1} \Rightarrow C_r$$
$$A_{2r-1} \Rightarrow D_r$$
$$D_r \Rightarrow B_{r-1}$$
$$E_6 \Rightarrow F_4$$
$$E_6 \Rightarrow C_4$$
$$D_4 \Rightarrow G_2$$

The last branching rule, $D_4 \Rightarrow G_2$ is not to a maximal subgroup since $D_4 \Rightarrow B_3 \Rightarrow G_2$, but it is included for convenience.

In some cases, two outer automorphisms that differ by an inner automorphism may have different fixed subgroups. Thus, while the Dynkin diagram of E_6 has a single involutory automorphism, there are two involutions of the group (differing by an inner automorphism) with fixed subgroups F_4 and C_4. Similarly $SL(2r)$, of Cartan type A_{2r-1}, has subgroups $SO(2r)$ and $Sp(2r)$, both fixed subgroups of outer automorphisms that differ from each other by an inner automorphism.

In many cases the Dynkin diagram of H can be obtained by folding the Dynkin diagram of G.

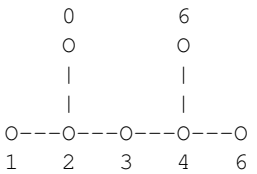
EXAMPLES:

```
sage: [w.branch(B2,rule="symmetric") for w in [A4(1,0,0,0,0),A4(1,1,0,0,0),A4(1,1,1,0,0),A4(2,0,
[B2(1,0), B2(1,1), B2(1,1), B2(0,0) + B2(2,0)]
sage: [A5(w).branch(C3,rule="symmetric") for w in A5.fundamental_weights()]
[C3(1,0,0), C3(0,0,0) + C3(1,1,0), C3(1,0,0) + C3(1,1,1), C3(0,0,0) + C3(1,1,0), C3(1,0,0)]
sage: [A5(w).branch(D3,rule="symmetric") for w in A5.fundamental_weights()]
[D3(1,0,0), D3(1,1,0), D3(1,1,-1) + D3(1,1,1), D3(1,1,0), D3(1,0,0)]
sage: [D4(x).branch(B3,rule="symmetric") for x in D4.fundamental_weights()]
[B3(0,0,0) + B3(1,0,0), B3(1,0,0) + B3(1,1,0), B3(1/2,1/2,1/2), B3(1/2,1/2,1/2)]
sage: [D4(x).branch(G2,rule="symmetric") for x in D4.fundamental_weights()]
[G2(0,0,0) + G2(1,0,-1), 2*G2(1,0,-1) + G2(2,-1,-1), G2(0,0,0) + G2(1,0,-1), G2(0,0,0) + G2(1,0,
sage: [E6(fw).branch(F4,rule="symmetric") for fw in E6.fundamental_weights()] # long time (4s)
[F4(0,0,0,0) + F4(0,0,0,1),
 F4(0,0,0,1) + F4(1,0,0,0),
 F4(0,0,0,1) + F4(1,0,0,0) + F4(0,0,1,0),
 F4(1,0,0,0) + 2*F4(0,0,1,0) + F4(1,0,0,1) + F4(0,1,0,0),
 F4(0,0,0,1) + F4(1,0,0,0) + F4(0,0,1,0),
 F4(0,0,0,0) + F4(0,0,0,1)]
sage: E6=WeylCharacterRing("E6",style="coroots")
sage: C4=WeylCharacterRing("C4",style="coroots")
sage: chi = E6(1,0,0,0,0,0); chi.degree()
27
sage: chi.branch(C4,rule="symmetric")
C4(0,1,0,0)
```

Extended Type

If removing a node from the extended Dynkin diagram results in a Dynkin diagram, then there is a branching rule. Use `rule="extended"` for these. We will also use this classification for some rules that are not of this type, mainly involving type B, such as $D_6 \Rightarrow B_3 \times B_3$.

Here is the extended Dynkin diagram for D_6:



Removing the node 3 results in an embedding $D_3 \times D_3 \Rightarrow D_6$. This corresponds to the embedding $SO(6) \times SO(6) \Rightarrow SO(12)$, and is of extended type. On the other hand the embedding $SO(5) \times SO(7) \Rightarrow SO(12)$ (e.g. $B_2 \times B_3 \Rightarrow D_6$) cannot be explained this way but for uniformity is implemented under `rule="extended"`.

The following rules are implemented as special cases of `rule="extended"`:

$$E_6 \Rightarrow A_5 \times A_1, A_2 \times A_2 \times A_2$$
$$E_7 \Rightarrow A_7, D_6 \times A_1, A_3 \times A_3 \times A_1$$
$$E_8 \Rightarrow A_8, D_8, E_7 \times A_1, A_4 \times A_4, D_5 \times A_3, E_6 \times A_2$$
$$F_4 \Rightarrow B_4, C_3 \times A_1, A_2 \times A_2, A_3 \times A_1$$
$$G_2 \Rightarrow A_1 \times A_1$$

Note that E_8 has only a limited number of representations of reasonably low degree.

EXAMPLES:

```
sage: [B3(x).branch(D3,rule="extended") for x in B3.fundamental_weights()]
[D3(0,0,0) + D3(1,0,0),
 D3(1,0,0) + D3(1,1,0),
 D3(1/2,1/2,-1/2) + D3(1/2,1/2,1/2)]
sage: [G2(w).branch(A2, rule="extended") for w in G2.fundamental_weights()]
[A2(0,0,0) + A2(1/3,1/3,-2/3) + A2(2/3,-1/3,-1/3),
 A2(1/3,1/3,-2/3) + A2(2/3,-1/3,-1/3) + A2(1,0,-1)]
sage: [F4(fw).branch(B4,rule="extended") for fw in F4.fundamental_weights()] # long time (2s)
[B4(1/2,1/2,1/2,1/2) + B4(1,1,0,0),
 B4(1,1,0,0) + B4(1,1,1,0) + B4(3/2,1/2,1/2,1/2) + B4(3/2,3/2,1/2,1/2) + B4(2,1,1,0),
 B4(1/2,1/2,1/2,1/2) + B4(1,0,0,0) + B4(1,1,0,0) + B4(1,1,1,0) + B4(3/2,1/2,1/2,1/2),
 B4(0,0,0,0) + B4(1/2,1/2,1/2,1/2) + B4(1,0,0,0)]

sage: E6 = WeylCharacterRing("E6", style="coroots")
sage: A2xA2xA2=WeylCharacterRing("A2xA2xA2",style="coroots")
sage: A5xA1=WeylCharacterRing("A5xA1",style="coroots")
sage: G2 = WeylCharacterRing("G2", style="coroots")
sage: A1xA1 = WeylCharacterRing("A1xA1", style="coroots")
sage: F4 = WeylCharacterRing("F4",style="coroots")
sage: A3xA1 = WeylCharacterRing("A3xA1", style="coroots")
sage: A2xA2 = WeylCharacterRing("A2xA2", style="coroots")
sage: A1xC3 = WeylCharacterRing("A1xC3",style="coroots")
sage: E6(1,0,0,0,0,0).branch(A5xA1,rule="extended") # (0.7s)
 A5xA1(0,0,0,1,0,0) + A5xA1(1,0,0,0,0,1)
sage: E6(1,0,0,0,0,0).branch(A2xA2xA2, rule="extended") # (0.7s)
A2xA2xA2(0,1,1,0,0,0) + A2xA2xA2(1,0,0,0,0,1) + A2xA2xA2(0,0,0,1,1,0)
sage: E7 = WeylCharacterRing("E7",style="coroots")
sage: A7 = WeylCharacterRing("A7",style="coroots")
sage: E7(1,0,0,0,0,0,0).branch(A7,rule="extended")
 A7(0,0,0,1,0,0,0) + A7(1,0,0,0,0,0,1)
sage: D6xA1 = WeylCharacterRing("D6xA1",style="coroots")
sage: E7(1,0,0,0,0,0,0).branch(D6xA1,rule="extended")
 D6xA1(0,0,0,0,1,0,1) + D6xA1(0,1,0,0,0,0,0) + D6xA1(0,0,0,0,0,0,2)
sage: A5xA2 =WeylCharacterRing("A5xA2",style="coroots")
sage: E7(1,0,0,0,0,0,0).branch(A5xA2,rule="extended")
A5xA2(0,0,0,1,0,1,0) + A5xA2(0,1,0,0,0,0,1) + A5xA2(1,0,0,0,1,0,0) + A5xA2(0,0,0,0,0,1,1)
sage: E8 = WeylCharacterRing("E8",style="coroots")
sage: D8 = WeylCharacterRing("D8",style="coroots")
sage: A8 = WeylCharacterRing("A8",style="coroots")
sage: E8(0,0,0,0,0,0,0,1).branch(D8,rule="extended") # long time (0.56s)
 D8(0,0,0,0,0,0,1,0) + D8(0,1,0,0,0,0,0,0)
sage: E8(0,0,0,0,0,0,0,1).branch(A8,rule="extended") # long time (0.73s)
A8(0,0,0,0,0,1,0,0) + A8(0,0,1,0,0,0,0,0) + A8(1,0,0,0,0,0,0,1)
sage: F4(1,0,0,0).branch(A1xC3,rule="extended") # (0.05s)
 A1xC3(1,0,0,1) + A1xC3(2,0,0,0) + A1xC3(0,2,0,0)
sage: G2(0,1).branch(A1xA1, rule="extended")
 A1xA1(2,0) + A1xA1(3,1) + A1xA1(0,2)
```

```
sage: F4(0,0,0,1).branch(A2xA2, rule="extended") # (0.4s)
 A2xA2(0,1,0,1) + A2xA2(1,0,1,0) + A2xA2(0,0,1,1)
sage: F4(0,0,0,1).branch(A3xA1,rule="extended") # (0.34s)
 A3xA1(0,0,0,0) + A3xA1(0,0,1,1) + A3xA1(0,1,0,0) + A3xA1(1,0,0,1) + A3xA1(0,0,0,2)
sage: D4=WeylCharacterRing("D4",style="coroots")
sage: D2xD2=WeylCharacterRing("D2xD2",style="coroots") # We get D4 => A1xA1xA1xA1 by remembering
sage: [D4(fw).branch(D2xD2, rule="extended") for fw in D4.fundamental_weights()]
 [D2xD2(1,1,0,0) + D2xD2(0,0,1,1),
 D2xD2(2,0,0,0) + D2xD2(0,2,0,0) + D2xD2(1,1,1,1) + D2xD2(0,0,2,0) + D2xD2(0,0,0,2),
 D2xD2(1,0,0,1) + D2xD2(0,1,1,0),
 D2xD2(1,0,1,0) + D2xD2(0,1,0,1)]
```

Orthogonal Sum

Using `rule="orthogonal_sum"`, for $n = a + b + c + \cdots$, you can get any branching rule

$$SO(n) \Rightarrow SO(a) \times SO(b) \times SO(c) \times \cdots,$$
$$Sp(2n) \Rightarrow Sp(2a) \times Sp(2b) \times Sp(2c)x \times \cdots,$$

where $O(a)$ is type D_r for $a = 2r$ or B_r for $a = 2r + 1$ and $Sp(2r)$ is type C_r. In some cases these are also of extended type, as in the case $D_3 \times D_3 \Rightarrow D_6$ discussed above. But in other cases, for example $B_3 \times B_3 \Rightarrow D_7$, they are not of extended type.

Tensor

There are branching rules:

$$\begin{aligned} A_{rs-1} &\Rightarrow A_{r-1} \times A_{s-1}, \\ B_{2rs+r+s} &\Rightarrow B_r \times B_s, \\ D_{2rs+s} &\Rightarrow B_r \times D_s, \\ D_{2rs} &\Rightarrow D_r \times D_s, \\ D_{2rs} &\Rightarrow C_r \times C_s, \\ C_{2rs+s} &\Rightarrow B_r \times C_s, \\ C_{2rs} &\Rightarrow C_r \times D_s. \end{aligned}$$

corresponding to the tensor product homomorphism. For type A, the homomorphism is $GL(r) \times GL(s) \Rightarrow GL(rs)$. For the classical types, the relevant fact is that if V, W are orthogonal or symplectic spaces, that is, spaces endowed with symmetric or skew-symmetric bilinear forms, then $V \otimes W$ is also an orthogonal space (if V and W are both orthogonal or both symplectic) or symplectic (if one of V and W is orthogonal and the other symplectic).

The corresponding branching rules are obtained using `rule="tensor"`.

EXAMPLES:

```
sage: A5=WeylCharacterRing("A5", style="coroots")
sage: A2xA1=WeylCharacterRing("A2xA1", style="coroots")
sage: [A5(hwv).branch(A2xA1, rule="tensor") for hwv in A5.fundamental_weights()]
[A2xA1(1,0,1),
A2xA1(0,1,2) + A2xA1(2,0,0),
A2xA1(1,1,1) + A2xA1(0,0,3),
A2xA1(1,0,2) + A2xA1(0,2,0),
A2xA1(0,1,1)]
```

```
sage: B4=WeylCharacterRing("B4",style="coroots")
sage: B1xB1=WeylCharacterRing("B1xB1",style="coroots")
sage: [B4(f).branch(B1xB1,rule="tensor") for f in B4.fundamental_weights()]
[B1xB1(2,2),
B1xB1(2,0) + B1xB1(2,4) + B1xB1(4,2) + B1xB1(0,2),
B1xB1(2,0) + B1xB1(2,2) + B1xB1(2,4) + B1xB1(4,2) + B1xB1(4,4) + B1xB1(6,0) + B1xB1(0,2) + B1xB1
B1xB1(1,3) + B1xB1(3,1)]
sage: D4=WeylCharacterRing("D4",style="coroots")
sage: C2xC1=WeylCharacterRing("C2xC1",style="coroots")
sage: [D4(f).branch(C2xC1,rule="tensor") for f in D4.fundamental_weights()]
[C2xC1(1,0,1),
C2xC1(0,1,2) + C2xC1(2,0,0) + C2xC1(0,0,2),
C2xC1(1,0,1),
C2xC1(0,1,0) + C2xC1(0,0,2)]
sage: C3=WeylCharacterRing("C3",style="coroots")
sage: B1xC1=WeylCharacterRing("B1xC1",style="coroots")
sage: [C3(f).branch(B1xC1,rule="tensor") for f in C3.fundamental_weights()]
[B1xC1(2,1), B1xC1(2,2) + B1xC1(4,0), B1xC1(4,1) + B1xC1(0,3)]
```

Symmetric Power

The k-th symmetric and exterior power homomorphisms map

$$GL(n) \Rightarrow GL\left(\binom{n+k-1}{k}\right) \times GL\left(\binom{n}{k}\right).$$

The corresponding branching rules are not implemented but a special case is. The k-th symmetric power homomorphism $SL(2) \Rightarrow GL(k+1)$ has its image inside of $SO(2r+1)$ if $k = 2r$ and inside of $Sp(2r)$ if $k = 2r-1$. Hence there are branching rules:

$$B_r \Rightarrow A_1$$
$$C_r \Rightarrow A_1$$

and these may be obtained using the rule "symmetric_power".

EXAMPLES:

```
sage: A1=WeylCharacterRing("A1",style="coroots")
sage: B3=WeylCharacterRing("B3",style="coroots")
sage: C3=WeylCharacterRing("C3",style="coroots")
sage: [B3(fw).branch(A1,rule="symmetric_power") for fw in B3.fundamental_weights()]
[A1(6), A1(2) + A1(6) + A1(10), A1(0) + A1(6)]
sage: [C3(fw).branch(A1,rule="symmetric_power") for fw in C3.fundamental_weights()]
[A1(5), A1(4) + A1(8), A1(3) + A1(9)]
```

Miscellaneous

Use `rule="miscellaneous"` for the following embeddings of maximal subgroups, all involving exceptional groups.

$$
\begin{aligned}
B_3 &\Rightarrow G_2,\\
E_6 &\Rightarrow G_2,\\
E_6 &\Rightarrow A_2,\\
F_4 &\Rightarrow G_2 \times A_1,\\
E_6 &\Rightarrow G_2 \times A_2,\\
E_7 &\Rightarrow G_2 \times C_3,\\
E_7 &\Rightarrow F_4 \times A_1,\\
E_7 &\Rightarrow A_1 \times A_1,\\
E_7 &\Rightarrow G_2 \times A_1,\\
E_8 &\Rightarrow G_2 \times F_4.\\
E_8 &\Rightarrow A2 \times A_1.\\
E_8 &\Rightarrow B2.
\end{aligned}
$$

Except for those embeddings available by `rule="extended"`, these are the only embeddings of these groups as maximal subgroups. There may be other embeddings besides these. For example, there are other more obvious embeddings of A_2 and G_2 into E_6. However the embeddings in this table are characterized as embeddings as maximal subgroups. Regarding the embeddings of A_2 and G_2 in E_6, the embeddings in question may be characterized by the condition that the 27-dimensional representations of E_6 restrict irreducibly to A_2 or G_2. Since G_2 has a subgroup isomorphic to A_2, it is worth mentioning that the composite branching rules:

```
branching_rule("E6","G2","miscellaneous")*branching_rule("G2","A2","extended")
branching_rule("E6","A2","miscellaneous")
```

are distinct.

These embeddings are described more completely (with references to the literature) in the thematic tutorial at:

http://www.sagemath.org/doc/thematic_tutorials/lie.html

EXAMPLES:

```
sage: G2 = WeylCharacterRing("G2")
sage: [fw1, fw2, fw3] = B3.fundamental_weights()
sage: B3(fw1+fw3).branch(G2, rule="miscellaneous")
G2(1,0,-1) + G2(2,-1,-1) + G2(2,0,-2)
sage: E6 = WeylCharacterRing("E6",style="coroots")
sage: G2 = WeylCharacterRing("G2",style="coroots")
sage: E6(1,0,0,0,0,0).branch(G2,"miscellaneous")
G2(2,0)
sage: A2=WeylCharacterRing("A2",style="coroots")
sage: E6(1,0,0,0,0,0).branch(A2,rule="miscellaneous")
A2(2,2)
sage: E6(0,1,0,0,0,0).branch(A2,rule="miscellaneous")
A2(1,1) + A2(1,4) + A2(4,1)
sage: E6(0,0,0,0,0,2).branch(G2,"miscellaneous") # long time (0.59s)
G2(0,0) + G2(2,0) + G2(1,1) + G2(0,2) + G2(4,0)
sage: F4=WeylCharacterRing("F4",style="coroots")
sage: G2xA1=WeylCharacterRing("G2xA1",style="coroots")
sage: F4(0,0,1,0).branch(G2xA1,rule="miscellaneous")
G2xA1(1,0,0) + G2xA1(1,0,2) + G2xA1(1,0,4) + G2xA1(1,0,6) + G2xA1(0,1,4) + G2xA1(2,0,2) + G2xA1(
sage: E6 = WeylCharacterRing("E6",style="coroots")
```

```
sage: A2xG2 = WeylCharacterRing("A2xG2",style="coroots")
sage: E6(1,0,0,0,0,0).branch(A2xG2,rule="miscellaneous")
A2xG2(0,1,1,0) + A2xG2(2,0,0,0)
sage: E7=WeylCharacterRing("E7",style="coroots")
sage: G2xC3=WeylCharacterRing("G2xC3",style="coroots")
sage: E7(0,1,0,0,0,0,0).branch(G2xC3,rule="miscellaneous") # long time (1.84s)
G2xC3(1,0,1,0,0) + G2xC3(1,0,1,1,0) + G2xC3(0,1,0,0,1) + G2xC3(2,0,1,0,0) + G2xC3(0,0,1,1,0)
sage: F4xA1=WeylCharacterRing("F4xA1",style="coroots")
sage: E7(0,0,0,0,0,0,1).branch(F4xA1,"miscellaneous")
F4xA1(0,0,0,1,1) + F4xA1(0,0,0,0,3)
sage: A1xA1=WeylCharacterRing("A1xA1",style="coroots")
sage: E7(0,0,0,0,0,0,1).branch(A1xA1,rule="miscellaneous")
A1xA1(2,5) + A1xA1(4,1) + A1xA1(6,3)
sage: A2=WeylCharacterRing("A2",style="coroots")
sage: E7(0,0,0,0,0,0,1).branch(A2,rule="miscellaneous")
A2(0,6) + A2(6,0)
sage: G2xA1=WeylCharacterRing("G2xA1",style="coroots")
sage: E7(1,0,0,0,0,0,0).branch(G2xA1,rule="miscellaneous")
G2xA1(1,0,4) + G2xA1(0,1,0) + G2xA1(2,0,2) + G2xA1(0,0,2)
sage: E8 = WeylCharacterRing("E8",style="coroots")
sage: G2xF4 = WeylCharacterRing("G2xF4",style="coroots")
sage: E8(0,0,0,0,0,0,0,1).branch(G2xF4,rule="miscellaneous") # long time (0.76s)
G2xF4(1,0,0,0,0,1) + G2xF4(0,1,0,0,0,0) + G2xF4(0,0,1,0,0,0)
sage: E8=WeylCharacterRing("E8",style="coroots")
sage: A1xA2=WeylCharacterRing("A1xA2",style="coroots")
sage: E8(0,0,0,0,0,0,0,1).branch(A1xA2,rule="miscellaneous") # long time (0.76s)
A1xA2(2,0,0) + A1xA2(2,2,2) + A1xA2(4,0,3) + A1xA2(4,3,0) + A1xA2(6,1,1) + A1xA2(0,1,1)
sage: B2=WeylCharacterRing("B2",style="coroots")
sage: E8(0,0,0,0,0,0,0,1).branch(B2,rule="miscellaneous") # long time (0.53s)
B2(0,2) + B2(0,6) + B2(3,2)
```

A1 maximal subgroups of exceptional groups

There are seven cases where the exceptional group G_2, F_4, E_7 or E_8 contains a maximal subgroup of type A_1. These are tabulated in Theorem 1 of Testerman, The construction of the maximal A1's in the exceptional algebraic groups, Proc. Amer. Math. Soc. 116 (1992), no. 3, 635-644. The names of these branching rules are roman numerals referring to the seven cases of her Theorem 1. Use these branching rules as in the following examples.

EXAMPLES:

```
sage: A1=WeylCharacterRing("A1",style="coroots")
sage: G2=WeylCharacterRing("G2",style="coroots")
sage: F4=WeylCharacterRing("F4",style="coroots")
sage: E7=WeylCharacterRing("E7",style="coroots")
sage: E8=WeylCharacterRing("E8",style="coroots")
sage: [G2(f).branch(A1,rule="i") for f in G2.fundamental_weights()]
[A1(6), A1(2) + A1(10)]
sage: F4(1,0,0,0).branch(A1,rule="ii")
A1(2) + A1(10) + A1(14) + A1(22)
sage: E7(0,0,0,0,0,0,1).branch(A1,rule="iii")
A1(9) + A1(17) + A1(27)
sage: E7(0,0,0,0,0,0,1).branch(A1,rule="iv")
A1(5) + A1(11) + A1(15) + A1(21)
sage: E8(0,0,0,0,0,0,0,1).branch(A1,rule="v") # long time (0.6s)
A1(2) + A1(14) + A1(22) + A1(26) + A1(34) + A1(38) + A1(46) + A1(58)
sage: E8(0,0,0,0,0,0,0,1).branch(A1,rule="vi") # long time (0.6s)
```

```
A1(2) + A1(10) + A1(14) + A1(18) + A1(22) + A1(26) + A1(28) + A1(34) + A1(38) + A1(46)
sage: E8(0,0,0,0,0,0,0,1).branch(A1,rule="vii") # long time (0.6s)
A1(2) + A1(6) + A1(10) + A1(14) + A1(16) + A1(18) + 2*A1(22) + A1(26) + A1(28) + A1(34) + A1(38)
```

Branching Rules From Plethysms

Nearly all branching rules $G \Rightarrow H$ where G is of type A, B, C or D are covered by the preceding rules. The function `branching_rule_from_plethysm()` covers the remaining cases.

This is a general rule that includes any branching rule from types A, B, C, or D as a special case. Thus it could be used in place of the above rules and would give the same results. However it is most useful when branching from G to a maximal subgroup H such that $\mathrm{rank}(H) < \mathrm{rank}(G) - 1$.

We consider a homomorphism $H \Rightarrow G$ where G is one of $SL(r+1)$, $SO(2r+1)$, $Sp(2r)$ or $SO(2r)$. The function `branching_rule_from_plethysm()` produces the corresponding branching rule. The main ingredient is the character χ of the representation of H that is the homomorphism to $GL(r+1)$, $GL(2r+1)$ or $GL(2r)$.

This rule is so powerful that it contains the other rules implemented above as special cases. First let us consider the symmetric fifth power representation of $SL(2)$.

```
sage: A1=WeylCharacterRing("A1",style="coroots")
sage: chi=A1([5])
sage: chi.degree()
 6
sage: chi.frobenius_schur_indicator()
-1
```

This confirms that the character has degree 6 and is symplectic, so it corresponds to a homomorphism $SL(2) \Rightarrow Sp(6)$, and there is a corresponding branching rule $C_3 \Rightarrow A_1$.

```
sage: C3 = WeylCharacterRing("C3",style="coroots")
sage: sym5rule = branching_rule_from_plethysm(chi,"C3")
sage: [C3(hwv).branch(A1,rule=sym5rule) for hwv in C3.fundamental_weights()]
[A1(5), A1(4) + A1(8), A1(3) + A1(9)]
```

This is identical to the results we would obtain using `rule="symmetric_power"`. The next example gives a branching not available by other standard rules.

```
sage: G2 = WeylCharacterRing("G2",style="coroots")
sage: D7 = WeylCharacterRing("D7",style="coroots")
sage: ad=G2(0,1); ad.degree(); ad.frobenius_schur_indicator()
 14
 1
sage: spin = D7(0,0,0,0,0,1,0); spin.degree()
 64
sage: spin.branch(G2, rule=branching_rule_from_plethysm(ad, "D7"))
 G2(1,1)
```

We have confirmed that the adjoint representation of G_2 gives a homomorphism into $SO(14)$, and that the pullback of the one of the two 64 dimensional spin representations to $SO(14)$ is an irreducible representation of G_2.

We do not actually have to create the character or its parent WeylCharacterRing to create the branching rule:

```
sage: b = branching_rule("C7","C3(0,0,1)","plethysm"); b
plethysm (along C3(0,0,1)) branching rule C7 => C3
```

Isomorphic Type

Although not usually referred to as a branching rule, the effects of the accidental isomorphisms may be handled using `rule="isomorphic"`:

$$B_2 \Rightarrow C_2$$
$$C_2 \Rightarrow B_2$$
$$A_3 \Rightarrow D_3$$
$$D_3 \Rightarrow A_3$$
$$D_2 \Rightarrow A_1 \Rightarrow A_1$$
$$B_1 \Rightarrow A_1$$
$$C_1 \Rightarrow A_1$$

EXAMPLES:

```
sage: B2 = WeylCharacterRing("B2")
sage: C2 = WeylCharacterRing("C2")
sage: [B2(x).branch(C2, rule="isomorphic") for x in B2.fundamental_weights()]
[C2(1,1), C2(1,0)]
sage: [C2(x).branch(B2, rule="isomorphic") for x in C2.fundamental_weights()]
[B2(1/2,1/2), B2(1,0)]
sage: D3 = WeylCharacterRing("D3")
sage: A3 = WeylCharacterRing("A3")
sage: [A3(x).branch(D3,rule="isomorphic") for x in A3.fundamental_weights()]
[D3(1/2,1/2,1/2), D3(1,0,0), D3(1/2,1/2,-1/2)]
sage: [D3(x).branch(A3,rule="isomorphic") for x in D3.fundamental_weights()]
[A3(1/2,1/2,-1/2,-1/2), A3(1/4,1/4,1/4,-3/4), A3(3/4,-1/4,-1/4,-1/4)]
```

Here $A_3(x,y,z,w)$ can be understood as a representation of $SL(4)$. The weights x,y,z,w and $x+t,y+t,z+t,w+t$ represent the same representation of $SL(4)$ - though not of $GL(4)$ - since $A_3(x+t,y+t,z+t,w+t)$ is the same as $A_3(x,y,z,w)$ tensored with $\det^t$. So as a representation of $SL(4)$, `A3(1/4,1/4,1/4,-3/4)` is the same as `A3(1,1,1,0)`. The exterior square representation $SL(4) \Rightarrow GL(6)$ admits an invariant symmetric bilinear form, so is a representation $SL(4) \Rightarrow SO(6)$ that lifts to an isomorphism $SL(4) \Rightarrow \mathrm{Spin}(6)$. Conversely, there are two isomorphisms $SO(6) \Rightarrow SL(4)$, of which we've selected one.

In cases like this you might prefer `style="coroots"`:

```
sage: A3 = WeylCharacterRing("A3",style="coroots")
sage: D3 = WeylCharacterRing("D3",style="coroots")
sage: [D3(fw) for fw in D3.fundamental_weights()]
[D3(1,0,0), D3(0,1,0), D3(0,0,1)]
sage: [D3(fw).branch(A3,rule="isomorphic") for fw in D3.fundamental_weights()]
[A3(0,1,0), A3(0,0,1), A3(1,0,0)]
sage: D2 = WeylCharacterRing("D2", style="coroots")
sage: A1xA1 = WeylCharacterRing("A1xA1", style="coroots")
sage: [D2(fw).branch(A1xA1,rule="isomorphic") for fw in D2.fundamental_weights()]
[A1xA1(1,0), A1xA1(0,1)]
```

Branching From a Reducible WeylCharacterRing

If the Cartan Type of R is reducible, we may project a character onto any of the components, or any combination of components. The rule to project on the first component is specified by the string `"proj1"`, the rule to project on the second component is `"proj2"`. To project on the first and third components, use ``"proj13"` and so on.

EXAMPLES:

```
sage: A2xG2=WeylCharacterRing("A2xG2",style="coroots")
sage: A2=WeylCharacterRing("A2",style="coroots")
sage: G2=WeylCharacterRing("G2",style="coroots")
sage: A2xG2(1,0,1,0).branch(A2,rule="proj1")
7*A2(1,0)
sage: A2xG2(1,0,1,0).branch(G2,rule="proj2")
3*G2(1,0)
sage: A2xA2xG2=WeylCharacterRing("A2xA2xG2",style="coroots")
sage: A2xA2xG2(0,1,1,1,0,1).branch(A2xG2,rule="proj13")
8*A2xG2(0,1,0,1)
```

A more general way of specifying a branching rule from a reducible type is to supply a *list* of rules, one *component rule* for each component type in the root system. In the following example, we branch the fundamental representations of D_4 down to $A_1 \times A_1 \times A_1 \times A_1$ through the intermediate group $D_2 \times D_2$. We use multiplicative notation to compose the branching rules. There is no need to construct the intermediate WeylCharacterRing with type $D_2 \times D_2$.

EXAMPLES:

```
sage: D4 = WeylCharacterRing("D4",style="coroots")
sage: A1xA1xA1xA1 = WeylCharacterRing("A1xA1xA1xA1",style="coroots")
sage: b = branching_rule("D2","A1xA1","isomorphic")
sage: br = branching_rule("D4","D2xD2","extended")*branching_rule("D2xD2","A1xA1xA1xA1",[b,b])
sage: [D4(fw).branch(A1xA1xA1xA1,rule=br) for fw in D4.fundamental_weights()]
[A1xA1xA1xA1(1,1,0,0) + A1xA1xA1xA1(0,0,1,1),
A1xA1xA1xA1(1,1,1,1) + A1xA1xA1xA1(2,0,0,0) + A1xA1xA1xA1(0,2,0,0) + A1xA1xA1xA1(0,0,2,0) + A1xA
A1xA1xA1xA1(1,0,0,1) + A1xA1xA1xA1(0,1,1,0),
A1xA1xA1xA1(1,0,1,0) + A1xA1xA1xA1(0,1,0,1)]
```

In the list of rules to be supplied in branching from a reducible root system, we may use two key words "omit" and "identity". The term "omit" means that we omit one factor, projecting onto the remaining factors. The term "identity" is supplied when the irreducible factor Cartan Types of both the target and the source are the same, and the component branching rule is to be the identity map. For example, we have projection maps from $A_3 \times A_2$ to A_3 and A_2, and the corresponding branching may be accomplished as follows. In this example the same could be accomplished using `rule="proj2"`.

EXAMPLES:

```
sage: A3xA2=WeylCharacterRing("A3xA2",style="coroots")
sage: A3=WeylCharacterRing("A3",style="coroots")
sage: chi = A3xA2(0,1,0,1,0)
sage: chi.branch(A3,rule=["identity","omit"])
3*A3(0,1,0)
sage: A2=WeylCharacterRing("A2",style="coroots")
sage: chi.branch(A2,rule=["omit","identity"])
6*A2(1,0)
```

Yet another way of branching from a reducible root system with repeated Cartan types is to embed along the diagonal. The branching rule is equivalent to the tensor product, as the example shows:

```
sage: G2=WeylCharacterRing("G2",style="coroots")
sage: G2xG2=WeylCharacterRing("G2xG2",style="coroots")
sage: G2=WeylCharacterRing("G2",style="coroots")
sage: G2xG2(1,0,0,1).branch(G2,rule="diagonal")
G2(1,0) + G2(2,0) + G2(1,1)
sage: G2xG2(1,0,0,1).branch(G2,rule="diagonal") == G2(1,0)*G2(0,1)
True
```

Writing Your Own (Branching) Rules

Suppose you want to branch from a group G to a subgroup H. Arrange the embedding so that a Cartan subalgebra U of H is contained in a Cartan subalgebra T of G. There is thus a mapping from the weight spaces $\mathrm{Lie}(T)^* \Rightarrow \mathrm{Lie}(U)^*$. Two embeddings will produce identical branching rules if they differ by an element of the Weyl group of H.

The *rule* is this map $\mathrm{Lie}(T)^*$, which is `G.space()`, to $\mathrm{Lie}(U)^*$, which is `H.space()`, which you may implement as a function. As an example, let us consider how to implement the branching rule $A_3 \Rightarrow C_2$. Here $H = C_2 = Sp(4)$ embedded as a subgroup in $A_3 = GL(4)$. The Cartan subalgebra U consists of diagonal matrices with eigenvalues $u_1, u_2, -u_2, -u_1$. The `C2.space()` is the two dimensional vector spaces consisting of the linear functionals u_1 and u_2 on U. On the other hand $\mathrm{Lie}(T)$ is $\mathbf{R}^4$. A convenient way to see the restriction is to think of it as the adjoint of the map $(u_1, u_2) \mapsto (u_1, u_2, -u_2, -u_1)$, that is, $(x_0, x_1, x_2, x_3) \Rightarrow (x_0 - x_3, x_1 - x_2)$. Hence we may encode the rule as follows:

```
def rule(x):
    return [x[0]-x[3],x[1]-x[2]]
```

or simply:

```
rule = lambda x : [x[0]-x[3],x[1]-x[2]]
```

We may now make and use the branching rule as follows.

EXAMPLES:

```
sage: br = BranchingRule("A3", "C2", lambda x : [x[0]-x[3],x[1]-x[2]], "homemade"); br
homemade branching rule A3 => C2
sage: [A3,C2]=[WeylCharacterRing(x,style="coroots") for x in ["A3","C2"]]
sage: A3(0,1,0).branch(C2,rule=br)
C2(0,0) + C2(0,1)
```

sage.combinat.root_system.branching_rules.**branching_rule**(*Rtype*, *Stype*, *rule='default'*)

Creates a branching rule.

INPUT:

- `R` – the Weyl Character Ring of G
- `S` – the Weyl Character Ring of H
- `rule` – a string describing the branching rule as a map from the weight space of S to the weight space of R.

If the rule parameter is omitted, in some cases, a default rule is supplied. See `branch_weyl_character()`.

EXAMPLES:

```
sage: rule = branching_rule(CartanType("A3"),CartanType("C2"),"symmetric")
sage: [rule(x) for x in WeylCharacterRing("A3").fundamental_weights()]
[[1, 0], [1, 1], [1, 0]]
```

sage.combinat.root_system.branching_rules.**branching_rule_from_plethysm**(*chi*, *cartan_type*, *return_matrix=False*)

Create the branching rule of a plethysm.

INPUT:

- chi – the character of an irreducible representation π of a group G
- cartan_type – a classical Cartan type (A,'B','C' or D).

It is assumed that the image of the irreducible representation pi naturally has its image in the group G.

Returns a branching rule for this plethysm.

EXAMPLES:

The adjoint representation $SL(3) \to GL(8)$ factors through $SO(8)$. The branching rule in question will describe how representations of $SO(8)$ composed with this homomorphism decompose into irreducible characters of $SL(3)$:

```
sage: A2 = WeylCharacterRing("A2")
sage: A2 = WeylCharacterRing("A2", style="coroots")
sage: ad = A2.adjoint_representation(); ad
A2(1,1)
sage: ad.degree()
8
sage: ad.frobenius_schur_indicator()
1
```

This confirms that ad has degree 8 and is orthogonal, hence factors through $SO(8)$ which is type D_4:

```
sage: br = branching_rule_from_plethysm(ad,"D4")
sage: D4 = WeylCharacterRing("D4")
sage: [D4(f).branch(A2,rule = br) for f in D4.fundamental_weights()]
[A2(1,1), A2(0,3) + A2(1,1) + A2(3,0), A2(1,1), A2(1,1)]
```

sage.combinat.root_system.branching_rules.**get_branching_rule**(*Rtype*, *Stype*, *rule='default'*)

Creates a branching rule.

INPUT:

- R – the Weyl Character Ring of G
- S – the Weyl Character Ring of H
- rule – a string describing the branching rule as a map from the weight space of S to the weight space of R.

If the rule parameter is omitted, in some cases, a default rule is supplied. See branch_weyl_character().

EXAMPLES:

```
sage: rule = branching_rule(CartanType("A3"),CartanType("C2"),"symmetric")
sage: [rule(x) for x in WeylCharacterRing("A3").fundamental_weights()]
[[1, 0], [1, 1], [1, 0]]
```

sage.combinat.root_system.branching_rules.**maximal_subgroups**(*ct*, *mode='print_rules'*)

Given a classical Cartan type (of rank less than or equal to 8) this prints the Cartan types of maximal subgroups, with a method of obtaining the branching rule. The string to the right of the colon in the output is a command to create a branching rule.

INPUT:

- ct – a classical irreducible Cartan type

Returns a list of maximal subgroups of ct.

EXAMPLES:

```
sage: from sage.combinat.root_system.branching_rules import maximal_subgroups
sage: maximal_subgroups("D4")
B3:branching_rule("D4","B3","symmetric")
A2:branching_rule("D4","A2(1,1)","plethysm")
A1xC2:branching_rule("D4","C1xC2","tensor")*branching_rule("C1xC2","A1xC2",[branching_rule("C1",
A1xA1xA1xA1:branching_rule("D4","D2xD2","orthogonal_sum")*branching_rule("D2xD2","A1xA1xA1xA1",[
```

5.1.196 Cartan matrices

AUTHORS:

- Travis Scrimshaw (2012-04-22): Nicolas M. Thiery moved matrix creation to `CartanType` to prepare `cartan_matrix()` for deprecation.
- Christian Stump, Travis Scrimshaw (2013-04-13): Created `CartanMatrix`.

class `sage.combinat.root_system.cartan_matrix.`**`CartanMatrix`**

Bases: `sage.matrix.matrix_integer_sparse.Matrix_integer_sparse`, `sage.combinat.root_system.cartan_type.CartanType_abstract`

A (generalized) Cartan matrix.

A matrix $A = (a_{ij})_{i,j \in I}$ for some index set I is a generalized Cartan matrix if it satisfies the following properties:

- $a_{ii} = 2$ for all i,
- $a_{ij} \leq 0$ for all $i \neq j$,
- $a_{ij} = 0$ if and only if $a_{ji} = 0$ for all $i \neq j$.

Additionally some reference assume that a Cartan matrix is *symmetrizable* (see `is_symmetrizable()`). However following Kac, we do not make that assumption here.

INPUT:

Can be anything which is accepted by `CartanType` or a matrix.

If given a matrix, one can also use the keyword `cartan_type` when giving a matrix to explicitly state the type. Otherwise this will try to check the input matrix against possible standard types of Cartan matrices. To disable this check, use the keyword `cartan_type_check = False`.

EXAMPLES:

```
sage: CartanMatrix(['A', 4])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]
sage: CartanMatrix(['B', 6])
[ 2 -1  0  0  0  0]
[-1  2 -1  0  0  0]
[ 0 -1  2 -1  0  0]
[ 0  0 -1  2 -1  0]
[ 0  0  0 -1  2 -1]
[ 0  0  0  0 -2  2]
sage: CartanMatrix(['C', 4])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -2]
[ 0  0 -1  2]
```

```
sage: CartanMatrix(['D', 6])
[ 2 -1  0  0  0  0]
[-1  2 -1  0  0  0]
[ 0 -1  2 -1  0  0]
[ 0  0 -1  2 -1 -1]
[ 0  0  0 -1  2  0]
[ 0  0  0 -1  0  2]
sage: CartanMatrix(['E',6])
[ 2  0 -1  0  0  0]
[ 0  2  0 -1  0  0]
[-1  0  2 -1  0  0]
[ 0 -1 -1  2 -1  0]
[ 0  0  0 -1  2 -1]
[ 0  0  0  0 -1  2]
sage: CartanMatrix(['E',7])
[ 2  0 -1  0  0  0  0]
[ 0  2  0 -1  0  0  0]
[-1  0  2 -1  0  0  0]
[ 0 -1 -1  2 -1  0  0]
[ 0  0  0 -1  2 -1  0]
[ 0  0  0  0 -1  2 -1]
[ 0  0  0  0  0 -1  2]
sage: CartanMatrix(['E', 8])
[ 2  0 -1  0  0  0  0  0]
[ 0  2  0 -1  0  0  0  0]
[-1  0  2 -1  0  0  0  0]
[ 0 -1 -1  2 -1  0  0  0]
[ 0  0  0 -1  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0 -1  2]
sage: CartanMatrix(['F', 4])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -2  2 -1]
[ 0  0 -1  2]
```

This is different from MuPAD-Combinat, due to different node convention?

```
sage: CartanMatrix(['G', 2])
[ 2 -3]
[-1  2]
sage: CartanMatrix(['A',1,1])
[ 2 -2]
[-2  2]
sage: CartanMatrix(['A', 3, 1])
[ 2 -1  0 -1]
[-1  2 -1  0]
[ 0 -1  2 -1]
[-1  0 -1  2]
sage: CartanMatrix(['B', 3, 1])
[ 2  0 -1  0]
[ 0  2 -1  0]
[-1 -1  2 -1]
[ 0  0 -2  2]
sage: CartanMatrix(['C', 3, 1])
[ 2 -1  0  0]
[-2  2 -1  0]
[ 0 -1  2 -2]
```

```
[ 0  0 -1  2]
sage: CartanMatrix(['D', 4, 1])
[ 2  0 -1  0  0]
[ 0  2 -1  0  0]
[-1 -1  2 -1 -1]
[ 0  0 -1  2  0]
[ 0  0 -1  0  2]
sage: CartanMatrix(['E', 6, 1])
[ 2  0 -1  0  0  0  0]
[ 0  2  0 -1  0  0  0]
[-1  0  2  0 -1  0  0]
[ 0 -1  0  2 -1  0  0]
[ 0  0 -1 -1  2 -1  0]
[ 0  0  0  0 -1  2 -1]
[ 0  0  0  0  0 -1  2]
sage: CartanMatrix(['E', 7, 1])
[ 2 -1  0  0  0  0  0  0]
[-1  2  0 -1  0  0  0  0]
[ 0  0  2  0 -1  0  0  0]
[ 0 -1  0  2 -1  0  0  0]
[ 0  0 -1 -1  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0 -1  2]
sage: CartanMatrix(['E', 8, 1])
[ 2  0  0  0  0  0  0  0 -1]
[ 0  2  0 -1  0  0  0  0  0]
[ 0  0  2  0 -1  0  0  0  0]
[ 0 -1  0  2 -1  0  0  0  0]
[ 0  0 -1 -1  2 -1  0  0  0]
[ 0  0  0  0 -1  2 -1  0  0]
[ 0  0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0  0 -1  2 -1]
[-1  0  0  0  0  0  0 -1  2]
sage: CartanMatrix(['F', 4, 1])
[ 2 -1  0  0  0]
[-1  2 -1  0  0]
[ 0 -1  2 -1  0]
[ 0  0 -2  2 -1]
[ 0  0  0 -1  2]
sage: CartanMatrix(['G', 2, 1])
[ 2  0 -1]
[ 0  2 -3]
[-1 -1  2]
```

Note: Since this is a matrix, `row()` and `column()` will return the standard row and column respectively. To get the row with the indices as in Dynkin diagrams/Cartan types, use `row_with_indices()` and `column_with_indices()` respectively.

---

cartan_matrix()

Return the Cartan matrix of `self`.

EXAMPLES:

```
sage: CartanMatrix(['C',3]).cartan_matrix()
[ 2 -1  0]
[-1  2 -2]
[ 0 -1  2]
```

cartan_type()
Return the Cartan type of `self` or `self` if unknown.

EXAMPLES:

```
sage: C = CartanMatrix(['A',4,1])
sage: C.cartan_type()
['A', 4, 1]
```

If the Cartan type is unknown:

```
sage: C = CartanMatrix([[2,-1,-2], [-1,2,-1], [-2,-1,2]])
sage: C.cartan_type()
[ 2 -1 -2]
[-1  2 -1]
[-2 -1  2]
```

column_with_indices(j)
Return the j^{th} column $(a_{i,j})_i$ of `self` as a container (or iterator) of tuples $(i, a_{i,j})$

EXAMPLES:

```
sage: M = CartanMatrix(['B',4])
sage: [ (i,a) for (i,a) in M.column_with_indices(3) ]
[(3, 2), (2, -1), (4, -2)]
```

dual()
Return the dual Cartan matrix of `self`, which is obtained by taking the transpose.

EXAMPLES:

```
sage: ct = CartanType(['C',3])
sage: M = CartanMatrix(ct); M
[ 2 -1  0]
[-1  2 -2]
[ 0 -1  2]
sage: M.dual()
[ 2 -1  0]
[-1  2 -1]
[ 0 -2  2]
sage: M.dual() == CartanMatrix(ct.dual())
True
sage: M.dual().cartan_type() == ct.dual()
True
```

An example with arbitrary Cartan matrices:

```
sage: cm = CartanMatrix([[2,-5], [-2, 2]]); cm
[ 2 -5]
[-2  2]
sage: cm.dual()
[ 2 -2]
[-5  2]
sage: cm.dual() == CartanMatrix(cm.transpose())
True
sage: cm.dual().dual() == cm
True
```

dynkin_diagram()
Return the Dynkin diagram corresponding to `self`.

EXAMPLES:

```
sage: C = CartanMatrix(['A',2])
sage: C.dynkin_diagram()
O---O
1   2
A2
sage: C = CartanMatrix(['F',4,1])
sage: C.dynkin_diagram()
O---O---O=>=O---O
0   1   2   3   4
F4~
sage: C = CartanMatrix([[2,-4],[-4,2]])
sage: C.dynkin_diagram()
Dynkin diagram of rank 2
```

indecomposable_blocks()

Return a tuple of all indecomposable blocks of `self`.

EXAMPLES:

```
sage: M = CartanMatrix(['A',2])
sage: M.indecomposable_blocks()
(
[ 2 -1]
[-1  2]
)
sage: M = CartanMatrix([['A',2,1],['A',3,1]])
sage: M.indecomposable_blocks()
(
[ 2 -1  0 -1]
[-1  2 -1  0]  [ 2 -1 -1]
[ 0 -1  2 -1]  [-1  2 -1]
[-1  0 -1  2], [-1 -1  2]
)
```

index_set()

Return the index set of `self`.

EXAMPLES:

```
sage: C = CartanMatrix(['A',1,1])
sage: C.index_set()
(0, 1)
sage: C = CartanMatrix(['E',6])
sage: C.index_set()
(1, 2, 3, 4, 5, 6)
```

is_affine()

Return `True` if `self` is an affine type or `False` otherwise.

A generalized Cartan matrix is affine if all of its indecomposable blocks are either finite (see `is_finite()`) or have zero determinant with all proper principal minors positive.

EXAMPLES:

```
sage: M = CartanMatrix(['C',4])
sage: M.is_affine()
False
sage: M = CartanMatrix(['D',4,1])
sage: M.is_affine()
True
```

```
sage: M = CartanMatrix([[2, -4], [-3, 2]])
sage: M.is_affine()
False
```

is_crystallographic()
Implements `CartanType_abstract.is_crystallographic()`.

A Cartan matrix is crystallographic if it is symmetrizable.

EXAMPLES:

```
sage: CartanMatrix(['F',4]).is_crystallographic()
True
```

is_finite()
Return `True` if `self` is a finite type or `False` otherwise.

A generalized Cartan matrix is finite if the determinant of all its principal submatrices (see `principal_submatrices()`) is positive. Such matrices have a positive definite symmetrized matrix. Note that a finite matrix may consist of multiple blocks of Cartan matrices each having finite Cartan type.

EXAMPLES:

```
sage: M = CartanMatrix(['C',4])
sage: M.is_finite()
True
sage: M = CartanMatrix(['D',4,1])
sage: M.is_finite()
False
sage: M = CartanMatrix([[2, -4], [-3, 2]])
sage: M.is_finite()
False
```

is_hyperbolic(*compact=False*)
Return if `True` if `self` is a (compact) hyperbolic type or `False` otherwise.

An indecomposable generalized Cartan matrix is hyperbolic if it has negative determinant and if any proper connected subdiagram of its Dynkin diagram is of finite or affine type. It is compact hyperbolic if any proper connected subdiagram has finite type.

INPUT:

- `compact` – if `True`, check if matrix is compact hyperbolic

EXAMPLES:

```
sage: M = CartanMatrix([[2,-2,0],[-2,2,-1],[0,-1,2]])
sage: M.is_hyperbolic()
True
sage: M.is_hyperbolic(compact=True)
False
sage: M = CartanMatrix([[2,-3],[-3,2]])
sage: M.is_hyperbolic()
True
sage: M = CartanMatrix(['C',4])
sage: M.is_hyperbolic()
False
```

is_indecomposable()
Return if `self` is an indecomposable matrix or `False` otherwise.

EXAMPLES:

```
sage: M = CartanMatrix(['A',5])
sage: M.is_indecomposable()
True
sage: M = CartanMatrix([[2,-1,0],[-1,2,0],[0,0,2]])
sage: M.is_indecomposable()
False
```

is_indefinite()

Return if `self` is an indefinite type or `False` otherwise.

EXAMPLES:

```
sage: M = CartanMatrix([[2,-3],[-3,2]])
sage: M.is_indefinite()
True
sage: M = CartanMatrix("A2")
sage: M.is_indefinite()
False
```

is_lorentzian()

Return `True` if `self` is a Lorentzian type or `False` otherwise.

A generalized Cartan matrix is Lorentzian if it has negative determinant and exactly one negative eigenvalue.

EXAMPLES:

```
sage: M = CartanMatrix([[2,-3],[-3,2]])
sage: M.is_lorentzian()
True
sage: M = CartanMatrix([[2,-1],[-1,2]])
sage: M.is_lorentzian()
False
```

is_simply_laced()

Implements `CartanType_abstract.is_simply_laced()`.

A Cartan matrix is simply-laced if all non diagonal entries are 0 or -1.

EXAMPLES:

```
sage: cm = CartanMatrix([[2, -1, -1, -1], [-1, 2, -1, -1], [-1, -1, 2, -1], [-1, -1, -1, 2]]
sage: cm.is_simply_laced()
True
```

principal_submatrices(*proper=False*)

Return a list of all principal submatrices of `self`.

INPUT:

- `proper` – if `True`, return only proper submatrices

EXAMPLES:

```
sage: M = CartanMatrix(['A',2])
sage: M.principal_submatrices()
[
              [ 2 -1]
[], [2], [2], [-1  2]
]
sage: M.principal_submatrices(proper=True)
[[], [2], [2]]
```

rank()
Return the rank of `self`.

EXAMPLES:

```
sage: CartanMatrix(['C',3]).rank()
3
sage: CartanMatrix(["A2","B2","F4"]).rank()
8
```

reflection_group(*type='matrix'*)
Return the reflection group corresponding to `self`.

EXAMPLES:

```
sage: C = CartanMatrix(['A',3])
sage: C.reflection_group()
Weyl Group of type ['A', 3] (as a matrix group acting on the root space)
```

relabel(*relabelling*)
Return the relabelled Cartan matrix.

EXAMPLES:

```
sage: CM = CartanMatrix(['C',3])
sage: R = CM.relabel({1:0, 2:4, 3:1}); R
[ 2  0 -1]
[ 0  2 -1]
[-1 -2  2]
sage: R.index_set()
(0, 1, 4)
sage: CM
[ 2 -1  0]
[-1  2 -2]
[ 0 -1  2]
```

root_space()
Return the root space corresponding to `self`.

EXAMPLES:

```
sage: C = CartanMatrix(['A',3])
sage: C.root_space()
Root space over the Rational Field of the Root system of type ['A', 3]
```

root_system()
Return the root system corresponding to `self`.

EXAMPLES:

```
sage: C = CartanMatrix(['A',3])
sage: C.root_system()
Root system of type ['A', 3]
```

row_with_indices(*i*)
Return the i^{th} row $(a_{i,j})_j$ of `self` as a container (or iterator) of tuples $(j, a_{i,j})$

EXAMPLES:

```
sage: M = CartanMatrix(['C',4])
sage: [ (i,a) for (i,a) in M.row_with_indices(3) ]
[(3, 2), (2, -1), (4, -2)]
```

subtype(*index_set*)

Return a subtype of `self` given by `index_set`.

A subtype can be considered the Dynkin diagram induced from the Dynkin diagram of `self` by `index_set`.

EXAMPLES:

```
sage: C = CartanMatrix(['F',4])
sage: S = C.subtype([1,2,3])
sage: S
[ 2 -1  0]
[-1  2 -1]
[ 0 -2  2]
sage: S.index_set()
(1, 2, 3)
```

symmetrized_matrix()

Return the symmetrized matrix of `self` if symmetrizable.

EXAMPLES:

```
sage: cm = CartanMatrix(['B',4,1])
sage: cm.symmetrized_matrix()
[ 4  0 -2  0  0]
[ 0  4 -2  0  0]
[-2 -2  4 -2  0]
[ 0  0 -2  4 -2]
[ 0  0  0 -2  2]
```

symmetrizer()

Return the symmetrizer of `self`.

EXAMPLES:

```
sage: cm = CartanMatrix([[2,-5],[-2,2]])
sage: cm.symmetrizer()
Finite family {0: 2, 1: 5}
```

TESTS:

Check that the symmetrizer computed from the Cartan matrix agrees with the values given by the Cartan type:

```
sage: ct = CartanType(['B',4,1])
sage: ct.symmetrizer()
Finite family {0: 2, 1: 2, 2: 2, 3: 2, 4: 1}
sage: ct.cartan_matrix().symmetrizer()
Finite family {0: 2, 1: 2, 2: 2, 3: 2, 4: 1}
```

sage.combinat.root_system.cartan_matrix.**find_cartan_type_from_matrix**(*CM*)

Find a Cartan type by direct comparison of Dynkin diagrams given from the generalized Cartan matrix `CM` and return `None` if not found.

INPUT:

- `CM` – a generalized Cartan matrix

EXAMPLES:

```
sage: from sage.combinat.root_system.cartan_matrix import find_cartan_type_from_matrix
sage: CM = CartanMatrix([[2,-1,-1], [-1,2,-1], [-1,-1,2]])
sage: find_cartan_type_from_matrix(CM)
['A', 2, 1]
sage: CM = CartanMatrix([[2,-1,0], [-1,2,-2], [0,-1,2]])
sage: find_cartan_type_from_matrix(CM)
['C', 3] relabelled by {1: 0, 2: 1, 3: 2}
sage: CM = CartanMatrix([[2,-1,-2], [-1,2,-1], [-2,-1,2]])
sage: find_cartan_type_from_matrix(CM)
```

sage.combinat.root_system.cartan_matrix.**is_generalized_cartan_matrix**(*M*)

Return `True` if `M` is a generalized Cartan matrix. For a definition of a generalized Cartan matrix, see `CartanMatrix`.

EXAMPLES:

```
sage: from sage.combinat.root_system.cartan_matrix import is_generalized_cartan_matrix
sage: M = matrix([[2,-1,-2], [-1,2,-1], [-2,-1,2]])
sage: is_generalized_cartan_matrix(M)
True
sage: M = matrix([[2,-1,-2], [-1,2,-1], [0,-1,2]])
sage: is_generalized_cartan_matrix(M)
False
sage: M = matrix([[1,-1,-2], [-1,2,-1], [-2,-1,2]])
sage: is_generalized_cartan_matrix(M)
False
```

A non-symmetrizable example:

```
sage: M = matrix([[2,-1,-2], [-1,2,-1], [-1,-1,2]])
sage: is_generalized_cartan_matrix(M)
True
```

5.1.197 Cartan types

Todo

Why does sphinx complain if I use sections here?

Introduction

Loosely speaking, Dynkin diagrams (or equivalently Cartan matrices) are graphs which are used to classify root systems, Coxeter and Weyl groups, Lie algebras, Lie groups, crystals, etc. up to an isomorphism. *Cartan types* are a standard set of names for those Dynkin diagrams (see Wikipedia article Dynkin_diagram).

Let us consider, for example, the Cartan type A_4:

```
sage: T = CartanType(['A', 4])
sage: T
['A', 4]
```

It is the name of the following Dynkin diagram:

```
sage: DynkinDiagram(T)
O---O---O---O
1   2   3   4
A4
```

Note: For convenience, the following shortcuts are available:

```
sage: DynkinDiagram(['A',4])
O---O---O---O
1   2   3   4
A4
sage: DynkinDiagram('A4')
O---O---O---O
1   2   3   4
A4
sage: T.dynkin_diagram()
O---O---O---O
1   2   3   4
A4
```

See DynkinDiagram for how to further manipulate Dynkin diagrams.

From this data (the *Cartan datum*), one can construct the associated root system:

```
sage: RootSystem(T)
Root system of type ['A', 4]
```

The associated Weyl group of A_n is the symmetric group S_{n+1}:

```
sage: W = WeylGroup(T)
sage: W
Weyl Group of type ['A', 4] (as a matrix group acting on the ambient space)
sage: W.cardinality()
120
```

while the Lie algebra is sl_{n+1}, and the Lie group SL_{n+1} (TODO: illustrate this once this is implemented).

One may also construct crystals associated to various Dynkin diagrams. For example:

```
sage: C = crystals.Letters(T)
sage: C
The crystal of letters for type ['A', 4]
sage: C.list()
[1, 2, 3, 4, 5]

sage: C = crystals.Tableaux(T, shape=[2])
sage: C
The crystal of tableaux of type ['A', 4] and shape(s) [[2]]
sage: C.cardinality()
15
```

Here is a sample of all the finite irreducible crystallographic Cartan types:

```
sage: CartanType.samples(finite = True, crystallographic = True)
[['A', 1], ['A', 5], ['B', 1], ['B', 5], ['C', 1], ['C', 5], ['D', 2], ['D', 3], ['D', 5],
 ['E', 6], ['E', 7], ['E', 8], ['F', 4], ['G', 2]]
```

One can also get latex representations of the crystallographic Cartan types and their corresponding Dynkin diagrams:

```
sage: [latex(ct) for ct in CartanType.samples(crystallographic=True)]
[A_{1}, A_{5}, B_{1}, B_{5}, C_{1}, C_{5}, D_{2}, D_{3}, D_{5},
 E_6, E_7, E_8, F_4, G_2,
 A_{1}^{(1)}, A_{5}^{(1)}, B_{1}^{(1)}, B_{5}^{(1)}, C_{1}^{(1)}, C_{5}^{(1)}, D_{3}^{(1)}, D_{5}^{(1
```

```
 E_6^{(1)}, E_7^{(1)}, E_8^{(1)}, F_4^{(1)}, G_2^{(1)},
 BC_{1}^{(2)}, BC_{5}^{(2)},
 B_{5}^{(1)\vee}, C_{4}^{(1)\vee}, F_4^{(1)\vee}, G_2^{(1)\vee}, BC_{1}^{(2)\vee}, BC_{5}^{(2)\vee}]
sage: view([DynkinDiagram(ct) for ct in CartanType.samples(crystallographic=True)]) # not tested
```

Non-crystallographic Cartan types are also partially supported:

```
sage: CartanType.samples(finite = True, crystallographic = False)
[['I', 5], ['H', 3], ['H', 4]]
```

In Sage, a Cartan type is used as a database of type-specific information and algorithms (see e.g. sage.combinat.root_system.type_A). This database includes how to construct the Dynkin diagram, the ambient space for the root system (see Wikipedia article Root_system), and further mathematical properties:

```
sage: T.is_finite(), T.is_simply_laced(), T.is_affine(), T.is_crystallographic()
(True, True, False, True)
```

In particular, a Sage Cartan type is endowed with a fixed choice of labels for the nodes of the Dynkin diagram. This choice follows the conventions of Nicolas Bourbaki, Lie Groups and Lie Algebras: Chapter 4-6, Elements of Mathematics, Springer (2002). ISBN 978-3540426509. For example:

```
sage: T = CartanType(['D', 4])
sage: DynkinDiagram(T)
    O 4
    |
    |
O---O---O
1   2   3
D4

sage: E6 = CartanType(['E',6])
sage: DynkinDiagram(E6)
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6
```

Note: The direction of the arrows is the **opposite** (i.e. the transpose) of Bourbaki's convention, but agrees with Kac's.

For example, in type C_2, we have:

```
sage: C2 = DynkinDiagram(['C',2]); C2
O=<=O
1   2
C2
sage: C2.cartan_matrix()
[ 2 -2]
[-1  2]
```

However Bourbaki would have the Cartan matrix as:

$$\begin{bmatrix} 2 & -1 \\ -2 & 2 \end{bmatrix}.$$

If desired, other node labelling conventions can be achieved. For example the Kac labelling for type E_6 can be obtained via:

```
sage: E6.relabel({1:1,2:6,3:2,4:3,5:4,6:5}).dynkin_diagram()
        O 6
        |
        |
O---O---O---O---O
1   2   3   4   5
E6 relabelled by {1: 1, 2: 6, 3: 2, 4: 3, 5: 4, 6: 5}
```

Contributions implementing other conventions are very welcome.

Another option is to build from scratch a new Dynkin diagram. The architecture has been designed to make it fairly easy to add other labelling conventions. In particular, we strived at choosing type free algorithms whenever possible, so in principle most features should remain available even with custom Cartan types. This has not been used much yet, so some rough corners certainly remain.

Here, we construct the hyperbolic example of Exercise 4.9 p. 57 of Kac, Infinite Dimensional Lie Algebras. We start with an empty Dynkin diagram, and add a couple nodes:

```
sage: g = DynkinDiagram()
sage: g.add_vertices([1,2,3])
```

Note that the diagonal of the Cartan matrix is already initialized:

```
sage: g.cartan_matrix()
[2 0 0]
[0 2 0]
[0 0 2]
```

Then we add a couple edges:

```
sage: g.add_edge(1,2,2)
sage: g.add_edge(1,3)
sage: g.add_edge(2,3)
```

and we get the desired Cartan matrix:

```
sage: g.cartan_matrix()
[2 0 0]
[0 2 0]
[0 0 2]
```

Oops, the Cartan matrix did not change! This is because it is cached for efficiency (see `cached_method`). In general, a Dynkin diagram should not be modified after having been used.

Warning: this is not checked currently

Todo

add a method `set_mutable()` as, say, for matrices

Here, we can work around this by clearing the cache:

```
sage: delattr(g, 'cartan_matrix')
```

Now we get the desired Cartan matrix:

```
sage: g.cartan_matrix()
[ 2 -1 -1]
[-2  2 -1]
[-1 -1  2]
```

Note that backward edges have been automatically added:

```
sage: g.edges()
[(1, 2, 2), (1, 3, 1), (2, 1, 1), (2, 3, 1), (3, 1, 1), (3, 2, 1)]
```

Reducible Cartan types

Reducible Cartan types can be specified by passing a sequence or list of irreducible Cartan types:

```
sage: CartanType(['A',2],['B',2])
A2xB2
sage: CartanType([['A',2],['B',2]])
A2xB2
sage: CartanType(['A',2],['B',2]).is_reducible()
True
```

or using the following short hand notation:

```
sage: CartanType("A2xB2")
A2xB2
sage: CartanType("A2","B2") == CartanType("A2xB2")
True
```

Degenerate cases

When possible, type I_n is automatically converted to the isomorphic crystallographic Cartan types (any reason not to do so?):

```
sage: CartanType(["I",1])
A1xA1
sage: CartanType(["I",3])
['A', 2]
sage: CartanType(["I",4])
['C', 2]
sage: CartanType(["I",6])
['G', 2]
```

The Dynkin diagrams for types B_1, C_1, D_2, and D_3 are isomorphic to that for A_1, A_1, $A_1 \times A_1$, and A_3, respectively. However their natural ambient space realizations (stemming from the corresponding infinite families of Lie groups) are different. Therefore, the Cartan types are considered as distinct:

```
sage: CartanType(['B',1])
['B', 1]
sage: CartanType(['C',1])
['C', 1]
sage: CartanType(['D',2])
['D', 2]
sage: CartanType(['D',3])
['D', 3]
```

Affine Cartan types

For affine types, we use the usual conventions for affine Coxeter groups: each affine type is either untwisted (that is arise from the natural affinisation of a finite Cartan type):

```
sage: CartanType(["A", 4, 1]).dynkin_diagram()
0
O-----------+
|           |
|           |
O---O---O---O
1   2   3   4
A4~
sage: CartanType(["B", 4, 1]).dynkin_diagram()
    O 0
    |
    |
O---O---O=>=O
1   2   3   4
B4~
```

or dual thereof:

```
sage: CartanType(["B", 4, 1]).dual().dynkin_diagram()
    O 0
    |
    |
O---O---O=<=O
1   2   3   4
B4~*
```

or is of type $\widetilde{BC}_n$ (which yields an irreducible, but nonreduced root system):

```
sage: CartanType(["BC", 4, 2]).dynkin_diagram()
O=<=O---O---O=<=O
0   1   2   3   4
BC4~
```

This includes the two degenerate cases:

```
sage: CartanType(["A", 1, 1]).dynkin_diagram()
O<=>O
0   1
A1~
sage: CartanType(["BC", 1, 2]).dynkin_diagram()
  4
O=<=O
0   1
BC1~
```

For the user convenience, Kac's notations for twisted affine types are automatically translated into the previous ones:

```
sage: CartanType(["A", 9, 2])
['B', 5, 1]^*
sage: CartanType(["A", 9, 2]).dynkin_diagram()
    O 0
    |
    |
```

```
O---O---O---O=<=O
1   2   3   4   5
B5~*
sage: CartanType(["A", 10, 2]).dynkin_diagram()
O=<=O---O---O---O=<=O
0   1   2   3   4   5
BC5~
sage: CartanType(["D", 5, 2]).dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
C4~*
sage: CartanType(["D", 4, 3]).dynkin_diagram()
  3
O=>=O---O
2   1   0
G2~* relabelled by {0: 0, 1: 2, 2: 1}
sage: CartanType(["E", 6, 2]).dynkin_diagram()
O---O---O=<=O---O
0   1   2   3   4
F4~*
```

Additionally one can set the notation global option to use Kac's notation:

```
sage: CartanType.global_options['notation'] = 'Kac'
sage: CartanType(["A", 9, 2])
['A', 9, 2]
sage: CartanType(["A", 9, 2]).dynkin_diagram()
    O 0
    |
    |
O---O---O---O=<=O
1   2   3   4   5
A9^2
sage: CartanType(["A", 10, 2]).dynkin_diagram()
O=<=O---O---O---O=<=O
0   1   2   3   4   5
A10^2
sage: CartanType(["D", 5, 2]).dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
D5^2
sage: CartanType(["D", 4, 3]).dynkin_diagram()
  3
O=>=O---O
2   1   0
D4^3
sage: CartanType(["E", 6, 2]).dynkin_diagram()
O---O---O=<=O---O
0   1   2   3   4
E6^2
sage: CartanType.global_options['notation'] = 'BC'
```

Todo

Should those indexes come before the introduction?

Abstract classes for cartan types

- CartanType_abstract
- CartanType_crystallographic
- CartanType_simply_laced
- CartanType_simple
- CartanType_finite
- CartanType_affine (see also *Root system data for affine Cartan types*)
- sage.combinat.root_system.cartan_type.CartanType
- *Root system data for dual Cartan types*
- *Root system data for reducible Cartan types*
- *Root system data for relabelled Cartan types*

Concrete classes for cartan types

- CartanType_standard_affine
- CartanType_standard_untwisted_affine

Type specific data

The data essentially consists of a description of the Dynkin/Coxeter diagram and, when relevant, of the natural embedding of the root system in an Euclidean space. Everything else is reconstructed from this data.

- *Root system data for type A*
- *Root system data for type B*
- *Root system data for type C*
- *Root system data for type D*
- *Root system data for type E*
- *Root system data for type F*
- *Root system data for type G*
- *Root system data for type H*
- *Root system data for type I*
- *Root system data for (untwisted) type A affine*
- *Root system data for (untwisted) type B affine*
- *Root system data for (untwisted) type C affine*
- *Root system data for (untwisted) type D affine*
- *Root system data for (untwisted) type E affine*
- *Root system data for (untwisted) type F affine*
- *Root system data for (untwisted) type G affine*
- *Root system data for type BC affine*

sage.combinat.root_system.cartan_type.**CartanType**(**args*)
Cartan types

Todo

Why does sphinx complain if I use sections here?

Introduction

Loosely speaking, Dynkin diagrams (or equivalently Cartan matrices) are graphs which are used to classify root systems, Coxeter and Weyl groups, Lie algebras, Lie groups, crystals, etc. up to an isomorphism. *Cartan types* are a standard set of names for those Dynkin diagrams (see Wikipedia article Dynkin_diagram).

Let us consider, for example, the Cartan type A_4:

```
sage: T = CartanType(['A', 4])
sage: T
['A', 4]
```

It is the name of the following Dynkin diagram:

```
sage: DynkinDiagram(T)
O---O---O---O
1   2   3   4
A4
```

Note: For convenience, the following shortcuts are available:

```
sage: DynkinDiagram(['A',4])
O---O---O---O
1   2   3   4
A4
sage: DynkinDiagram('A4')
O---O---O---O
1   2   3   4
A4
sage: T.dynkin_diagram()
O---O---O---O
1   2   3   4
A4
```

See `DynkinDiagram` for how to further manipulate Dynkin diagrams.

From this data (the *Cartan datum*), one can construct the associated root system:

```
sage: RootSystem(T)
Root system of type ['A', 4]
```

The associated Weyl group of A_n is the symmetric group S_{n+1}:

```
sage: W = WeylGroup(T)
sage: W
Weyl Group of type ['A', 4] (as a matrix group acting on the ambient space)
sage: W.cardinality()
120
```

while the Lie algebra is sl_{n+1}, and the Lie group SL_{n+1} (TODO: illustrate this once this is implemented).

One may also construct crystals associated to various Dynkin diagrams. For example:

```
sage: C = crystals.Letters(T)
sage: C
The crystal of letters for type ['A', 4]
sage: C.list()
[1, 2, 3, 4, 5]

sage: C = crystals.Tableaux(T, shape=[2])
```

```
sage: C
The crystal of tableaux of type ['A', 4] and shape(s) [[2]]
sage: C.cardinality()
15
```

Here is a sample of all the finite irreducible crystallographic Cartan types:

```
sage: CartanType.samples(finite = True, crystallographic = True)
[['A', 1], ['A', 5], ['B', 1], ['B', 5], ['C', 1], ['C', 5], ['D', 2], ['D', 3], ['D', 5],
 ['E', 6], ['E', 7], ['E', 8], ['F', 4], ['G', 2]]
```

One can also get latex representations of the crystallographic Cartan types and their corresponding Dynkin diagrams:

```
sage: [latex(ct) for ct in CartanType.samples(crystallographic=True)]
[A_{1}, A_{5}, B_{1}, B_{5}, C_{1}, C_{5}, D_{2}, D_{3}, D_{5},
 E_6, E_7, E_8, F_4, G_2,
 A_{1}^{(1)}, A_{5}^{(1)}, B_{1}^{(1)}, B_{5}^{(1)}, C_{1}^{(1)}, C_{5}^{(1)}, D_{3}^{(1)}, D_{5
 E_6^{(1)}, E_7^{(1)}, E_8^{(1)}, F_4^{(1)}, G_2^{(1)},
 BC_{1}^{(2)}, BC_{5}^{(2)},
 B_{5}^{(1)\vee}, C_{4}^{(1)\vee}, F_4^{(1)\vee}, G_2^{(1)\vee}, BC_{1}^{(2)\vee}, BC_{5}^{(2)\v
sage: view([DynkinDiagram(ct) for ct in CartanType.samples(crystallographic=True)]) # not tested
```

Non-crystallographic Cartan types are also partially supported:

```
sage: CartanType.samples(finite = True, crystallographic = False)
[['I', 5], ['H', 3], ['H', 4]]
```

In Sage, a Cartan type is used as a database of type-specific information and algorithms (see e.g. `sage.combinat.root_system.type_A`). This database includes how to construct the Dynkin diagram, the ambient space for the root system (see Wikipedia article Root_system), and further mathematical properties:

```
sage: T.is_finite(), T.is_simply_laced(), T.is_affine(), T.is_crystallographic()
(True, True, False, True)
```

In particular, a Sage Cartan type is endowed with a fixed choice of labels for the nodes of the Dynkin diagram. This choice follows the conventions of Nicolas Bourbaki, Lie Groups and Lie Algebras: Chapter 4-6, Elements of Mathematics, Springer (2002). ISBN 978-3540426509. For example:

```
sage: T = CartanType(['D', 4])
sage: DynkinDiagram(T)
    O 4
    |
    |
O---O---O
1   2   3
D4

sage: E6 = CartanType(['E',6])
sage: DynkinDiagram(E6)
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6
```

Note: The direction of the arrows is the **opposite** (i.e. the transpose) of Bourbaki's convention, but agrees with Kac's.

For example, in type C_2, we have:

```
sage: C2 = DynkinDiagram(['C',2]); C2
O=<=O
1   2
C2
sage: C2.cartan_matrix()
[ 2 -2]
[-1  2]
```

However Bourbaki would have the Cartan matrix as:

$$\begin{bmatrix} 2 & -1 \\ -2 & 2 \end{bmatrix}.$$

If desired, other node labelling conventions can be achieved. For example the Kac labelling for type E_6 can be obtained via:

```
sage: E6.relabel({1:1,2:6,3:2,4:3,5:4,6:5}).dynkin_diagram()
        O 6
        |
        |
O---O---O---O---O
1   2   3   4   5
E6 relabelled by {1: 1, 2: 6, 3: 2, 4: 3, 5: 4, 6: 5}
```

Contributions implementing other conventions are very welcome.

Another option is to build from scratch a new Dynkin diagram. The architecture has been designed to make it fairly easy to add other labelling conventions. In particular, we strived at choosing type free algorithms whenever possible, so in principle most features should remain available even with custom Cartan types. This has not been used much yet, so some rough corners certainly remain.

Here, we construct the hyperbolic example of Exercise 4.9 p. 57 of Kac, Infinite Dimensional Lie Algebras. We start with an empty Dynkin diagram, and add a couple nodes:

```
sage: g = DynkinDiagram()
sage: g.add_vertices([1,2,3])
```

Note that the diagonal of the Cartan matrix is already initialized:

```
sage: g.cartan_matrix()
[2 0 0]
[0 2 0]
[0 0 2]
```

Then we add a couple edges:

```
sage: g.add_edge(1,2,2)
sage: g.add_edge(1,3)
sage: g.add_edge(2,3)
```

and we get the desired Cartan matrix:

```
sage: g.cartan_matrix()
[2 0 0]
[0 2 0]
[0 0 2]
```

Oops, the Cartan matrix did not change! This is because it is cached for efficiency (see `cached_method`). In general, a Dynkin diagram should not be modified after having been used.

Warning: this is not checked currently

Todo

add a method `set_mutable()` as, say, for matrices

Here, we can work around this by clearing the cache:

```
sage: delattr(g, 'cartan_matrix')
```

Now we get the desired Cartan matrix:

```
sage: g.cartan_matrix()
[ 2 -1 -1]
[-2  2 -1]
[-1 -1  2]
```

Note that backward edges have been automatically added:

```
sage: g.edges()
[(1, 2, 2), (1, 3, 1), (2, 1, 1), (2, 3, 1), (3, 1, 1), (3, 2, 1)]
```

Reducible Cartan types

Reducible Cartan types can be specified by passing a sequence or list of irreducible Cartan types:

```
sage: CartanType(['A',2],['B',2])
A2xB2
sage: CartanType([['A',2],['B',2]])
A2xB2
sage: CartanType(['A',2],['B',2]).is_reducible()
True
```

or using the following short hand notation:

```
sage: CartanType("A2xB2")
A2xB2
sage: CartanType("A2","B2") == CartanType("A2xB2")
True
```

Degenerate cases

When possible, type I_n is automatically converted to the isomorphic crystallographic Cartan types (any reason not to do so?):

```
sage: CartanType(["I",1])
A1xA1
sage: CartanType(["I",3])
['A', 2]
sage: CartanType(["I",4])
['C', 2]
sage: CartanType(["I",6])
['G', 2]
```

The Dynkin diagrams for types B_1, C_1, D_2, and D_3 are isomorphic to that for A_1, A_1, $A_1 \times A_1$, and A_3, respectively. However their natural ambient space realizations (stemming from the corresponding infinite families

of Lie groups) are different. Therefore, the Cartan types are considered as distinct:

```
sage: CartanType(['B',1])
['B', 1]
sage: CartanType(['C',1])
['C', 1]
sage: CartanType(['D',2])
['D', 2]
sage: CartanType(['D',3])
['D', 3]
```

Affine Cartan types

For affine types, we use the usual conventions for affine Coxeter groups: each affine type is either untwisted (that is arise from the natural affinisation of a finite Cartan type):

```
sage: CartanType(["A", 4, 1]).dynkin_diagram()
0
O------------+
|            |
|            |
O---O---O---O
1   2   3   4
A4~
sage: CartanType(["B", 4, 1]).dynkin_diagram()
    O 0
    |
    |
O---O---O=>=O
1   2   3   4
B4~
```

or dual thereof:

```
sage: CartanType(["B", 4, 1]).dual().dynkin_diagram()
    O 0
    |
    |
O---O---O=<=O
1   2   3   4
B4~*
```

or is of type $\widetilde{BC}_n$ (which yields an irreducible, but nonreduced root system):

```
sage: CartanType(["BC", 4, 2]).dynkin_diagram()
O=<=O---O---O=<=O
0   1   2   3   4
BC4~
```

This includes the two degenerate cases:

```
sage: CartanType(["A", 1, 1]).dynkin_diagram()
O<=>O
0   1
A1~
sage: CartanType(["BC", 1, 2]).dynkin_diagram()
  4
O=<=O
0   1
```

```
BC1~
```

For the user convenience, Kac's notations for twisted affine types are automatically translated into the previous ones:

```
sage: CartanType(["A", 9, 2])
['B', 5, 1]^*
sage: CartanType(["A", 9, 2]).dynkin_diagram()
    O 0
    |
    |
O---O---O---O=<=O
1   2   3   4   5
B5~*
sage: CartanType(["A", 10, 2]).dynkin_diagram()
O=<=O---O---O---O=<=O
0   1   2   3   4   5
BC5~
sage: CartanType(["D", 5, 2]).dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
C4~*
sage: CartanType(["D", 4, 3]).dynkin_diagram()
  3
O=>=O---O
2   1   0
G2~* relabelled by {0: 0, 1: 2, 2: 1}
sage: CartanType(["E", 6, 2]).dynkin_diagram()
O---O---O=<=O---O
0   1   2   3   4
F4~*
```

Additionally one can set the notation global option to use Kac's notation:

```
sage: CartanType.global_options['notation'] = 'Kac'
sage: CartanType(["A", 9, 2])
['A', 9, 2]
sage: CartanType(["A", 9, 2]).dynkin_diagram()
    O 0
    |
    |
O---O---O---O=<=O
1   2   3   4   5
A9^2
sage: CartanType(["A", 10, 2]).dynkin_diagram()
O=<=O---O---O---O=<=O
0   1   2   3   4   5
A10^2
sage: CartanType(["D", 5, 2]).dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
D5^2
sage: CartanType(["D", 4, 3]).dynkin_diagram()
  3
O=>=O---O
2   1   0
D4^3
sage: CartanType(["E", 6, 2]).dynkin_diagram()
O---O---O=<=O---O
```

```
0   1   2   3   4
E6^2
sage: CartanType.global_options['notation'] = 'BC'
```

Todo

Should those indexes come before the introduction?

Abstract classes for cartan types

- CartanType_abstract
- CartanType_crystallographic
- CartanType_simply_laced
- CartanType_simple
- CartanType_finite
- CartanType_affine (see also *Root system data for affine Cartan types*)
- sage.combinat.root_system.cartan_type.CartanType
- *Root system data for dual Cartan types*
- *Root system data for reducible Cartan types*
- *Root system data for relabelled Cartan types*

Concrete classes for cartan types

- CartanType_standard_affine
- CartanType_standard_untwisted_affine

Type specific data

The data essentially consists of a description of the Dynkin/Coxeter diagram and, when relevant, of the natural embedding of the root system in an Euclidean space. Everything else is reconstructed from this data.

- *Root system data for type A*
- *Root system data for type B*
- *Root system data for type C*
- *Root system data for type D*
- *Root system data for type E*
- *Root system data for type F*
- *Root system data for type G*
- *Root system data for type H*
- *Root system data for type I*
- *Root system data for (untwisted) type A affine*
- *Root system data for (untwisted) type B affine*
- *Root system data for (untwisted) type C affine*
- *Root system data for (untwisted) type D affine*
- *Root system data for (untwisted) type E affine*

- *Root system data for (untwisted) type F affine*
- *Root system data for (untwisted) type G affine*
- *Root system data for type BC affine*

class sage.combinat.root_system.cartan_type.**CartanTypeFactory**
Bases: sage.structure.sage_object.SageObject

classmethod color(*i*)
Default color scheme for the vertices of a Dynkin diagram (and associated objects)

EXAMPLES:

```
sage: CartanType.color(1)
'blue'
sage: CartanType.color(2)
'red'
sage: CartanType.color(3)
'green'
```

The default color is black:

```
sage: CartanType.color(0)
'black'
```

Negative indices get the same color as their positive counterparts:

```
sage: CartanType.color(-1)
'blue'
sage: CartanType.color(-2)
'red'
sage: CartanType.color(-3)
'green'
```

global_options(**get_value*, ***set_value*)
Sets and displays the global options for Cartan types. If no parameters are set, then the function returns a copy of the options dictionary.

The `options` to partitions can be accessed as the method `CartanType.global_options` of `CartanType`.

OPTIONS:

- `dual_latex` – (default: `\vee`) The latex used for dual CartanTypes when latexing
- `dual_str` – (default: `*`) The string used for dual Cartan types when printing
- `latex_marked` – (default: `True`) Indicate in the latex output if a Cartan type has been marked
- `latex_relabel` – (default: `True`) Indicate in the latex output if a Cartan type has been relabelled
- `mark_special_node` – (default: `none`) Make the special nodes
  - `both` – both in latex and printing
  - `latex` – only in latex
  - `none` – no markup
  - `printing` – only in printing
- `marked_node_str` – (default: `X`) The string used to indicate a marked node when printing
- `notation` – (default: `Stembridge`) Specifies which notation Cartan types should use when printed

– `BC` – alias for `Stembridge`

– `Kac` – use Kac's notation

– `Stembridge` – use Stembridge's notation

– `tilde` – alias for `Stembridge`

– `twisted` – alias for `Kac`

• `special_node_str` – (default: `@`) The string used to indicate which node is special when printing

EXAMPLES:

```
sage: ct = CartanType(['D',5,2]); ct
['C', 4, 1]^*
sage: ct.dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
C4~*
sage: latex(ct)
C_{4}^{(1)\vee}
sage: CartanType.global_options(dual_str='#', dual_latex='\\ast',)
sage: ct
['C', 4, 1]^#
sage: ct.dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
C4~#
sage: latex(ct)
C_{4}^{(1)\ast}
sage: CartanType.global_options(notation='kac', mark_special_node='both')
sage: ct
['D', 5, 2]
sage: ct.dynkin_diagram()
@=<=O---O---O=>=O
0   1   2   3   4
D5^2
sage: latex(ct)
D_{5}^{(2)}
```

For type $A_{2n}^{(2)\dagger}$, the dual string/latex options are automatically overriden:

```
sage: dct = CartanType(['A',8,2]).dual(); dct
['A', 8, 2]^+
sage: latex(dct)
A_{8}^{(2)\dagger}
sage: dct.dynkin_diagram()
@=>=O---O---O=>=O
0   1   2   3   4
A8^2+
sage: CartanType.global_options.reset()
```

See `GlobalOptions` for more features of these options.

samples (*finite=None*, *affine=None*, *crystallographic=None*)

Return a sample of the available Cartan types.

INPUT:

• `finite` – a boolean or `None` (default: `None`)

• `affine` – a boolean or `None` (default: `None`)

- `crystallographic` – a boolean or `None` (default: `None`)

The sample contains all the exceptional finite and affine Cartan types, as well as typical representatives of the infinite families.

EXAMPLES:

```
sage: CartanType.samples()
[['A', 1], ['A', 5], ['B', 1], ['B', 5], ['C', 1], ['C', 5], ['D', 2], ['D', 3], ['D', 5],
 ['E', 6], ['E', 7], ['E', 8], ['F', 4], ['G', 2], ['I', 5], ['H', 3], ['H', 4],
 ['A', 1, 1], ['A', 5, 1], ['B', 1, 1], ['B', 5, 1],
 ['C', 1, 1], ['C', 5, 1], ['D', 3, 1], ['D', 5, 1],
 ['E', 6, 1], ['E', 7, 1], ['E', 8, 1], ['F', 4, 1], ['G', 2, 1], ['BC', 1, 2], ['BC', 5, 2]
 ['B', 5, 1]^*, ['C', 4, 1]^*, ['F', 4, 1]^*, ['G', 2, 1]^*, ['BC', 1, 2]^*, ['BC', 5, 2]^*]
```

The finite, affine and crystallographic options allow respectively for restricting to (non) finite, (non) affine, and (non) crystallographic Cartan types:

```
sage: CartanType.samples(finite=True)
[['A', 1], ['A', 5], ['B', 1], ['B', 5], ['C', 1], ['C', 5], ['D', 2], ['D', 3], ['D', 5],
 ['E', 6], ['E', 7], ['E', 8], ['F', 4], ['G', 2], ['I', 5], ['H', 3], ['H', 4]]

sage: CartanType.samples(affine=True)
[['A', 1, 1], ['A', 5, 1], ['B', 1, 1], ['B', 5, 1],
 ['C', 1, 1], ['C', 5, 1], ['D', 3, 1], ['D', 5, 1],
 ['E', 6, 1], ['E', 7, 1], ['E', 8, 1], ['F', 4, 1], ['G', 2, 1], ['BC', 1, 2], ['BC', 5, 2]
 ['B', 5, 1]^*, ['C', 4, 1]^*, ['F', 4, 1]^*, ['G', 2, 1]^*, ['BC', 1, 2]^*, ['BC', 5, 2]^*]

sage: CartanType.samples(crystallographic=True)
[['A', 1], ['A', 5], ['B', 1], ['B', 5], ['C', 1], ['C', 5], ['D', 2], ['D', 3], ['D', 5],
 ['E', 6], ['E', 7], ['E', 8], ['F', 4], ['G', 2],
 ['A', 1, 1], ['A', 5, 1], ['B', 1, 1], ['B', 5, 1],
 ['C', 1, 1], ['C', 5, 1], ['D', 3, 1], ['D', 5, 1],
 ['E', 6, 1], ['E', 7, 1], ['E', 8, 1], ['F', 4, 1], ['G', 2, 1], ['BC', 1, 2], ['BC', 5, 2]
 ['B', 5, 1]^*, ['C', 4, 1]^*, ['F', 4, 1]^*, ['G', 2, 1]^*, ['BC', 1, 2]^*, ['BC', 5, 2]^*]

sage: CartanType.samples(crystallographic=False)
[['I', 5], ['H', 3], ['H', 4]]
```

Todo

add some reducible Cartan types (suggestions?)

TESTS:

```
sage: for ct in CartanType.samples(): TestSuite(ct).run()
```

class sage.combinat.root_system.cartan_type.**CartanType_abstract**

Bases: `object`

Abstract class for Cartan types

Subclasses should implement:

- `dynkin_diagram()`
- `cartan_matrix()`
- `is_finite()`
- `is_affine()`

- is_irreducible()

as_folding(*folding_of=None*, *sigma=None*)

Return self realized as a folded Cartan type.

For finite and affine types, this is realized by the Dynkin diagram foldings:

$$\begin{aligned} C_n^{(1)}, A_{2n}^{(2)}, A_{2n}^{(2)\dagger}, D_{n+1}^{(2)} &\hookrightarrow A_{2n-1}^{(1)}, \\ A_{2n-1}^{(2)}, B_n^{(1)} &\hookrightarrow D_{n+1}^{(1)}, \\ E_6^{(2)}, F_4^{(1)} &\hookrightarrow E_6^{(1)}, \\ D_4^{(3)}, G_2^{(1)} &\hookrightarrow D_4^{(1)}, \\ C_n &\hookrightarrow A_{2n-1}, \\ B_n &\hookrightarrow D_{n+1}, \\ F_4 &\hookrightarrow E_6, \\ G_2 &\hookrightarrow D_4. \end{aligned}$$

For general types, this returns self as a folded type of self with σ as the identity map.

For more information on these foldings and folded Cartan types, see sage.combinat.root_system.type_folded.CartanTypeFolded.

If the optional inputs folding_of and sigma are specified, then this returns the folded Cartan type of self in folding_of given by the automorphism sigma.

EXAMPLES:

```
sage: CartanType(['B', 3, 1]).as_folding()
['B', 3, 1] as a folding of  ['D', 4, 1]
sage: CartanType(['F', 4]).as_folding()
['F', 4] as a folding of  ['E', 6]
sage: CartanType(['BC', 3, 2]).as_folding()
['BC', 3, 2] as a folding of  ['A', 5, 1]
sage: CartanType(['D', 4, 3]).as_folding()
['G', 2, 1]^* relabelled by {0: 0, 1: 2, 2: 1} as a folding of ['D', 4, 1]
```

coxeter_diagram()

Return the Coxeter diagram for self.

EXAMPLES:

```
sage: CartanType(['B',3]).coxeter_diagram()
Graph on 3 vertices
sage: CartanType(['A',3]).coxeter_diagram().edges()
[(1, 2, 3), (2, 3, 3)]
sage: CartanType(['B',3]).coxeter_diagram().edges()
[(1, 2, 3), (2, 3, 4)]
sage: CartanType(['G',2]).coxeter_diagram().edges()
[(1, 2, 6)]
sage: CartanType(['F',4]).coxeter_diagram().edges()
[(1, 2, 3), (2, 3, 4), (3, 4, 3)]
```

coxeter_matrix()

Return the Coxeter matrix for self.

EXAMPLES:

```
sage: CartanType(['A', 4]).coxeter_matrix()
[1 3 2 2]
[3 1 3 2]
[2 3 1 3]
[2 2 3 1]
```

coxeter_type()
Return the Coxeter type for `self`.

EXAMPLES:

```
sage: CartanType(['A', 4]).coxeter_type()
Coxeter type of ['A', 4]
```

dual()
Return the dual Cartan type, possibly just as a formal dual.

EXAMPLES:

```
sage: CartanType(['A',3]).dual()
['A', 3]
sage: CartanType(["B", 3]).dual()
['C', 3]
sage: CartanType(['C',2]).dual()
['B', 2]
sage: CartanType(['D',4]).dual()
['D', 4]
sage: CartanType(['E',8]).dual()
['E', 8]
sage: CartanType(['F',4]).dual()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

global_options(**get_value*, ***set_value*)
Sets and displays the global options for Cartan types. If no parameters are set, then the function returns a copy of the options dictionary.

The `options` to partitions can be accessed as the method `CartanType.global_options` of `CartanType`.

OPTIONS:

- `dual_latex` – (default: `\vee`) The latex used for dual CartanTypes when latexing
- `dual_str` – (default: `*`) The string used for dual Cartan types when printing
- `latex_marked` – (default: `True`) Indicate in the latex output if a Cartan type has been marked
- `latex_relabel` – (default: `True`) Indicate in the latex output if a Cartan type has been relabelled
- `mark_special_node` – (default: `none`) Make the special nodes
  - `both` – both in latex and printing
  - `latex` – only in latex
  - `none` – no markup
  - `printing` – only in printing
- `marked_node_str` – (default: `X`) The string used to indicate a marked node when printing
- `notation` – (default: `Stembridge`) Specifies which notation Cartan types should use when printed
  - `BC` – alias for `Stembridge`
  - `Kac` – use Kac's notation
  - `Stembridge` – use Stembridge's notation
  - `tilde` – alias for `Stembridge`

– `twisted` – alias for `Kac`

• `special_node_str` – (default: `@`) The string used to indicate which node is special when printing

EXAMPLES:

```
sage: ct = CartanType(['D',5,2]); ct
['C', 4, 1]^*
sage: ct.dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
C4~*
sage: latex(ct)
C_{4}^{(1)\vee}
sage: CartanType.global_options(dual_str='#', dual_latex='\\ast',)
sage: ct
['C', 4, 1]^#
sage: ct.dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
C4~#
sage: latex(ct)
C_{4}^{(1)\ast}
sage: CartanType.global_options(notation='kac', mark_special_node='both')
sage: ct
['D', 5, 2]
sage: ct.dynkin_diagram()
@=<=O---O---O=>=O
0   1   2   3   4
D5^2
sage: latex(ct)
D_{5}^{(2)}
```

For type $A_{2n}^{(2)\dagger}$, the dual string/latex options are automatically overriden:

```
sage: dct = CartanType(['A',8,2]).dual(); dct
['A', 8, 2]^+
sage: latex(dct)
A_{8}^{(2)\dagger}
sage: dct.dynkin_diagram()
@=>=O---O---O=>=O
0   1   2   3   4
A8^2+
sage: CartanType.global_options.reset()
```

See `GlobalOptions` for more features of these options.

index_set()

Return the index set for `self`.

This is the list of the nodes of the associated Coxeter or Dynkin diagram.

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).index_set()
(0, 1, 2, 3)
sage: CartanType(['D', 4]).index_set()
(1, 2, 3, 4)
sage: CartanType(['A', 7, 2]).index_set()
(0, 1, 2, 3, 4)
sage: CartanType(['A', 7, 2]).index_set()
(0, 1, 2, 3, 4)
```

```
sage: CartanType(['A', 6, 2]).index_set()
(0, 1, 2, 3)
sage: CartanType(['D', 6, 2]).index_set()
(0, 1, 2, 3, 4, 5)
sage: CartanType(['E', 6, 1]).index_set()
(0, 1, 2, 3, 4, 5, 6)
sage: CartanType(['E', 6, 2]).index_set()
(0, 1, 2, 3, 4)
sage: CartanType(['A', 2, 2]).index_set()
(0, 1)
sage: CartanType(['G', 2, 1]).index_set()
(0, 1, 2)
sage: CartanType(['F', 4, 1]).index_set()
(0, 1, 2, 3, 4)
```

is_affine()

Return whether `self` is affine.

EXAMPLES:

```
sage: CartanType(['A', 3]).is_affine()
False
sage: CartanType(['A', 3, 1]).is_affine()
True
```

is_atomic()

This method is usually equivalent to `is_reducible()`, except for the Cartan type D_2.

D_2 is not a standard Cartan type. It is equivalent to type $A_1 \times A_1$ which is reducible; however the isomorphism from its ambient space (for the orthogonal group of degree 4) to that of $A_1 \times A_1$ is non trivial, and it is useful to have it.

From a programming point of view its implementation is more similar to the irreducible types, and so the method `is_atomic()` is supplied.

EXAMPLES:

```
sage: CartanType("D2").is_atomic()
True
sage: CartanType("D2").is_irreducible()
False
```

TESTS:

```
sage: all( T.is_irreducible() == T.is_atomic() for T in CartanType.samples() if T != CartanT
True
```

is_compound()

A short hand for not `is_atomic()`.

TESTS:

```
sage: all( T.is_compound() == (not T.is_atomic()) for T in CartanType.samples())
True
```

is_crystallographic()

Return whether this Cartan type is crystallographic.

This returns `False` by default. Derived class should override this appropriately.

EXAMPLES:

```
sage: [ [t, t.is_crystallographic() ] for t in CartanType.samples(finite=True) ]
[[['A', 1], True], [['A', 5], True],
 [['B', 1], True], [['B', 5], True],
 [['C', 1], True], [['C', 5], True],
 [['D', 2], True], [['D', 3], True], [['D', 5], True],
 [['E', 6], True], [['E', 7], True], [['E', 8], True],
 [['F', 4], True], [['G', 2], True],
 [['I', 5], False], [['H', 3], False], [['H', 4], False]]
```

is_finite()

Return whether this Cartan type is finite.

EXAMPLES:

```
sage: from sage.combinat.root_system.cartan_type import CartanType_abstract
sage: C = CartanType_abstract()
sage: C.is_finite()
Traceback (most recent call last):
...
NotImplementedError: <abstract method is_finite at ...>

sage: CartanType(['A',4]).is_finite()
True
sage: CartanType(['A',4, 1]).is_finite()
False
```

is_implemented()

Check whether the Cartan datum for `self` is actually implemented.

EXAMPLES:

```
sage: CartanType(["A",4,1]).is_implemented()
True
sage: CartanType(['H',3]).is_implemented()
True
```

is_irreducible()

Report whether this Cartan type is irreducible (i.e. simple). This should be overridden in any subclass.

This returns `False` by default. Derived class should override this appropriately.

EXAMPLES:

```
sage: from sage.combinat.root_system.cartan_type import CartanType_abstract
sage: C = CartanType_abstract()
sage: C.is_irreducible()
False
```

is_reducible()

Report whether the root system is reducible (i.e. not simple), that is whether it can be factored as a product of root systems.

EXAMPLES:

```
sage: CartanType("A2xB3").is_reducible()
True
sage: CartanType(['A',2]).is_reducible()
False
```

is_simply_laced()

Return whether this Cartan type is simply laced.

This returns `False` by default. Derived class should override this appropriately.

EXAMPLES:

```
sage: [ [t, t.is_simply_laced() ] for t in CartanType.samples() ]
[[['A', 1], True], [['A', 5], True],
 [['B', 1], True], [['B', 5], False],
 [['C', 1], True], [['C', 5], False],
 [['D', 2], True], [['D', 3], True], [['D', 5], True],
 [['E', 6], True], [['E', 7], True], [['E', 8], True],
 [['F', 4], False], [['G', 2], False], [['I', 5], False], [['H', 3], False], [['H', 4], Fals
 [['A', 1, 1], False], [['A', 5, 1], True],
 [['B', 1, 1], False], [['B', 5, 1], False],
 [['C', 1, 1], False], [['C', 5, 1], False],
 [['D', 3, 1], True], [['D', 5, 1], True],
 [['E', 6, 1], True], [['E', 7, 1], True], [['E', 8, 1], True],
 [['F', 4, 1], False], [['G', 2, 1], False],
 [['BC', 1, 2], False], [['BC', 5, 2], False],
 [['B', 5, 1]^*, False], [['C', 4, 1]^*, False], [['F', 4, 1]^*, False], [['G', 2, 1]^*, Fal
 [['BC', 1, 2]^*, False], [['BC', 5, 2]^*, False]]
```

marked_nodes (*marked_nodes*)

Return a Cartan type with the nodes `marked_nodes` marked.

INPUT:

- `marked_nodes` – a list of nodes to mark

EXAMPLES:

```
sage: CartanType(['F',4]).marked_nodes([1, 3]).dynkin_diagram()
X---O=>=X---O
1   2   3   4
F4 with nodes (1, 3) marked
```

rank()

Return the rank of `self`.

This is the number of nodes of the associated Coxeter or Dynkin diagram.

EXAMPLES:

```
sage: CartanType(['A', 4]).rank()
4
sage: CartanType(['A', 7, 2]).rank()
5
sage: CartanType(['I', 8]).rank()
2
```

relabel (*relabelling*)

Return a relabelled copy of this Cartan type.

INPUT:

- `relabelling` – a function (or a list or dictionary)

OUTPUT:

an isomorphic Cartan type obtained by relabelling the nodes of the Dynkin diagram. Namely, the node with label `i` is relabelled `f(i)` (or, by `f[i]` if `f` is a list or dictionary).

EXAMPLES:

```
sage: CartanType(['F',4]).relabel({ 1:4, 2:3, 3:2, 4:1 }).dynkin_diagram()
O---O=>=O---O
4   3   2   1
F4 relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

root_system()
Return the root system associated to `self`.

EXAMPLES:

```
sage: CartanType(['A',4]).root_system()
Root system of type ['A', 4]
```

subtype(*index_set*)
Return a subtype of `self` given by `index_set`.

A subtype can be considered the Dynkin diagram induced from the Dynkin diagram of `self` by `index_set`.

EXAMPLES:

```
sage: ct = CartanType(['A',6,2])
sage: ct.dynkin_diagram()
O=<=O---O=<=O
0   1   2   3
BC3~
sage: ct.subtype([1,2,3])
['C', 3]
```

type()
Return the type of `self`, or `None` if unknown.

This method should be overridden in any subclass.

EXAMPLES:

```
sage: from sage.combinat.root_system.cartan_type import CartanType_abstract
sage: C = CartanType_abstract()
sage: C.type() is None
True
```

class sage.combinat.root_system.cartan_type.**CartanType_affine**
Bases: sage.combinat.root_system.cartan_type.CartanType_simple, sage.combinat.root_system.cartan_type.CartanType_crystallographic

An abstract class for simple affine Cartan types

AmbientSpace
alias of AmbientSpace

a()
Return the unique minimal non trivial annihilating linear combination of $\alpha_0^\vee, \alpha^\vee, \ldots, \alpha^\vee$ with nonnegative coefficients (or alternatively, the unique minimal non trivial annihilating linear combination of the columns of the Cartan matrix with non-negative coefficients).

Throw an error if the existence or uniqueness does not hold

FIXME: the current implementation assumes that the Cartan matrix is indexed by $[0, 1, ...]$, in the same order as the index set.

EXAMPLES:

```
sage: RootSystem(['C',2,1]).cartan_type().a()
Finite family {0: 1, 1: 2, 2: 1}
sage: RootSystem(['D',4,1]).cartan_type().a()
Finite family {0: 1, 1: 1, 2: 2, 3: 1, 4: 1}
sage: RootSystem(['F',4,1]).cartan_type().a()
Finite family {0: 1, 1: 2, 2: 3, 3: 4, 4: 2}
sage: RootSystem(['BC',4,2]).cartan_type().a()
Finite family {0: 2, 1: 2, 2: 2, 3: 2, 4: 1}
```

a is a shortcut for col_annihilator:

```
sage: RootSystem(['BC',4,2]).cartan_type().col_annihilator()
Finite family {0: 2, 1: 2, 2: 2, 3: 2, 4: 1}
```

acheck(*m=None*)

Return the unique minimal non trivial annihilating linear combination of $\alpha_0, \alpha_1, \ldots, \alpha_n$ with nonnegative coefficients (or alternatively, the unique minimal non trivial annihilating linear combination of the rows of the Cartan matrix with non-negative coefficients).

Throw an error if the existence of uniqueness does not hold

The optional argument m is for internal use only.

EXAMPLES:

```
sage: RootSystem(['C',2,1]).cartan_type().acheck()
Finite family {0: 1, 1: 1, 2: 1}
sage: RootSystem(['D',4,1]).cartan_type().acheck()
Finite family {0: 1, 1: 1, 2: 2, 3: 1, 4: 1}
sage: RootSystem(['F',4,1]).cartan_type().acheck()
Finite family {0: 1, 1: 2, 2: 3, 3: 2, 4: 1}
sage: RootSystem(['BC',4,2]).cartan_type().acheck()
Finite family {0: 1, 1: 2, 2: 2, 3: 2, 4: 2}
```

acheck is a shortcut for row_annihilator:

```
sage: RootSystem(['BC',4,2]).cartan_type().row_annihilator()
Finite family {0: 1, 1: 2, 2: 2, 3: 2, 4: 2}
```

FIXME:

- The current implementation assumes that the Cartan matrix is indexed by $[0, 1, ...]$, in the same order as the index set.
- This really should be a method of CartanMatrix.

basic_untwisted()

Return the basic untwisted Cartan type associated with this affine Cartan type.

Given an affine type $X_n^{(r)}$, the basic untwisted type is X_n. In other words, it is the classical Cartan type that is twisted to obtain self.

EXAMPLES:

```
sage: CartanType(['A', 1, 1]).basic_untwisted()
['A', 1]
sage: CartanType(['A', 3, 1]).basic_untwisted()
['A', 3]
sage: CartanType(['B', 3, 1]).basic_untwisted()
['B', 3]
sage: CartanType(['E', 6, 1]).basic_untwisted()
['E', 6]
```

```
sage: CartanType(['G', 2, 1]).basic_untwisted()
['G', 2]

sage: CartanType(['A', 2, 2]).basic_untwisted()
['A', 2]
sage: CartanType(['A', 4, 2]).basic_untwisted()
['A', 4]
sage: CartanType(['A', 11, 2]).basic_untwisted()
['A', 11]
sage: CartanType(['D', 5, 2]).basic_untwisted()
['D', 5]
sage: CartanType(['E', 6, 2]).basic_untwisted()
['E', 6]
sage: CartanType(['D', 4, 3]).basic_untwisted()
['D', 4]
```

c()

Returns the family (c_i)_i of integer coefficients defined by $c_i = max(1, a_i/a^vee_i)$ (see e.g. [FSS07] p. 3)

FIXME: the current implementation assumes that the Cartan matrix is indexed by $[0, 1, ...]$, in the same order as the index set.

EXAMPLES:

```
sage: RootSystem(['C',2,1]).cartan_type().c()
Finite family {0: 1, 1: 2, 2: 1}
sage: RootSystem(['D',4,1]).cartan_type().c()
Finite family {0: 1, 1: 1, 2: 1, 3: 1, 4: 1}
sage: RootSystem(['F',4,1]).cartan_type().c()
Finite family {0: 1, 1: 1, 2: 1, 3: 2, 4: 2}
sage: RootSystem(['BC',4,2]).cartan_type().c()
Finite family {0: 2, 1: 1, 2: 1, 3: 1, 4: 1}
```

TESTS:

```
sage: CartanType(["B", 3, 1]).c().map(parent)
Finite family {0: Integer Ring, 1: Integer Ring, 2: Integer Ring, 3: Integer Ring}
```

REFERENCES:

classical()

Return the classical Cartan type associated with this affine Cartan type.

EXAMPLES:

```
sage: CartanType(['A', 1, 1]).classical()
['A', 1]
sage: CartanType(['A', 3, 1]).classical()
['A', 3]
sage: CartanType(['B', 3, 1]).classical()
['B', 3]

sage: CartanType(['A', 2, 2]).classical()
['C', 1]
sage: CartanType(['BC', 1, 2]).classical()
['C', 1]
sage: CartanType(['A', 4, 2]).classical()
['C', 2]
sage: CartanType(['BC', 2, 2]).classical()
```

```
['C', 2]
sage: CartanType(['A', 10, 2]).classical()
['C', 5]
sage: CartanType(['BC', 5, 2]).classical()
['C', 5]

sage: CartanType(['D', 5, 2]).classical()
['B', 4]
sage: CartanType(['E', 6, 1]).classical()
['E', 6]
sage: CartanType(['G', 2, 1]).classical()
['G', 2]
sage: CartanType(['E', 6, 2]).classical()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
sage: CartanType(['D', 4, 3]).classical()
['G', 2]
```

We check that `classical()`, `sage.combinat.root_system.cartan_type.CartanType_crystallograp` and `special_node()` are consistent:

```
sage: for ct in CartanType.samples(affine = True):
...       g1 = ct.classical().dynkin_diagram()
...       g2 = ct.dynkin_diagram()
...       g2.delete_vertex(ct.special_node())
...       assert sorted(g1.vertices()) == sorted(g2.vertices())
...       assert sorted(g1.edges()) == sorted(g2.edges())
```

col_annihilator()

Return the unique minimal non trivial annihilating linear combination of $\alpha_0^\vee, \alpha^\vee, \ldots, \alpha^\vee$ with nonnegative coefficients (or alternatively, the unique minimal non trivial annihilating linear combination of the columns of the Cartan matrix with non-negative coefficients).

Throw an error if the existence or uniqueness does not hold

FIXME: the current implementation assumes that the Cartan matrix is indexed by $[0, 1, ...]$, in the same order as the index set.

EXAMPLES:

```
sage: RootSystem(['C',2,1]).cartan_type().a()
Finite family {0: 1, 1: 2, 2: 1}
sage: RootSystem(['D',4,1]).cartan_type().a()
Finite family {0: 1, 1: 1, 2: 2, 3: 1, 4: 1}
sage: RootSystem(['F',4,1]).cartan_type().a()
Finite family {0: 1, 1: 2, 2: 3, 3: 4, 4: 2}
sage: RootSystem(['BC',4,2]).cartan_type().a()
Finite family {0: 2, 1: 2, 2: 2, 3: 2, 4: 1}
```

`a` is a shortcut for col_annihilator:

```
sage: RootSystem(['BC',4,2]).cartan_type().col_annihilator()
Finite family {0: 2, 1: 2, 2: 2, 3: 2, 4: 1}
```

is_affine()

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).is_affine()
True
```

is_finite()

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).is_finite()
False
```

is_untwisted_affine()

Return whether `self` is untwisted affine

A Cartan type is untwisted affine if it is the canonical affine extension of some finite type. Every affine type is either untwisted affine, dual thereof, or of type `BC`.

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).is_untwisted_affine()
True
sage: CartanType(['A', 3, 1]).dual().is_untwisted_affine() # this one is self dual!
True
sage: CartanType(['B', 3, 1]).dual().is_untwisted_affine()
False
sage: CartanType(['BC', 3, 2]).is_untwisted_affine()
False
```

other_affinization()

Return the other affinization of the same classical type.

EXAMPLES:

```
sage: CartanType(["A", 3, 1]).other_affinization()
['A', 3, 1]
sage: CartanType(["B", 3, 1]).other_affinization()
['C', 3, 1]^*
sage: CartanType(["C", 3, 1]).dual().other_affinization()
['B', 3, 1]
```

Is this what we want?:

```
sage: CartanType(["BC", 3, 2]).dual().other_affinization()
['B', 3, 1]
```

row_annihilator(*m=None*)

Return the unique minimal non trivial annihilating linear combination of $\alpha_0, \alpha_1, \ldots, \alpha_n$ with nonnegative coefficients (or alternatively, the unique minimal non trivial annihilating linear combination of the rows of the Cartan matrix with non-negative coefficients).

Throw an error if the existence of uniqueness does not hold

The optional argument `m` is for internal use only.

EXAMPLES:

```
sage: RootSystem(['C',2,1]).cartan_type().acheck()
Finite family {0: 1, 1: 1, 2: 1}
sage: RootSystem(['D',4,1]).cartan_type().acheck()
Finite family {0: 1, 1: 1, 2: 2, 3: 1, 4: 1}
sage: RootSystem(['F',4,1]).cartan_type().acheck()
Finite family {0: 1, 1: 2, 2: 3, 3: 2, 4: 1}
sage: RootSystem(['BC',4,2]).cartan_type().acheck()
Finite family {0: 1, 1: 2, 2: 2, 3: 2, 4: 2}
```

`acheck` is a shortcut for row_annihilator:

```
sage: RootSystem(['BC',4,2]).cartan_type().row_annihilator()
Finite family {0: 1, 1: 2, 2: 2, 3: 2, 4: 2}
```

FIXME:

- The current implementation assumes that the Cartan matrix is indexed by $[0, 1, ...]$, in the same order as the index set.
- This really should be a method of `CartanMatrix`.

special_node()

Return a special node of the Dynkin diagram.

A *special* node is a node of the Dynkin diagram such that pruning it yields a Dynkin diagram for the associated classical type (see `classical()`).

This method returns the label of some special node. This is usually 0 in the standard conventions.

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).special_node()
0
```

The choice is guaranteed to be consistent with the indexing of the nodes of the classical Dynkin diagram:

```
sage: CartanType(['A', 3, 1]).index_set()
(0, 1, 2, 3)
sage: CartanType(['A', 3, 1]).classical().index_set()
(1, 2, 3)
```

special_nodes()

Return the set of special nodes of the affine Dynkin diagram.

EXAMPLES:

```
sage: CartanType(['A',3,1]).special_nodes()
(0, 1, 2, 3)
sage: CartanType(['C',2,1]).special_nodes()
(0, 2)
sage: CartanType(['D',4,1]).special_nodes()
(0, 1, 3, 4)
sage: CartanType(['E',6,1]).special_nodes()
(0, 1, 6)
sage: CartanType(['D',3,2]).special_nodes()
(0, 2)
sage: CartanType(['A',4,2]).special_nodes()
(0,)
```

translation_factors()

Returns the translation factors for `self`. Those are the smallest factors t_i such that the translation by $t_i \alpha_i$ maps the fundamental polygon to another polygon in the alcove picture.

OUTPUT: a dictionary from `self.index_set()` to $\mathbf{Z}$ (or $\mathbf{Q}$ for affine type BC)

Those coefficients are all 1 for dual untwisted, and in particular for simply laced. They coincide with the usual c_i coefficients (see `c()`) for untwisted and dual thereof. See the discussion below for affine type BC.

Note: one usually realizes the alcove picture in the coweight lattice, with translations by coroots; in that case, one will use the translation factors for the dual Cartan type.

FIXME: the current implementation assumes that the Cartan matrix is indexed by $[0, 1, ...]$, in the same order as the index set.

EXAMPLES:

```
sage: CartanType(['C',2,1]).translation_factors()
Finite family {0: 1, 1: 2, 2: 1}
sage: CartanType(['C',2,1]).dual().translation_factors()
Finite family {0: 1, 1: 1, 2: 1}
sage: CartanType(['D',4,1]).translation_factors()
Finite family {0: 1, 1: 1, 2: 1, 3: 1, 4: 1}
sage: CartanType(['F',4,1]).translation_factors()
Finite family {0: 1, 1: 1, 2: 1, 3: 2, 4: 2}
sage: CartanType(['BC',4,2]).translation_factors()
Finite family {0: 1, 1: 1, 2: 1, 3: 1, 4: 1/2}
```

We proceed with systematic tests taken from MuPAD-Combinat's testsuite:

```
sage: list(CartanType(["A", 1, 1]).translation_factors())
[1, 1]
sage: list(CartanType(["A", 5, 1]).translation_factors())
[1, 1, 1, 1, 1, 1]
sage: list(CartanType(["B", 5, 1]).translation_factors())
[1, 1, 1, 1, 1, 2]
sage: list(CartanType(["C", 5, 1]).translation_factors())
[1, 2, 2, 2, 2, 1]
sage: list(CartanType(["D", 5, 1]).translation_factors())
[1, 1, 1, 1, 1, 1]
sage: list(CartanType(["E", 6, 1]).translation_factors())
[1, 1, 1, 1, 1, 1, 1]
sage: list(CartanType(["E", 7, 1]).translation_factors())
[1, 1, 1, 1, 1, 1, 1, 1]
sage: list(CartanType(["E", 8, 1]).translation_factors())
[1, 1, 1, 1, 1, 1, 1, 1, 1]
sage: list(CartanType(["F", 4, 1]).translation_factors())
[1, 1, 1, 2, 2]
sage: list(CartanType(["G", 2, 1]).translation_factors())
[1, 3, 1]
sage: list(CartanType(["A", 2, 2]).translation_factors())
[1, 1/2]
sage: list(CartanType(["A", 2, 2]).dual().translation_factors())
[1/2, 1]
sage: list(CartanType(["A", 10, 2]).translation_factors())
[1, 1, 1, 1, 1, 1/2]
sage: list(CartanType(["A", 10, 2]).dual().translation_factors())
[1/2, 1, 1, 1, 1, 1]
sage: list(CartanType(["A", 9, 2]).translation_factors())
[1, 1, 1, 1, 1, 1]
sage: list(CartanType(["D", 5, 2]).translation_factors())
[1, 1, 1, 1, 1]
sage: list(CartanType(["D", 4, 3]).translation_factors())
[1, 1, 1]
sage: list(CartanType(["E", 6, 2]).translation_factors())
[1, 1, 1, 1, 1]
```

We conclude with a discussion of the appropriate value for affine type BC. Let us consider the alcove picture realized in the weight lattice. It is obtained by taking the level-1 affine hyperplane in the weight lattice, and projecting it along Λ_0:

```
sage: R = RootSystem(["BC",2,2])
sage: alpha = R.weight_space().simple_roots()
sage: alphacheck = R.coroot_space().simple_roots()
sage: Lambda = R.weight_space().fundamental_weights()
```

Here are the levels of the fundamental weights:

```
sage: Lambda[0].level(), Lambda[1].level(), Lambda[2].level()
(1, 2, 2)
```

So the "center" of the fundamental polygon at level 1 is:

```
sage: O = Lambda[0]
sage: O.level()
1
```

We take the projection ω_1 at level 0 of Λ_1 as unit vector on the x-axis, and the projection ω_2 at level 0 of Λ_2 as unit vector of the y-axis:

```
sage: omega1 = Lambda[1]-2*Lambda[0]
sage: omega2 = Lambda[2]-2*Lambda[0]
sage: omega1.level(), omega2.level()
(0, 0)
```

The projections of the simple roots can be read off:

```
sage: alpha[0]
2*Lambda[0] - Lambda[1]
sage: alpha[1]
-2*Lambda[0] + 2*Lambda[1] - Lambda[2]
sage: alpha[2]
-2*Lambda[1] + 2*Lambda[2]
```

Namely $\alpha_0 = -\omega_1$, $\alpha_1 = 2\omega_1 - \omega_2$ and $\alpha_2 = -2\omega_1 + 2\omega_2$.

The reflection hyperplane defined by $\alpha_0^\vee$ goes through the points $O + 1/2\omega_1$ and $O + 1/2\omega_2$:

```
sage: (O+(1/2)*omega1).scalar(alphacheck[0])
0
sage: (O+(1/2)*omega2).scalar(alphacheck[0])
0
```

Hence, the fundamental alcove is the triangle $(O, O + 1/2\omega_1, O + 1/2\omega_2)$. By successive reflections, one can tile the full plane. This induces a tiling of the full plane by translates of the fundamental polygon.

Todo

Add the picture here, once root system plots in the weight lattice will be implemented. In the mean time, the reader may look up the dual picture on Figure 2 of [HST09] which was produced by MuPAD-Combinat.

From this picture, one can read that translations by α_0, α_1, and $1/2\alpha_2$ map the fundamental polygon to translates of it in the alcove picture, and are smallest with this property. Hence, the translation factors for affine type BC are $t_0 = 1, t_1 = 1, t_2 = 1/2$:

```
sage: CartanType(['BC',2,2]).translation_factors()
Finite family {0: 1, 1: 1, 2: 1/2}
```

TESTS:

```
sage: CartanType(["B", 3, 1]).translation_factors().map(parent)
Finite family {0: Integer Ring, 1: Integer Ring, 2: Integer Ring, 3: Integer Ring}
sage: CartanType(["BC", 3, 2]).translation_factors().map(parent)
Finite family {0: Integer Ring, 1: Integer Ring, 2: Integer Ring, 3: Rational Field}
```

REFERENCES:

class sage.combinat.root_system.cartan_type.**CartanType_crystallographic**
Bases: sage.combinat.root_system.cartan_type.CartanType_abstract

An abstract class for crystallographic Cartan types.

ascii_art(*label='lambda x: x'*, *node=None*)
Return an ascii art representation of the Dynkin diagram.

INPUT:

- label – (default: the identity) a relabeling function for the nodes
- node – (optional) a function which returns the character for a node

EXAMPLES:

```
sage: cartan_type = CartanType(['B',5,1])
sage: print cartan_type.ascii_art()
    O 0
    |
    |
O---O---O---O=>=O
1   2   3   4   5
```

The label option is useful to visualize various statistics on the nodes of the Dynkin diagram:

```
sage: a = cartan_type.col_annihilator(); a
Finite family {0: 1, 1: 1, 2: 2, 3: 2, 4: 2, 5: 2}
sage: print CartanType(['B',5,1]).ascii_art(label=a.__getitem__)
    O 1
    |
    |
O---O---O---O=>=O
1   2   2   2   2
```

cartan_matrix()
Return the Cartan matrix associated with self.

EXAMPLES:

```
sage: CartanType(['A',4]).cartan_matrix()
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]
```

coxeter_diagram()
Return the Coxeter diagram for self.

This implementation constructs it from the Dynkin diagram.

See also:

CartanType_abstract.coxeter_diagram()

EXAMPLES:

```
sage: CartanType(['A',3]).coxeter_diagram()
Graph on 3 vertices
sage: CartanType(['A',3]).coxeter_diagram().edges()
[(1, 2, 3), (2, 3, 3)]
sage: CartanType(['B',3]).coxeter_diagram().edges()
[(1, 2, 3), (2, 3, 4)]
```

```
sage: CartanType(['G',2]).coxeter_diagram().edges()
[(1, 2, 6)]
sage: CartanType(['F',4]).coxeter_diagram().edges()
[(1, 2, 3), (2, 3, 4), (3, 4, 3)]
sage: CartanType(['A',2,2]).coxeter_diagram().edges()
[(0, 1, +Infinity)]
```

dynkin_diagram()

Return the Dynkin diagram associated with `self`.

EXAMPLES:

```
sage: CartanType(['A',4]).dynkin_diagram()
O---O---O---O
1   2   3   4
A4
```

Note: Derived subclasses should typically implement this as a cached method.

---

index_set_bipartition()

Return a bipartition $\{L, R\}$ of the vertices of the Dynkin diagram.

For i and j both in L (or both in R), the simple reflections s_i and s_j commute.

Of course, the Dynkin diagram should be bipartite. This is always the case for all finite types.

EXAMPLES:

```
sage: CartanType(['A',5]).index_set_bipartition()
({1, 3, 5}, {2, 4})

sage: CartanType(['A',2,1]).index_set_bipartition()
Traceback (most recent call last):
...
ValueError: the Dynkin diagram must be bipartite
```

is_crystallographic()

Implements `CartanType_abstract.is_crystallographic()` by returning `True`.

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).is_crystallographic()
True
```

symmetrizer()

Return the symmetrizer of the Cartan matrix of `self`.

A Cartan matrix M is symmetrizable if there exists a non trivial diagonal matrix D such that DM is a symmetric matrix, that is $DM = M^t D$. In that case, D is unique, up to a scalar factor for each connected component of the Dynkin diagram.

This method computes the unique minimal such D with positive integral coefficients. If D exists, it is returned as a family. Otherwise `None` is returned.

The coefficients are coerced to `base_ring`.

EXAMPLES:

```
sage: CartanType(["B",5]).symmetrizer()
Finite family {1: 2, 2: 2, 3: 2, 4: 2, 5: 1}
```

Here is a neat trick to visualize it better:

```
sage: T = CartanType(["B",5])
sage: print T.ascii_art(T.symmetrizer().__getitem__)
O---O---O---O=>=O
2   2   2   2   1

sage: T = CartanType(["BC",5, 2])
sage: print T.ascii_art(T.symmetrizer().__getitem__)
O=<=O---O---O---O=<=O
1   2   2   2   2   4
```

Here is the symmetrizer of some reducible Cartan types:

```
sage: T = CartanType(["D", 2])
sage: print T.ascii_art(T.symmetrizer().__getitem__)
O   O
1   1

sage: T = CartanType(["B",5],["BC",5, 2])
sage: print T.ascii_art(T.symmetrizer().__getitem__)
O---O---O---O=>=O
2   2   2   2   1
O=<=O---O---O---O=<=O
1   2   2   2   2   4
```

Property: up to an overall scalar factor, this gives the norm of the simple roots in the ambient space::

```
sage: T = CartanType(["C",5])
sage: print T.ascii_art(T.symmetrizer().__getitem__)
O---O---O---O=<=O
1   1   1   1   2

sage: alpha = RootSystem(T).ambient_space().simple_roots()
sage: print T.ascii_art(lambda i: alpha[i].scalar(alpha[i]))
O---O---O---O=<=O
2   2   2   2   4
```

class sage.combinat.root_system.cartan_type.**CartanType_decorator**(*ct*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.sage_object.SageObject, sage.combinat.root_system.cartan_type.CartanType_

Concrete base class for Cartan types that decorate another Cartan type.

index_set()

EXAMPLES:

```
sage: ct = CartanType(['F', 4, 1]).dual()
sage: ct.index_set()
(0, 1, 2, 3, 4)
```

is_affine()

EXAMPLES:

```
sage: ct = CartanType(['G', 2]).relabel({1:2,2:1})
sage: ct.is_affine()
False
```

is_crystallographic()

EXAMPLES:

```
sage: ct = CartanType(['G', 2]).relabel({1:2,2:1})
sage: ct.is_crystallographic()
True
```

is_finite()

EXAMPLES:

```
sage: ct = CartanType(['G', 2]).relabel({1:2,2:1})
sage: ct.is_finite()
True
```

is_irreducible()

EXAMPLES:

```
sage: ct = CartanType(['G', 2]).relabel({1:2,2:1})
sage: ct.is_irreducible()
True
```

rank()

EXAMPLES:

```
sage: ct = CartanType(['G', 2]).relabel({1:2,2:1})
sage: ct.rank()
2
```

class sage.combinat.root_system.cartan_type.**CartanType_finite**

Bases: sage.combinat.root_system.cartan_type.CartanType_abstract

An abstract class for simple affine Cartan types.

is_affine()

EXAMPLES:

```
sage: CartanType(["A", 3]).is_affine()
False
```

is_finite()

EXAMPLES:

```
sage: CartanType(["A", 3]).is_finite()
True
```

class sage.combinat.root_system.cartan_type.**CartanType_simple**

Bases: sage.combinat.root_system.cartan_type.CartanType_abstract

An abstract class for simple Cartan types.

is_irreducible()

Return whether `self` is irreducible, which is `True`.

EXAMPLES:

```
sage: CartanType(['A', 3]).is_irreducible()
True
```

class sage.combinat.root_system.cartan_type.**CartanType_simple_finite**
Bases: object

class sage.combinat.root_system.cartan_type.**CartanType_simply_laced**
Bases: sage.combinat.root_system.cartan_type.CartanType_crystallographic

An abstract class for simply laced Cartan types.

dual()
Simply laced Cartan types are self-dual, so return self.

EXAMPLES:

```
sage: CartanType(["A", 3]).dual()
['A', 3]
sage: CartanType(["A", 3, 1]).dual()
['A', 3, 1]
sage: CartanType(["D", 3]).dual()
['D', 3]
sage: CartanType(["D", 4, 1]).dual()
['D', 4, 1]
sage: CartanType(["E", 6]).dual()
['E', 6]
sage: CartanType(["E", 6, 1]).dual()
['E', 6, 1]
```

is_simply_laced()
Return whether self is simply laced, which is True.

EXAMPLES:

```
sage: CartanType(['A',3,1]).is_simply_laced()
True
sage: CartanType(['A',2]).is_simply_laced()
True
```

class sage.combinat.root_system.cartan_type.**CartanType_standard_affine**(*letter*, *n*, *affine=1*)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.sage_object.SageObject, sage.combinat.root_system.cartan_type.CartanType_

A concrete class for affine simple Cartan types.

index_set()
Implements CartanType_abstract.index_set().

The index set for all standard affine Cartan types is of the form $\{0, \ldots, n\}$.

EXAMPLES:

```
sage: CartanType(['A', 5, 1]).index_set()
(0, 1, 2, 3, 4, 5)
```

rank()
Return the rank of self which for type $X_n^{(1)}$ is $n + 1$.

EXAMPLES:

```
sage: CartanType(['A', 4, 1]).rank()
5
sage: CartanType(['B', 4, 1]).rank()
5
```

```
sage: CartanType(['C', 3, 1]).rank()
4
sage: CartanType(['D', 4, 1]).rank()
5
sage: CartanType(['E', 6, 1]).rank()
7
sage: CartanType(['E', 7, 1]).rank()
8
sage: CartanType(['F', 4, 1]).rank()
5
sage: CartanType(['G', 2, 1]).rank()
3
sage: CartanType(['A', 2, 2]).rank()
2
sage: CartanType(['A', 6, 2]).rank()
4
sage: CartanType(['A', 7, 2]).rank()
5
sage: CartanType(['D', 5, 2]).rank()
5
sage: CartanType(['E', 6, 2]).rank()
5
sage: CartanType(['D', 4, 3]).rank()
3
```

special_node()

Implement `CartanType_abstract.special_node()`.

With the standard labelling conventions, 0 is always a special node.

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).special_node()
0
```

type()

Return the type of `self`.

EXAMPLES:

```
sage: CartanType(['A', 4, 1]).type()
'A'
```

class sage.combinat.root_system.cartan_type.**CartanType_standard_finite**(*letter*, *n*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.sage_object.SageObject, sage.combinat.root_system.cartan_type.CartanType_

A concrete base class for the finite standard Cartan types.

This includes for example A_3, D_4, or E_8.

affine()

Return the corresponding untwisted affine Cartan type.

EXAMPLES:

```
sage: CartanType(['A',3]).affine()
['A', 3, 1]
```

coxeter_number()

Return the Coxeter number associated with `self`.

The Coxeter number is the order of a Coxeter element of the corresponding Weyl group.

See Bourbaki, Lie Groups and Lie Algebras V.6.1 or Wikipedia article Coxeter_element for more information.

EXAMPLES:

```
sage: CartanType(['A',4]).coxeter_number()
5
sage: CartanType(['B',4]).coxeter_number()
8
sage: CartanType(['C',4]).coxeter_number()
8
```

dual_coxeter_number()

Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['A',4]).dual_coxeter_number()
5
sage: CartanType(['B',4]).dual_coxeter_number()
7
sage: CartanType(['C',4]).dual_coxeter_number()
5
```

index_set()

Implements `CartanType_abstract.index_set()`.

The index set for all standard finite Cartan types is of the form $\{1, \ldots, n\}$. (See `type_I` for a slight abuse of this).

EXAMPLES:

```
sage: CartanType(['A', 5]).index_set()
(1, 2, 3, 4, 5)
```

opposition_automorphism()

Returns the opposition automorphism

The *opposition automorphism* is the automorphism $i \mapsto i^*$ of the vertices Dynkin diagram such that, for w_0 the longest element of the Weyl group, and any simple root α_i, one has $\alpha_{i^*} = -w_0(\alpha_i)$.

The automorphism is returned as a `Family`.

EXAMPLES:

```
sage: ct = CartanType(['A', 5])
sage: ct.opposition_automorphism()
Finite family {1: 5, 2: 4, 3: 3, 4: 2, 5: 1}

sage: ct = CartanType(['D', 4])
sage: ct.opposition_automorphism()
Finite family {1: 1, 2: 2, 3: 3, 4: 4}

sage: ct = CartanType(['D', 5])
sage: ct.opposition_automorphism()
Finite family {1: 1, 2: 2, 3: 3, 4: 5, 5: 4}

sage: ct = CartanType(['C', 4])
sage: ct.opposition_automorphism()
Finite family {1: 1, 2: 2, 3: 3, 4: 4}
```

rank()
Return the rank of `self` which for type X_n is n.

EXAMPLES:

```
sage: CartanType(['A', 3]).rank()
3
sage: CartanType(['B', 3]).rank()
3
sage: CartanType(['C', 3]).rank()
3
sage: CartanType(['D', 4]).rank()
4
sage: CartanType(['E', 6]).rank()
6
```

type()
Returns the type of `self`.

EXAMPLES:

```
sage: CartanType(['A', 4]).type()
'A'
sage: CartanType(['A', 4, 1]).type()
'A'
```

class sage.combinat.root_system.cartan_type.**CartanType_standard_untwisted_affine**(*letter*, *n*, *affine=1*)
Bases: sage.combinat.root_system.cartan_type.CartanType_standard_affine

A concrete class for the standard untwisted affine Cartan types.

basic_untwisted()
Return the basic_untwisted Cartan type associated with this affine Cartan type.

Given an affine type $X_n^{(r)}$, the basic_untwisted type is X_n. In other words, it is the classical Cartan type that is twisted to obtain `self`.

EXAMPLES:

```
sage: CartanType(['A', 1, 1]).basic_untwisted()
['A', 1]
sage: CartanType(['A', 3, 1]).basic_untwisted()
['A', 3]
sage: CartanType(['B', 3, 1]).basic_untwisted()
['B', 3]
sage: CartanType(['E', 6, 1]).basic_untwisted()
['E', 6]
sage: CartanType(['G', 2, 1]).basic_untwisted()
['G', 2]
```

classical()
Return the classical Cartan type associated with `self`.

EXAMPLES:

```
sage: CartanType(['A', 3, 1]).classical()
['A', 3]
sage: CartanType(['B', 3, 1]).classical()
['B', 3]
sage: CartanType(['C', 3, 1]).classical()
```

```
['C', 3]
sage: CartanType(['D', 4, 1]).classical()
['D', 4]
sage: CartanType(['E', 6, 1]).classical()
['E', 6]
sage: CartanType(['F', 4, 1]).classical()
['F', 4]
sage: CartanType(['G', 2, 1]).classical()
['G', 2]
```

is_untwisted_affine()

Implement CartanType_affine.is_untwisted_affine() by returning True.

EXAMPLES:

```
sage: CartanType(['B', 3, 1]).is_untwisted_affine()
True
```

5.1.198 Coxeter Groups

sage.combinat.root_system.coxeter_group.**CoxeterGroup**(*data*, *implementation='reflection'*, *base_ring=None*, *index_set=None*)

Return an implementation of the Coxeter group given by data.

INPUT:

- data – a Cartan type (or coercible into; see CartanType) or a Coxeter matrix or graph
- implementation – (default: 'reflection') can be one of the following:
  - 'permutation' - as a permutation representation
  - 'matrix' - as a Weyl group (as a matrix group acting on the root space); if this is not implemented, this uses the "reflection" implementation
  - 'coxeter3' - using the coxeter3 package
  - 'reflection' - as elements in the reflection representation; see CoxeterMatrixGroup
- base_ring – (optional) the base ring for the 'reflection' implementation
- index_set – (optional) the index set for the 'reflection' implementation

EXAMPLES:

Now assume that data represents a Cartan type. If implementation is not specified, the reflection representation is returned:

```
sage: W = CoxeterGroup(["A",2])
sage: W
Finite Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[1 3]
[3 1]

sage: W = CoxeterGroup(["A",3,1]); W
Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[1 3 2 3]
[3 1 3 2]
[2 3 1 3]
```

```
[3 2 3 1]

sage: W = CoxeterGroup(['H',3]); W
Finite Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[1 3 2]
[3 1 5]
[2 5 1]
```

We now use the `implementation` option:

```
sage: W = CoxeterGroup(["A",2], implementation = "permutation") # optional - chevie
sage: W                                                         # optional - chevie
Permutation Group with generators [(1,3)(2,5)(4,6), (1,4)(2,3)(5,6)]
sage: W.category()                          # optional - chevie
Join of Category of finite permutation groups and Category of finite coxeter groups

sage: W = CoxeterGroup(["A",2], implementation="matrix")
sage: W
Weyl Group of type ['A', 2] (as a matrix group acting on the ambient space)

sage: W = CoxeterGroup(["H",3], implementation="matrix")
sage: W
Finite Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[1 3 2]
[3 1 5]
[2 5 1]

sage: W = CoxeterGroup(["H",3], implementation="reflection")
sage: W
Finite Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[1 3 2]
[3 1 5]
[2 5 1]

sage: W = CoxeterGroup(["A",4,1], implementation="permutation")
Traceback (most recent call last):
...
NotImplementedError: Coxeter group of type ['A', 4, 1] as permutation group not implemented
```

We use the different options for the "reflection" implementation:

```
sage: W = CoxeterGroup(["H",3], implementation="reflection", base_ring=RR)
sage: W
Finite Coxeter group over Real Field with 53 bits of precision with Coxeter matrix:
[1 3 2]
[3 1 5]
[2 5 1]
sage: W = CoxeterGroup([[1,10],[10,1]], implementation="reflection", index_set=['a','b'], base_r
sage: W
Finite Coxeter group over Symbolic Ring with Coxeter matrix:
[ 1 10]
[10  1]
```

TESTS:

```
sage: W = groups.misc.CoxeterGroup(["H",3])
```

class sage.combinat.root_system.coxeter_group.**CoxeterGroupAsPermutationGroup**(*cartan_type*)

Bases: sage.structure.unique_representation.UniqueRepresentation,

sage.groups.perm_gps.permgroup.PermutationGroup_generic

Construct this Coxeter group as a Sage permutation group, by fetching the permutation representation of the generators from Chevie's database.

TESTS:

```
sage: from sage.combinat.root_system.coxeter_group import CoxeterGroupAsPermutationGroup
sage: W = CoxeterGroupAsPermutationGroup(CartanType(["H",3])) # optional - chevie
sage: TestSuite(W).run()              # optional - chevie
```

class Element

Bases: sage.groups.perm_gps.permgroup_element.PermutationGroupElement

has_descent(*i*, *side='right'*, *positive=False*)

Returns whether i is a (left/right) descent of self.

See sage.categories.coxeter_groups.CoxeterGroups.ElementMethods.descents() for a description of the options.

EXAMPLES:

```
sage: W = CoxeterGroup(["A",3])
sage: s = W.simple_reflections()
sage: w = s[1] * s[2] * s[3]
sage: w.has_descent(3)
True
sage: [ w.has_descent(i)                    for i in [1,2,3] ]
[False, False, True]
sage: [ w.has_descent(i, side = 'left')     for i in [1,2,3] ]
[True, False, False]
sage: [ w.has_descent(i, positive = True) for i in [1,2,3] ]
[True, True, False]
```

This implementation is a plain copy of that of CoxeterGroups. It is there as a workaround since *PermutationGroupElement* currently redefines abusively has_descent() as if the group was the full symmetric group.

has_left_descent(*i*)

Returns whether i is a descent of self by testing whether i is mapped to a negative root.

EXAMPLES:

```
sage: W = CoxeterGroup(["A",3], implementation = "permutation") # optional - chevie
sage: s = W.simple_reflections() # optional - chevie
sage: (s[1]*s[2]).has_left_descent(1) # optional - chevie
True
sage: (s[1]*s[2]).has_left_descent(2) # optional - chevie
False
```

CoxeterGroupAsPermutationGroup.**index_set**()

Returns the index set of this Coxeter group

EXAMPLES:

```
sage: W = CoxeterGroup(["H",3], implementation = "permutation")  # optional - chevie
sage: W.index_set() # optional - chevie
[1, 2, 3]
```

CoxeterGroupAsPermutationGroup.**reflection**(*i*)

Returns the i-th reflection of self.

For i in $1, \ldots, n$, this gives the i-th simple reflection of self.

EXAMPLES:

```
sage: W = CoxeterGroup(["H",3], implementation = "permutation") # optional - chevie
sage: W.simple_reflection(1) # optional - chevie
(1,16)(2,5)(4,7)(6,9)(8,10)(11,13)(12,14)(17,20)(19,22)(21,24)(23,25)(26,28)(27,29)
sage: W.simple_reflection(2) # optional - chevie
(1,4)(2,17)(3,6)(5,7)(9,11)(10,12)(14,15)(16,19)(18,21)(20,22)(24,26)(25,27)(29,30)
sage: W.simple_reflection(3) # optional - chevie
(2,6)(3,18)(4,8)(5,9)(7,10)(11,12)(13,14)(17,21)(19,23)(20,24)(22,25)(26,27)(28,29)
sage: W.reflection(4)        # optional - chevie
(1,5)(2,22)(3,11)(4,19)(7,17)(8,12)(9,13)(10,15)(16,20)(18,26)(23,27)(24,28)(25,30)
sage: W.reflection(5)        # optional - chevie
(1,22)(2,4)(3,9)(5,20)(6,13)(7,16)(8,14)(12,15)(17,19)(18,24)(21,28)(23,29)(27,30)
sage: W.reflection(6)        # optional - chevie
(1,8)(2,18)(3,17)(5,12)(6,21)(7,11)(9,10)(13,15)(16,23)(20,27)(22,26)(24,25)(28,30)
```

CoxeterGroupAsPermutationGroup.**simple_reflection**(i)

Returns the i-th reflection of `self`.

For i in $1, \ldots, n$, this gives the i-th simple reflection of `self`.

EXAMPLES:

```
sage: W = CoxeterGroup(["H",3], implementation = "permutation") # optional - chevie
sage: W.simple_reflection(1) # optional - chevie
(1,16)(2,5)(4,7)(6,9)(8,10)(11,13)(12,14)(17,20)(19,22)(21,24)(23,25)(26,28)(27,29)
sage: W.simple_reflection(2) # optional - chevie
(1,4)(2,17)(3,6)(5,7)(9,11)(10,12)(14,15)(16,19)(18,21)(20,22)(24,26)(25,27)(29,30)
sage: W.simple_reflection(3) # optional - chevie
(2,6)(3,18)(4,8)(5,9)(7,10)(11,12)(13,14)(17,21)(19,23)(20,24)(22,25)(26,27)(28,29)
sage: W.reflection(4)        # optional - chevie
(1,5)(2,22)(3,11)(4,19)(7,17)(8,12)(9,13)(10,15)(16,20)(18,26)(23,27)(24,28)(25,30)
sage: W.reflection(5)        # optional - chevie
(1,22)(2,4)(3,9)(5,20)(6,13)(7,16)(8,14)(12,15)(17,19)(18,24)(21,28)(23,29)(27,30)
sage: W.reflection(6)        # optional - chevie
(1,8)(2,18)(3,17)(5,12)(6,21)(7,11)(9,10)(13,15)(16,23)(20,27)(22,26)(24,25)(28,30)
```

sage.combinat.root_system.coxeter_group.**is_chevie_available**()

Tests whether the GAP3 Chevie package is available

EXAMPLES:

```
sage: from sage.combinat.root_system.coxeter_group import is_chevie_available
sage: is_chevie_available() # random
False
sage: is_chevie_available() in [True, False]
True
```

5.1.199 Coxeter Matrices

class sage.combinat.root_system.coxeter_matrix.**CoxeterMatrix**(*parent*, *data*, *coxeter_type*, *index_set*)

Bases: sage.combinat.root_system.coxeter_type.CoxeterType

A Coxeter matrix.

A Coxeter matrix $M = (m_{ij})_{i,j \in I}$ is a matrix encoding a Coxeter system (W, S), where the relations are given by $(s_i s_j)^{m_{ij}}$. Thus M is symmetric and has entries in $\{1, 2, 3, \ldots, \infty\}$ with $m_{ij} = 1$ if and only if $i = j$.

We represent $m_{ij} = \infty$ by any number $m_{ij} \leq -1$. In particular, we can construct a bilinear form $B = (b_{ij})_{i,j \in I}$ from M by

$$b_{ij} = \begin{cases} m_{ij} & m_{ij} < 0 \text{ (i.e., } m_{ij} = \infty), \\ -\cos\left(\frac{\pi}{m_{ij}}\right) & \text{otherwise.} \end{cases}$$

EXAMPLES:

```
sage: CoxeterMatrix(['A', 4])
[1 3 2 2]
[3 1 3 2]
[2 3 1 3]
[2 2 3 1]
sage: CoxeterMatrix(['B', 4])
[1 3 2 2]
[3 1 3 2]
[2 3 1 4]
[2 2 4 1]
sage: CoxeterMatrix(['C', 4])
[1 3 2 2]
[3 1 3 2]
[2 3 1 4]
[2 2 4 1]
sage: CoxeterMatrix(['D', 4])
[1 3 2 2]
[3 1 3 3]
[2 3 1 2]
[2 3 2 1]

sage: CoxeterMatrix(['E', 6])
[1 2 3 2 2 2]
[2 1 2 3 2 2]
[3 2 1 3 2 2]
[2 3 3 1 3 2]
[2 2 2 3 1 3]
[2 2 2 2 3 1]

sage: CoxeterMatrix(['F', 4])
[1 3 2 2]
[3 1 4 2]
[2 4 1 3]
[2 2 3 1]

sage: CoxeterMatrix(['G', 2])
[1 6]
[6 1]
```

By default, entries representing ∞ are given by -1 in the Coxeter matrix:

```
sage: G = Graph([(0,1,None), (1,2,4), (0,2,oo)])
sage: CoxeterMatrix(G)
[ 1  3 -1]
[ 3  1  4]
[-1  4  1]
```

It is possible to give a number ≤ -1 to represent an infinite label:

```
sage: CoxeterMatrix([[1,-1],[-1,1]])
[ 1 -1]
[-1  1]
```

```
sage: CoxeterMatrix([[1,-3/2],[-3/2,1]])
[   1 -3/2]
[-3/2    1]
```

bilinear_form(*R=None*)

Return the bilinear form of `self`.

EXAMPLES:

```
sage: CoxeterType(['A', 2, 1]).bilinear_form()
[   1 -1/2 -1/2]
[-1/2    1 -1/2]
[-1/2 -1/2    1]
sage: CoxeterType(['H', 3]).bilinear_form()
[                     1                     -1/2                                0]
[                  -1/2                        1 1/2*E(5)^2 + 1/2*E(5)^3]
[                     0 1/2*E(5)^2 + 1/2*E(5)^3                                1]
sage: C = CoxeterMatrix([[1,-1,-1],[-1,1,-1],[-1,-1,1]])
sage: C.bilinear_form()
[ 1 -1 -1]
[-1  1 -1]
[-1 -1  1]
```

coxeter_graph()

Return the Coxeter graph of `self`.

EXAMPLES:

```
sage: C = CoxeterMatrix(['A',3])
sage: C.coxeter_graph()
Graph on 3 vertices

sage: C = CoxeterMatrix([['A',3],['A',1]])
sage: C.coxeter_graph()
Graph on 4 vertices
```

coxeter_matrix()

Return the Coxeter matrix of `self`.

EXAMPLES:

```
sage: CoxeterMatrix(['C',3]).coxeter_matrix()
[1 3 2]
[3 1 4]
[2 4 1]
```

coxeter_type()

Return the Coxeter type of `self` or `self` if unknown.

EXAMPLES:

```
sage: C = CoxeterMatrix(['A',4,1])
sage: C.coxeter_type()
Coxeter type of ['A', 4, 1]
```

If the Coxeter type is unknown:

```
sage: C = CoxeterMatrix([[1,3,4], [3,1,-1], [4,-1,1]])
sage: C.coxeter_type()
[ 1  3  4]
```

```
[ 3  1 -1]
[ 4 -1  1]
```

index_set()
Return the index set of `self`.

EXAMPLES:

```
sage: C = CoxeterMatrix(['A',1,1])
sage: C.index_set()
(0, 1)
sage: C = CoxeterMatrix(['E',6])
sage: C.index_set()
(1, 2, 3, 4, 5, 6)
```

is_affine()
Return if `self` is an affine type or `False` if unknown.

EXAMPLES:

```
sage: M = CoxeterMatrix(['C',4])
sage: M.is_affine()
False
sage: M = CoxeterMatrix(['D',4,1])
sage: M.is_affine()
True
sage: M = CoxeterMatrix([[1, 3],[3,1]])
sage: M.is_affine()
False
sage: M = CoxeterMatrix([[1, -1, 7], [-1, 1, 3], [7, 3, 1]])
sage: M.is_affine()
False
```

is_crystallographic()
Return if `self` is crystallographic.

A Coxeter matrix is crystallographic if all non-diagonal entries are either 2, 4, or 6.

EXAMPLES:

```
sage: CoxeterMatrix(['F',4]).is_crystallographic()
True
sage: CoxeterMatrix(['H',3]).is_crystallographic()
False
```

is_finite()
Return if `self` is a finite type or `False` if unknown.

EXAMPLES:

```
sage: M = CoxeterMatrix(['C',4])
sage: M.is_finite()
True
sage: M = CoxeterMatrix(['D',4,1])
sage: M.is_finite()
False
sage: M = CoxeterMatrix([[1, -1], [-1, 1]])
sage: M.is_finite()
False
```

is_simply_laced()
Return if `self` is simply-laced.

A Coxeter matrix is simply-laced if all non-diagonal entries are either 2 or 3.

EXAMPLES:
```
sage: cm = CoxeterMatrix([[1,3,3,3], [3,1,3,3], [3,3,1,3], [3,3,3,1]])
sage: cm.is_simply_laced()
True
```

rank()
Return the rank of `self`.

EXAMPLES:
```
sage: CoxeterMatrix(['C',3]).rank()
3
sage: CoxeterMatrix(["A2","B2","F4"]).rank()
8
```

relabel(*relabelling*)
Return a relabelled copy of this Coxeter matrix.

INPUT:

- `relabelling` – a function (or dictionary)

OUTPUT:

an isomorphic Coxeter type obtained by relabelling the nodes of the Coxeter graph. Namely, the node with label `i` is relabelled `f(i)` (or, by `f[i]` if `f` is a dictionary).

EXAMPLES:
```
sage: CoxeterMatrix(['F',4]).relabel({ 1:2, 2:3, 3:4, 4:1})
[1 4 2 3]
[4 1 3 2]
[2 3 1 2]
[3 2 2 1]
sage: CoxeterMatrix(['F',4]).relabel(lambda x: x+1 if x<4 else 1)
[1 4 2 3]
[4 1 3 2]
[2 3 1 2]
[3 2 2 1]
```

classmethod samples(*finite=None*, *affine=None*, *crystallographic=None*, *higher_rank=None*)
Return a sample of the available Coxeter types.

INPUT:

- `finite` – (default: `None`) a boolean or `None`
- `affine` – (default: `None`) a boolean or `None`
- `crystallographic` – (default: `None`) a boolean or `None`
- `higher_rank` – (default: `None`) a boolean or `None`

The sample contains all the exceptional finite and affine Coxeter types, as well as typical representatives of the infinite families.

Here the `higher_rank` term denotes non-finite, non-affine, Coxeter groups (including hyperbolic types).

Todo

Implement the hyperbolic and compact hyperbolic in the samples.

EXAMPLES:

```
sage: [CM.coxeter_type() for CM in CoxeterMatrix.samples()]
[
Coxeter type of ['A', 1], Coxeter type of ['A', 5],

Coxeter type of ['B', 5], Coxeter type of ['D', 4],

Coxeter type of ['D', 5], Coxeter type of ['E', 6],

Coxeter type of ['E', 7], Coxeter type of ['E', 8],

Coxeter type of ['F', 4], Coxeter type of ['H', 3],

Coxeter type of ['H', 4], Coxeter type of ['I', 10],

Coxeter type of ['A', 2, 1], Coxeter type of ['B', 5, 1],

Coxeter type of ['C', 5, 1], Coxeter type of ['D', 5, 1],

Coxeter type of ['E', 6, 1], Coxeter type of ['E', 7, 1],

Coxeter type of ['E', 8, 1], Coxeter type of ['F', 4, 1],

                                                          [ 1 -1 -1]
                                                          [-1  1 -1]
Coxeter type of ['G', 2, 1], Coxeter type of ['A', 1, 1], [-1 -1  1],

          [ 1 -2  3  2]
[1 2 3]   [-2  1  2  3]
[2 1 7]   [ 3  2  1 -8]
[3 7 1], [ 2  3 -8  1]
]
```

The finite, affine and crystallographic options allow respectively for restricting to (non) finite, (non) affine, and (non) crystallographic Cartan types:

```
sage: [CM.coxeter_type() for CM in CoxeterMatrix.samples(finite=True)]
[Coxeter type of ['A', 1], Coxeter type of ['A', 5],
 Coxeter type of ['B', 5], Coxeter type of ['D', 4],
 Coxeter type of ['D', 5], Coxeter type of ['E', 6],
 Coxeter type of ['E', 7], Coxeter type of ['E', 8],
 Coxeter type of ['F', 4], Coxeter type of ['H', 3],
 Coxeter type of ['H', 4], Coxeter type of ['I', 10]]

sage: [CM.coxeter_type() for CM in CoxeterMatrix.samples(affine=True)]
[Coxeter type of ['A', 2, 1], Coxeter type of ['B', 5, 1],
 Coxeter type of ['C', 5, 1], Coxeter type of ['D', 5, 1],
 Coxeter type of ['E', 6, 1], Coxeter type of ['E', 7, 1],
 Coxeter type of ['E', 8, 1], Coxeter type of ['F', 4, 1],
 Coxeter type of ['G', 2, 1], Coxeter type of ['A', 1, 1]]

sage: [CM.coxeter_type() for CM in CoxeterMatrix.samples(crystallographic=True)]
[Coxeter type of ['A', 1], Coxeter type of ['A', 5],
 Coxeter type of ['B', 5], Coxeter type of ['D', 4],
 Coxeter type of ['D', 5], Coxeter type of ['E', 6],
 Coxeter type of ['E', 7], Coxeter type of ['E', 8],
```

```
 Coxeter type of ['F', 4], Coxeter type of ['A', 2, 1],
 Coxeter type of ['B', 5, 1], Coxeter type of ['C', 5, 1],
 Coxeter type of ['D', 5, 1], Coxeter type of ['E', 6, 1],
 Coxeter type of ['E', 7, 1], Coxeter type of ['E', 8, 1],
 Coxeter type of ['F', 4, 1], Coxeter type of ['G', 2, 1]]
```

```
sage: CoxeterMatrix.samples(crystallographic=False)
[
          [1 3 2 2]
[1 3 2]   [3 1 3 2]                     [ 1 -1 -1]   [1 2 3]
[3 1 5]   [2 3 1 5]   [ 1 10]   [ 1 -1]   [-1  1 -1]   [2 1 7]
[2 5 1], [2 2 5 1], [10  1], [-1  1], [-1 -1  1], [3 7 1],

[ 1 -2  3  2]
[-2  1  2  3]
[ 3  2  1 -8]
[ 2  3 -8  1]
]
```

Todo

add some reducible Coxeter types (suggestions?)

TESTS:

```
sage: for ct in CoxeterMatrix.samples(): TestSuite(ct).run()
```

sage.combinat.root_system.coxeter_matrix.**check_coxeter_matrix**(*m*)

Check if m represents a generalized Coxeter matrix and raise and error if not.

EXAMPLES:

```
sage: from sage.combinat.root_system.coxeter_matrix import check_coxeter_matrix
sage: m = matrix([[1,3,2],[3,1,-1],[2,-1,1]])
sage: check_coxeter_matrix(m)

sage: m = matrix([[1,3],[3,1],[2,-1]])
sage: check_coxeter_matrix(m)
Traceback (most recent call last):
...
ValueError: not a square matrix

sage: m = matrix([[1,3,2],[3,1,-1],[2,-1,2]])
sage: check_coxeter_matrix(m)
Traceback (most recent call last):
...
ValueError: the matrix diagonal is not all 1

sage: m = matrix([[1,3,3],[3,1,-1],[2,-1,1]])
sage: check_coxeter_matrix(m)
Traceback (most recent call last):
...
ValueError: the matrix is not symmetric

sage: m = matrix([[1,3,1/2],[3,1,-1],[1/2,-1,1]])
sage: check_coxeter_matrix(m)
Traceback (most recent call last):
...
ValueError: invalid Coxeter label 1/2
```

```
sage: m = matrix([[1,3,1],[3,1,-1],[1,-1,1]])
sage: check_coxeter_matrix(m)
Traceback (most recent call last):
...
ValueError: invalid Coxeter label 1
```

sage.combinat.root_system.coxeter_matrix.**coxeter_matrix**(*t*)

This was deprecated in trac ticket #17798 for CartanMatrix.

EXAMPLES:

```
sage: coxeter_matrix(['A', 4])
doctest:...: DeprecationWarning: coxeter_matrix() is deprecated. Use CoxeterMatrix() instead
See http://trac.sagemath.org/17798 for details.
[1 3 2 2]
[3 1 3 2]
[2 3 1 3]
[2 2 3 1]
```

sage.combinat.root_system.coxeter_matrix.**coxeter_matrix_as_function**(*t*)

Return the Coxeter matrix, as a function.

INPUT:

- t – a Cartan type

EXAMPLES:

```
sage: from sage.combinat.root_system.coxeter_matrix import coxeter_matrix_as_function
sage: f = coxeter_matrix_as_function(['A',4])
sage: matrix([[f(i,j) for j in range(1,5)] for i in range(1,5)])
[1 3 2 2]
[3 1 3 2]
[2 3 1 3]
[2 2 3 1]
```

sage.combinat.root_system.coxeter_matrix.**recognize_coxeter_type_from_matrix**(*coxeter_matrix*, *index_set*)

Return the Coxeter type of coxeter_matrix if known, otherwise return None.

EXAMPLES:

Some infinite ones:

```
sage: C = CoxeterMatrix([[1,3,2],[3,1,-1],[2,-1,1]])
sage: C.is_finite()  # indirect doctest
False
sage: C = CoxeterMatrix([[1,-1,-1],[-1,1,-1],[-1,-1,1]])
sage: C.is_finite()  # indirect doctest
False
```

Some finite ones:

```
sage: m = matrix(CoxeterMatrix(['D', 4]))
sage: CoxeterMatrix(m).is_finite()  # indirect doctest
True
sage: m = matrix(CoxeterMatrix(['H', 4]))
sage: CoxeterMatrix(m).is_finite()  # indirect doctest
True
```

```
sage: CoxeterMatrix(CoxeterType(['A',10]).coxeter_graph()).coxeter_type()
Coxeter type of ['A', 10]
sage: CoxeterMatrix(CoxeterType(['B',10]).coxeter_graph()).coxeter_type()
Coxeter type of ['B', 10]
sage: CoxeterMatrix(CoxeterType(['C',10]).coxeter_graph()).coxeter_type()
Coxeter type of ['B', 10]
sage: CoxeterMatrix(CoxeterType(['D',10]).coxeter_graph()).coxeter_type()
Coxeter type of ['D', 10]
sage: CoxeterMatrix(CoxeterType(['E',6]).coxeter_graph()).coxeter_type()
Coxeter type of ['E', 6]
sage: CoxeterMatrix(CoxeterType(['E',7]).coxeter_graph()).coxeter_type()
Coxeter type of ['E', 7]
sage: CoxeterMatrix(CoxeterType(['E',8]).coxeter_graph()).coxeter_type()
Coxeter type of ['E', 8]
sage: CoxeterMatrix(CoxeterType(['F',4]).coxeter_graph()).coxeter_type()
Coxeter type of ['F', 4]
sage: CoxeterMatrix(CoxeterType(['G',2]).coxeter_graph()).coxeter_type()
Coxeter type of ['G', 2]
sage: CoxeterMatrix(CoxeterType(['H',3]).coxeter_graph()).coxeter_type()
Coxeter type of ['H', 3]
sage: CoxeterMatrix(CoxeterType(['H',4]).coxeter_graph()).coxeter_type()
Coxeter type of ['H', 4]
sage: CoxeterMatrix(CoxeterType(['I',100]).coxeter_graph()).coxeter_type()
Coxeter type of ['I', 100]
```

Some affine graphs:

```
sage: CoxeterMatrix(CoxeterType(['A',1,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['A', 1, 1]
sage: CoxeterMatrix(CoxeterType(['A',10,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['A', 10, 1]
sage: CoxeterMatrix(CoxeterType(['B',10,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['B', 10, 1]
sage: CoxeterMatrix(CoxeterType(['C',10,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['C', 10, 1]
sage: CoxeterMatrix(CoxeterType(['D',10,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['D', 10, 1]
sage: CoxeterMatrix(CoxeterType(['E',6,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['E', 6, 1]
sage: CoxeterMatrix(CoxeterType(['E',7,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['E', 7, 1]
sage: CoxeterMatrix(CoxeterType(['E',8,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['E', 8, 1]
sage: CoxeterMatrix(CoxeterType(['F',4,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['F', 4, 1]
sage: CoxeterMatrix(CoxeterType(['G',2,1]).coxeter_graph()).coxeter_type()
Coxeter type of ['G', 2, 1]
```

TESTS:

Check that we detect relabellings:

```
sage: M = CoxeterMatrix([[1,2,3],[2,1,6],[3,6,1]], index_set=['a', 'b', 'c'])
sage: M.coxeter_type()
Coxeter type of ['G', 2, 1] relabelled by {0: 'a', 1: 'b', 2: 'c'}

sage: from sage.combinat.root_system.coxeter_matrix import recognize_coxeter_type_from_matrix
sage: for C in CoxeterMatrix.samples():
....:     relabelling_perm = Permutations(C.index_set()).random_element()
```

```
....:     relabelling_dict = {C.index_set()[i]: relabelling_perm[i] for i in range(C.rank())}
....:     relabeled_matrix = C.relabel(relabelling_dict)._matrix
....:     recognized_type = recognize_coxeter_type_from_matrix(relabeled_matrix, relabelling_per
....:     if C.is_finite() or C.is_affine():
....:         assert recognized_type == C.coxeter_type()
```

5.1.200 Coxeter Types

class sage.combinat.root_system.coxeter_type.**CoxeterType**

Bases: sage.structure.sage_object.SageObject

Abstract class for Coxeter types.

bilinear_form(*R=None*)

Return the bilinear form over R associated to self.

INPUT:

- R – (default: universal cyclotomic field) a ring used to compute the bilinear form

EXAMPLES:

```
sage: CoxeterType(['A', 2, 1]).bilinear_form()
[   1 -1/2 -1/2]
[-1/2    1 -1/2]
[-1/2 -1/2    1]
sage: CoxeterType(['H', 3]).bilinear_form()
[                      1                      -1/2                                0]
[                   -1/2                         1 1/2*E(5)^2 + 1/2*E(5)^3]
[                      0 1/2*E(5)^2 + 1/2*E(5)^3                                1]
sage: C = CoxeterMatrix([[1,-1,-1],[-1,1,-1],[-1,-1,1]])
sage: C.bilinear_form()
[ 1 -1 -1]
[-1  1 -1]
[-1 -1  1]
```

coxeter_graph()

Return the Coxeter graph associated to self.

EXAMPLES:

```
sage: CoxeterType(['A', 3]).coxeter_graph()
Graph on 3 vertices
sage: CoxeterType(['A', 3, 1]).coxeter_graph()
Graph on 4 vertices
```

coxeter_matrix()

Return the Coxeter matrix associated to self.

EXAMPLES:

```
sage: CoxeterType(['A', 3]).coxeter_matrix()
[1 3 2]
[3 1 3]
[2 3 1]
sage: CoxeterType(['A', 3, 1]).coxeter_matrix()
[1 3 2 3]
[3 1 3 2]
[2 3 1 3]
[3 2 3 1]
```

index_set()

Return the index set for `self`.

This is the list of the nodes of the associated Coxeter graph.

EXAMPLES:

```
sage: CoxeterType(['A', 3, 1]).index_set()
(0, 1, 2, 3)
sage: CoxeterType(['D', 4]).index_set()
(1, 2, 3, 4)
sage: CoxeterType(['A', 7, 2]).index_set()
(0, 1, 2, 3, 4)
sage: CoxeterType(['A', 7, 2]).index_set()
(0, 1, 2, 3, 4)
sage: CoxeterType(['A', 6, 2]).index_set()
(0, 1, 2, 3)
sage: CoxeterType(['D', 6, 2]).index_set()
(0, 1, 2, 3, 4, 5)
sage: CoxeterType(['E', 6, 1]).index_set()
(0, 1, 2, 3, 4, 5, 6)
sage: CoxeterType(['E', 6, 2]).index_set()
(0, 1, 2, 3, 4)
sage: CoxeterType(['A', 2, 2]).index_set()
(0, 1)
sage: CoxeterType(['G', 2, 1]).index_set()
(0, 1, 2)
sage: CoxeterType(['F', 4, 1]).index_set()
(0, 1, 2, 3, 4)
```

is_affine()

Return whether `self` is affine.

EXAMPLES:

```
sage: CoxeterType(['A', 3]).is_affine()
False
sage: CoxeterType(['A', 3, 1]).is_affine()
True
```

is_crystallographic()

Return whether `self` is crystallographic.

This returns `False` by default. Derived class should override this appropriately.

EXAMPLES:

```
sage: [ [t, t.is_crystallographic() ] for t in CartanType.samples(finite=True) ]
[[['A', 1], True], [['A', 5], True],
 [['B', 1], True], [['B', 5], True],
 [['C', 1], True], [['C', 5], True],
 [['D', 2], True], [['D', 3], True], [['D', 5], True],
 [['E', 6], True], [['E', 7], True], [['E', 8], True],
 [['F', 4], True], [['G', 2], True],
 [['I', 5], False], [['H', 3], False], [['H', 4], False]]
```

is_finite()

Return whether `self` is finite.

EXAMPLES:

```
sage: CoxeterType(['A',4]).is_finite()
True
sage: CoxeterType(['A',4, 1]).is_finite()
False
```

is_simply_laced()

Return whether `self` is simply laced.

This returns `False` by default. Derived class should override this appropriately.

EXAMPLES:

```
sage: [ [t, t.is_simply_laced() ] for t in CartanType.samples() ]
[[['A', 1], True], [['A', 5], True],
 [['B', 1], True], [['B', 5], False],
 [['C', 1], True], [['C', 5], False],
 [['D', 2], True], [['D', 3], True], [['D', 5], True],
 [['E', 6], True], [['E', 7], True], [['E', 8], True],
 [['F', 4], False], [['G', 2], False],
 [['I', 5], False], [['H', 3], False], [['H', 4], False],
 [['A', 1, 1], False], [['A', 5, 1], True],
 [['B', 1, 1], False], [['B', 5, 1], False],
 [['C', 1, 1], False], [['C', 5, 1], False],
 [['D', 3, 1], True], [['D', 5, 1], True],
 [['E', 6, 1], True], [['E', 7, 1], True], [['E', 8, 1], True],
 [['F', 4, 1], False], [['G', 2, 1], False],
 [['BC', 1, 2], False], [['BC', 5, 2], False],
 [['B', 5, 1]^*, False], [['C', 4, 1]^*, False],
 [['F', 4, 1]^*, False], [['G', 2, 1]^*, False],
 [['BC', 1, 2]^*, False], [['BC', 5, 2]^*, False]]
```

rank()

Return the rank of `self`.

This is the number of nodes of the associated Coxeter graph.

EXAMPLES:

```
sage: CoxeterType(['A', 4]).rank()
4
sage: CoxeterType(['A', 7, 2]).rank()
5
sage: CoxeterType(['I', 8]).rank()
2
```

classmethod samples(*finite=None*, *affine=None*, *crystallographic=None*)

Return a sample of the available Coxeter types.

INPUT:

- `finite` – a boolean or `None` (default: `None`)
- `affine` – a boolean or `None` (default: `None`)
- `crystallographic` – a boolean or `None` (default: `None`)

The sample contains all the exceptional finite and affine Coxeter types, as well as typical representatives of the infinite families.

EXAMPLES:

```
sage: CoxeterType.samples()
[Coxeter type of ['A', 1], Coxeter type of ['A', 5],
 Coxeter type of ['B', 1], Coxeter type of ['B', 5],
 Coxeter type of ['C', 1], Coxeter type of ['C', 5],
 Coxeter type of ['D', 4], Coxeter type of ['D', 5],
 Coxeter type of ['E', 6], Coxeter type of ['E', 7],
 Coxeter type of ['E', 8], Coxeter type of ['F', 4],
 Coxeter type of ['H', 3], Coxeter type of ['H', 4],
 Coxeter type of ['I', 10], Coxeter type of ['A', 2, 1],
 Coxeter type of ['B', 5, 1], Coxeter type of ['C', 5, 1],
 Coxeter type of ['D', 5, 1], Coxeter type of ['E', 6, 1],
 Coxeter type of ['E', 7, 1], Coxeter type of ['E', 8, 1],
 Coxeter type of ['F', 4, 1], Coxeter type of ['G', 2, 1],
 Coxeter type of ['A', 1, 1]]
```

The finite, affine and crystallographic options allow respectively for restricting to (non) finite, (non) affine, and (non) crystallographic Cartan types:

```
sage: CoxeterType.samples(finite=True)
[Coxeter type of ['A', 1], Coxeter type of ['A', 5],
 Coxeter type of ['B', 1], Coxeter type of ['B', 5],
 Coxeter type of ['C', 1], Coxeter type of ['C', 5],
 Coxeter type of ['D', 4], Coxeter type of ['D', 5],
 Coxeter type of ['E', 6], Coxeter type of ['E', 7],
 Coxeter type of ['E', 8], Coxeter type of ['F', 4],
 Coxeter type of ['H', 3], Coxeter type of ['H', 4],
 Coxeter type of ['I', 10]]

sage: CoxeterType.samples(affine=True)
[Coxeter type of ['A', 2, 1], Coxeter type of ['B', 5, 1],
 Coxeter type of ['C', 5, 1], Coxeter type of ['D', 5, 1],
 Coxeter type of ['E', 6, 1], Coxeter type of ['E', 7, 1],
 Coxeter type of ['E', 8, 1], Coxeter type of ['F', 4, 1],
 Coxeter type of ['G', 2, 1], Coxeter type of ['A', 1, 1]]

sage: CoxeterType.samples(crystallographic=True)
[Coxeter type of ['A', 1], Coxeter type of ['A', 5],
 Coxeter type of ['B', 1], Coxeter type of ['B', 5],
 Coxeter type of ['C', 1], Coxeter type of ['C', 5],
 Coxeter type of ['D', 4], Coxeter type of ['D', 5],
 Coxeter type of ['E', 6], Coxeter type of ['E', 7],
 Coxeter type of ['E', 8], Coxeter type of ['F', 4],
 Coxeter type of ['A', 2, 1], Coxeter type of ['B', 5, 1],
 Coxeter type of ['C', 5, 1], Coxeter type of ['D', 5, 1],
 Coxeter type of ['E', 6, 1], Coxeter type of ['E', 7, 1],
 Coxeter type of ['E', 8, 1], Coxeter type of ['F', 4, 1],
 Coxeter type of ['G', 2, 1], Coxeter type of ['A', 1, 1]]

sage: CoxeterType.samples(crystallographic=False)
[Coxeter type of ['H', 3],
 Coxeter type of ['H', 4],
 Coxeter type of ['I', 10]]
```

Todo

add some reducible Coxeter types (suggestions?)

TESTS:

```
sage: for ct in CoxeterType.samples(): TestSuite(ct).run()
```

class sage.combinat.root_system.coxeter_type.**CoxeterTypeFromCartanType**(*cartan_type*)

Bases: sage.combinat.root_system.coxeter_type.CoxeterType, sage.structure.unique_representation.UniqueRepresentation

A Coxeter type associated to a Cartan type.

cartan_type()

Return the Cartan type used to construct `self`.

EXAMPLES:

```
sage: C = CoxeterType(['C',3])
sage: C.cartan_type()
['C', 3]
```

coxeter_graph()

Return the Coxeter graph of `self`.

EXAMPLES:

```
sage: C = CoxeterType(['H',3])
sage: C.coxeter_graph().edges()
[(1, 2, 3), (2, 3, 5)]
```

coxeter_matrix()

Return the Coxeter matrix associated to `self`.

EXAMPLES:

```
sage: C = CoxeterType(['H',3])
sage: C.coxeter_matrix()
[1 3 2]
[3 1 5]
[2 5 1]
```

index_set()

Return the index set of `self`.

EXAMPLES:

```
sage: C = CoxeterType(['A', 4])
sage: C.index_set()
(1, 2, 3, 4)
```

is_affine()

Return if `self` is an affine type.

EXAMPLES:

```
sage: C = CoxeterType(['F', 4, 1])
sage: C.is_affine()
True
```

is_crystallographic()

Return if `self` is crystallographic.

EXAMPLES:

```
sage: C = CoxeterType(['C', 3])
sage: C.is_crystallographic()
```

```
True

sage: C = CoxeterType(['H', 3])
sage: C.is_crystallographic()
False
```

is_finite()
Return if `self` is a finite type.

EXAMPLES:

```
sage: C = CoxeterType(['E', 6])
sage: C.is_finite()
True
```

is_simply_laced()
Return if `self` is simply-laced.

EXAMPLES:

```
sage: C = CoxeterType(['A', 5])
sage: C.is_simply_laced()
True

sage: C = CoxeterType(['B', 3])
sage: C.is_simply_laced()
False
```

rank()
Return the rank of `self`.

EXAMPLES:

```
sage: C = CoxeterType(['I', 16])
sage: C.rank()
2
```

relabel(*relabelling*)
Return a relabelled copy of `self`.

EXAMPLES:

```
sage: ct = CoxeterType(['A',2])
sage: ct.relabel({1:-1, 2:-2})
Coxeter type of ['A', 2] relabelled by {1: -1, 2: -2}
```

5.1.201 Dynkin diagrams

AUTHORS:

- Travis Scrimshaw (2012-04-22): Nicolas M. Thiery moved Cartan matrix creation to here and I cached results for speed.
- Travis Scrimshaw (2013-06-11): Changed inputs of Dynkin diagrams to handle other Dynkin diagrams and graphs. Implemented remaining Cartan type methods.
- Christian Stump, Travis Scrimshaw (2013-04-11): Added Cartan matrix as possible input for Dynkin diagrams.

sage.combinat.root_system.dynkin_diagram.**DynkinDiagram**(**args*, ***kwds*)
Return the Dynkin diagram corresponding to the input.

INPUT:

The input can be one of the following:

- empty to obtain an empty Dynkin diagram
- a Cartan type
- a Cartan matrix
- a Cartan matrix and an indexing set

One can also input an indexing set by passing a tuple using the optional argument `index_set`.

The edge multiplicities are encoded as edge labels. For the corresponding Cartan matrices, this uses the convention in Hong and Kang, Kac, Fulton and Harris, and crystals. This is the **opposite** convention in Bourbaki and Wikipedia's Dynkin diagram (Wikipedia article Dynkin_diagram). That is for $i \neq j$:

```
i <--k-- j <==> a_ij = -k
           <==> -scalar(coroot[i], root[j]) = k
           <==> multiple arrows point from the longer root
                to the shorter one
```

For example, in type C_2, we have:

```
sage: C2 = DynkinDiagram(['C',2]); C2
O=<=O
1   2
C2
sage: C2.cartan_matrix()
[ 2 -2]
[-1  2]
```

However Bourbaki would have the Cartan matrix as:

$$\begin{bmatrix} 2 & -1 \\ -2 & 2 \end{bmatrix}.$$

EXAMPLES:

```
sage: DynkinDiagram(['A', 4])
O---O---O---O
1   2   3   4
A4

sage: DynkinDiagram(['A',1],['A',1])
O
1
O
2
A1xA1

sage: R = RootSystem("A2xB2xF4")
sage: DynkinDiagram(R)
O---O
1   2
O=>=O
3   4
O---O=>=O---O
5   6   7   8
A2xB2xF4

sage: R = RootSystem("A2xB2xF4")
```

```
sage: CM = R.cartan_matrix(); CM
[ 2 -1| 0  0| 0  0  0  0]
[-1  2| 0  0| 0  0  0  0]
[-----+-----+-----------]
[ 0  0| 2 -1| 0  0  0  0]
[ 0  0|-2  2| 0  0  0  0]
[-----+-----+-----------]
[ 0  0| 0  0| 2 -1  0  0]
[ 0  0| 0  0|-1  2 -1  0]
[ 0  0| 0  0| 0 -2  2 -1]
[ 0  0| 0  0| 0  0 -1  2]
sage: DD = DynkinDiagram(CM); DD
O---O
1   2
O=>=O
3   4
O---O=>=O---O
5   6   7   8
A2xB2xF4
sage: DD.cartan_matrix()
[ 2 -1  0  0  0  0  0  0]
[-1  2  0  0  0  0  0  0]
[ 0  0  2 -1  0  0  0  0]
[ 0  0 -2  2  0  0  0  0]
[ 0  0  0  0  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -2  2 -1]
[ 0  0  0  0  0  0 -1  2]
```

We can also create Dynkin diagrams from arbitrary Cartan matrices:

```
sage: C = CartanMatrix([[2, -3], [-4, 2]])
sage: DynkinDiagram(C)
Dynkin diagram of rank 2
sage: C.index_set()
(0, 1)
sage: CI = CartanMatrix([[2, -3], [-4, 2]], [3, 5])
sage: DI = DynkinDiagram(CI)
sage: DI.index_set()
(3, 5)
sage: CII = CartanMatrix([[2, -3], [-4, 2]])
sage: DII = DynkinDiagram(CII, ('y', 'x'))
sage: DII.index_set()
('x', 'y')
```

See also:

`CartanType()` for a general discussion on Cartan types and in particular node labeling conventions.

TESTS:

Check that trac ticket #15277 is fixed by not having edges from 0's:

```
sage: CM = CartanMatrix([[2,-1,0,0],[-3,2,-2,-2],[0,-1,2,-1],[0,-1,-1,2]])
sage: CM
[ 2 -1  0  0]
[-3  2 -2 -2]
[ 0 -1  2 -1]
[ 0 -1 -1  2]
sage: CM.dynkin_diagram().edges()
```

```
[(0, 1, 3),
 (1, 0, 1),
 (1, 2, 1),
 (1, 3, 1),
 (2, 1, 2),
 (2, 3, 1),
 (3, 1, 2),
 (3, 2, 1)]
```

class sage.combinat.root_system.dynkin_diagram.**DynkinDiagram_class**(*t=None*, *index_set=None*, ***options*)

Bases: sage.graphs.digraph.DiGraph, sage.combinat.root_system.cartan_type.CartanType_abstr

A Dynkin diagram.

See also:

DynkinDiagram()

INPUT:

- t – a Cartan type, Cartan matrix, or None

EXAMPLES:

```
sage: DynkinDiagram(['A', 3])
O---O---O
1   2   3
A3
sage: C = CartanMatrix([[2, -3], [-4, 2]])
sage: DynkinDiagram(C)
Dynkin diagram of rank 2
sage: C.dynkin_diagram().cartan_matrix() == C
True
```

TESTS:

Check that the correct type is returned when copied:

```
sage: d = DynkinDiagram(['A', 3])
sage: type(copy(d))
<class 'sage.combinat.root_system.dynkin_diagram.DynkinDiagram_class'>
```

We check that trac ticket #14655 is fixed:

```
sage: cd = copy(d)
sage: cd.add_vertex(4)
sage: d.vertices() != cd.vertices()
True
```

Implementation note: if a Cartan type is given, then the nodes are initialized from the index set of this Cartan type.

add_edge(*i*, *j*, *label=1*)

EXAMPLES:

```
sage: from sage.combinat.root_system.dynkin_diagram import DynkinDiagram_class
sage: d = DynkinDiagram_class(CartanType(['A',3]))
sage: list(sorted(d.edges()))
[]
sage: d.add_edge(2, 3)
```

```
sage: list(sorted(d.edges()))
[(2, 3, 1), (3, 2, 1)]
```

static an_instance()
Returns an example of Dynkin diagram

EXAMPLES:

```
sage: from sage.combinat.root_system.dynkin_diagram import DynkinDiagram_class
sage: g = DynkinDiagram_class.an_instance()
sage: g
Dynkin diagram of rank 3
sage: g.cartan_matrix()
[ 2 -1 -1]
[-2  2 -1]
[-1 -1  2]
```

cartan_matrix()
Returns the Cartan matrix for this Dynkin diagram

EXAMPLES:

```
sage: DynkinDiagram(['C',3]).cartan_matrix()
[ 2 -1  0]
[-1  2 -2]
[ 0 -1  2]
```

cartan_type()
EXAMPLES:

```
sage: DynkinDiagram("A2","B2","F4").cartan_type()
A2xB2xF4
```

column(j)
Returns the j^{th} column $(a_{i,j})_i$ of the Cartan matrix corresponding to this Dynkin diagram, as a container (or iterator) of tuples $(i, a_{i,j})$

EXAMPLES:

```
sage: g = DynkinDiagram(["B",4])
sage: [ (i,a) for (i,a) in g.column(3) ]
[(3, 2), (2, -1), (4, -2)]
```

dual()
Returns the dual Dynkin diagram, obtained by reversing all edges.

EXAMPLES:

```
sage: D = DynkinDiagram(['C',3])
sage: D.edges()
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]
sage: D.dual()
O---O=>=O
1   2   3
B3
sage: D.dual().edges()
[(1, 2, 1), (2, 1, 1), (2, 3, 2), (3, 2, 1)]
sage: D.dual() == DynkinDiagram(['B',3])
True
```

TESTS:

```
sage: D = DynkinDiagram(['A',0]); D
A0
sage: D.edges()
[]
sage: D.dual()
A0
sage: D.dual().edges()
[]
sage: D = DynkinDiagram(['A',1])
sage: D.edges()
[]
sage: D.dual()
O
1
A1
sage: D.dual().edges()
[]
```

dynkin_diagram()

EXAMPLES:

```
sage: DynkinDiagram(['C',3]).dynkin_diagram()
O---O=<=O
1   2   3
C3
```

index_set()

EXAMPLES:

```
sage: DynkinDiagram(['C',3]).index_set()
(1, 2, 3)
sage: DynkinDiagram("A2","B2","F4").index_set()
(1, 2, 3, 4, 5, 6, 7, 8)
```

is_affine()

Check if `self` corresponds to an affine root system.

EXAMPLES:

```
sage: CartanType(['F',4]).dynkin_diagram().is_affine()
False
sage: D = DynkinDiagram(CartanMatrix([[2, -4], [-3, 2]]))
sage: D.is_affine()
False
```

is_crystallographic()

Implements `CartanType_abstract.is_crystallographic()`

A Dynkin diagram always corresponds to a crystallographic root system.

EXAMPLES:

```
sage: CartanType(['F',4]).dynkin_diagram().is_crystallographic()
True
```

TESTS:

```
sage: CartanType(['G',2]).dynkin_diagram().is_crystallographic()
True
```

is_finite()

Check if `self` corresponds to a finite root system.

EXAMPLES:

```
sage: CartanType(['F',4]).dynkin_diagram().is_finite()
True
sage: D = DynkinDiagram(CartanMatrix([[2, -4], [-3, 2]]))
sage: D.is_finite()
False
```

is_irreducible()

Check if `self` corresponds to an irreducible root system.

EXAMPLES:

```
sage: CartanType(['F',4]).dynkin_diagram().is_irreducible()
True
```

rank()

Returns the index set for this Dynkin diagram

EXAMPLES:

```
sage: DynkinDiagram(['C',3]).rank()
3
sage: DynkinDiagram("A2","B2","F4").rank()
8
```

relabel(*relabelling*, *inplace=False*, ***kwds*)

Return the relabelling Dynkin diagram of `self`.

EXAMPLES:

```
sage: D = DynkinDiagram(['C',3])
sage: D.relabel({1:0, 2:4, 3:1})
O---O=<=O
0   4   1
C3 relabelled by {1: 0, 2: 4, 3: 1}
sage: D
O---O=<=O
1   2   3
C3
```

row(i)

Returns the i^{th} row $(a_{i,j})_j$ of the Cartan matrix corresponding to this Dynkin diagram, as a container (or iterator) of tuples $(j, a_{i,j})$

EXAMPLES:

```
sage: g = DynkinDiagram(["C",4])
sage: [ (i,a) for (i,a) in g.row(3) ]
[(3, 2), (2, -1), (4, -2)]
```

subtype(*index_set*)

Return a subtype of `self` given by `index_set`.

A subtype can be considered the Dynkin diagram induced from the Dynkin diagram of `self` by `index_set`.

EXAMPLES:

```
sage: D = DynkinDiagram(['A',6,2]); D
O=<=O---O=<=O
```

```
0   1   2   3
BC3~
sage: D.subtype([1,2,3])
Dynkin diagram of rank 3
```

symmetrizer()

Return the symmetrizer of the corresponding Cartan matrix.

EXAMPLES:

```
sage: d = DynkinDiagram()
sage: d.add_edge(1,2,3)
sage: d.add_edge(2,3)
sage: d.add_edge(3,4,3)
sage: d.symmetrizer()
Finite family {1: 9, 2: 3, 3: 3, 4: 1}
```

TESTS:

We check that trac ticket #15740 is fixed:

```
sage: d = DynkinDiagram()
sage: d.add_edge(1,2,3)
sage: d.add_edge(2,3)
sage: d.add_edge(3,4,3)
sage: L = d.root_system().root_lattice()
sage: al = L.simple_roots()
sage: al[1].associated_coroot()
alphacheck[1]
sage: al[1].reflection(al[2])
alpha[1] + 3*alpha[2]
```

sage.combinat.root_system.dynkin_diagram.**precheck**(*t*, *letter=None*, *length=None*, *affine=None*, *n_ge=None*, *n=None*)

EXAMPLES:

```
sage: from sage.combinat.root_system.dynkin_diagram import precheck
sage: ct = CartanType(['A',4])
sage: precheck(ct, letter='C')
Traceback (most recent call last):
...
ValueError: t[0] must be = 'C'
sage: precheck(ct, affine=1)
Traceback (most recent call last):
...
ValueError: t[2] must be = 1
sage: precheck(ct, length=3)
Traceback (most recent call last):
...
ValueError: len(t) must be = 3
sage: precheck(ct, n=3)
Traceback (most recent call last):
...
ValueError: t[1] must be = 3
sage: precheck(ct, n_ge=5)
Traceback (most recent call last):
...
ValueError: t[1] must be >= 5
```

5.1.202 Hecke algebra representations

class sage.combinat.root_system.hecke_algebra_representation.**CherednikOperatorsEigenvectors**(*T*, *T*, *n*, *n*, *i*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.sage_object.SageObject

A class for the family of eigenvectors of the Y Cherednik operators for a module over a (Double) Affine Hecke algebra

INPUT:

- `T` – a family $(T_i)_{i\in I}$ implementing the action of the generators of an affine Hecke algebra on `self`. The intertwiner operators are built from these.
- `T_Y` – a family $(T_i^Y)_{i\in I}$ implementing the action of the generators of an affine Hecke algebra on `self`. By default, this is `T`. But this can be used to get the action of the full Double Affine Hecke Algebra. The Y operators are built from the `T_Y`.

This returns a function $\mu \mapsto E_\mu$ which uses intertwining operators to calculate recursively eigenvectors E_μ for the action of the torus of the affine Hecke algebra with eigenvalue given by f. Namely:

$$E_\mu.Y^{\lambda^\vee} = f(\lambda^\vee, \mu)E_\mu$$

Assumptions:

- `seed(mu)` initializes the recurrence by returning an appropriate eigenvector E_μ for μ trivial enough. For example, for nonsymmetric Macdonald polynomials `seed(mu)` returns the monomial X^μ for a minuscule weight μ.
- f is almost equivariant. Namely, $f(\lambda^\vee, \mu) = f(\lambda^\vee s_i, twist(\mu, i))$ whenever i is a descent of μ.
- $twist(\mu, i)$ maps μ closer to the dominant chamber whenever i is a descent of μ.

Todo

Add tests for the above assumptions, and also that the classical operators $T_1, \ldots, T_n$ from T and T_Y coincide.

Y()

Return the Cherednik operators.

EXAMPLES:

```
sage: W = WeylGroup(["B",2])
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: E.Y()
Lazy family (...)_{i in Coroot lattice of the Root system of type ['B', 2, 1]}
```

affine_lift(*mu*)

Lift the index `\mu` to a space admitting an action of the affine Weyl group.

INPUT:

- `mu` – an element μ of the indexing set

In this space, one should have `first_descent` and `apply_simple_reflection` act properly.

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: w = W.an_element(); w
123
sage: E.affine_lift(w)
121
```

affine_retract(*mu*)

Retract μ from a space admitting an action of the affine Weyl group.

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: w = W.an_element(); w
123
sage: E.affine_retract(E.affine_lift(w)) == w
True
```

cartan_type()

Return Cartan type of `self`.

EXAMPLES:

```
sage: W = WeylGroup(["B",3])
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: E.cartan_type()
['B', 3, 1]

sage: NonSymmetricMacdonaldPolynomials(["B", 2, 1]).cartan_type()
['B', 2, 1]
```

domain()

The module on which the affine Hecke algebra acts.

EXAMPLES:

```
sage: W = WeylGroup(["B",3])
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: E.domain()
Group algebra of Weyl Group of type ['B', 3] (as a matrix group acting on the ambient space)
```

eigenvalue(*mu*, *l*)

Return the eigenvalue of $Y_{\lambda^\vee}$ on E_μ computed by applying $Y_{\lambda^\vee}$ on E_μ.

INPUT:

- mu – the index μ of an eigenvector, or a tentative eigenvector
- l – the index $\lambda^\vee$ of a Cherednik operator in self.Y_index_set()

This default implementation applies explicitly Y_μ to E_λ.

EXAMPLES:

```
sage: W = WeylGroup(["B",2])
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: w0 = W.long_element()
sage: Y = E.Y()
sage: alphacheck = Y.keys().simple_roots()
sage: E.eigenvalue(w0, alphacheck[1])
q1/(-q2)
sage: E.eigenvalue(w0, alphacheck[2])
q1/(-q2)
sage: E.eigenvalue(w0, alphacheck[0])
q2^2/q1^2
```

The following checks that all E_w are eigenvectors, with eigenvalue given by Lemma 7.5 of [HST2008] (checked for A_2, A_3):

```
sage: Pcheck = Y.keys()
sage: Wcheck = Pcheck.weyl_group()
sage: P0check = Pcheck.classical()
sage: def height(root):
....:     return sum(P0check(root).coefficients())
sage: def eigenvalue(w, mu):
....:     return (-q2/q1)^height(Wcheck.from_reduced_word(w.reduced_word()).action(mu))
sage: all(E.eigenvalue(w, a) == eigenvalue(w, a) for w in E.keys() for a in Y.keys().simple_
True
```

eigenvalues(*mu*)

Return the eigenvalues of $Y_{\alpha_0}, \ldots, Y_{\alpha_n}$ on E_μ.

INPUT:

- mu – the index μ of an eigenvector or a tentative eigenvector

EXAMPLES:

```
sage: W = WeylGroup(["B",2])
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: w0 = W.long_element()
sage: E.eigenvalues(w0)
[q2^2/q1^2, q1/(-q2), q1/(-q2)]
sage: w = W.an_element()
sage: E.eigenvalues(w)
[(-q2)/q1, (-q2^2)/(-q1^2), q1^3/(-q2^3)]
```

hecke_parameters(*i*)

Return the Hecke parameters for index i.

EXAMPLES:

```
sage: W = WeylGroup(["B",3])
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: T = KW.demazure_lusztig_operators(q1, q2, affine=True)
sage: E = T.Y_eigenvectors()
sage: E.hecke_parameters(1)
(q1, q2)
sage: E.hecke_parameters(2)
(q1, q2)
sage: E.hecke_parameters(0)
(q1, q2)
```

keys()

The index set for the eigenvectors.

By default, this assumes that the eigenvectors span the full affine Hecke algebra module and that the eigenvectors have the same indexing as the basis of this module.

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: E.keys()
Weyl Group of type ['A', 3] (as a matrix group acting on the ambient space)
```

recursion(*mu*)

Return the indices used in the recursion.

INPUT:

- mu – the index μ of an eigenvector

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: w0 = W.long_element()
sage: E.recursion(w0)
[]
sage: w = W.an_element(); w
123
sage: E.recursion(w)
[1, 2, 1]
```

seed(*mu*)

Return the eigenvector for μ minuscule.

INPUT:

- mu – an element μ of the indexing set

OUTPUT: an element of `T.domain()`

This default implementation returns the monomial indexed by μ.

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1, q2)
sage: E.seed(W.long_element())
B[123121]
```

twist (*mu*, *i*)

Act by s_i on μ.

By default, this calls the method `apply_simple_reflection`.

EXAMPLES:

```
sage: W = WeylGroup(["B",3])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: T = KW.demazure_lusztig_operators(q1, q2, affine=True)
sage: E = T.Y_eigenvectors()
sage: w = W.an_element(); w
123
sage: E.twist(w,1)
1231
```

class sage.combinat.root_system.hecke_algebra_representation.**HeckeAlgebraRepresentation**(*domain*, *on_basis*, *cartan_type*, *q1*, *q2*, *q=1*, *side='r*

Bases: `sage.misc.fast_methods.WithEqualityById`, `sage.structure.sage_object.SageObject`

A representation of an (affine) Hecke algebra given by the action of the T generators

Let F_i be a family of operators implementing an action of the operators $(T_i)_{i\in I}$ of the Hecke algebra on some vector space `domain`, given by their action on the basis of `domain`. This constructs the family of operators $(F_w)_{w\in W}$ describing the action of all elements of the basis $(T_w)_{w\in W}$ of the Hecke algebra. This is achieved by linearity on the first argument, and applying recursively the F_i along a reduced word for $w = s_{i_1}\cdots s_{i_k}$:

$$F_w(x) = F_{i_k} \circ \cdots \circ F_{i_1}(x).$$

INPUT:

- `domain` – a vector space
- `f` – a function `f(l,i)` taking a basis element l of `domain` and an index i, and returning F_i
- `cartan_type` – The Cartan type of the Hecke algebra
- `q1,q2` – The eigenvalues of the generators T of the Hecke algebra
- `side` – "left" or "right" (default: "right") whether this is a left or right representation

EXAMPLES:

```
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = WeylGroup(["A",3]).algebra(QQ)
sage: H = KW.demazure_lusztig_operators(q1,q2); H
A representation of the (q1, q2)-Hecke algebra of type ['A', 3, 1]
on Group algebra of Weyl Group of type ['A', 3] (as a matrix group acting on the ambient space)
```

Among other things, it implements the T_w operators, their inverses and compositions thereof:

```
sage: H.Tw((1,2))
Generic endomorphism of Group algebra of Weyl Group of type ['A', 3] (as a matrix group acting c
```

and the Cherednik operators $Y^{\lambda^\vee}$:

```
sage: H.Y()
Lazy family (...)_{i in Coroot lattice of the Root system of type ['A', 3, 1]}
```

TESTS:

```
sage: from sage.combinat.root_system.hecke_algebra_representation import HeckeAlgebraRepresentat
sage: W = SymmetricGroup(3)
sage: domain = W.algebra(QQ)
sage: action = lambda x,i: domain.monomial(x.apply_simple_reflection(i, side="right"))
sage: r = HeckeAlgebraRepresentation(domain, action, CartanType(["A",2]), 1, -1)
sage: hash(r) # random
3
```

REFERENCES:

Ti_inverse_on_basis(x, i)

The T_i^{-1} operators, on basis elements

INPUT:

- `x` – the index of a basis element
- `i` – the index of a generator

EXAMPLES:

```
sage: W = WeylGroup("A3")
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: rho = KW.demazure_lusztig_operators(q1,q2)
sage: w = W.an_element()
sage: rho.Ti_inverse_on_basis(w, 1)
-1/q2*B[1231] + ((q1+q2)/(q1*q2))*B[123]
```

Ti_on_basis(x, i)

The T_i operators, on basis elements.

INPUT:

- `x` – the index of a basis element
- `i` – the index of a generator

EXAMPLES:

```
sage: W = WeylGroup("A3")
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: rho = KW.demazure_lusztig_operators(q1,q2)
sage: w = W.an_element()
sage: rho.Ti_on_basis(w,1)
q1*B[1231]
```

Tw(*word*, *signs=None*, *scalar=None*)

Return T_w.

INPUT:

- `word` – a word $i_1, \ldots, i_k$ for some element w of the Weyl group. See `straighten_word()` for how this word can be specified.
- `signs` – a list $\epsilon_1, \ldots, \epsilon_k$ of the same length as `word` with $\epsilon_i = \pm 1$ or `None` for $1, \ldots, 1$ (default: `None`)
- `scalar` – an element c of the base ring or `None` for 1 (default: `None`)

OUTPUT:

a module morphism implementing

$$T_w = T_{i_k} \circ \cdots \circ T_{i_1}$$

in left action notation; that is T_{i_1} is applied first, then T_{i_2}, etc.

More generally, if `scalar` or `signs` is specified, the morphism implements

$$cT_{i_k}^{\epsilon_k} \circ \cdots \circ T_{i_1}^{\epsilon_k}$$

EXAMPLES:

```
sage: W = WeylGroup("A3")
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: x = KW.an_element(); x
2*B[12321] + 3*B[1231] + B[123] + B[]

sage: T = KW.demazure_lusztig_operators(q1,q2)
sage: T12 = T.Tw( (1,2) )
sage: T12(KW.one())
q1^2*B[12]
```

This is $T_2 \circ T_1$:

```
sage: T[2](T[1](KW.one()))
q1^2*B[12]
sage: T[1](T[2](KW.one()))
q1^2*B[21]
sage: T12(x) == T[2](T[1](x))
True
```

Now with signs and scalar coefficient we construct $3T_2 \circ T_1^{-1}$:

```
sage: phi = T.Tw((1,2), (-1,1), 3)
sage: phi(KW.one())
((-3*q1)/q2)*B[12] + ((3*q1+3*q2)/q2)*B[2]
sage: phi(T[1](x)) == 3*T[2](x)
True
```

For debugging purposes, one can recover the input data:

```
sage: phi.word
(1, 2)
sage: phi.signs
(-1, 1)
sage: phi.scalar
3
```

Tw_inverse(*word*)

Return T_w^{-1}.

This is essentially a shorthand for `Tw()` with all minus signs.

Todo

Add an example where $T_i \neq T_i^{-1}$

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: W.element_class._repr_ = lambda x: "".join(str(i) for i in x.reduced_word())
sage: KW = W.algebra(QQ)
sage: rho = KW.demazure_lusztig_operators(1, -1)
sage: x = KW.monomial(W.an_element()); x
B[123]
sage: word = [1,2]
sage: rho.Tw(word)(x)
B[12312]
sage: rho.Tw_inverse(word)(x)
B[12321]

sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: rho = KW.demazure_lusztig_operators(q1, q2)
sage: x = KW.monomial(W.an_element()); x
B[123]
sage: rho.Tw_inverse(word)(x)
1/q2^2*B[12321] + ((-q1-q2)/(q1*q2^2))*B[1231] + ((-q1-q2)/(q1*q2^2))*B[1232] + ((q1^2+2*q1*
sage: rho.Tw(word)(_)
B[123]
```

Y(*base_ring=Integer Ring*)

Return the Cherednik operators Y for this representation of an affine Hecke algebra.

INPUT:

- `self` – a representation of an affine Hecke algebra
- `base_ring` – the base ring of the coroot lattice

This is a family of operators indexed by the coroot lattice for this Cartan type. In practice this is currently indexed instead by the affine coroot lattice, even if this indexing is not one to one, in order to allow for

$Y[\alpha_0^\vee]$.

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: rho = KW.demazure_lusztig_operators(q2, q1)
sage: Y = rho.Y(); Y
Lazy family (...(i))_{i in Coroot lattice of the Root system of type ['A', 3, 1]}
```

Y_eigenvectors()

Return the family of eigenvectors for the Cherednik operators Y of this representation of an affine Hecke algebra.

INPUT:

- `self` – a representation of an affine Hecke algebra
- `base_ring` – the base ring of the coroot lattice

EXAMPLES:

```
sage: W = WeylGroup(["B",2])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: rho = KW.demazure_lusztig_operators(q1, q2, affine=True)
sage: E = rho.Y_eigenvectors()
sage: E.keys()
Weyl Group of type ['B', 2] (as a matrix group acting on the ambient space)
sage: w0 = W.long_element()
```

To set the recurrence up properly, one often needs to customize the `CherednikOperatorsEigenvectors.affine_lift()` and `CherednikOperatorsEigenvectors.affine_retract()` methods. This would usually be done by subclassing `CherednikOperatorsEigenvectors`; here we just override the methods directly.

In this particular case, we multiply by w_0 to take into account that w_0 is the seed for the recursion:

```
sage: E.affine_lift = w0._mul_
sage: E.affine_retract = w0._mul_

sage: E[w0]
B[2121]
sage: E.eigenvalues(E[w0])
[q2^2/q1^2, q1/(-q2), q1/(-q2)]
```

This step is taken care of automatically if one instead calls the specialization `sage.coxeter_groups.CoxeterGroups.Algebras.demazure_lusztig_eigenvectors()`.

Now we can compute all eigenvectors:

```
sage: [E[w] for w in W]
[B[2121] - B[121] - B[212] + B[12] + B[21] - B[1] - B[2] + B[],
-B[2121] + B[212],
(q2/(q1-q2))*B[2121] + (q2/(-q1+q2))*B[121] + (q2/(-q1+q2))*B[212] - B[12] + ((-q2)/(-q1+q2)
((-q2^2)/(-q1^2+q1*q2-q2^2))*B[2121] - B[121] + (q2^2/(-q1^2+q1*q2-q2^2))*B[212] + B[21],
((q1^2+q2^2)/(-q1^2+q1*q2-q2^2))*B[2121] + ((-q1^2-q2^2)/(-q1^2+q1*q2-q2^2))*B[121] + ((-q2^
```

```
B[2121],
(q2/(-q1+q2))*B[2121] + ((-q2)/(-q1+q2))*B[121] - B[212] + B[12],
-B[2121] + B[121]]
```

Y_lambdacheck(*lambdacheck*)

Return the Cherednik operators $Y^{\lambda^\vee}$ for this representation of an affine Hecke algebra.

INPUT:

- `lambdacheck` – an element of the coroot lattice for this cartan type

EXAMPLES:

```
sage: W = WeylGroup(["B",2])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
```

We take q_2 and q_1 as eigenvalues to match with the notations of [HST2008]

```
sage: rho = KW.demazure_lusztig_operators(q2, q1)
sage: L = rho.Y().keys()
sage: alpha = L.simple_roots()
sage: Y0 = rho.Y_lambdacheck(alpha[0])
sage: Y1 = rho.Y_lambdacheck(alpha[1])
sage: Y2 = rho.Y_lambdacheck(alpha[2])

sage: x = KW.monomial(W.an_element()); x
B[12]
sage: Y1(x)
((-q1^2-2*q1*q2-q2^2)/(-q2^2))*B[2121] + ((q1^3+q1^2*q2+q1*q2^2+q2^3)/(-q1*q2^2))*B[121] + (
sage: Y2(x)
((-q1^4-q1^3*q2-q1*q2^3-q2^4)/(-q1^3*q2))*B[2121] + ((q1^3+q1^2*q2+q1*q2^2+q2^3)/(-q1^2*q2))
sage: Y1(Y2(x))
((q1*q2+q2^2)/q1^2)*B[212] + ((-q2)/q1)*B[12]
sage: Y2(Y1(x))
((q1*q2+q2^2)/q1^2)*B[212] + ((-q2)/q1)*B[12]
```

The Y operators commute:

```
sage: Y0(Y1(x)) - Y1(Y0(x))
0
sage: Y2(Y1(x)) - Y1(Y2(x))
0
```

In the classical root lattice, $\alpha_0 + \alpha_1 + \alpha_2 = 0$:

```
sage: Y0(Y1(Y2(x)))
B[12]
```

Lemma 7.2 of [HST2008]:

```
sage: w0 = KW.monomial(W.long_element())
sage: rho.Tw(0)(w0)
q2*B[1]
sage: rho.Tw_inverse(1)(w0)
1/q2*B[212]
sage: rho.Tw_inverse(2)(w0)
1/q2*B[121]
```

Lemma 7.5 of [HST2008]:

```
sage: Y0(w0)
q1^2/q2^2*B[2121]
sage: Y1(w0)
(q2/(-q1))*B[2121]
sage: Y2(w0)
(q2/(-q1))*B[2121]
```

Todo

Add more tests

Add tests in type BC affine where the null coroot $\delta^\vee$ can have non trivial coefficient in term of α_0

See also:

- [HST2008] for the formula in terms of q_1, q_2

cartan_type()

Return the Cartan type of `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.hecke_algebra_representation import HeckeAlgebraRepresentation
sage: KW = SymmetricGroup(3).algebra(QQ)
sage: action = lambda x,i: KW.monomial(x.apply_simple_reflection(i, side="right"))
sage: H = HeckeAlgebraRepresentation(KW, action, CartanType(["A",2]), 1, -1)
sage: H.cartan_type()
['A', 2]

sage: H = WeylGroup(["A",3]).algebra(QQ).demazure_lusztig_operators(-1,1)
sage: H.cartan_type()
['A', 3, 1]
```

domain()

Return the domain of `self`.

EXAMPLES:

```
sage: H = WeylGroup(["A",3]).algebra(QQ).demazure_lusztig_operators(-1,1)
sage: H.domain()
Group algebra of Weyl Group of type ['A', 3] (as a matrix group acting on the ambient space)
```

on_basis(*x*, *word*, *signs=None*, *scalar=None*)

Action of product of T_i and T_i^{-1} on `x`.

INPUT:

- `x` – the index of a basis element
- `word` – word of indices of generators
- `signs` – (default: None) sequence of signs of same length as `word`; determines which operators are supposed to be taken as inverses.
- `scalar` – (default: None) scalar to multiply the answer by

EXAMPLES:

```
sage: from sage.combinat.root_system.hecke_algebra_representation import HeckeAlgebraRepresentation
sage: W = SymmetricGroup(3)
```

```
sage: domain = W.algebra(QQ)
sage: action = lambda x,i: domain.monomial(x.apply_simple_reflection(i, side="right"))
sage: rho = HeckeAlgebraRepresentation(domain, action, CartanType(["A",2]), 1, -1)

sage: rho.on_basis(W.one(), (1,2,1))
(1,3)

sage: word = (1,2)
sage: u = W.from_reduced_word(word)
sage: for w in W:  assert rho.on_basis(w, word) == domain.monomial(w*u)
```

The next example tests the signs:

```
sage: W = WeylGroup("A3")
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: rho = KW.demazure_lusztig_operators(q1,q2)
sage: w = W.an_element(); w
123
sage: rho.on_basis(w, (1,),  signs=(-1,))
-1/q2*B[1231] + ((q1+q2)/(q1*q2))*B[123]
sage: rho.on_basis(w, (1,),  signs=( 1,))
q1*B[1231]
sage: rho.on_basis(w, (1,1), signs=(1,-1))
B[123]
sage: rho.on_basis(w, (1,1), signs=(-1,1))
B[123]
```

parameters(*i*)

Return q_1, q_2 such that $(T_i - q_1)(T_i - q_2) = 0$.

EXAMPLES:

```
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = WeylGroup(["A",3]).algebra(QQ)
sage: H = KW.demazure_lusztig_operators(q1,q2)
sage: H.parameters(1)
(q1, q2)

sage: H = KW.demazure_lusztig_operators(1,-1)
sage: H.parameters(1)
(1, -1)
```

Todo

At this point, this method is constant. It's meant as a starting point for implementing parameters depending on the node i of the Dynkin diagram.

---

straighten_word(*word*)

Return a tuple of indices of generators after some straightening.

INPUT:

- `word` – a list/tuple of indices of generators, the index of a generator, or an object with a reduced word method

OUTPUT: a tuple of indices of generators

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: H = W.algebra(QQ).demazure_lusztig_operators(-1,1)
sage: H.straighten_word(1)
(1,)
sage: H.straighten_word((2,1))
(2, 1)
sage: H.straighten_word([2,1])
(2, 1)
sage: H.straighten_word(W.an_element())
(1, 2, 3)
```

5.1.203 Integrable Representations of Affine Lie Algebras

class sage.combinat.root_system.integrable_representations.**IntegrableRepresentation**(*Lam*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.category_object.CategoryObject

An irreducible integrable highest weight representation of an affine Lie algebra.

INPUT:

- Lam – a dominant weight in an extended weight lattice of affine type

REFERENCES:

If Λ is a dominant integral weight for an affine root system, there exists a unique integrable representation $V = V_\Lambda$ of highest weight Λ. If μ is another weight, let $m(\mu)$ denote the multiplicity of the weight μ in this representation. The set $\mathrm{supp}(V)$ of μ such that $m(\mu) > 0$ is contained in the paraboloid

$$(\Lambda + \rho|\Lambda + \rho) - (\mu + \rho|\mu + \rho) \geq 0$$

where $(\,|\,)$ is the invariant inner product on the weight lattice and ρ is the Weyl vector. Moreover if $m(\mu) > 0$ then $\mu \in \mathrm{supp}(V)$ differs from Λ by an element of the root lattice ([Kac], Propositions 11.3 and 11.4).

Let δ be the nullroot, which is the lowest positive imaginary root. Then by [Kac], Proposition 11.3 or Corollary 11.9, for fixed μ the function $m(\mu - k\delta)$ is a monotone increasing function of k. It is useful to take μ to be such that this function is nonzero if and only if $k \geq 0$. Therefore we make the following definition. If μ is such that $m(\mu) \neq 0$ but $m(\mu + \delta) = 0$ then μ is called *maximal*.

Since δ is fixed under the action of the affine Weyl group, and since the weight multiplicities are Weyl group invariant, the function $k \mapsto m(\mu - k\delta)$ is unchanged if μ is replaced by an equivalent weight. Therefore in tabulating these functions, we may assume that μ is dominant. There are only a finite number of dominant maximal weights.

Since every nonzero weight multiplicity appears in the string $\mu - k\delta$ for one of the finite number of dominant maximal weights μ, it is important to be able to compute these. We may do this as follows.

EXAMPLES:

```
sage: Lambda = RootSystem(['A',3,1]).weight_lattice(extended=true).fundamental_weights()
sage: IntegrableRepresentation(Lambda[1]+Lambda[2]+Lambda[3]).print_strings()
2*Lambda[0] + Lambda[2]: 4 31 161 665 2380 7658 22721 63120 166085 417295 1007601 2349655
Lambda[0] + 2*Lambda[1]: 2 18 99 430 1593 5274 16005 45324 121200 308829 754884 1779570
Lambda[0] + 2*Lambda[3]: 2 18 99 430 1593 5274 16005 45324 121200 308829 754884 1779570
Lambda[1] + Lambda[2] + Lambda[3]: 1 10 60 274 1056 3601 11199 32354 88009 227555 563390 1343178
3*Lambda[2] - delta: 3 21 107 450 1638 5367 16194 45687 121876 310056 757056 1783324
sage: Lambda = RootSystem(['D',4,1]).weight_lattice(extended=true).fundamental_weights()
```

```
sage: IntegrableRepresentation(Lambda[0]+Lambda[1]).print_strings()                    # lon
Lambda[0] + Lambda[1]: 1 10 62 293 1165 4097 13120 38997 109036 289575 735870 1799620
Lambda[3] + Lambda[4] - delta: 3 25 136 590 2205 7391 22780 65613 178660 463842 1155717 2777795
```

In this example, we construct the extended weight lattice of Cartan type $A_3^{(1)}$, then define `Lambda` to be the fundamental weights $(\Lambda_i)_{i\in I}$. We find there are 5 maximal dominant weights in irreducible representation of highest weight $\Lambda_1 + \Lambda_2 + \Lambda_3$, and we determine their strings.

It was shown in [KacPeterson] that each string is the set of Fourier coefficients of a modular form.

Every weight μ such that the weight multiplicity $m(\mu)$ is nonzero has the form

$$\Lambda - n_0\alpha_0 - n_1\alpha_1 - \cdots,$$

where the n_i are nonnegative integers. This is represented internally as a tuple $(n_0, n_1, n_2, \ldots)$. If you want an individual multiplicity you use the method `m()` and supply it with this tuple:

```
sage: Lambda = RootSystem(['C',2,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(2*Lambda[0]); V
Integrable representation of ['C', 2, 1] with highest weight 2*Lambda[0]
sage: V.m((3,5,3))
18
```

The `IntegrableRepresentation` class has methods `to_weight()` and `from_weight()` to convert between this internal representation and the weight lattice:

```
sage: delta = V.weight_lattice().null_root()
sage: V.to_weight((4,3,2))
-3*Lambda[0] + 6*Lambda[1] - Lambda[2] - 4*delta
sage: V.from_weight(-3*Lambda[0] + 6*Lambda[1] - Lambda[2] - 4*delta)
(4, 3, 2)
```

To get more values, use the depth parameter:

```
sage: L0 = RootSystem(["A",1,1]).weight_lattice(extended=true).fundamental_weight(0); L0
Lambda[0]
sage: IntegrableRepresentation(4*L0).print_strings(depth=20)
4*Lambda[0]: 1 1 3 6 13 23 44 75 131 215 354 561 889 1368 2097 3153 4712 6936 10151 14677
2*Lambda[0] + 2*Lambda[1] - delta: 1 2 5 10 20 36 66 112 190 310 501 788 1230 1880 2850 4256 630
4*Lambda[1] - 2*delta: 1 2 6 11 23 41 75 126 215 347 561 878 1368 2082 3153 4690 6936 10121 1467
```

An example in type $C_2^{(1)}$:

```
sage: Lambda = RootSystem(['C',2,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(2*Lambda[0])
sage: V.print_strings()     # long time
2*Lambda[0]: 1 2 9 26 77 194 477 1084 2387 5010 10227 20198
Lambda[0] + Lambda[2] - delta: 1 5 18 55 149 372 872 1941 4141 8523 17005 33019
2*Lambda[1] - delta: 1 4 15 44 122 304 721 1612 3469 7176 14414 28124
2*Lambda[2] - 2*delta: 2 7 26 72 194 467 1084 2367 5010 10191 20198 38907
```

Examples for twisted affine types:

```
sage: Lambda = RootSystem(["A",2,2]).weight_lattice(extended=True).fundamental_weights()
sage: IntegrableRepresentation(Lambda[0]).strings()
{Lambda[0]: [1, 1, 2, 3, 5, 7, 11, 15, 22, 30, 42, 56]}
sage: Lambda = RootSystem(['G',2,1]).dual.weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(Lambda[0]+Lambda[1]+Lambda[2])
```

```
sage: V.print_strings() # long time
6*Lambdacheck[0]: 4 28 100 320 944 2460 6064 14300 31968 69020 144676 293916
4*Lambdacheck[0] + Lambdacheck[2]: 4 22 84 276 800 2124 5288 12470 28116 61056 128304 261972
3*Lambdacheck[0] + Lambdacheck[1]: 2 16 58 192 588 1568 3952 9520 21644 47456 100906 207536
Lambdacheck[0] + Lambdacheck[1] + Lambdacheck[2]: 1 6 26 94 294 832 2184 5388 12634 28390 61488
2*Lambdacheck[1] - deltacheck: 2 8 32 120 354 980 2576 6244 14498 32480 69776 145528
2*Lambdacheck[0] + 2*Lambdacheck[2]: 2 12 48 164 492 1344 3428 8256 18960 41844 89208 184512
3*Lambdacheck[2] - deltacheck: 4 16 60 208 592 1584 4032 9552 21728 47776 101068 207888
sage: Lambda = RootSystem(['A',6,2]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(Lambda[0]+2*Lambda[1])
sage: V.print_strings() # long time
5*Lambda[0]: 3 42 378 2508 13707 64650 272211 1045470 3721815 12425064 39254163 118191378
3*Lambda[0] + Lambda[2]: 1 23 234 1690 9689 47313 204247 800029 2893198 9786257 31262198 9503535
Lambda[0] + 2*Lambda[1]: 1 14 154 1160 6920 34756 153523 612354 2248318 7702198 24875351 7634163
Lambda[0] + Lambda[1] + Lambda[3] - 2*delta: 6 87 751 4779 25060 113971 464842 1736620 6034717 1
Lambda[0] + 2*Lambda[2] - 2*delta: 3 54 499 3349 18166 84836 353092 1341250 4725259 15625727 489
Lambda[0] + 2*Lambda[3] - 4*delta: 15 195 1539 9186 45804 200073 789201 2866560 9723582 31120281
```

branch (*i=None*, *weyl_character_ring=None*, *sequence=None*, *depth=5*)

Return the branching rule on `self`.

Removing any node from the extended Dynkin diagram of the affine Lie algebra results in the Dynkin diagram of a classical Lie algebra, which is therefore a Lie subalgebra. For example removing the 0 node from the Dynkin diagram of type `[X, r, 1]` produces the classical Dynkin diagram of `[X, r]`.

Thus for each i in the index set, we may restrict `self` to the corresponding classical subalgebra. Of course `self` is an infinite dimensional representation, but each weight μ is assigned a grading by the number of times the simple root α_i appears in $\Lambda - \mu$. Thus the branched representation is graded and we get sequence of finite-dimensional representations which this method is able to compute.

OPTIONAL:

- `i` – (default: 0) an element of the index set
- `weyl_character_ring` – a WeylCharacterRing
- `sequence` – a dictionary
- `depth` – (default: 5) an upper bound for k determining how many terms to give

In the default case where $i = 0$, you do not need to specify anything else, though you may want to increase the depth if you need more terms.

EXAMPLES:

```
sage: Lambda = RootSystem(['A',2,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(2*Lambda[0])
sage: b = V.branch(); b
[A2(0,0),
 A2(1,1),
 A2(0,0) + 2*A2(1,1) + A2(2,2),
 2*A2(0,0) + 2*A2(0,3) + 4*A2(1,1) + 2*A2(3,0) + 2*A2(2,2),
 4*A2(0,0) + 3*A2(0,3) + 10*A2(1,1) + 3*A2(3,0) + A2(1,4) + 6*A2(2,2) + A2(4,1),
 6*A2(0,0) + 9*A2(0,3) + 20*A2(1,1) + 9*A2(3,0) + 3*A2(1,4) + 12*A2(2,2) + 3*A2(4,1) + A2(3,
```

If the parameter `weyl_character_ring` is omitted, the ring may be recovered as the parent of one of the branched coefficients:

```
sage: A2 = b[0].parent(); A2
The Weyl Character Ring of Type A2 with Integer Ring coefficients
```

If i is not zero then you should specify the `WeylCharacterRing` that you are branching to. This is determined by the Dynkin diagram:

```
sage: Lambda = RootSystem(['B',3,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(Lambda[0])
sage: V.cartan_type().dynkin_diagram()
    O 0
    |
    |
O---O=>=O
1   2   3
B3~
```

In this example, we observe that removing the $i = 2$ node from the Dynkin diagram produces a reducible diagram of type `A1xA1xA1`. Thus we have a branching to $\mathfrak{sl}(2) \times \mathfrak{sl}(2) \times \mathfrak{sl}(2)$:

```
sage: A1xA1xA1 = WeylCharacterRing("A1xA1xA1",style="coroots")
sage: V.branch(i=2,weyl_character_ring=A1xA1xA1)
[A1xA1xA1(1,0,0),
 A1xA1xA1(0,1,2),
 A1xA1xA1(1,0,0) + A1xA1xA1(1,2,0) + A1xA1xA1(1,0,2),
 A1xA1xA1(2,1,2) + A1xA1xA1(0,1,0) + 2*A1xA1xA1(0,1,2),
 3*A1xA1xA1(1,0,0) + 2*A1xA1xA1(1,2,0) + A1xA1xA1(1,2,2) + 2*A1xA1xA1(1,0,2) + A1xA1xA1(1,0,
 A1xA1xA1(2,1,0) + 3*A1xA1xA1(2,1,2) + 2*A1xA1xA1(0,1,0) + 5*A1xA1xA1(0,1,2) + A1xA1xA1(0,1,
```

If the nodes of the two Dynkin diagrams are not in the same order, you must specify an additional parameter, `sequence` which gives a dictionary to the affine Dynkin diagram to the classical one.

EXAMPLES:

```
sage: Lambda = RootSystem(['F',4,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(Lambda[0])
sage: V.cartan_type().dynkin_diagram()
O---O---O=>=O---O
0   1   2   3   4
F4~
sage: A1xC3=WeylCharacterRing("A1xC3",style="coroots")
sage: A1xC3.dynkin_diagram()
O
1
O---O=<=O
2   3   4
A1xC3
```

Observe that removing the $i = 1$ node from the `F4~` Dynkin diagram gives the `A1xC3` diagram, but the roots are in a different order. The nodes $0, 2, 3, 4$ of `F4~` correspond to `1, 4, 3, 2` of `A1xC3` and so we encode this in a dictionary:

```
sage: V.branch(i=1,weyl_character_ring=A1xC3,sequence={0:1,2:4,3:3,4:2}) # long time
[A1xC3(1,0,0,0),
 A1xC3(0,0,0,1),
 A1xC3(1,0,0,0) + A1xC3(1,2,0,0),
 A1xC3(2,0,0,1) + A1xC3(0,0,0,1) + A1xC3(0,1,1,0),
 2*A1xC3(1,0,0,0) + A1xC3(1,0,1,0) + 2*A1xC3(1,2,0,0) + A1xC3(1,0,2,0) + A1xC3(3,0,0,0),
 2*A1xC3(2,0,0,1) + A1xC3(2,1,1,0) + A1xC3(0,1,0,0) + 3*A1xC3(0,0,0,1) + 2*A1xC3(0,1,1,0) +
```

The branch method gives a way of computing the graded dimension of the integrable representation:

```
sage: Lambda = RootSystem("A1~").weight_lattice(extended=true).fundamental_weights()
sage: V=IntegrableRepresentation(Lambda[0])
sage: r = [x.degree() for x in V.branch(depth=15)]; r
```

```
[1, 3, 4, 7, 13, 19, 29, 43, 62, 90, 126, 174, 239, 325, 435, 580]
sage: oeis(r)                                                    # optional -- internet
0: A029552: Expansion of phi(x) / f(-x) in powers of x where phi(), f() are Ramanujan theta
```

cartan_type()

Return the Cartan type of `self`.

EXAMPLES:

```
sage: Lambda = RootSystem(['F',4,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(Lambda[0])
sage: V.cartan_type()
['F', 4, 1]
```

coxeter_number()

Return the Coxeter number of the Cartan type of `self`.

The Coxeter number is defined in [Kac] Chapter 6, and commonly denoted h.

EXAMPLES:

```
sage: Lambda = RootSystem(['F',4,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(Lambda[0])
sage: V.coxeter_number()
12
```

dominant_maximal_weights()

Return the dominant maximal weights of `self`.

A weight μ is *maximal* if it has nonzero multiplicity but $\mu + \delta$‘ has multiplicity zero. There are a finite number of dominant maximal weights. Indeed, [Kac] Proposition 12.6 shows that the dominant maximal weights are in bijection with the classical weights in $k \cdot F$ where F is the fundamental alcove and k is the level. The construction used in this method is based on that Proposition.

EXAMPLES:

```
sage: Lambda = RootSystem(['C',3,1]).weight_lattice(extended=true).fundamental_weights()
sage: IntegrableRepresentation(2*Lambda[0]).dominant_maximal_weights()
(2*Lambda[0],
 Lambda[0] + Lambda[2] - delta,
 2*Lambda[1] - delta,
 Lambda[1] + Lambda[3] - 2*delta,
 2*Lambda[2] - 2*delta,
 2*Lambda[3] - 3*delta)
```

dual_coxeter_number()

Return the dual Coxeter number of the Cartan type of `self`.

The dual Coxeter number is defined in [Kac] Chapter 6, and commonly denoted $h^\vee$.

EXAMPLES:

```
sage: Lambda = RootSystem(['F',4,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(Lambda[0])
sage: V.dual_coxeter_number()
9
```

from_weight(*mu*)

Return the tuple $(n_0, n_1, ...)$‘ such that `mu` equals $\Lambda - \sum_{i \in I} n_i \alpha_i$ in `self`, where Λ is the highest weight of `self`.

EXAMPLES:

```
sage: Lambda = RootSystem(['A',2,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(2*Lambda[2])
sage: V.to_weight((1,0,0))
-2*Lambda[0] + Lambda[1] + 3*Lambda[2] - delta
sage: delta = V.weight_lattice().null_root()
sage: V.from_weight(-2*Lambda[0] + Lambda[1] + 3*Lambda[2] - delta)
(1, 0, 0)
```

highest_weight()

Returns the highest weight of `self`.

EXAMPLES:

```
sage: Lambda = RootSystem(['D',4,1]).weight_lattice(extended=true).fundamental_weights()
sage: IntegrableRepresentation(Lambda[0]+2*Lambda[2]).highest_weight()
Lambda[0] + 2*Lambda[2]
```

level()

Return the level of `self`.

The level of a highest weight representation V_Λ is defined as $(\Lambda|\delta)$ See [Kac] section 12.4.

EXAMPLES:

```
sage: Lambda = RootSystem(['G',2,1]).weight_lattice(extended=true).fundamental_weights()
sage: [IntegrableRepresentation(Lambda[i]).level() for i in [0,1,2]]
[1, 1, 2]
```

m(n)

Return the multiplicity of the weight μ in `self`, where $\mu = \Lambda - \sum_i n_i \alpha_i$.

INPUT:

- `n` – a tuple representing a weight μ.

EXAMPLES:

```
sage: Lambda = RootSystem(['E',6,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(Lambda[0])
sage: u = V.highest_weight() - V.weight_lattice().null_root()
sage: V.from_weight(u)
(1, 1, 2, 2, 3, 2, 1)
sage: V.m(V.from_weight(u))
6
```

modular_characteristic(*mu=None*)

Return the modular characteristic of `self`.

The modular characteristic is a rational number introduced by Kac and Peterson [KacPeterson], required to interpret the string functions as Fourier coefficients of modular forms. See [Kac] Section 12.7. Let k be the level, and let $h^\vee$ be the dual Coxeter number. Then

$$m_\Lambda = \frac{|\Lambda+\rho|^2}{2(k+h^\vee)} - \frac{|\rho|^2}{2h^\vee}$$

If μ is a weight, then

$$m_{\Lambda,\mu} = m_\Lambda - \frac{|\mu|^2}{2k}.$$

OPTIONAL:

- mu – a weight; or alternatively:
- n – a tuple representing a weight μ.

If no optional parameter is specified, this returns m_Λ. If mu is specified, it returns $m_{\Lambda,\mu}$. You may use the tuple n to specify μ. If you do this, μ is $\Lambda - \sum_i n_i \alpha_i$.

EXAMPLES:

```
sage: Lambda = RootSystem(['A',1,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(3*Lambda[0]+2*Lambda[1])
sage: [V.modular_characteristic(x) for x in V.dominant_maximal_weights()]
[11/56, -1/280, 111/280]
```

print_strings(*depth=12*)

Print the strings of self.

See also:

strings()

EXAMPLES:

```
sage: Lambda = RootSystem(['A',1,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(2*Lambda[0])
sage: V.print_strings(depth=25)
2*Lambda[0]: 1 1 3 5 10 16 28 43 70 105 161 236 350 501 722 1016 1431 1981 2741 3740 5096 68
2*Lambda[1] - delta: 1 2 4 7 13 21 35 55 86 130 196 287 420 602 858 1206 1687 2331 3206 4368
```

root_lattice()

Return the root lattice associated to self.

EXAMPLES:

```
sage: V=IntegrableRepresentation(RootSystem(['F',4,1]).weight_lattice(extended=true).fundame
sage: V.root_lattice()
Root lattice of the Root system of type ['F', 4, 1]
```

s(*n*, *i*)

Return the action of the i-th simple reflection on the internal representation of weights by tuples n in self.

EXAMPLES:

```
sage: V = IntegrableRepresentation(RootSystem(['A',2,1]).weight_lattice(extended=true).funda
sage: [V.s((0,0,0),i) for i in V._index_set]
[(1, 0, 0), (0, 0, 0), (0, 0, 0)]
```

string(*max_weight*, *depth=12*)

Return the list of multiplicities $m(\Lambda - k\delta)$ in self, where Λ is max_weight and k runs from 0 to depth.

INPUT:

- max_weight – a dominant maximal weight
- depth – (default: 12) the maximum value of k

EXAMPLES:

```
sage: Lambda = RootSystem(['A',2,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(2*Lambda[0])
sage: V.string(2*Lambda[0])
[1, 2, 8, 20, 52, 116, 256, 522, 1045, 1996, 3736, 6780]
```

```
sage: V.string(Lambda[1] + Lambda[2])
[0, 1, 4, 12, 32, 77, 172, 365, 740, 1445, 2736, 5041]
```

strings(*depth=12*)

Return the set of dominant maximal weights of `self`, together with the string coefficients for each.

OPTIONAL:

- `depth` – (default: 12) a parameter indicating how far to push computations

EXAMPLES:

```
sage: Lambda = RootSystem(['A',1,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(2*Lambda[0])
sage: S = V.strings(depth=25)
sage: for k in S:
....:     print "{}: {}".format(k, ' '.join(str(x) for x in S[k]))
2*Lambda[0]: 1 1 3 5 10 16 28 43 70 105 161 236 350 501 722 1016 1431 1981 2741 3740 5096 68
2*Lambda[1] - delta: 1 2 4 7 13 21 35 55 86 130 196 287 420 602 858 1206 1687 2331 3206 4368
```

to_dominant(*n*)

Return the dominant weight in `self` equivalent to `n` under the affine Weyl group.

EXAMPLES:

```
sage: Lambda = RootSystem(['A',2,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(3*Lambda[0])
sage: n = V.to_dominant((13,11,7)); n
(4, 3, 3)
sage: V.to_weight(n)
Lambda[0] + Lambda[1] + Lambda[2] - 4*delta
```

to_weight(*n*)

Return the weight associated to the tuple `n` in `self`.

If `n` is the tuple $(n_1, n_2, \ldots)$, then the associated weight is $\Lambda - \sum_i n_i \alpha_i$, where Λ is the weight of the representation.

INPUT:

- `n` – a tuple representing a weight

EXAMPLES:

```
sage: Lambda = RootSystem(['A',2,1]).weight_lattice(extended=true).fundamental_weights()
sage: V = IntegrableRepresentation(2*Lambda[2])
sage: V.to_weight((1,0,0))
-2*Lambda[0] + Lambda[1] + 3*Lambda[2] - delta
```

weight_lattice()

Return the weight lattice associated to `self`.

EXAMPLES:

```
sage: V=IntegrableRepresentation(RootSystem(['E',6,1]).weight_lattice(extended=true).fundame
sage: V.weight_lattice()
Extended weight lattice of the Root system of type ['E', 6, 1]
```

5.1.204 Nonsymmetric Macdonald polynomials

AUTHORS:

- Anne Schilling and Nicolas M. Thiery (2013): initial version

ACKNOWLEDGEMENTS:

The initial version of this code (together with root_lattice_realization_algebras.Algebras and hecke_algebra_representation.HeckeAlgebraRepresentation) was written by Anne Schilling and Nicolas M. Thiery during the ICERM Semester Program on "Automorphic Forms, Combinatorial Representation Theory and Multiple Dirichlet Series" (January 28, 2013 - May 3, 2013) with the help of Dan Bump, Ben Brubaker, Bogdan Ion, Dan Orr, Arun Ram, Siddhartha Sahi, and Mark Shimozono. Special thanks go to Bogdan Ion and Mark Shimozono for their patient explanations and hand computations to check the code.

class sage.combinat.root_system.non_symmetric_macdonald_polynomials.**NonSymmetricMacdonaldPoly**

Bases: sage.combinat.root_system.hecke_algebra_representation.CherednikOperatorsEigenvect

Nonsymmetric Macdonald polynomials

INPUT:

- KL – an affine Cartan type or the group algebra of a realization of the affine weight lattice
- q, q1, q2 – parameters in the base ring of the group algebra (default: q, q1, q2)
- normalized – a boolean (default: True) whether to normalize the result to have leading coefficient 1

This implementation covers all reduced affine root systems. The polynomials are constructed recursively by the application of intertwining operators.

Todo

- Non-reduced case (Koornwinder polynomials).
- Non-equal parameters for the affine Hecke algebra.
- Choice of convention (dominant/anti-dominant, ...).
- More uniform implementation of the $T_0^\vee$ operator.
- Optimizations, in particular in the calculation of the eigenvalues for the recursion.

EXAMPLES:

We construct the family of nonsymmetric Macdonald polynomials in three variables in type A:

```
sage: E = NonSymmetricMacdonaldPolynomials(["A",2,1])
```

They are constructed as elements of the group algebra of the classical weight lattice L_0 (or one of its realizations, such as the ambient space, which is used here) and indexed by elements of L_0:

```
sage: L0 = E.keys(); L0
Ambient space of the Root system of type ['A', 2]
```

Here is the nonsymmetric Macdonald polynomial with leading term $[2,0,1]$:

```
sage: E[L0([2,0,1])]
((-q*q1-q*q2)/(-q*q1-q2))*B[(1, 1, 1)] + ((-q1-q2)/(-q*q1-q2))*B[(2, 1, 0)] + B[(2, 0, 1)]
```

It can be seen as a polynomial (or in general a Laurent polynomial) by interpreting each term as an exponent vector. The parameter q is the exponential of the null (co)root, whereas q_1 and q_2 are the two eigenvalues of each generator T_i of the affine Hecke algebra (see the background section for details).

By setting $q_1 = t, q_2 = -1$ and using the root_lattice_realization_algebras.Algebras.ElementMethods.e method, we recover the nonsymmetric Macdonald polynomial as computed by [HHL06]'s combinatorial formula:

```
sage: K = QQ['q,t'].fraction_field()
sage: q,t = K.gens()
sage: E = NonSymmetricMacdonaldPolynomials(["A",2,1], q=q, q1=t, q2=-1)
sage: vars = K['x0,x1,x2'].gens()
sage: E[L0([2,0,1])].expand(vars)
((-t + 1)/(-q*t + 1))*x0^2*x1 + x0^2*x2 + ((-q*t + q)/(-q*t + 1))*x0*x1*x2

sage: from sage.combinat.sf.ns_macdonald import E
sage: E([2,0,1])
((-t + 1)/(-q*t + 1))*x0^2*x1 + x0^2*x2 + ((-q*t + q)/(-q*t + 1))*x0*x1*x2
```

Here is a type $G_2^{(1)}$ nonsymmetric Macdonald polynomial:

```
sage: E = NonSymmetricMacdonaldPolynomials(["G",2,1])
sage: L0 = E.keys()
sage: omega = L0.fundamental_weights()
sage: E[ omega[2]-omega[1] ]
((-q*q1^3*q2-q*q1^2*q2^2)/(q*q1^4-q2^4))*B[(0, 0, 0)] + B[(1, -1, 0)] + ((-q1*q2^3-q2^4)/(q*q1^4
```

Many more examples are given after the background section.

See also:

- sage.combinat.sf.ns_macdonald.E()
- SymmetricFunctions.macdonald()

Background

The polynomial module

The nonsymmetric Macdonald polynomials are a distinguished basis of the "polynomial" module of the affine Hecke algebra. Given:

```
- a ground ring `K`, which contains the input parameters `q, q_1, q_2`
- an affine root system, specified by a Cartan type `C`
- a realization `L` of the weight lattice of type `C`
```

the polynomial module is the group algebra $K[L_0]$ of the classical weight lattice L_0 with coefficients in K. It is isomorphic to the Laurent polynomial ring over K generated by the formal exponentials of any basis of L_0.

In our running example L is the ambient space of type $C_2^{(1)}$:

```
sage: K = QQ['q,q1,q2'].fraction_field()
sage: q, q1, q2 = K.gens()
sage: C = CartanType(["C",2,1])
sage: L = RootSystem(C).ambient_space(); L
Ambient space of the Root system of type ['C', 2, 1]

sage: L.simple_roots()
Finite family {0: -2*e[0] + e['delta'], 1: e[0] - e[1], 2: 2*e[1]}
```

```
sage: omega = L.fundamental_weights(); omega
Finite family {0: e['deltacheck'], 1: e[0] + e['deltacheck'], 2: e[0] + e[1] + e['deltacheck']}

sage: L0 = L.classical(); L0
Ambient space of the Root system of type ['C', 2]
sage: KL0 = L0.algebra(K); KL0
Group algebra of the Ambient space of the Root system of type ['C', 2]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational Field
```

Affine Hecke algebra

The affine Hecke algebra is generated by elements T_i for `i` in the set of affine Dynkin nodes. They satisfy the same braid relations as the simple reflections s_i of the affine Weyl group. The T_i satisfy the quadratic relation

$$(T_i - q_1) \circ (T_i - q_2) = 0$$

where q_1 and q_2 are the input parameters. Some of the representation theory requires that q_1 and q_2 satisfy additional relations; typically one uses the specializations $q_1 = u$ and $q_2 = -1/u$ or $q_1 = t$ and $q_2 = -1$). This can be achieved by constructing an appropriate field and passing q_1 and q_2 appropriately; see the examples. In principle, the parameter(s) could further depend on `i`; this is not yet implemented but the code has been designed in such a way that this feature is easy to add.

Demazure-Lusztig operators

The `i`-th Demazure-Lusztig operator is an operator on $K[L]$ which interpolates between the reflection s_i and the Demazure operator π_i (see `root_lattice_realization.RootLatticeRealization.Algebras.ParentMethods.demazure_lusztig`

```
sage: KL = L.algebra(K); KL
Group algebra of the Ambient space of the Root system of type ['C', 2, 1]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational Field
sage: T = KL.demazure_lusztig_operators(q1, q2)
sage: x = KL.monomial(omega[1]); x
B[e[0] + e['deltacheck']]
sage: T[2](x)
q1*B[e[0] + e['deltacheck']]
sage: T[1](x)
(q1+q2)*B[e[0] + e['deltacheck']] + q1*B[e[1] + e['deltacheck']]
sage: T[0](x)
q1*B[e[0] + e['deltacheck']]
```

The affine Hecke algebra acts on $K[L]$ by letting the generators T_i act by the Demazure-Lusztig operators. The class `sage.combinat.root_system.hecke_algebra_representation.HeckeAlgebraRepresentation` implements some simple generic features for representations of affine Hecke algebras defined by the action of their T-generators.:

```
sage: T
A representation of the (q1, q2)-Hecke algebra of type ['C', 2, 1] on Group algebra of the Ambie
sage: type(T)
<class 'sage.combinat.root_system.hecke_algebra_representation.HeckeAlgebraRepresentation'>
sage: T._test_relations()                    # long time (1.3s)
```

Here we construct the operator $q_1 T_2^{-1} \circ T_1^{-1} T_0$ from a signed reduced word:

```
sage: T.Tw([0,1,2],[1,-1,-1], q1^2)
Generic endomorphism of Group algebra of the Ambient space of the Root system of type ['C', 2, 1
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational Field
```

(note the reversal of the word). Inverses are computed using the quadratic relation.

Cherednik operators

The affine Hecke algebra contains elements Y_λ indexed by the coroot lattice. Their action on $K[L]$ is implemented in Sage:

```
sage: Y = T.Y(); Y
Lazy family (...)_{i in Coroot lattice of the Root system of type ['C', 2, 1]}
sage: alphacheck = Y.keys().simple_roots()
sage: Y1 = Y[alphacheck[1]]
sage: Y1(x)
((q1^3+q1^2*q2)/(-q2^3))*B[-e[0] - 2*e[1] - e['delta'] + e['deltacheck']]
+ ((q1^3+2*q1^2*q2+q1*q2^2)/(-q2^3))*B[-e[0] - e['delta'] + e['deltacheck']]
+ ((q1^3+q1^2*q2)/(-q2^3))*B[e[0] - 2*e[1] - 2*e['delta'] + e['deltacheck']]
+ ((q1^3+q1^2*q2)/(-q2^3))*B[e[0] - 2*e[1] - e['delta'] + e['deltacheck']]
+ ((q1^3+2*q1^2*q2+q1*q2^2)/(-q2^3))*B[e[0] - 2*e['delta'] + e['deltacheck']]
+ ((q1^3+q1^2*q2)/(-q2^3))*B[e[0] - e['delta'] + e['deltacheck']] + ((q1^2+2*q1*q2+q2^2)/(-q1*q2
+ ((q1^3+q1^2*q2)/(-q2^3))*B[2*e[0] - e[1] - 2*e['delta'] + e['deltacheck']]
+ ((-q1^2-q1*q2)/(-q2^2))*B[2*e[0] - e[1] - e['delta'] + e['deltacheck']] + (q1^3/(-q2^3))*B[3*e
+ ((q1^3+q1^2*q2)/(-q2^3))*B[3*e[0] - 3*e['delta'] + e['deltacheck']]
+ ((q1^3+2*q1^2*q2+q1*q2^2)/(-q2^3))*B[-e[1] - e['delta'] + e['deltacheck']]
+ ((-q1^2-2*q1*q2-q2^2)/(-q2^2))*B[-e[1] + e['deltacheck']] + ((q1+q2)/(-q2))*B[e[1] + e['deltac
```

The Cherednik operators span a Laurent polynomial ring inside the affine Hecke algebra; namely $\lambda \mapsto Y_\lambda$ is a group isomorphism from the classical root lattice (viewed additively) to the affine Hecke algebra (viewed multiplicatively). In practice, Y_λ is constructed by computing combinatorially its signed reduced word (and an overall scalar factor) using the periodic orientation of the alcove model in the coweight lattice (see `hecke_algebra_representation.HeckeAlgebraRepresentation.Y_lambdacheck()`):

```
sage: Lcheck = L.root_system.coweight_lattice()
sage: w = Lcheck.reduced_word_of_translation(Lcheck(alphacheck[1])); w
[0, 2, 1, 0, 2, 1]
sage: Lcheck.signs_of_alcovewalk(w)
[1, -1, 1, -1, 1, 1]
```

Level zero representation of the affine Hecke algebra

The action of the affine Hecke algebra on $K[L]$ induces an action on $K[L_0]$: the action of T_i on X^λ for λ a classical weight in L_0 is obtained by embedding the weight at level zero in the affine weight lattice (see `weight_lattice_realizations.WeightLatticeRealizations.ParentMethods.embed_at_level()`) applying the Demazure-Lusztig operator there, and projecting from $K[L] \to K[L_0]$ mapping the exponential of δ to q (see `root_lattice_realization_algebras.Algebras.ParentMethods.q_project()`). This is implemented in `root_lattice_realization_algebras.Algebras.ParentMethods.demazure_lusztig`

```
sage: T = KL.demazure_lusztig_operators_on_classical(q, q1,q2)
sage: omega = L0.fundamental_weights()
sage: x = KL0.monomial(omega[1])
sage: T[0](x)
(-q*q2)*B[(-1, 0)]
```

For classical nodes these are the usual Demazure-Lusztig operators:

```
sage: T[1](x)
(q1+q2)*B[(1, 0)] + q1*B[(0, 1)]
```

Nonsymmetric Macdonald polynomials

We can now finally define the nonsymmetric Macdonald polynomials. Because the Cherednik operators commute (and there is no radical), they can be simultaneously diagonalized; namely, $K[L_0]$ admits a K-basis of joint eigenvectors for the Y_λ. For $\mu \in L_0$, the nonsymmetric Macdonald polynomial E_μ is the unique eigenvector of the family of Cherednik operators Y_λ having μ as leading term:

```
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, q1, q2); E
The family of the Macdonald polynomials of type ['C', 2, 1] with parameters q, q1, q2
```

Or for short:

```
sage: E = NonSymmetricMacdonaldPolynomials(C)
```

Recursive construction of the nonsymmetric Macdonald polynomials

The generators T_i of the affine Hecke algebra almost skew commute with the Cherednik operators. More precisely, one can deform them into the so-called intertwining operators:

$$\tau_i = T_i - (q_1 + q_2)\frac{Y_i^{a-1}}{1 - Y_i^a}\,.$$

(where $a = 1$ except for $i = 0$ in type BC where $a = a_0 = 2$) which satisfy the following skew commutation relations:

$$\tau_i Y_\lambda = \tau_i Y_{s_i\lambda}\,.$$

If $s_i\mu \neq \mu$, applying τ_i on an eigenvector E_μ produces a new eigenvector (essentially $E_{s_i\mu}$) with a distinct eigenvalue. It follows that the eigenvectors indexed by an affine Weyl orbit of weights, may be recursively computed from a single weight in the orbit.

In the case at hand, there is a little complication: namely, the simple reflections s_i acting at level 0 do not act transitively on classical weights; in fact the orbits for the classical Weyl group and for the affine Weyl group are the same. Thus, one can construct the nonsymmetric Macdonald polynomials for all weights from those for the classical dominant weights, but one is lacking a creation operator to construct the nonsymmetric Macdonald polynomials for dominant weights.

Twisted Demazure-Lusztig operators

To compensate for this, one needs to consider another affinization of the action of the classical Demazure-Lusztig operators $T_1, \ldots, T_n$, which gives rise to the double affine Hecke algebra. Following Cherednik, one adds another operator $T_0^\vee$ implemented in: root_lattice_realization_algebras.Algebras.ParentMethods.T0_check_on_basis().
See also: root_lattice_realization_algebras.Algebras.ParentMethods.twisted_demazure_luszti

Depending on the type (untwisted or not), this is a representation of the affine Hecke algebra for another affinization of the classical Cartan type. The corresponding action of the affine Weyl group – which is used to compute the recursion on μ – occurs in the corresponding weight lattice realization:

```
sage: E.L()
Ambient space of the Root system of type ['C', 2, 1]
sage: E.L_prime()
Coambient space of the Root system of type ['B', 2, 1]
sage: E.L_prime().classical()
Ambient space of the Root system of type ['C', 2]
```

See `L_prime()` and `cartan_type.CartanType_affine.other_affinization()`.

REFERENCES:

More examples

We show how to create the nonsymmetric Macdonald polynomials in two different ways and check that they are the same:

```
sage: K = QQ['q,u'].fraction_field()
sage: q, u = K.gens()
sage: E = NonSymmetricMacdonaldPolynomials(['D',3,1], q, u, -1/u)
sage: omega = E.keys().fundamental_weights()
sage: E[omega[1]+omega[3]]
((-q*u^2+q)/(-q*u^4+1))*B[(1/2, -1/2, 1/2)] + ((-q*u^2+q)/(-q*u^4+1))*B[(1/2, 1/2, -1/2)] + B[(3

sage: KL = RootSystem(["D",3,1]).ambient_space().algebra(K)
sage: P = NonSymmetricMacdonaldPolynomials(KL, q, u, -1/u)
sage: E[omega[1]+omega[3]] == P[omega[1]+omega[3]]
True
sage: E[E.keys()((0,1,-1))]
((-q*u^2+q)/(-q*u^2+1))*B[(0, 0, 0)] + ((-u^2+1)/(-q*u^2+1))*B[(1, 1, 0)]
+ ((-u^2+1)/(-q*u^2+1))*B[(1, 0, -1)] + B[(0, 1, -1)]
```

In type A, there is also a combinatorial implementation of the nonsymmetric Macdonald polynomials in terms of augmented diagram fillings as in [HHL06]. See `sage.combinat.sf.ns_macdonald.E()`. First we check that these polynomials are indeed eigenvectors of the Cherednik operators:

```
sage: K = QQ['q,t'].fraction_field()
sage: q,t = K.gens()
sage: q1 = t; q2 = -1
sage: KL = RootSystem(["A",2,1]).ambient_space().algebra(K)
sage: KL0 = KL.classical()
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, q1, q2)
sage: omega = E.keys().fundamental_weights()
sage: w = omega[1]
sage: import sage.combinat.sf.ns_macdonald as NS
sage: p = NS.E([1,0,0]); p
x0
sage: pp = KL0.from_polynomial(p)
sage: E.eigenvalues(KL0.from_polynomial(p))
[t, (-1)/(-q*t^2), t]

sage: def eig(l): return E.eigenvalues(KL0.from_polynomial(NS.E(l)))

sage: eig([1,0,0])
[t, (-1)/(-q*t^2), t]
sage: eig([2,0,0])
[q*t, (-1)/(-q^2*t^2), t]
sage: eig([3,0,0])
[q^2*t, (-1)/(-q^3*t^2), t]
```

```
sage: eig([2,0,4])
[(-1)/(-q^3*t), 1/(q^2*t), q^4*t^2]
```

Next we check explicitly that they agree with the current implementation:

```
sage: K = QQ['q','t'].fraction_field()
sage: q,t = K.gens()
sage: KL = RootSystem(["A",1,1]).ambient_lattice().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t, -1)
sage: L0 = E.keys()
sage: KL0 = KL.classical()
sage: P = K['x0,x1']
sage: def EE(weight): return E[L0(weight)].expand(P.gens())
sage: import sage.combinat.sf.ns_macdonald as NS
sage: EE([0,0])
1
sage: NS.E([0,0])
1
sage: EE([1,0])
x0
sage: NS.E([1,0])
x0
sage: EE([0,1])
((-t + 1)/(-q*t + 1))*x0 + x1
sage: NS.E([0,1])
((-t + 1)/(-q*t + 1))*x0 + x1

sage: NS.E([2,0])
x0^2 + ((-q*t + q)/(-q*t + 1))*x0*x1
sage: EE([2,0])
x0^2 + ((-q*t + q)/(-q*t + 1))*x0*x1
```

The same, directly in the ambient lattice with several shifts:

```
sage: E[L0([2,0])]
((-q*t+q)/(-q*t+1))*B[(1, 1)] + B[(2, 0)]
sage: E[L0([1,-1])]
((-q*t+q)/(-q*t+1))*B[(0, 0)] + B[(1, -1)]
sage: E[L0([0,-2])]
((-q*t+q)/(-q*t+1))*B[(-1, -1)] + B[(0, -2)]
```

Systematic checks with Sage's implementation of [HHL06]:

```
sage: assert all(EE([x,y]) == NS.E([x,y]) for d in range(5) for x,y in IntegerVectors(d,2))
```

With the current implementation, we can compute nonsymmetric Macdonald polynomials for any type, for example for type $E_6^{(1)}$:

```
sage: K=QQ['q,u'].fraction_field()
sage: q, u = K.gens()
sage: KL = RootSystem(["E",6,1]).weight_space(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q,u,-1/u)
sage: L0 = E.keys()

sage: E[L0.fundamental_weight(1).weyl_action([2,4,3,2,1])]
((-u^2+1)/(-q*u^16+1))*B[-Lambda[1] + Lambda[3]] + ((-u^2+1)/(-q*u^16+1))*B[Lambda[1]]
+ B[-Lambda[2] + Lambda[5]] + ((-u^2+1)/(-q*u^16+1))*B[Lambda[2] - Lambda[4] + Lambda[5]]
+ ((-u^2+1)/(-q*u^16+1))*B[-Lambda[3] + Lambda[4]]
```

```
sage: E[L0.fundamental_weight(2).weyl_action([2,5,3,4,2])]  # long time (6s)
((-q^2*u^20+q^2*u^18+q*u^2-q)/(-q^2*u^32+2*q*u^16-1))*B[0]
+ B[Lambda[1] - Lambda[3] + Lambda[4] - Lambda[5] + Lambda[6]]
+ ((-u^2+1)/(-q*u^16+1))*B[Lambda[1] - Lambda[3] + Lambda[5]]
+ ((-q*u^20+q*u^18+u^2-1)/(-q^2*u^32+2*q*u^16-1))*B[-Lambda[2] + Lambda[4]]
+ ((-q*u^20+q*u^18+u^2-1)/(-q^2*u^32+2*q*u^16-1))*B[Lambda[2]]
+ ((u^4-2*u^2+1)/(q^2*u^32-2*q*u^16+1))*B[Lambda[3] - Lambda[4] + Lambda[5]]
+ ((-u^2+1)/(-q*u^16+1))*B[Lambda[3] - Lambda[5] + Lambda[6]]

sage: E[L0.fundamental_weight(1)+L0.fundamental_weight(6)]  # long time (13s)
((q^2*u^10-q^2*u^8-q^2*u^2+q^2)/(q^2*u^26-q*u^16-q*u^10+1))*B[0]
+ ((-q*u^2+q)/(-q*u^10+1))*B[Lambda[1] - Lambda[2] + Lambda[6]]
+ ((-q*u^2+q)/(-q*u^10+1))*B[Lambda[1] + Lambda[2] - Lambda[4] + Lambda[6]]
+ ((-q*u^2+q)/(-q*u^10+1))*B[Lambda[1] - Lambda[3] + Lambda[4] - Lambda[5] + Lambda[6]]
+ ((-q*u^2+q)/(-q*u^10+1))*B[Lambda[1] - Lambda[3] + Lambda[5]] + B[Lambda[1] + Lambda[6]]
+ ((-q*u^2+q)/(-q*u^10+1))*B[-Lambda[2] + Lambda[4]] + ((-q*u^2+q)/(-q*u^10+1))*B[Lambda[2]]
+ ((-q*u^2+q)/(-q*u^10+1))*B[Lambda[3] - Lambda[4] + Lambda[5]]
+ ((-q*u^2+q)/(-q*u^10+1))*B[Lambda[3] - Lambda[5] + Lambda[6]]
```

We test various other types:

```
sage: K=QQ['q,u'].fraction_field()
sage: q, u = K.gens()
sage: KL = RootSystem(["A",5,2]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, u, -1/u)
sage: L0 = E.keys()
sage: E[L0.fundamental_weight(2)]
((-q*u^2+q)/(-q*u^8+1))*B[(0, 0, 0)] + B[(1, 1, 0)]
sage: E[L0((0,-1,1))]                                          # long time (1.5s)
((-q^2*u^10+q^2*u^8-q*u^6+q*u^4+q*u^2+u^2-q-1)/(-q^3*u^12+q^2*u^8+q*u^4-1))*B[(0, 0, 0)]
+ ((-u^2+1)/(-q*u^4+1))*B[(1, -1, 0)]
+ ((u^6-u^4-u^2+1)/(q^3*u^12-q^2*u^8-q*u^4+1))*B[(1, 1, 0)]
+ ((u^4-2*u^2+1)/(q^3*u^12-q^2*u^8-q*u^4+1))*B[(1, 0, -1)]
+ ((q^2*u^12-q^2*u^10-u^2+1)/(q^3*u^12-q^2*u^8-q*u^4+1))*B[(1, 0, 1)] + B[(0, -1, 1)]
+ ((-u^2+1)/(-q^2*u^8+1))*B[(0, 1, -1)] + ((-u^2+1)/(-q^2*u^8+1))*B[(0, 1, 1)]

sage: K=QQ['q,u'].fraction_field()
sage: q, u = K.gens()
sage: KL = RootSystem(["E",6,2]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q,u,-1/u)
sage: L0 = E.keys()
sage: E[L0.fundamental_weight(4)]                              # long time (5s)
((-q^3*u^20+q^3*u^18+q^2*u^2-q^2)/(-q^3*u^28+q^2*u^22+q*u^6-1))*B[(0, 0, 0, 0)]
+ ((-q*u^2+q)/(-q*u^6+1))*B[(1/2, 1/2, -1/2, -1/2)] + ((-q*u^2+q)/(-q*u^6+1))*B[(1/2, 1/2, -1/2,
+ ((-q*u^2+q)/(-q*u^6+1))*B[(1/2, 1/2, 1/2, -1/2)] + ((-q*u^2+q)/(-q*u^6+1))*B[(1/2, 1/2, 1/2, 1
+ ((q*u^2-q)/(q*u^6-1))*B[(1, 0, 0, 0)] + B[(1, 1, 0, 0)] + ((-q*u^2+q)/(-q*u^6+1))*B[(0, 1, 0,
sage: E[L0((1,-1,0,0))]                                        # long time (23s)
((q^3*u^18-q^3*u^16+q*u^4-q^2*u^2-2*q*u^2+q^2+q)/(q^3*u^18-q^2*u^12-q*u^6+1))*B[(0, 0, 0, 0)]
+ ((-q^3*u^18+q^3*u^16+q*u^2-q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(1/2, -1/2, -1/2, -1/2)]
+ ((-q^3*u^18+q^3*u^16+q*u^2-q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(1/2, -1/2, -1/2, 1/2)]
+ ((q^3*u^18-q^3*u^16-q*u^2+q)/(q^3*u^18-q^2*u^12-q*u^6+1))*B[(1/2, -1/2, 1/2, -1/2)]
+ ((q^3*u^18-q^3*u^16-q*u^2+q)/(q^3*u^18-q^2*u^12-q*u^6+1))*B[(1/2, -1/2, 1/2, 1/2)]
+ ((q*u^8-q*u^6-q*u^2+q)/(q^3*u^18-q^2*u^12-q*u^6+1))*B[(1/2, 1/2, -1/2, -1/2)]
+ ((q*u^8-q*u^6-q*u^2+q)/(q^3*u^18-q^2*u^12-q*u^6+1))*B[(1/2, 1/2, -1/2, 1/2)]
+ ((-q*u^8+q*u^6+q*u^2-q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(1/2, 1/2, 1/2, -1/2)]
+ ((-q*u^8+q*u^6+q*u^2-q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(1/2, 1/2, 1/2, 1/2)]
+ ((-q^2*u^18+q^2*u^16-q*u^8+q*u^6+q*u^2+u^2-q-1)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(1, 0, 0, 0)]
+ B[(1, -1, 0, 0)] + ((-u^2+1)/(-q^2*u^12+1))*B[(1, 1, 0, 0)] + ((-u^2+1)/(-q^2*u^12+1))*B[(1, 0
```

```
+ ((u^2-1)/(q^2*u^12-1))*B[(1, 0, 1, 0)] + ((-u^2+1)/(-q^2*u^12+1))*B[(1, 0, 0, -1)]
+ ((-u^2+1)/(-q^2*u^12+1))*B[(1, 0, 0, 1)] + ((-q*u^2+q)/(-q*u^6+1))*B[(0, -1, 0, 0)]
+ ((-q*u^4+2*q*u^2-q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(0, 1, 0, 0)]
+ ((-q*u^4+2*q*u^2-q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(0, 0, -1, 0)]
+ ((-q*u^4+2*q*u^2-q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(0, 0, 1, 0)]
+ ((-q*u^4+2*q*u^2-q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(0, 0, 0, -1)]
+ ((-q*u^4+2*q*u^2-q)/(-q^3*u^18+q^2*u^12+q*u^6-1))*B[(0, 0, 0, 1)]
```

Next we test a twisted type (checked against Maple computation by Bogdan Ion for $q_1 = t^2$ and $q_2 = -1$):

```
sage: E = NonSymmetricMacdonaldPolynomials(["A",5,2])
sage: omega = E.keys()

sage: E[omega[1]]
B[(1, 0, 0)]

sage: E[-omega[1]]
B[(-1, 0, 0)] + ((-q*q1^6-q*q1^5*q2-q1*q2^5-q2^6)/(-q^3*q1^6-q^2*q1^5*q2-q*q1*q2^5-q2^6))*B[(1,
+ ((q1+q2)/(q*q1+q2))*B[(0, 1, 0)] + ((-q1-q2)/(-q*q1-q2))*B[(0, 0, -1)] + ((-q1-q2)/(-q*q1-q2))

sage: E[omega[2]]
((-q1*q2^3-q2^4)/(q*q1^4-q2^4))*B[(1, 0, 0)] + B[(0, 1, 0)]

sage: E[-omega[2]]
((q^2*q1^7+q^2*q1^6*q2-q1*q2^6-q2^7)/(q^3*q1^7-q^2*q1^5*q2^2+q*q1^2*q2^5-q2^7))*B[(1, 0, 0)] + B
+ ((q*q1^5*q2^2+q*q1^4*q2^3-q1*q2^6-q2^7)/(q^3*q1^7-q^2*q1^5*q2^2+q*q1^2*q2^5-q2^7))*B[(0, 1, 0)
+ ((-q1*q2-q2^2)/(q*q1^2-q2^2))*B[(0, 0, -1)] + ((q1*q2+q2^2)/(-q*q1^2+q2^2))*B[(0, 0, 1)]

sage: E[-omega[1]-omega[2]]
((-q^3*q1^6-q^3*q1^5*q2-2*q^2*q1^6-3*q^2*q1^5*q2+q^2*q1^4*q2^2+2*q^2*q1^3*q2^3+q*q1^5*q2+2*q*q1^
+ ((q*q1^4+q*q1^3*q2+q1*q2^3+q2^4)/(q^3*q1^4+q^2*q1^3*q2+q*q1*q2^3+q2^4))*B[(-1, 1, 0)] + ((q1+q
+ ((q*q1^4+q*q1^3*q2+q1*q2^3+q2^4)/(q^3*q1^4+q^2*q1^3*q2+q*q1*q2^3+q2^4))*B[(1, -1, 0)]
+ ((-q^2*q1^6-q^2*q1^5*q2-q*q1^5*q2+q*q1^3*q2^3+q1^5*q2+q1^4*q2^2-q1^3*q2^3-q1^2*q2^4+q1*q2^5+q2
+ ((-q*q1^4-2*q*q1^3*q2-q*q1^2*q2^2+q1^3*q2+q1^2*q2^2-q1*q2^3-q2^4)/(-q^3*q1^4-q^2*q1^3*q2-q*q1*
+ ((-q*q1^4-2*q*q1^3*q2-q*q1^2*q2^2+q1^3*q2+q1^2*q2^2-q1*q2^3-q2^4)/(-q^3*q1^4-q^2*q1^3*q2-q*q1*
+ ((-q1-q2)/(-q*q1-q2))*B[(0, -1, 1)] + ((q*q1^4+2*q*q1^3*q2+q*q1^2*q2^2-q1^3*q2-q1^2*q2^2+q1*q2
+ ((q*q1^4+2*q*q1^3*q2+q*q1^2*q2^2-q1^3*q2-q1^2*q2^2+q1*q2^3+q2^4)/(q^3*q1^4+q^2*q1^3*q2+q*q1*q2

sage: E[omega[1]-omega[2]]
((q^3*q1^7+q^3*q1^6*q2-q*q1*q2^6-q*q2^7)/(q^3*q1^7-q^2*q1^5*q2^2+q*q1^2*q2^5-q2^7))*B[(0, 0, 0)]
+ ((q*q1^5*q2^2+q*q1^4*q2^3-q1*q2^6-q2^7)/(q^3*q1^7-q^2*q1^5*q2^2+q*q1^2*q2^5-q2^7))*B[(1, 1, 0)
+ ((q1*q2+q2^2)/(-q*q1^2+q2^2))*B[(1, 0, 1)]

sage: E[omega[3]]
((-q1*q2^2-q2^3)/(-q*q1^3-q2^3))*B[(1, 0, 0)] + ((-q1*q2^2-q2^3)/(-q*q1^3-q2^3))*B[(0, 1, 0)] +

sage: E[-omega[3]]
((q*q1^4*q2+q*q1^3*q2^2-q1*q2^4-q2^5)/(-q^2*q1^5-q2^5))*B[(1, 0, 0)] + ((q*q1^4*q2+q*q1^3*q2^2-q
+ B[(0, 0, -1)] + ((-q1*q2^4-q2^5)/(-q^2*q1^5-q2^5))*B[(0, 0, 1)]
```

Comparison with the energy function of crystals

Next we test that the nonsymmetric Macdonald polynomials at $t = 0$ match with the one-dimensional configuration sums involving Kirillov-Reshetikhin crystals for various types. See [LNSSS12]:

```
sage: K = QQ['q,t'].fraction_field()
sage: q,t = K.gens()
```

```
sage: KL = RootSystem(["A",5,2]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, t, -1)
sage: omega = E.keys().fundamental_weights()
sage: E[-omega[1]].map_coefficients(lambda x:x.subs(t=0))
B[(-1, 0, 0)] + B[(1, 0, 0)] + B[(0, -1, 0)] + B[(0, 1, 0)] + B[(0, 0, -1)] + B[(0, 0, 1)]
sage: E[-omega[2]].map_coefficients(lambda x:x.subs(t=0))    # long time (3s)
(q+2)*B[(0, 0, 0)] + B[(-1, -1, 0)] + B[(-1, 1, 0)] + B[(-1, 0, -1)]
+ B[(-1, 0, 1)] + B[(1, -1, 0)] + B[(1, 1, 0)] + B[(1, 0, -1)] + B[(1, 0, 1)]
+ B[(0, -1, -1)] + B[(0, -1, 1)] + B[(0, 1, -1)] + B[(0, 1, 1)]

sage: KL = RootSystem(["C",3,1]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t,-1)
sage: omega = E.keys().fundamental_weights()
sage: E[-omega[2]].map_coefficients(lambda x:x.subs(t=0))    # long time (5s)
2*B[(0, 0, 0)] + B[(-1, -1, 0)] + B[(-1, 1, 0)] + B[(-1, 0, -1)]
+ B[(-1, 0, 1)] + B[(1, -1, 0)] + B[(1, 1, 0)] + B[(1, 0, -1)] + B[(1, 0, 1)]
+ B[(0, -1, -1)] + B[(0, -1, 1)] + B[(0, 1, -1)] + B[(0, 1, 1)]

sage: R = RootSystem(['C',3,1])
sage: KL = R.weight_lattice(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t,-1)
sage: omega = E.keys().fundamental_weights()
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: E[-2*omega[1]].map_coefficients(lambda x:x.subs(t=0)) == LS.one_dimensional_configuration_
True
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[2])
sage: E[-omega[1]-omega[2]].map_coefficients(lambda x:x.subs(t=0)) == LS.one_dimensional_configu
True

sage: R = RootSystem(['C',2,1])
sage: KL = R.weight_lattice(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t,-1)
sage: omega = E.keys().fundamental_weights()
sage: La = R.weight_space().basis()
sage: for d in range(1,3):                                  # long time (10s)
....:     for x,y in IntegerVectors(d,2):
....:         weight = x*La[1]+y*La[2]
....:         weight0 = -x*omega[1]-y*omega[2]
....:         LS = crystals.ProjectedLevelZeroLSPaths(weight)
....:         assert E[weight0].map_coefficients(lambda x:x.subs(t=0)) == LS.one_dimensional_con

sage: R = RootSystem(['B',3,1])
sage: KL = R.weight_lattice(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t,-1)
sage: omega = E.keys().fundamental_weights()
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: E[-2*omega[1]].map_coefficients(lambda x:x.subs(t=0)) == LS.one_dimensional_configuration_
True
sage: B = crystals.KirillovReshetikhin(['B',3,1],1,1)
sage: T = crystals.TensorProduct(B,B)
sage: T.one_dimensional_configuration_sum(q) == LS.one_dimensional_configuration_sum(q) # long t
True

sage: R = RootSystem(['BC',3,2])
sage: KL = R.weight_lattice(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t,-1)
```

```
sage: omega = E.keys().fundamental_weights()
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: E[-2*omega[1]].map_coefficients(lambda x:x.subs(t=0)) == LS.one_dimensional_configuration_
True

sage: R = RootSystem(CartanType(['BC',3,2]).dual())
sage: KL = R.weight_space(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t,-1)
sage: omega = E.keys().fundamental_weights()
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: g = E[-2*omega[1]].map_coefficients(lambda x:x.subs(t=0)) # long time (30s)
sage: f = LS.one_dimensional_configuration_sum(q)           # long time (1.5s)
sage: P = g.support()[0].parent()                           # long time
sage: B = P.algebra(q.parent())                             # long time
sage: sum(p[1]*B(P(p[0])) for p in f) == g                  # long time
True

sage: C = CartanType(['G',2,1])
sage: R = RootSystem(C.dual())
sage: K = QQ['q,t'].fraction_field()
sage: q,t = K.gens()
sage: KL = R.weight_lattice(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, t,-1)
sage: omega = E.keys().fundamental_weights()
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: E[-2*omega[1]].map_coefficients(lambda x:x.subs(t=0)) == LS.one_dimensional_configuration_
True
sage: LS = crystals.ProjectedLevelZeroLSPaths(La[1]+La[2])
sage: E[-omega[1]-omega[2]].map_coefficients(lambda x:x.subs(t=0)) == LS.one_dimensional_configu
True
```

The next test breaks if the energy is not scaled by the translation factor for dual type $G_2^{(1)}$:

```
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1]+La[2])
sage: E[-2*omega[1]-omega[2]].map_coefficients(lambda x:x.subs(t=0)) == LS.one_dimensional_confi
True

sage: R = RootSystem(['D',4,1])
sage: KL = R.weight_lattice(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL, q, t,-1)
sage: omega = E.keys().fundamental_weights()
sage: La = R.weight_space().basis()
sage: for d in range(1,2):                                  # long time (41s)
....:     for a,b,c,d in IntegerVectors(d,4):
....:         weight = a*La[1]+b*La[2]+c*La[3]+d*La[4]
....:         weight0 = -a*omega[1]-b*omega[2]-c*omega[3]-d*omega[4]
....:         LS = crystals.ProjectedLevelZeroLSPaths(weight)
....:         assert E[weight0].map_coefficients(lambda x:x.subs(t=0)) == LS.one_dimensional_con
```

TESTS:

Calculations checked with Bogdan Ion 2013/04/18:

```
sage: K = QQ['q,t'].fraction_field()
sage: q,t=K.gens()
sage: E = NonSymmetricMacdonaldPolynomials(["B",2,1], q=q,q1=t,q2=-1/t)
sage: L0 = E.keys()
```

```
sage: omega = L0.fundamental_weights()

sage: E[omega[1]]
((-q*t^4+q*t^2)/(-q*t^6+1))*B[(0, 0)] + B[(1, 0)]
sage: E[omega[2]]
B[(1/2, 1/2)]
sage: E[-omega[1]]
((-q^2*t^8+q^2*t^6-q*t^6+2*q*t^4-q*t^2+t^2-1)/(-q^3*t^8+q^2*t^6+q*t^2-1))*B[(0, 0)] + B[(-1, 0)]
sage: E[L0([0,1])]
((-q*t^4+q*t^2)/(-q*t^4+1))*B[(0, 0)] + ((-t^2+1)/(-q*t^4+1))*B[(1, 0)] + B[(0, 1)]
sage: E[L0([1,1])]
((q*t^2-q)/(q*t^2-1))*B[(0, 0)] + ((-q*t^2+q)/(-q*t^2+1))*B[(1, 0)] + B[(1, 1)] + ((-q*t^2+q)/(-

sage: E = NonSymmetricMacdonaldPolynomials(["A",2,1], q=q,q1=t,q2=-1/t)
sage: L0 = E.keys()
sage: factor(E[L0([-1,0,1])][L0.zero()])
(t - 1) * (t + 1) * (q*t^2 - 1)^-3 * (q*t^2 + 1)^-1 * (q^3*t^6 + 2*q^2*t^6 - 3*q^2*t^4 - 2*q*t^2
```

Checking step by step calculations in type BC with Bogdan Ion 2013/04/18:

```
sage: K = QQ['q,t'].fraction_field()
sage: q,t=K.gens()
sage: E = NonSymmetricMacdonaldPolynomials(["BC",1,2], q=q,q1=t,q2=-1/t)
sage: KL0 = E.domain()
sage: L0 = E.keys()
sage: omega = L0.fundamental_weights()
sage: e = L0.basis()
sage: E._T_Y[1] ( KL0.monomial(e[0]) )
1/t*B[(-1)]
sage: E._T_Y[0] ( KL0.monomial(L0.zero()) )
t*B[(0)]
sage: E._T_Y[0] ( KL0.monomial(-e[0]))
((-t^2+1)/(q*t))*B[(0)] + 1/(q^2*t)*B[(1)]

sage: Y = E.Y()
sage: alphacheck = Y.keys().simple_roots()
sage: Y0 = Y[alphacheck[0]]
sage: Y1 = Y[alphacheck[1]]
sage: Y0
Generic endomorphism of Group algebra of the Ambient space of the Root system of type ['C', 1]
over Fraction Field of Multivariate Polynomial Ring in q, t over Rational Field
sage: Y0.word, Y0.signs, Y0.scalar
((0, 1), (-1, -1), 1/q)
sage: Y1.word, Y1.signs, Y1.scalar
((1, 0), (1, 1), 1)

sage: T0_check = E._T[0]
```

Comparing with Bogdan Ion's hand calculations for type BC, 2013/05/13:

Todo

add his notes in latex

```
sage: K = QQ['q,q1,q2'].fraction_field()
sage: q,q1,q2=K.gens()
sage: L = RootSystem(["A",4,2]).ambient_space()
sage: L.cartan_type()
```

```
['BC', 2, 2]
sage: L.null_root()
2*e['delta']
sage: L.simple_roots()
Finite family {0: -e[0] + e['delta'], 1: e[0] - e[1], 2: 2*e[1]}
sage: KL = L.algebra(K)
sage: KL0 = KL.classical()
sage: L0 = L.classical()
sage: L0.cartan_type()
['C', 2]

sage: E = NonSymmetricMacdonaldPolynomials(KL, q=q,q1=q1,q2=q2)
sage: E.keys()
Ambient space of the Root system of type ['C', 2]
sage: E.keys().simple_roots()
Finite family {1: (1, -1), 2: (0, 2)}
sage: omega = E.keys().fundamental_weights()

sage: E[0*omega[1]]
B[(0, 0)]
sage: E[omega[1]]
((-q*q1*q2^3-q*q2^4)/(q^2*q1^4-q2^4))*B[(0, 0)] + B[(1, 0)]

sage: E[2*omega[2]]      # long time # not checked against Bogdan's notes, but a good self-consi
((-q^12*q1^6-q^12*q1^5*q2+2*q^10*q1^5*q2+5*q^10*q1^4*q2^2+3*q^10*q1^3*q2^3+2*q^8*q1^5*q2+4*q^8*q
sage: E.recursion(2*omega[2])
[0, 1, 0, 2, 1, 0, 2, 1, 0]
```

Some tests that the T s are implemented properly by hand defining the Y s in terms of them:

```
sage: T = E._T_Y
sage: Ye1     = T.Tw((1,2,1,0), scalar = (-1/(q1*q2))^2)
sage: Ye2     = T.Tw((2,1,0,1), signs = (1,1,1,-1), scalar = (-1/(q1*q2)))
sage: Yalpha0 = T.Tw((0,1,2,1), signs = (-1,-1,-1,-1), scalar = q^-1*(-q1*q2)^2)
sage: Yalpha1 = T.Tw((1,2,0,1,2,0), signs=(1,1,-1,1,-1,1), scalar = -1/(q1*q2))
sage: Yalpha2 = T.Tw((2,1,0,1,2,1,0,1), signs = (1,1,1,-1,1,1,1,-1), scalar = (1/(q1*q2))^2)

sage: Ye1(KL0.one())
q1^2/q2^2*B[(0, 0)]
sage: Ye2(KL0.one())
((-q1)/q2)*B[(0, 0)]

sage: Yalpha0(KL0.one())
q2^2/(q*q1^2)*B[(0, 0)]
sage: Yalpha1(KL0.one())
((-q1)/q2)*B[(0, 0)]
sage: Yalpha2(KL0.one())
q1^2/q2^2*B[(0, 0)]
```

Testing the Y s directly:

```
sage: Y = E.Y()
sage: Y.keys()
Coroot lattice of the Root system of type ['BC', 2, 2]
sage: alpha = Y.keys().simple_roots()
sage: L(alpha[0])
-2*e[0] + e['deltacheck']
sage: L(alpha[1])
e[0] - e[1]
```

```
sage: L(alpha[2])
e[1]

sage: Y[alpha[0]].word
(0, 1, 2, 1)
sage: Y[alpha[0]].signs
(-1, -1, -1, -1)
sage: Y[alpha[0]].scalar # mind that Sage's q is the usual q^{1/2}
q1^2*q2^2/q
sage: Y[alpha[0]](KL0.one())
q2^2/(q*q1^2)*B[(0, 0)]

sage: Y[alpha[1]].word
(1, 2, 0, 1, 2, 0)
sage: Y[alpha[1]].signs
(1, 1, -1, 1, -1, 1)
sage: Y[alpha[1]].scalar
1/(-q1*q2)

sage: Y[alpha[2]].word    # Bogdan says it should be the square of that; do we need to take tran
(2, 1, 0, 1)
sage: Y[alpha[2]].signs
(1, 1, 1, -1)
sage: Y[alpha[2]].scalar
1/(-q1*q2)
```

Checking the provided nonsymmetric Macdonald polynomial:

```
sage: E10 = KL0.monomial(L0((1,0))) + KL0( q*(1-(-q1/q2)) / (1-q^2*(-q1/q2)^4) )
sage: E10 == E[omega[1]]
True
sage: E.eigenvalues(E10)  # not checked
[q*q1^2/q2^2, q2^3/(-q^2*q1^3), q1/(-q2)]
```

Checking T0check:

```
sage: T0check_on_basis = KL.T0_check_on_basis(q1,q2, convention="dominant")
sage: T0check_on_basis.phi # note: this is in fact a0 phi
(2, 0)
sage: T0check_on_basis.v   # what to match it with?
(1,)
sage: T0check_on_basis.j   # what to match it with?
2
sage: T0check_on_basis(KL0.basis().keys().zero())
((-q1^2)/q2)*B[(1, 0)]

sage: T0check = E._T[0]
sage: T0check(KL0.one())
((-q1^2)/q2)*B[(1, 0)]
```

Systematic tests of nonsymmetric Macdonald polynomials in type $A_1^{(1)}$, in the weight lattice. Each time, we specify the eigenvalues for the action of Y_{α_0}, and Y_{α_1}:

```
sage: K = QQ['q','t'].fraction_field()
sage: q,t = K.gens()
sage: KL = RootSystem(["A",1,1]).weight_lattice(extended=True).algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t, -1)
sage: omega = E.keys().fundamental_weights()
```

```
sage: x = E[0*omega[1]]; x
B[0]
sage: E.eigenvalues(x)
[1/(q*t), t]
sage: x.is_one()
True
sage: x.parent()
Group algebra of the Weight lattice of the Root system of type ['A', 1]
over Fraction Field of Multivariate Polynomial Ring in q, t over Rational Field

sage: E[omega[1]]
B[Lambda[1]]
sage: E.eigenvalues(_)
[t, 1/(q*t)]
sage: E[2*omega[1]]
((-q*t+q)/(-q*t+1))*B[0] + B[2*Lambda[1]]
sage: E.eigenvalues(_)
[q*t, 1/(q^2*t)]
sage: E[3*omega[1]]
((-q^2*t+q^2)/(-q^2*t+1))*B[-Lambda[1]] + ((-q^2*t+q^2-q*t+q)/(-q^2*t+1))*B[Lambda[1]] + B[3*Lam
sage: E.eigenvalues(_)
[q^2*t, 1/(q^3*t)]
sage: E[4*omega[1]]
((q^5*t^2-q^5*t+q^4*t^2-2*q^4*t+q^3*t^2+q^4-2*q^3*t+q^3-q^2*t+q^2)/(q^5*t^2-q^3*t-q^2*t+1))*B[0]
sage: E.eigenvalues(_)
[q^3*t, 1/(q^4*t)]
sage: E[6*omega[1]]
((-q^12*t^3+q^12*t^2-q^11*t^3+2*q^11*t^2-2*q^10*t^3-q^11*t+4*q^10*t^2-2*q^9*t^3-2*q^10*t+5*q^9*t
sage: E.eigenvalues(_)
[q^5*t, 1/(q^6*t)]
sage: E[-omega[1]]
B[-Lambda[1]] + ((-t+1)/(-q*t+1))*B[Lambda[1]]
sage: E.eigenvalues(_)
[(-1)/(-q^2*t), q*t]
```

As expected, $e^{-\omega}$ is not an eigenvector:

```
sage: E.eigenvalues(KL.classical().monomial(-omega[1]))
Traceback (most recent call last):
...
AssertionError
```

We proceed by comparing against the examples from the appendix of [HHL06] in type $A_2^{(1)}$:

```
sage: K = QQ['q','t'].fraction_field()
sage: q,t = K.gens()
sage: KL = RootSystem(["A",2,1]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t, -1)
sage: L0 = E.keys()
sage: omega = L0.fundamental_weights()
sage: P = K['x1,x2,x3']
sage: def EE(weight): return E[L0(weight)].expand(P.gens())

sage: EE([0,0,0])
1
sage: EE([1,0,0])
x1
sage: EE([0,1,0])
((-t + 1)/(-q*t^2 + 1))*x1 + x2
```

```
sage: EE([0,0,1])
((-t + 1)/(-q*t + 1))*x1 + ((-t + 1)/(-q*t + 1))*x2 + x3
sage: EE([1,1,0])
x1*x2
sage: EE([1,0,1])
((-t + 1)/(-q*t^2 + 1))*x1*x2 + x1*x3
sage: EE([0,1,1])
((-t + 1)/(-q*t + 1))*x1*x2 + ((-t + 1)/(-q*t + 1))*x1*x3 + x2*x3
sage: EE([2,0,0])
x1^2 + ((-q*t + q)/(-q*t + 1))*x1*x2 + ((-q*t + q)/(-q*t + 1))*x1*x3

sage: EE([0,2,0])
((-t + 1)/(-q^2*t^2 + 1))*x1^2 + ((-q^2*t^3 + q^2*t^2 - q*t^2 + 2*q*t - q + t - 1)/(-q^3*t^3 + q
```

Systematic checks with Sage's implementation of [HHL06]:

```
sage: import sage.combinat.sf.ns_macdonald as NS
sage: assert all(EE([x,y,z]) == NS.E([x,y,z]) for d in range(5) for x,y,z in IntegerVectors(d,3)
```

We check that we get eigenvectors for generic q_1, q_2:

```
sage: K = QQ['q,q1,q2'].fraction_field()
sage: q,q1,q2 = K.gens()
sage: KL = RootSystem(["A",2,1]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, q1, q2)
sage: L0 = E.keys()
sage: omega = L0.fundamental_weights()
sage: E[2*omega[2]]
((q*q1+q*q2)/(q*q1+q2))*B[(1, 2, 1)] + ((q*q1+q*q2)/(q*q1+q2))*B[(2, 1, 1)] + B[(2, 2, 0)]
sage: for d in range(4):                                    # long time (9s)
...       for weight in IntegerVectors(d,3).map(list).map(L0):
...           eigenvalues = E.eigenvalues(E[L0(weight)])
```

Some type C calculations:

```
sage: K = QQ['q','t'].fraction_field()
sage: q, t = K.gens()
sage: KL = RootSystem(["C",2,1]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t, -1)
sage: L0 = E.keys()
sage: omega = L0.fundamental_weights()
sage: E[0*omega[1]]
B[(0, 0)]
sage: E.eigenvalues(_)  # checked for i=0 with previous calculation
[1/(q*t^3), t, t]
sage: E[omega[1]]
B[(1, 0)]
sage: E.eigenvalues(_)  # not checked
[t, 1/(q*t^3), t]

sage: E[-omega[1]]          # consistent with before refactoring
B[(-1, 0)] + ((-t+1)/(-q*t+1))*B[(1, 0)] + ((-t+1)/(-q*t+1))*B[(0, -1)] + ((t-1)/(q*t-1))*B[(0,
sage: E.eigenvalues(_)  # not checked
[(-1)/(-q^2*t^3), q*t, t]
sage: E[-omega[1]+omega[2]] # consistent with before refactoring
((-t+1)/(-q*t^3+1))*B[(1, 0)] + B[(0, 1)]
sage: E.eigenvalues(_)  # not checked
[t, q*t^3, (-1)/(-q*t^2)]
sage: E[omega[1]-omega[2]]  # consistent with before refactoring
```

```
((-t+1)/(-q*t^2+1))*B[(1, 0)] + B[(0, -1)] + ((-t+1)/(-q*t^2+1))*B[(0, 1)]
sage: E.eigenvalues(_)  # not checked
[1/(q^2*t^3), 1/(q*t), q*t^2]

sage: E[-omega[2]]
((-q^2*t^4+q^2*t^3-q*t^3+2*q*t^2-q*t+t-1)/(-q^3*t^4+q^2*t^3+q*t-1))*B[(0, 0)] + B[(-1, -1)] + ((
sage: E.eigenvalues(_)  # not checked                    # long time (1s)
[1/(q^3*t^3), t, q*t]
sage: E[-omega[2]].map_coefficients(lambda c: c.subs(t=0))     # checking againsts crystals
B[(0, 0)] + B[(-1, -1)] + B[(-1, 1)] + B[(1, -1)] + B[(1, 1)]

sage: E[2*omega[2]]
((-q^6*t^7+q^6*t^6-q^5*t^6+2*q^5*t^5-q^4*t^5-q^5*t^3+3*q^4*t^4-3*q^4*t^3+q^3*t^4+q^4*t^2-2*q^3*t
sage: E.eigenvalues(_)  # not checked
[q^3*t^3, t, (-1)/(-q^2*t^2)]
```

The following computations were calculated by hand:

```
sage: KL0 = KL.classical()
sage: E11 = KL0.sum_of_terms([[L0([1,1]), 1], [L0([0,0]), (-q*t^2 + q*t)/(1-q*t^3)]])
sage: E11 == E[omega[2]]
True
sage: E.eigenvalues(E11)
[q*t^3, t, (-1)/(-q*t^2)]

sage: E1m1 = KL0.sum_of_terms([[L0([1,-1]), 1], [L0([1,1]), (1-t)/(1-q*t^2)], [L0([0,0]), q*t*(1
sage: E1m1 == E[2*omega[1]-omega[2]]
True
sage: E.eigenvalues(E1m1)
[1/(q*t), 1/(q^2*t^3), q*t^2]
```

Now we present an example for a twisted affine root system. The results are eigenvectors:

```
sage: K = QQ['q','t'].fraction_field()
sage: q, t = K.gens()
sage: KL = RootSystem("C2~*").ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, t, -1)
sage: omega = E.keys().fundamental_weights()
sage: E[0*omega[1]]
B[(0, 0)]
sage: E.eigenvalues(_)
[1/(q*t^2), t, t]
sage: E[omega[1]]
((-q*t+q)/(-q*t^2+1))*B[(0, 0)] + B[(1, 0)]
sage: E.eigenvalues(_)
[q*t^2, 1/(q^2*t^3), t]

sage: E[-omega[1]]
((-q*t+q-t+1)/(-q^2*t+1))*B[(0, 0)] + B[(-1, 0)] + ((-t+1)/(-q^2*t+1))*B[(1, 0)] + ((-t+1)/(-q^2
sage: E.eigenvalues(_)
[(-1)/(-q^3*t^2), q^2*t, t]
sage: E[-omega[1]+omega[2]]
B[(-1/2, 1/2)] + ((-t+1)/(-q^2*t^3+1))*B[(1/2, -1/2)] + ((-q*t^3+q*t^2-t+1)/(-q^2*t^3+1))*B[(1/2
sage: E.eigenvalues(_)
[(-1)/(-q^2*t^2), q^2*t^3, (-1)/(-q*t)]
sage: E[omega[1]-omega[2]]
B[(1/2, -1/2)] + ((-t+1)/(-q*t^2+1))*B[(1/2, 1/2)]
sage: E.eigenvalues(_)
[t, 1/(q^2*t^3), q*t^2]
```

Type BC, comparison with calculations with Maple by Bogdan Ion:

```
sage: K = QQ['q','t'].fraction_field()
sage: q,t = K.gens()
sage: def to_SR(x): return x.expand([SR.var('x%s'%i) for i in range(1,x.parent().basis().keys().
sage: var('x1,x2,x3')
(x1, x2, x3)

sage: E = NonSymmetricMacdonaldPolynomials(["BC",2,2], q=q, q1=t^2,q2=-1)
sage: omega=E.keys().fundamental_weights()
sage: expected = (t-1)*(t+1)*(2+q^4+2*q^2-2*t^2-2*q^2*t^2-t^4*q^2-q^4*t^4+t^4-3*q^6*t^6-2*q^4*t^
sage: to_SR(E[2*omega[2]]) - expected                    # long time (3.5s)
0

sage: E = NonSymmetricMacdonaldPolynomials(["BC",3,2], q=q, q1=t^2,q2=-1)
sage: omega=E.keys().fundamental_weights()
sage: mu = -3*omega[1] + 3*omega[2] - omega[3]; mu
(-1, 2, -1)
sage: expected = (t-1)^2*(t+1)^2*(3*q^2+q^4+1+t^2*q^4+q^2*t^2-3*t^4*q^2-5*t^6*q^4+2*t^8*q^4-4*t^
sage: to_SR(E[mu]) - expected                            # long time (20s)
0

sage: E = NonSymmetricMacdonaldPolynomials(["BC",1,2], q=q, q1=t^2,q2=-1)
sage: omega=E.keys().fundamental_weights()
sage: mu = -4*omega[1]; mu
(-4)
sage: expected = (t-1)*(t+1)*(-1+q^2*t^2-q^2-3*q^10-7*q^26*t^8+5*t^2*q^6-q^16-3*q^4+4*t^10*q^30-
sage: to_SR(E[mu]) - expected
0
```

Type BC dual, comparison with hand calculations by Bogdan Ion:

```
sage: K = QQ['q,q1,q2'].fraction_field()
sage: q,q1,q2 = K.gens()
sage: ct = CartanType(["BC",2,2]).dual()
sage: E = NonSymmetricMacdonaldPolynomials(ct, q=q, q1=q1, q2=q2)
sage: KL = E.domain(); KL
Group algebra of the Ambient space of the Root system of type ['B', 2]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational Field
sage: alpha = E.keys().simple_roots(); alpha
Finite family {1: (1, -1), 2: (0, 1)}
sage: omega=E.keys().fundamental_weights(); omega
Finite family {1: (1, 0), 2: (1/2, 1/2)}
sage: epsilon = E.keys().basis(); epsilon
Finite family {0: (1, 0), 1: (0, 1)}
```

Note: Sage's q is the usual q^2:

```
sage: E.L().null_root()
e['delta']
sage: E.L().null_coroot()
2*e['deltacheck']
```

Some eigenvectors:

```
sage: E[0*omega[1]]
B[(0, 0)]
sage: E[omega[1]]
((-q^2*q1^3*q2-q^2*q1^2*q2^2)/(q^2*q1^4-q2^4))*B[(0, 0)] + B[(1, 0)]
```

```
sage: Eomega1 = KL.one() * (q^2*(-q1/q2)^2*(1-(-q1/q2))) / (1-q^2*(-q1/q2)^4) + KL.monomial(omeg
sage: E[omega[1]] == Eomega1
True
```

Checking the Y s:

```
sage: Y = E.Y()
sage: alphacheck = Y.keys().simple_roots()
sage: Y0 = Y[alphacheck[0]]
sage: Y1 = Y[alphacheck[1]]
sage: Y2 = Y[alphacheck[2]]

sage: Y0.word, Y0.signs, Y0.scalar
((0, 1, 2, 1, 0, 1, 2, 1), (-1, -1, -1, -1, -1, -1, -1, -1), q1^4*q2^4/q^2)
sage: Y1.word, Y1.signs, Y1.scalar
((1, 2, 0, 1, 2, 0), (1, 1, -1, 1, -1, 1), 1/(-q1*q2))
sage: Y2.word, Y2.signs, Y2.scalar
((2, 1, 0, 1), (1, 1, 1, -1), 1/(-q1*q2))

sage: E.eigenvalues(0*omega[1])
[q2^4/(q^2*q1^4), q1/(-q2), q1/(-q2)]
```

Checking the T and T^{-1} s:

```
sage: T = E._T_Y
sage: Tinv0 = T.Tw_inverse([0])
sage: Tinv1 = T.Tw_inverse([1])
sage: Tinv2 = T.Tw_inverse([2])

sage: for x in [0*epsilon[0], -epsilon[0], -epsilon[1], epsilon[0], epsilon[1]]:
....:     x = KL.monomial(x)
....:     assert Tinv0(T[0](x)) == x and T[0](Tinv0(x)) == x
....:     assert Tinv1(T[1](x)) == x and T[1](Tinv1(x)) == x
....:     assert Tinv2(T[2](x)) == x and T[2](Tinv2(x)) == x

sage: start = E[omega[1]]; start
((-q^2*q1^3*q2-q^2*q1^2*q2^2)/(q^2*q1^4-q2^4))*B[(0, 0)] + B[(1, 0)]
sage: Tinv1(Tinv2(Tinv1(Tinv0(Tinv1(Tinv2(Tinv1(Tinv0(start))))))) * (q1*q2)^4/q^2 == Y0(start)
True
sage: Y0(start) == q^2*q1^4/q2^4 * start
True
```

Checking the relation between the Y s:

```
sage: q^2 * Y0(Y1(Y1(Y2(Y2(start))))) == start
True
sage: for x in [0*epsilon[0], -epsilon[0], -epsilon[1], epsilon[0], epsilon[1]]:
....:     x = KL.monomial(x)
....:     assert q^2 * Y0(Y1(Y1(Y2(Y2(start))))) == start
```

KL0()

Return the group algebra where the nonsymmetric Macdonald polynomials live.

EXAMPLES:

```
sage: NonSymmetricMacdonaldPolynomials("B2~").KL0()
Group algebra of the Ambient space of the Root system of type ['B', 2]
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational Field
sage: NonSymmetricMacdonaldPolynomials("B2~*").KL0()
Group algebra of the Ambient space of the Root system of type ['C', 2]
```

```
over Fraction Field of Multivariate Polynomial Ring in q, q1, q2 over Rational Field
```

L()

Return the affinization of the classical weight space.

EXAMPLES:

```
sage: NonSymmetricMacdonaldPolynomials(["B", 2, 1]).L()
Ambient space of the Root system of type ['B', 2, 1]
```

L0()

Return the space indexing the monomials of the nonsymmetric Macdonald polynomials.

EXAMPLES:

```
sage: NonSymmetricMacdonaldPolynomials("B2~").L0()
Ambient space of the Root system of type ['B', 2]
sage: NonSymmetricMacdonaldPolynomials("B2~*").L0()
Ambient space of the Root system of type ['C', 2]
```

L_check()

Return the other affinization of the classical weight space.

Todo

should this just return L in the simply laced case?

EXAMPLES:

```
sage: NonSymmetricMacdonaldPolynomials(["B", 2, 1]).L_check()
Coambient space of the Root system of type ['C', 2, 1]
sage: NonSymmetricMacdonaldPolynomials(["B", 2, 1]).L_check().classical()
Ambient space of the Root system of type ['B', 2]
```

L_prime()

The affine space where classical weights are lifted for the recursion.

Also the parent of ρ'.

EXAMPLES:

In the twisted case, this is the affinization of the classical ambient space:

```
sage: NonSymmetricMacdonaldPolynomials("B2~*").L()
Ambient space of the Root system of type ['B', 2, 1]^*
sage: NonSymmetricMacdonaldPolynomials("B2~*").L().classical()
Ambient space of the Root system of type ['C', 2]

sage: NonSymmetricMacdonaldPolynomials("B2~*").L_prime()
Ambient space of the Root system of type ['B', 2, 1]^*
sage: NonSymmetricMacdonaldPolynomials("B2~*").L_prime().classical()
Ambient space of the Root system of type ['C', 2]
```

In the untwisted case, this is the other affinization of the classical ambient space:

```
sage: NonSymmetricMacdonaldPolynomials("B2~").L()
Ambient space of the Root system of type ['B', 2, 1]
sage: NonSymmetricMacdonaldPolynomials("B2~").L().classical()
Ambient space of the Root system of type ['B', 2]

sage: NonSymmetricMacdonaldPolynomials("B2~").L_prime()
```

```
Coambient space of the Root system of type ['C', 2, 1]
sage: NonSymmetricMacdonaldPolynomials("B2~").L_prime().classical()
Ambient space of the Root system of type ['B', 2]
```

For simply laced, the two affinizations coincide:

```
sage: NonSymmetricMacdonaldPolynomials("A2~").L()
Ambient space of the Root system of type ['A', 2, 1]
sage: NonSymmetricMacdonaldPolynomials("A2~").L().classical()
Ambient space of the Root system of type ['A', 2]

sage: NonSymmetricMacdonaldPolynomials("A2~").L_prime()
Coambient space of the Root system of type ['A', 2, 1]
sage: NonSymmetricMacdonaldPolynomials("A2~").L_prime().classical()
Ambient space of the Root system of type ['A', 2]
```

Note: do we want the coambient space of type $A_2^{(1)}$ instead?

For type BC:

```
sage: NonSymmetricMacdonaldPolynomials(["BC",3,2]).L_prime()
Ambient space of the Root system of type ['BC', 3, 2]
```

Q_to_Qcheck()

The reindexing of the index set of the Y's by the coroot lattice.

EXAMPLES:

```
sage: E = NonSymmetricMacdonaldPolynomials("C2~")
sage: alphacheck = E.Y().keys().simple_roots()
sage: E.Q_to_Qcheck(alphacheck[0])
alphacheck[0] - alphacheck[2]
sage: E.Q_to_Qcheck(alphacheck[1])
alphacheck[1]
sage: E.Q_to_Qcheck(alphacheck[2])
alphacheck[2]

sage: x = alphacheck[1] + 2*alphacheck[2]
sage: x.parent()
Root lattice of the Root system of type ['B', 2, 1]
sage: E.Q_to_Qcheck(x)
alphacheck[1] + 2*alphacheck[2]
sage: _.parent()
Coroot lattice of the Root system of type ['C', 2, 1]
```

Y()

Return the family of Y operators whose eigenvectors are the nonsymmetric Macdonald polynomials.

EXAMPLES:

```
sage: NonSymmetricMacdonaldPolynomials("C2~").Y()
Lazy family (<lambda>(i))_{i in Root lattice of the Root system of type ['B', 2, 1]}
sage: _.keys().classical()
Root lattice of the Root system of type ['B', 2]

sage: NonSymmetricMacdonaldPolynomials("C2~*").Y()
Lazy family (<...Y_lambdacheck...>(i))_{i in Coroot lattice of the Root system of type ['C',
sage: _.keys().classical()
Root lattice of the Root system of type ['C', 2]
```

```
sage: NonSymmetricMacdonaldPolynomials(["BC", 3, 2]).Y()
Lazy family (<...Y_lambdacheck...>(i))_{i in Coroot lattice of the Root system of type ['BC'
sage: _.keys().classical()
Root lattice of the Root system of type ['B', 3]
```

affine_lift(*mu*)

Return the affinization of μ in L'.

INPUT:

- mu – a classical weight μ

See also:

- hecke_algebra_representation.CherednikOperatorsEigenvectors.affine_lift()
- affine_retract()
- L_prime()

EXAMPLES:

In the untwisted case, this is the other affinization at level 1:

```
sage: E = NonSymmetricMacdonaldPolynomials("B2~")
sage: L0 = E.keys(); L0
Ambient space of the Root system of type ['B', 2]
sage: omega = L0.fundamental_weights()
sage: E.affine_lift(omega[1])
e[0] + e['deltacheck']
sage: E.affine_lift(omega[1]).parent()
Coambient space of the Root system of type ['C', 2, 1]
```

In the twisted case, this is the usual affinization at level 1:

```
sage: E = NonSymmetricMacdonaldPolynomials("B2~*")
sage: L0 = E.keys(); L0
Ambient space of the Root system of type ['C', 2]
sage: omega = L0.fundamental_weights()
sage: E.affine_lift(omega[1])
e[0] + e['deltacheck']
sage: E.affine_lift(omega[1]).parent()
Ambient space of the Root system of type ['B', 2, 1]^*
```

affine_retract(*mu*)

Retract the affine weight μ into a classical weight.

INPUT:

- mu – an affine weight μ in L'

See also:

- hecke_algebra_representation.HeckeAlgebraRepresentation.affine_retract()
- affine_lift()
- L_prime()

EXAMPLES:

```
sage: E = NonSymmetricMacdonaldPolynomials("B2~")
sage: L0 = E.keys(); L0
Ambient space of the Root system of type ['B', 2]
sage: omega = L0.fundamental_weights()
sage: E.affine_lift(omega[1])
e[0] + e['deltacheck']
sage: E.affine_retract(E.affine_lift(omega[1]))
(1, 0)
```

cartan_type()

Return Cartan type of `self`.

EXAMPLES:

```
sage: NonSymmetricMacdonaldPolynomials(["B", 2, 1]).cartan_type()
['B', 2, 1]
```

eigenvalue_experimental(*mu*, *l*)

Return the eigenvalue of $Y^{\lambda^\vee}$ acting on the macdonald polynomial E_μ.

INPUT:

- `mu` – the index μ of an eigenvector
- l – an index $\lambda^\vee$ of some Y

Note:

- This method is currently not used; most tests below even test the naive method. They are left here as a basis for a future implementation.
- This is actually equivariant, as long as s_i does not fix λ.
- This method is only really needed for $\lambda^\vee = \alpha_i^\vee$ with $i = 0, ..., n$.

See Corollary 6.11 of [Haiman06].

EXAMPLES:

```
sage: K = QQ['q,t'].fraction_field()
sage: q,t = K.gens()
sage: q1 = t
sage: q2 = -1
sage: KL = RootSystem(["A",1,1]).ambient_space().algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, q1, q2)
sage: L0 = E.keys()
sage: E.eigenvalues(L0([0,0])) # Checked by hand by Mark and Arun
[1/(q*t), t]
sage: alpha = E.Y().keys().simple_roots()
sage: E.eigenvalue_experimental(L0([0,0]), alpha[0]) # todo: not implemented
1/(q*t)
sage: E.eigenvalue_experimental(L0([0,0]), alpha[1])
t
```

Some examples of eigenvalues (not mathematically checked!!!):

```
sage: E.eigenvalues(L0([1,0]))
[t, 1/(q*t)]
sage: E.eigenvalues(L0([0,1]))
[1/(q^2*t), q*t]
sage: E.eigenvalues(L0([1,1]))
```

```
[1/(q*t), t]
sage: E.eigenvalues(L0([2,1]))
[t, 1/(q*t)]
sage: E.eigenvalues(L0([-1,1]))
[(-1)/(-q^3*t), q^2*t]
sage: E.eigenvalues(L0([-2,1]))
[(-1)/(-q^4*t), q^3*t]
sage: E.eigenvalues(L0([-2,0]))
[(-1)/(-q^3*t), q^2*t]
```

Some type B examples:

```
sage: K = QQ['q,t'].fraction_field()
sage: q,t = K.gens()
sage: q1 = t
sage: q2 = -1
sage: L = RootSystem(["B",2,1]).ambient_space()
sage: KL = L.algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, q1, q2)
sage: L0 = E.keys()
sage: alpha = L.simple_coroots()
sage: E.eigenvalue(L0((0,0)), alpha[0]) # not checked # not tested
q/t
sage: E.eigenvalue(L0((1,0)), alpha[1]) # What Mark got by hand # not tested
q
sage: E.eigenvalue(L0((1,0)), alpha[2]) # not checked # not tested
t
sage: E.eigenvalue(L0((1,0)), alpha[0]) # not checked # not tested
1

sage: L = RootSystem("B2~*").ambient_space()
sage: KL = L.algebra(K)
sage: E = NonSymmetricMacdonaldPolynomials(KL,q, q1, q2)
sage: L0 = E.keys()
sage: alpha = L.simple_coroots()
sage: E.eigenvalue(L0((0,0)), alpha[0]) # assuming Mark's calculation is correct, one should
1/(q*t^2)
```

The expected value can more or less be read off from equation (37), Corollary 6.15 of [Haiman06]

Todo

- Use proposition 6.9 of [Haiman06] to check the action of the Y s on monomials.
- Generalize to any q_1, q_2.
- Check claim by Mark: all scalar products should occur in the finite weight lattice, with alpha 0 being the appropriate projection of the affine alpha 0. Question: can this be emulated by being at level 0?

---

rho_prime()

Return the level 0 sum of the classical fundamental weights in L'.

See also:

L_prime()

EXAMPLES:

Untwisted case:

```
sage: NonSymmetricMacdonaldPolynomials("B2~").rho_prime() # CHECKME
3/2*e[0] + 1/2*e[1]
sage: NonSymmetricMacdonaldPolynomials("B2~").rho_prime().parent()
Coambient space of the Root system of type ['C', 2, 1]
```

Twisted case:

```
sage: NonSymmetricMacdonaldPolynomials("B2~*").rho_prime() # CHECKME
2*e[0] + e[1]
sage: NonSymmetricMacdonaldPolynomials("B2~*").rho_prime().parent()
Ambient space of the Root system of type ['B', 2, 1]^*
```

seed(*mu*)

Return E_μ for μ minuscule, i.e. in the fundamental alcove.

INPUT:

- mu – the index μ of an eigenvector

EXAMPLES:

```
sage: E = NonSymmetricMacdonaldPolynomials(["A",2,1])
sage: omega = E.keys().fundamental_weights()
sage: E.seed(omega[1])
B[(1, 0, 0)]
```

symmetric_macdonald_polynomial(*mu*)

Return the symmetric Macdonald polynomial indexed by μ.

INPUT:

- mu – a dominant weight μ

Warning: The result is Weyl-symmetric only for Hecke parameters of the form $q_1 = v$ and $q_2 = -1/v$. In general the value of v below, should be the square root of $-q_1/q_2$, but the use of $q_1 = t$ and $q_2 = -1$ results in nonintegral powers of t.

EXAMPLES:

```
sage: K = QQ['q,v,t'].fraction_field()
sage: q,v,t = K.gens()
sage: E = NonSymmetricMacdonaldPolynomials(['A',2,1], q, v, -1/v)
sage: om = E.L0().fundamental_weights()
sage: E.symmetric_macdonald_polynomial(om[2])
B[(1, 1, 0)] + B[(1, 0, 1)] + B[(0, 1, 1)]
sage: E.symmetric_macdonald_polynomial(2*om[1])
((q*v^2+v^2-q-1)/(q*v^2-1))*B[(1, 1, 0)] + ((q*v^2+v^2-q-1)/(q*v^2-1))*B[(1, 0, 1)] + B[(2,
sage: f = E.symmetric_macdonald_polynomial(E.L0()((2,1,0))); f
((2*q*v^4+v^4-q*v^2+v^2-q-2)/(q*v^4-1))*B[(1, 1, 1)] + B[(1, 2, 0)] + B[(1, 0, 2)] + B[(2, 1
```

We compare with the type A Macdonald polynomials coming from symmetric functions:

```
sage: P = SymmetricFunctions(K).macdonald().P()
sage: g = P[2,1].expand(3); g
x0^2*x1 + x0*x1^2 + x0^2*x2 + ((-2*q*t^2 + q*t - t^2 + q - t + 2)/(-q*t^2 + 1))*x0*x1*x2 + x
sage: fe =f.expand(g.parent().gens()); fe
x0^2*x1 + x0*x1^2 + x0^2*x2 + ((2*q*v^4 + v^4 - q*v^2 + v^2 - q - 2)/(q*v^4 - 1))*x0*x1*x2 +
sage: g.map_coefficients(lambda x: x.subs(t=v*v)) == fe
True
```

```
sage: E = NonSymmetricMacdonaldPolynomials(['C',3,1], q, v, -1/v)
sage: om = E.L0().fundamental_weights()
sage: E.symmetric_macdonald_polynomial(om[1]+om[2])
B[(-2, -1, 0)] + B[(-2, 1, 0)] + B[(-2, 0, -1)] + B[(-2, 0, 1)] + ((4*q^3*v^14+2*q^2*v^14-2*
```

An example for type G:

```
sage: E = NonSymmetricMacdonaldPolynomials(['G',2,1], q, v, -1/v)
sage: om = E.L0().fundamental_weights()
sage: E.symmetric_macdonald_polynomial(2*om[1])
((3*q^6*v^22+3*q^5*v^22-3*q^6*v^20+q^4*v^22-4*q^5*v^20+q^4*v^18-q^5*v^16+q^3*v^18-2*q^4*v^16
```

twist (*mu*, *i*)

Act by s_i on the affine weight μ.

This calls `simple_reflection`; which is semantically the same as the default implementation.

EXAMPLES:

```
sage: W = WeylGroup(["B",3])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in x.reduced_word())
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: T = KW.demazure_lusztig_operators(q1, q2, affine=True)
sage: E = T.Y_eigenvectors()
sage: w = W.an_element(); w
123
sage: E.twist(w,1)
1231
```

5.1.205 Pieri Factors

class sage.combinat.root_system.pieri_factors.**PieriFactors**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

An abstract class for sets of Pieri factors, used for constructing Stanley symmetric functions. The set of Pieri factors for a given type can be realized as an order ideal of the Bruhat order poset generated by a certain set of maximal elements.

See also:

- `WeylGroups.ParentMethods.pieri_factors()`
- `WeylGroups.ElementMethods.stanley_symmetric_function()`

EXAMPLES:

```
sage: W = WeylGroup(['A',4])
sage: PF = W.pieri_factors()
sage: PF.an_element().reduced_word()
[4, 3, 2, 1]
sage: Waff = WeylGroup(['A',4,1])
sage: PFaff = Waff.pieri_factors()
sage: Waff.from_reduced_word(PF.an_element().reduced_word()) in PFaff
True
```

```
sage: W = WeylGroup(['B',3,1])
sage: PF = W.pieri_factors()
sage: W.from_reduced_word([2,3,2]) in PF.elements()
True
sage: PF.cardinality()
47

sage: W = WeylGroup(['C',3,1])
sage: PF = W.pieri_factors()
sage: PF.generating_series()
6*z^6 + 14*z^5 + 18*z^4 + 15*z^3 + 9*z^2 + 4*z + 1
sage: [w.reduced_word() for w in PF if w.length() == 2]
[[2, 0], [0, 1], [2, 3], [1, 2], [3, 2], [3, 1], [2, 1], [3, 0], [1, 0]]
```

REFERENCES:

default_weight()

Returns the function $i \mapsto z^i$, where z is the generator of `QQ['z']`.

EXAMPLES:

```
sage: W = WeylGroup(["A", 3, 1])
sage: weight = W.pieri_factors().default_weight()
sage: weight(1)
z
sage: weight(5)
z^5
```

TESTS:

```
sage: weight(4) in QQ['z']
True
sage: weight(0) in QQ['z']
True
sage: weight(0).parent() == QQ['z']  # todo: not implemented
True
```

elements()

Returns the elements of `self`

Those are constructed as the elements below the maximal elements of `self` in Bruhat order.

OUTPUT: a `RecursivelyEnumeratedSet_generic` object

EXAMPLES:

```
sage: PF = WeylGroup(['A',3]).pieri_factors()
sage: [w.reduced_word() for w in PF.elements()]
[[3, 2, 1], [2, 1], [1], [], [3, 1], [3], [3, 2], [2]]
```

See also:

`maximal_elements()`

Todo

Possibly remove this method and instead have this class inherit from `RecursivelyEnumeratedSet_generic`.

---

generating_series(*weight=None*)

Returns a length generating series for the elements of `self`

EXAMPLES:

```
sage: PF = WeylGroup(['C',3,1]).pieri_factors()
sage: PF.generating_series()
6*z^6 + 14*z^5 + 18*z^4 + 15*z^3 + 9*z^2 + 4*z + 1

sage: PF = WeylGroup(['B',4]).pieri_factors()
sage: PF.generating_series()
z^7 + 6*z^6 + 14*z^5 + 18*z^4 + 15*z^3 + 9*z^2 + 4*z + 1
```

max_length()

Return the maximal length of a Pieri factor.

EXAMPLES:

In type A and A affine, this is n:

```
sage: WeylGroup(['A',5]).pieri_factors().max_length()
5
sage: WeylGroup(['A',5,1]).pieri_factors().max_length()
5
```

In type B and B affine, this is $2n-1$:

```
sage: WeylGroup(['B',5,1]).pieri_factors().max_length()
9
sage: WeylGroup(['B',5]).pieri_factors().max_length()
9
```

In type C affine this is $2n$:

```
sage: WeylGroup(['C',5,1]).pieri_factors().max_length()
10
```

In type D affine this is $2n-2$:

```
sage: WeylGroup(['D',5,1]).pieri_factors().max_length()
8
```

class sage.combinat.root_system.pieri_factors.**PieriFactors_affine_type**

Bases: sage.combinat.root_system.pieri_factors.PieriFactors

maximal_elements()

Return the maximal elements of `self` with respect to Bruhat order.

The current implementation is via a conjectural type-free formula. Use maximal_elements_combinatorial() for proven type-specific implementations. To compare type-free and type-specific (combinatorial) implementations, use method `_test_maximal_elements()`.

EXAMPLES:

```
sage: W = WeylGroup(['A',4,1])
sage: PF = W.pieri_factors()
sage: sorted([w.reduced_word() for w in PF.maximal_elements()], key=str)
[[0, 4, 3, 2], [1, 0, 4, 3], [2, 1, 0, 4], [3, 2, 1, 0], [4, 3, 2, 1]]

sage: W = WeylGroup(RootSystem(["C",3,1]).weight_space())
sage: PF = W.pieri_factors()
sage: sorted([w.reduced_word() for w in PF.maximal_elements()], key=str)
[[0, 1, 2, 3, 2, 1], [1, 0, 1, 2, 3, 2], [1, 2, 3, 2, 1, 0],
```

```
 [2, 1, 0, 1, 2, 3], [2, 3, 2, 1, 0, 1], [3, 2, 1, 0, 1, 2]]

sage: W = WeylGroup(RootSystem(["B",3,1]).weight_space())
sage: PF = W.pieri_factors()
sage: sorted([w.reduced_word() for w in PF.maximal_elements()], key=str)
[[0, 2, 3, 2, 0], [1, 0, 2, 3, 2], [1, 2, 3, 2, 1],
 [2, 1, 0, 2, 3], [2, 3, 2, 1, 0], [3, 2, 1, 0, 2]]

sage: W = WeylGroup(['D',4,1])
sage: PF = W.pieri_factors()
sage: sorted([w.reduced_word() for w in PF.maximal_elements()], key=str)
[[0, 2, 4, 3, 2, 0], [1, 0, 2, 4, 3, 2], [1, 2, 4, 3, 2, 1],
 [2, 1, 0, 2, 4, 3], [2, 4, 3, 2, 1, 0], [3, 2, 1, 0, 2, 3],
 [4, 2, 1, 0, 2, 4], [4, 3, 2, 1, 0, 2]]
```

class sage.combinat.root_system.pieri_factors.**PieriFactors_finite_type**

Bases: sage.combinat.root_system.pieri_factors.PieriFactors

The Pieri factors of finite type A are the restriction of the Pieri factors of affine type A to finite permutations (under the canonical embedding of finite type A into the affine Weyl group), and the Pieri factors of finite type B are the restriction of the Pieri factors of affine type C. The finite type D Pieri factors are (weakly) conjectured to be the restriction of the Pieri factors of affine type D.

maximal_elements()

The current algorithm uses the fact that the maximal Pieri factors of affine type A,B,C, or D either contain a finite Weyl group element, or contain an affine Weyl group element whose reflection by s_0 gets a finite Weyl group element, and that either of these finite group elements will serve as a maximal element for finite Pieri factors. A better algorithm is desirable.

EXAMPLES:

```
sage: PF = WeylGroup(['A',5]).pieri_factors()
sage: [v.reduced_word() for v in PF.maximal_elements()]
[[5, 4, 3, 2, 1]]

sage: WeylGroup(['B',4]).pieri_factors().maximal_elements()
[
[-1  0  0  0]
[ 0  1  0  0]
[ 0  0  1  0]
[ 0  0  0  1]
]
```

class sage.combinat.root_system.pieri_factors.**PieriFactors_type_A**(*W*)

Bases: sage.combinat.root_system.pieri_factors.PieriFactors_finite_type

The set of Pieri factors for finite type A.

This is the set of elements of the Weyl group that have a reduced word that is strictly decreasing. May also be viewed as the restriction of affine type A Pieri factors to finite Weyl group elements.

maximal_elements_combinatorial()

Returns the maximal Pieri factors, using the type A combinatorial description

EXAMPLES:

```
sage: W = WeylGroup(['A',4])
sage: PF = W.pieri_factors()
sage: PF.maximal_elements_combinatorial()[0].reduced_word()
[4, 3, 2, 1]
```

stanley_symm_poly_weight(*w*)

EXAMPLES:

```
sage: W = WeylGroup(['A',4])
sage: PF = W.pieri_factors()
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([3,1]))
0
```

class sage.combinat.root_system.pieri_factors.**PieriFactors_type_A_affine**(*W*, *min_length*, *max_length*, *min_support*, *max_support*)

Bases: sage.combinat.root_system.pieri_factors.PieriFactors_affine_type

The set of Pieri factors for type A affine, that is the set of elements of the Weyl Group which are cyclically decreasing.

Those are used for constructing (affine) Stanley symmetric functions.

The Pieri factors are in bijection with the proper subsets of the index_set. The bijection is given by the support. Namely, let f be a Pieri factor, and red a reduced word for f. No simple reflection appears twice in red, and the support S of red (that is the i such that s_i appears in red) does not depend on the reduced word).

cardinality()

EXAMPLES:

```
sage: WeylGroup(["A", 3, 1]).pieri_factors().cardinality()
15
```

generating_series(*weight=None*)

Returns a length generating series for the elements of `self`

EXAMPLES:

```
sage: W = WeylGroup(["A", 3, 1])
sage: W.pieri_factors().cardinality()
15
sage: W.pieri_factors().generating_series()
4*z^3 + 6*z^2 + 4*z + 1
```

maximal_elements_combinatorial()

Returns the maximal Pieri factors, using the affine type A combinatorial description

EXAMPLES:

```
sage: W = WeylGroup(['A',4,1])
sage: PF = W.pieri_factors()
sage: [w.reduced_word() for w in PF.maximal_elements_combinatorial()]
[[3, 2, 1, 0], [2, 1, 0, 4], [1, 0, 4, 3], [0, 4, 3, 2], [4, 3, 2, 1]]
```

stanley_symm_poly_weight(*w*)

Weight used in computing (affine) Stanley symmetric polynomials for affine type A.

EXAMPLES:

```
sage: W = WeylGroup(['A',5,1])
sage: PF = W.pieri_factors()
sage: PF.stanley_symm_poly_weight(W.one())
0
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([5,4,2,1,0]))
0
```

subset(*length*)

INPUT:

- `length` – a non-negative integer

Returns the subset of the elements of `self` of length `length`

sage: PF = WeylGroup(["A", 3, 1]).pieri_factors(); PF Pieri factors for Weyl Group of type ['A', 3, 1] (as a matrix group acting on the root space) sage: PF3 = PF.subset(length = 2) sage: PF3.cardinality() 6

TESTS:

We check that there is no reference effect (there was at some point!):

```
sage: PF.cardinality()
15
```

class sage.combinat.root_system.pieri_factors.**PieriFactors_type_B**(*W*)
Bases: sage.combinat.root_system.pieri_factors.PieriFactors_finite_type

The type B finite Pieri factors are realized as the set of elements that have a reduced word that is a subword of 12...(n-1)n(n-1)...21. They are the restriction of the type C affine Pieri factors to the set of finite Weyl group elements under the usual embedding.

maximal_elements_combinatorial()
Returns the maximal Pieri factors, using the type B combinatorial description

EXAMPLES:

```
sage: PF = WeylGroup(['B',4]).pieri_factors()
sage: PF.maximal_elements_combinatorial()[0].reduced_word()
[1, 2, 3, 4, 3, 2, 1]
```

stanley_symm_poly_weight(*w*)
Weight used in computing Stanley symmetric polynomials of type B. The weight for finite type B is the number of components of the support of an element minus the number of occurrences of n in a reduced word.

EXAMPLES:

```
sage: W = WeylGroup(['B',5])
sage: PF = W.pieri_factors()
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([3,1,5]))
2
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([3,4,5]))
0
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([1,2,3,4,5,4]))
0
```

class sage.combinat.root_system.pieri_factors.**PieriFactors_type_B_affine**(*W*)
Bases: sage.combinat.root_system.pieri_factors.PieriFactors_affine_type

The type B affine Pieri factors are realized as the order ideal (in Bruhat order) generated by the following elements:

•cyclic rotations of the element with reduced word 234...(n-1)n(n-1)...3210, except for 123...n...320 and 023...n...321.

•123...(n-1)n(n-1)...321

•023...(n-1)n(n-1)...320

EXAMPLES:

```
sage: W = WeylGroup(['B',4,1])
sage: PF = W.pieri_factors()
sage: W.from_reduced_word([2,3,4,3,2,1,0]) in PF.maximal_elements()
True
sage: W.from_reduced_word([0,2,3,4,3,2,1]) in PF.maximal_elements()
False
sage: W.from_reduced_word([1,0,2,3,4,3,2]) in PF.maximal_elements()
True
sage: W.from_reduced_word([0,2,3,4,3,2,0]) in PF.maximal_elements()
True
sage: W.from_reduced_word([0,2,0]) in PF
True
```

maximal_elements_combinatorial()

Returns the maximal Pieri factors, using the affine type B combinatorial description

EXAMPLES:

```
sage: W = WeylGroup(['B',4,1])
sage: [u.reduced_word() for u in W.pieri_factors().maximal_elements_combinatorial()]
[[1, 0, 2, 3, 4, 3, 2], [2, 1, 0, 2, 3, 4, 3], [3, 2, 1, 0, 2, 3, 4], [4, 3, 2, 1, 0, 2, 3],
```

stanley_symm_poly_weight(*w*)

Returns the weight of a Pieri factor to be used in the definition of Stanley symmetric functions. For type B, this weight involves the number of components of the complement of the support of an element, where we consider 0 and 1 to be one node – if 1 is in the support, then we pretend 0 in the support, and vice versa. We also consider 0 and 1 to be one node for the purpose of counting components of the complement (as if the Dynkin diagram were that of type C). Let n be the rank of the affine Weyl group in question (if type ['B',k,1] then we have n = k+1). Let chi(v.length() < n-1) be the indicator function that is 1 if the length of v is smaller than n-1, and 0 if the length of v is greater than or equal to n-1. If we say c'(v) = the number of components of the complement of the support of v, then the type B weight is given by weight = c'(v) - chi(v.length() < n-1).

EXAMPLES:

```
sage: W = WeylGroup(['B',5,1])
sage: PF = W.pieri_factors()
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([0,3]))
1
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([0,1,3]))
1
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([2,3]))
1
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([2,3,4,5]))
0
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([0,5]))
0
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([2,4,5,4,3,0]))
-1
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([4,5,4,3,0]))
0
```

class sage.combinat.root_system.pieri_factors.**PieriFactors_type_C_affine**(*W*)

Bases: sage.combinat.root_system.pieri_factors.PieriFactors_affine_type

The type C affine Pieri factors are realized as the order ideal (in Bruhat order) generated by cyclic rotations of the element with unique reduced word 123...(n-1)n(n-1)...3210.

EXAMPLES:

```
sage: W = WeylGroup(['C',3,1])
sage: PF = W.pieri_factors()
sage: sorted([u.reduced_word() for u in PF.maximal_elements()], key=str)
[[0, 1, 2, 3, 2, 1], [1, 0, 1, 2, 3, 2], [1, 2, 3, 2, 1, 0],
 [2, 1, 0, 1, 2, 3], [2, 3, 2, 1, 0, 1], [3, 2, 1, 0, 1, 2]]
```

maximal_elements_combinatorial()

Returns the maximal Pieri factors, using the affine type C combinatorial description

EXAMPLES:

```
sage: PF = WeylGroup(['C',3,1]).pieri_factors()
sage: [w.reduced_word() for w in PF.maximal_elements_combinatorial()]
[[0, 1, 2, 3, 2, 1], [1, 0, 1, 2, 3, 2], [2, 1, 0, 1, 2, 3], [3, 2, 1, 0, 1, 2], [2, 3, 2, 1
```

stanley_symm_poly_weight(*w*)

Returns the weight of a Pieri factor to be used in the definition of Stanley symmetric functions. For type C, this weight is the number of connected components of the support (the indices appearing in a reduced word) of an element.

EXAMPLES:

```
sage: W = WeylGroup(['C',5,1])
sage: PF = W.pieri_factors()
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([1,3]))
2
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([1,3,2,0]))
1
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([5,3,0]))
3
sage: PF.stanley_symm_poly_weight(W.one())
0
```

class sage.combinat.root_system.pieri_factors.**PieriFactors_type_D_affine**(*W*)

Bases: sage.combinat.root_system.pieri_factors.PieriFactors_affine_type

The type D affine Pieri factors are realized as the order ideal (in Bruhat order) generated by the following elements:

- cyclic rotations of the element with reduced word 234...(n-2)n(n-1)(n-2)...3210 such that 1 and 0 are always adjacent and (n-1) and n are always adjacent.
- 123...(n-2)n(n-1)(n-2)...321
- 023...(n-2)n(n-1)(n-2)...320
- n(n-2)...2102...(n-2)n
- (n-1)(n-2)...2102...(n-2)(n-1)

EXAMPLES:

```
sage: W = WeylGroup(['D',5,1])
sage: PF = W.pieri_factors()
sage: W.from_reduced_word([3,2,1,0]) in PF
True
sage: W.from_reduced_word([0,3,2,1]) in PF
False
sage: W.from_reduced_word([0,1,3,2]) in PF
True
sage: W.from_reduced_word([2,0,1,3]) in PF
```

```
True
sage: sorted([u.reduced_word() for u in PF.maximal_elements()], key=str)
[[0, 2, 3, 5, 4, 3, 2, 0], [1, 0, 2, 3, 5, 4, 3, 2], [1, 2, 3, 5, 4, 3, 2, 1],
 [2, 1, 0, 2, 3, 5, 4, 3], [2, 3, 5, 4, 3, 2, 1, 0], [3, 2, 1, 0, 2, 3, 5, 4],
 [3, 5, 4, 3, 2, 1, 0, 2], [4, 3, 2, 1, 0, 2, 3, 4], [5, 3, 2, 1, 0, 2, 3, 5],
 [5, 4, 3, 2, 1, 0, 2, 3]]
```

maximal_elements_combinatorial()

Returns the maximal Pieri factors, using the affine type D combinatorial description

EXAMPLES:

```
sage: W = WeylGroup(['D',5,1])
sage: PF = W.pieri_factors()
sage: set(PF.maximal_elements_combinatorial()) == set(PF.maximal_elements())
True
```

stanley_symm_poly_weight(w)

INPUT:

- w – a pieri factor for this type

Returns the weight of w, to be used in the definition of Stanley symmetric functions. For type D, this weight involves the number of components of the complement of the support of an element, where we consider 0 and 1 to be one node – if 1 is in the support, then we pretend 0 in the support, and vice versa. Similarly with $n-1$ and n. We also consider 0 and 1, n-1 and n to be one node for the purpose of counting components of the complement (as if the Dynkin diagram were that of type C).

Type D Stanley symmetric polynomial weights are still conjectural. The given weight comes from conditions on elements of the affine Fomin-Stanley subalgebra, but work is needed to show this weight is correct for affine Stanley symmetric functions – see [LSS2009, Pon2010] for details.

EXAMPLES:

```
sage: W = WeylGroup(['D', 5, 1])
sage: PF = W.pieri_factors()
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([5,2,1]))
0
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([5,2,1,0]))
0
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([5,2]))
1
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([]))
0

sage: W = WeylGroup(['D',7,1])
sage: PF = W.pieri_factors()
sage: PF.stanley_symm_poly_weight(W.from_reduced_word([2,4,6]))
2
```

5.1.206 Tutorial: visualizing root systems

Root systems encode the positions of collections of hyperplanes in space, and form the fundamental combinatorial data underlying Coxeter and Weyl groups, Lie algebras and groups, etc. The theory can be a bit intimidating at first because of the many technical gadgets (roots, coroots, weights, ...). Vizualizing them goes a long way toward building a geometric intuition.

This tutorial starts from simple plots and guides you all the way to advanced plots with your own combinatorial data drawn on top of it.

See also:

- *Root systems* – An overview of root systems in Sage
- `RootLatticeRealizations.ParentMethods.plot()` – the main plotting function, with pointers to all the subroutines

First plots

In this first plot, we draw the root system for type A_2 in the ambient space. It is generated from two hyperplanes at a 120 degree angle:

```
sage: L = RootSystem(["A",2]).ambient_space()
sage: L.plot()
Graphics object consisting of 13 graphics primitives
```

Each of those hyperplane $H_{\alpha_i^\vee}$ is described by a linear form $\alpha_i^\vee$ called simple coroot. To each such hyperplane corresponds a reflection along a vector called root. In this picture, the reflections are orthogonal and the two simple roots α_1 and α_2 are vectors which are normal to the reflection hyperplanes. The same color code is used uniformly: blue for 1, red for 2, green for 3, ... (see `CartanType.color()`). The fundamental weights, Λ_1 and Λ_2 form the dual basis of the coroots.

The two reflections generate a group of order six which is nothing but the usual symmetric group S_3, in its natural action by permutations of the coordinates of the ambient space. Wait, but the ambient space should be of dimension 3 then? That's perfectly right. Here is the full picture in 3D:

```
sage: L = RootSystem(["A",2]).ambient_space()
sage: L.plot(projection=False)
Graphics3d Object
```

However in this space, the line $(1,1,1)$ is fixed by the action of the group. Therefore, the so called barycentric projection orthogonal to $(1,1,1)$ gives a convenient 2D picture which contains all the essential information. The same projection is used by default in type G_2:

```
sage: L = RootSystem(["G",2]).ambient_space()
sage: L.plot(reflection_hyperplanes="all")
Graphics object consisting of 21 graphics primitives
```

The group is now the dihedral group of order 12, generated by the two reflections s_1 and s_2. The picture displays the hyperplanes for all 12 reflections of the group. Those reflections delimit 12 chambers which are in one to one correspondance with the elements of the group. The fundamental chamber, which is grayed out, is associated with the identity of the group.

Warning: The fundamental chamber is currently plotted as the cone generated by the fundamental weights. As can be seen on the previous 3D picture this is not quite correct if the fundamental weights do not span the space. Another caveat is that some plotting features may require manipulating elements with rational coordinates which will fail if one is working in, say, the weight lattice. It is therefore recommended to use the root, weight, or ambient spaces for plotting purposes rather than their lattice counterparts.

Coming back to the symmetric group, here is the picture in the weight space, with all roots and all reflection hyperplanes; remark that, unlike in the ambient space, a root is not necessarily orthogonal to its corresponding reflection hyperplane:

```
sage: L = RootSystem(["A",2]).weight_space()
sage: L.plot(roots = "all", reflection_hyperplanes="all").show(figsize=15)
```

Note: Setting a larger figure size as above can help reduce the overlap between the text labels when the figure gets crowded.

One can further customize which roots to display, as in the following example showing the positive roots in the weight space for type ['G',2], labelled by their coordinates in the root lattice:

```
sage: Q = RootSystem(["G",2]).root_space()
sage: L = RootSystem(["G",2]).ambient_space()
sage: L.plot(roots=list(Q.positive_roots()), fundamental_weights=False)
Graphics object consisting of 17 graphics primitives
```

One can also customize the projection by specifying a function. Here, we display all the roots for type E_8 using the projection from its eight dimensional ambient space onto 3D described on Wikipedia's E8 3D picture:

```
sage: M = matrix([[0., -0.556793440452, 0.19694925177, -0.19694925177, 0.0805477263944, -0.385290876
...               [0.180913155536, 0., 0.160212955043, 0.160212955043, 0., 0.0990170516545, 0.766360
...               [0.338261212718, 0, 0, -0.338261212718, 0.672816364803, 0.171502564281, 0, -0.1715
sage: L = RootSystem(["E",8]).ambient_space()
sage: L.dimension()
8
sage: L.plot(roots="all", reflection_hyperplanes=False, projection=lambda v: M*vector(v), labels=Fals
Graphics3d Object
```

The projection function should be linear or affine, and return a vector with rational coordinates. The rationale for the later constraint is to allow for using the PPL exact library for manipulating polytopes. Indeed exact calculations give cleaner pictures (adjacent objects, intersection with the bounding box, ...). Besides the interface to PPL is indeed currently faster than that for CDD, and it is likely to become even more so.

Exercise

Draw all finite root systems in 2D, using the canonical projection onto their Coxeter plane. See Stembridge's page.

Alcoves and chambers

We now draw the root system for type G_2, with its alcoves (in finite type, those really are the chambers) and the corresponding elements of the Weyl group. We enlarge a bit the bounding box to make sure everything fits in the picture:

```
sage: RootSystem(["G",2]).ambient_space().plot(alcoves=True, alcove_labels=True, bounding_box=5)
Graphics object consisting of 37 graphics primitives
```

The same picture in 3D, for type B_3:

```
sage: RootSystem(["B",3]).ambient_space().plot(alcoves=True, alcove_labels=True)
Graphics3d Object
```

Exercise

Can you spot the fundamental chamber? The fundamental weights? The simple roots? The longest element of the Weyl group?

Alcove pictures for affine types

We now draw the usual alcove picture for affine type $A_2^{(1)}$:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: L.plot()                                  # long time
Graphics object consisting of 160 graphics primitives
```

This picture is convenient because it is low dimensional and contains most of the relevant information. Beside, by choosing the ambient space, the elements of the Weyl group act as orthogonal affine maps. In particular, reflections are usual (affine) orthogonal reflections. However this is in fact only a slice of the real picture: the Weyl group actually acts by linear maps on the full ambient space. Those maps stabilize the so-called level l hyperplanes, and we are visualizing here what's happening at level 1. Here is the full picture in 3D:

```
sage: L.plot(bounding_box=[[-3,3],[-3,3],[-1,1]], affine=False) # long time
Graphics3d Object
```

In fact, in type A, this really is a picture in 4D, but as usual the barycentric projection kills the boring extra dimension for us.

It's usually more readable to only draw the intersection of the reflection hyperplanes with the level 1 hyperplane:

```
sage: L.plot(affine=False, level=1)             # long time
Graphics3d Object
```

Such 3D pictures are useful to better understand technicalities, like the fact that the fundamental weights do not necessarily all live at level 1:

```
sage: L = RootSystem(["G",2,1]).ambient_space()
sage: L.plot(affine=False, level=1)
Graphics3d Object
```

Note: Such pictures may tend to be a bit flat, and it may be helpful to play with the aspect_ratio and more generaly with the various options of the `show()` method:

```
sage: p = L.plot(affine=False, level=1)
sage: p.show(aspect_ratio=[1,1,2], frame=False)
```

Exercise

Draw the alcove picture at level 1, and compare the position of the fundamental weights and the vertices of the fundamental alcove.

As for finite root systems, the alcoves are indexed by the elements of the Weyl group W. Two alcoves indexed by u and v respectively share a wall if u and v are neighbors in the right Cayley graph: $u = vs_i$; the color of that wall is given by i:

```
sage: L = RootSystem(["C",2,1]).ambient_space()
sage: L.plot(coroots="simple", alcove_labels=True) # long time
Graphics object consisting of 216 graphics primitives
```

Even 2D pictures of the rank $1 + 1$ cases can give some food for thought. Here, we draw the root lattice, with the positive roots of small height in the root poset:

```
sage: L = RootSystem(["A",1,1]).root_lattice()
sage: seed = L.simple_roots()
sage: succ = attrcall("pred")
sage: positive_roots = RecursivelyEnumeratedSet(seed, succ, structure='graded')
sage: it = iter(positive_roots)
sage: first_positive_roots = [next(it) for i in range(10)]
sage: L.plot(roots=first_positive_roots, affine=False, alcoves=False)
Graphics object consisting of 24 graphics primitives
```

Exercises

1. Use the same trick to draw the reflection hyperplanes in the weight lattice for the coroots of small height. Add the indexing of the alcoves by elements of the Weyl group. See below for a solution.
2. Draw the positive roots in the weight lattice and in the extended weight lattice.
3. Draw the reflection hyperplanes in the root lattice
4. Recreate John Stembridge's "Sandwich" arrangement pictures by choosing appropriate coroots for the reflection hyperplanes.

Here is a polished solution for the first exercise:

```
sage: L = RootSystem(["A",1,1]).weight_space()
sage: seed = L.simple_coroots()
sage: succ = attrcall("pred")
sage: positive_coroots = RecursivelyEnumeratedSet(seed, succ, structure='graded')
sage: it = iter(positive_coroots)
sage: first_positive_coroots = [next(it) for i in range(20)]
sage: p = L.plot(fundamental_chamber=True, reflection_hyperplanes=first_positive_coroots,
...              affine=False, alcove_labels=1,
...              bounding_box=[[-9,9],[-1,2]],
...              projection=lambda x: matrix([[1,-1],[1,1]])*vector(x))
sage: p.show(figsize=20)                          # long time
```

Higher dimension affine pictures

We now do some plots for rank 4 affine types, at level 1. The space is tiled by the alcoves, each of which is a 3D simplex:

```
sage: L = RootSystem(["A",3,1]).ambient_space()
sage: L.plot(reflection_hyperplanes=False, bounding_box=85/100) # long time
Graphics3d Object
```

It is recommended to use a small bounding box here, for otherwise the number of simplices grows quicker than what Sage can handle smoothly. It can help to specify explicitly which alcoves to visualize. Here is the fundamental alcove, specified by an element of the Weyl group:

```
sage: W = L.weyl_group()
sage: L.plot(reflection_hyperplanes=False, alcoves=[W.one()], bounding_box=2)
Graphics3d Object
```

and the fundamental polygon, specified by the coordinates of its center in the root lattice:

```
sage: W = L.weyl_group()
sage: L.plot(reflection_hyperplanes=False, alcoves=[[0,0]], bounding_box=2)
Graphics3d Object
```

Finally, we draw the alcoves in the classical fundamental chambers, using that those are indexed by the elements of the Weyl group having no other left descent than 0. In order to see the inner structure, we only draw the wireframe of the facets of the alcoves. Specifying the `wireframe` option requires a more flexible syntax for plots which will be explained later on in this tutorial:

```
sage: L = RootSystem(["B",3,1]).ambient_space()
sage: W = L.weyl_group()
sage: alcoves = [~w for d in range(12) for w in W.affine_grassmannian_elements_of_given_length(d)]
sage: p = L.plot_fundamental_chamber("classical")
sage: p += L.plot_alcoves(alcoves=alcoves, wireframe=True)
sage: p += L.plot_fundamental_weights()
sage: p.show(frame=False)
```

Exercises

1. Draw the fundamental alcove in the ambient space, just by itself (no reflection hyperplane, root, ...). The automorphism group of the Dynkin diagram for $A_3^{(1)}$ (a cycle of length 4) is the dihedral group. Visualize the corresponding symmetries of the fundamental alcove.
2. Draw the fundamental alcoves for the other rank 4 affine types, and recover the automorphism groups of their Dynkin diagram from the pictures.

Drawing on top of a root system plot

The root system plots have been designed to be used as wallpaper on top of which to draw more information. In the following example, we draw an alcove walk, specified by a word of indices of simple reflections, on top of the weight lattice in affine type $A_{2,1}$:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: w1 = [0,2,1,2,0,2,1,0,2,1,2,1,2,0,2,0,1,2,0]
sage: L.plot(alcove_walk=w1, bounding_box=6)        # long time
Graphics object consisting of 535 graphics primitives
```

Now, what about drawing several alcove walks, and specifying some colors? A single do-it-all plot method would be cumbersome; so instead, it is actually built on top of many methods (see the list below) that can be called independently and combined at will:

```
sage: L.plot_roots() + L.plot_reflection_hyperplanes()
Graphics object consisting of 12 graphics primitives
```

Note: By default the axes are disabled in root system plots since they tend to polute the picture. Annoyingly they come back when combining them. Here is a workaround:

```
sage: p = L.plot_roots() + L.plot_reflection_hyperplanes()
sage: p.axes(False)
sage: p
Graphics object consisting of 12 graphics primitives
```

In order to specify common information for all the pieces of a root system plot (choice of projection, bounding box, color code for the index set, ...), the easiest is to create an option object using `plot_parse_options()`, and pass it down to each piece. We use this to plot our two walks:

```
sage: plot_options = L.plot_parse_options(bounding_box=[[-2,5],[-2,6]])
sage: w2 = [2,1,2,0,2,0,2,1,2,0,1,2,1,2,1,0,1,2,0,2,0,1,2,0,2]
sage: p = L.plot_alcoves(plot_options=plot_options) # long time
sage: p += L.plot_alcove_walk(w1, color="green", plot_options=plot_options)
sage: p += L.plot_alcove_walk(w2, color="orange", plot_options=plot_options)
sage: p
Graphics object consisting of ... graphics primitives
```

And another with some foldings:

```
sage: p += L.plot_alcove_walk([0,1,2,0,2,0,1,2,0,1],
...                           foldings= [False, False, True, False, False, False, True, False, True,
...                           color="purple")
sage: p.axes(False)
sage: p.show(figsize=20)
```

Here we show a weight at level 0 and the reduced word implementing the translation by this weight:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: P = RootSystem(["A",2,1]).weight_space(extended=True)
sage: Lambda = P.fundamental_weights()
sage: t = 6*Lambda[1] - 2*Lambda[2] - 4*Lambda[0]
sage: walk = L.reduced_word_of_translation(L(t))
sage: plot_options = L.plot_parse_options(bounding_box=[[-2,5],[-2,5]])
sage: p = L.plot(plot_options=plot_options)        # long time
sage: p += L.plot_alcove_walk(walk, color="green", plot_options=plot_options)
sage: p += plot_options.family_of_vectors({t: L(t)})
sage: plot_options.finalize(p)
Graphics object consisting of ... graphics primitives
sage: p
Graphics object consisting of ... graphics primitives
```

Note that the coloring of the translated alcove does not match with that of the fundamental alcove: the translation actually lives in the extended Weyl group and is the composition of the simple reflections indexed by the alcove walk together with a rotation implementing an automorphism of the Dynkin diagram.

We conclude with a rank $3 + 1$ alcove walk:

```
sage: L = RootSystem(["B",3,1]).ambient_space()
sage: w3 = [0,2,1,3,2,0,2,1,0,2,3,1,2,1,3,2,0,2,0,1,2,0]
sage: L.plot_fundamental_weights() + L.plot_reflection_hyperplanes(bounding_box=2) + L.plot_alcove_wa
Graphics3d Object
```

Exercise

1. Draw the tiling of 3D space by the fundamental polygons for types A,B,C,D. Hints: use the `wireframe` option of `RootLatticeRealizations.ParentMethods.plot_alcoves()` and the `color` option of `plot()` to only draw the alcove facets indexed by 0.

Solution

```
sage: L = RootSystem(["A",3,1]).ambient_space()
sage: alcoves = cartesian_product([[0,1],[0,1],[0,1]])
sage: color = lambda i: "black" if i==0 else None
sage: L.plot_alcoves(alcoves=alcoves, color=color, bounding_box=10,wireframe=True).show(frame=False)
```

Hand drawing on top of a root system plot (aka Coxeter graph paper)

Taken from John Stembridge's excellent data archive:

"If you've ever worked with affine reflection groups, you've probably wasted lots of time drawing the reflecting hyperplanes of the rank 2 groups on scraps of paper. You may also have wished you had pads of graph paper with these lines drawn in for you. If so, you've come to the right place. Behold! Coxeter graph paper!".

Now you can create your own customized color Coxeter graph paper:

```
sage: L = RootSystem(["C",2,1]).ambient_space()
sage: p = L.plot(bounding_box=[[-8,9],[-5,7]], coroots="simple") # long time (10 s)
sage: p
Graphics object consisting of ... graphics primitives
```

By default Sage's plot are bitmap pictures which would come out ugly if printed on paper. Instead, we recommend saving the picture in postscript or svg before printing it:

```
sage: p.save("C21paper.eps")        # not tested
```

Note: Drawing pictures with a large number of alcoves is currently somewhat ridiculously slow. This is due to the use of generic code that works uniformly in all dimension rather than taylor-made code for 2D. Things should improve with the upcoming fast interface to the PPL library (see e.g. trac ticket #12553).

Drawing custom objects on top of a root system plot

So far so good. Now, what if one wants to draw, on top of a root system plot, some object for which there is no preexisting plot method? Again, the `plot_options` object come in handy, as it can be used to compute appropriate coordinates. Here we draw the permutohedron, that is the Cayley graph of the symmetric group W, by positioning each element w at $w(\rho)$, where ρ is in the fundamental alcove:

```
sage: L = RootSystem(["A",2]).ambient_space()
sage: rho = L.rho()
sage: plot_options = L.plot_parse_options()
sage: W = L.weyl_group()
sage: g = W.cayley_graph(side="right")
sage: positions = {w: plot_options.projection(w.action(rho)) for w in W}
sage: p = L.plot_alcoves()
```

```
sage: p += g.plot(pos = positions, vertex_size=0,
...               color_by_label=plot_options.color)
sage: p.axes(False)
sage: p
Graphics object consisting of 30 graphics primitives
```

Todo

Could we have nice LaTeX labels in this graph?

The same picture for A_3 gives a nice 3D permutohedron:

```
sage: L = RootSystem(["A",3]).ambient_space()
sage: rho = L.rho()
sage: plot_options = L.plot_parse_options()
sage: W = L.weyl_group()
sage: g = W.cayley_graph(side="right")
sage: positions = {w: plot_options.projection(w.action(rho)) for w in W}
sage: p = L.plot_roots()
sage: p += g.plot3d(pos3d = positions, color_by_label=plot_options.color)
sage: p
Graphics3d Object
```

Exercises

1. Locate the identity element of W in the previous picture
2. Rotate the picture appropriately to highlight the various symmetries of the permutohedron.
3. Make a function out of the previous example, and explore the Cayley graphs of all rank 2 and 3 Weyl groups.
4. Draw the root poset for type B_2 and B_3
5. Draw the root poset for type E_8 to recover the picture from Wikipedia article File:E8_3D.png

Similarly, we display a crystal graph by positioning each element according to its weight:

```
sage: C = crystals.Tableaux(["A",2], shape=[4,2])
sage: L = C.weight_lattice_realization()
sage: plot_options = L.plot_parse_options()

sage: g = C.digraph()
sage: positions = {x: plot_options.projection(x.weight()) for x in C}
sage: p = L.plot()
sage: p += g.plot(pos = positions,
...               color_by_label=plot_options.color, vertex_size=0)
sage: p.axes(False)
sage: p.show(figsize=15)
```

Note: In the above picture, many pairs of tableaux have the same weight and are thus superposed (look for example near the center). Some more layout logic would be needed to separate those nodes properly, but the foundations are laid firmly and uniformly accross all types of root systems for writing such extensions.

Here is an analogue picture in 3D:

```
sage: C = crystals.Tableaux(["A",3], shape=[3,2,1])
sage: L = C.weight_lattice_realization()
```

```
sage: plot_options = L.plot_parse_options()
sage: g = C.digraph()
sage: positions = {x:plot_options.projection(x.weight()) for x in C}
sage: p = L.plot(reflection_hyperplanes=False, fundamental_weights=False)
sage: p += g.plot3d(pos3d = positions, vertex_labels=True,
...                 color_by_label=plot_options.color, edge_labels=True)
sage: p
Graphics3d Object
```

Exercise

Explore the previous picture and notice how the edges of the crystal graph are parallel to the simple roots.

Enjoy and please post your best pictures on the Sage-Combinat wiki.

class sage.combinat.root_system.plot.**PlotOptions**(*space*, *projection=True*, *bounding_box=3*, *color=<bound method type.color of <class 'sage.combinat.root_system.cartan_type.CartanTypeFactory'>>*, *labels=True*, *level=None*, *affine=None*, *arrowsize=5*)

A class for plotting options for root lattice realizations.

See also:

- RootLatticeRealizations.ParentMethods.plot() for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

color(*i*)

Return the color to be used for objects indexed by i.

INPUT:

- i – an index

See also:

index_of_object()

EXAMPLES:

```
sage: L = RootSystem(["A",2]).root_lattice()
sage: options = L.plot_parse_options(labels=False)
sage: alpha = L.simple_roots()
sage: options.color(1)
'blue'
sage: options.color(2)
'red'
sage: for alpha in L.roots():
...       print alpha, options.color(alpha)
alpha[1]             blue
alpha[2]             red
alpha[1] + alpha[2]  black
-alpha[1]            black
-alpha[2]            black
-alpha[1] - alpha[2] black
```

cone(*rays=[ ], lines=[ ], color='black', thickness=1, alpha=1, wireframe=False, label=None, draw_degenerate=True, as_polyhedron=False*)

Return the cone generated by the given rays and lines.

INPUT:

- `rays`, `lines` – lists of elements of the root lattice realization (default: `[]`)
- `color` – a color (default: `"black"`)
- `alpha` – a number in the interval $[0, 1]$ (default: 1) the desired transparency
- `label` – an object to be used as for this cone. The label itself will be constructed by calling `latex()` or `repr()` on the object depending on the graphics backend.
- `draw_degenerate` – a boolean (default: `True`) whether to draw cones with a degenerate intersection with the bounding box
- `as_polyhedron` – a boolean (default: `False`) whether to return the result as a polyhedron, without clipping it to the bounding box, and without making a plot out of it (for testing purposes)

OUTPUT:

A graphic object, a polyhedron, or `0`.

EXAMPLES:

```
sage: L = RootSystem(["A",2]).root_lattice()
sage: options = L.plot_parse_options()
sage: alpha = L.simple_roots()
sage: p = options.cone(rays=[alpha[1]], lines=[alpha[2]], color='green', label=2)
sage: p
Graphics object consisting of 2 graphics primitives
sage: list(p)
[Polygon defined by 4 points,
 Text '$2$' at the point (3.15,3.15)]
sage: options.cone(rays=[alpha[1]], lines=[alpha[2]], color='green', label=2, as_polyhedron=
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex, 1 ray, 1 line
```

An empty result, being outside of the bounding box:

```
sage: options = L.plot_parse_options(labels=True, bounding_box=[[-10,-9]]*2)
sage: options.cone(rays=[alpha[1]], lines=[alpha[2]], color='green', label=2)
0
```

Test that the options are properly passed down:

```
sage: L = RootSystem(["A",2]).root_lattice()
sage: options = L.plot_parse_options()
sage: p = options.cone(rays=[alpha[1]+alpha[2]], color='green', label=2, thickness=4, alpha=
sage: list(p)
[Line defined by 2 points, Text '$2$' at the point (3.15,3.15)]
sage: sorted(p[0].options().items())
[('alpha', 0.500000000000000), ('legend_color', None),
 ('legend_label', None), ('rgbcolor', 'green'), ('thickness', 4),
 ('zorder', 1)]
```

This method is tested indirectly but extensively by the various plot methods of root lattice realizations.

empty(**args*)

Return an empty plot.

EXAMPLES:

```
sage: L = RootSystem(["A",2]).root_lattice()
sage: options = L.plot_parse_options(labels=True)
```

This currently returns `int(0)`:

```
sage: options.empty()
0
```

This is not a plot, so may cause some corner cases. On the other hand, 0 behaves as a fast neutral element, which is important given the typical idioms used in the plotting code:

```
sage: p = point([0,0])
sage: p + options.empty() is p
True
```

family_of_vectors(*vectors*)

Return a plot of a family of vectors.

INPUT:

- `vectors` – family or vectors in `self`

The vectors are labelled by their index.

EXAMPLES:

```
sage: L = RootSystem(["A",2]).root_lattice()
sage: options = L.plot_parse_options()
sage: alpha = L.simple_roots()
sage: p = options.family_of_vectors(alpha); p
Graphics object consisting of 4 graphics primitives
sage: list(p)
[Arrow from (0.0,0.0) to (1.0,0.0),
 Text '$1$' at the point (1.05,0.0),
 Arrow from (0.0,0.0) to (0.0,1.0),
 Text '$2$' at the point (0.0,1.05)]
```

Handling of colors and labels:

```
sage: color=lambda i: "purple" if i==1 else None
sage: options = L.plot_parse_options(labels=False, color=color)
sage: p = options.family_of_vectors(alpha)
sage: list(p)
[Arrow from (0.0,0.0) to (1.0,0.0)]
sage: p[0].options()['rgbcolor']
'purple'
```

Matplotlib emits a warning for arrows of length 0 and draws nothing anyway. So we do not draw them at all:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: options = L.plot_parse_options()
sage: Lambda = L.fundamental_weights()
sage: p = options.family_of_vectors(Lambda); p
Graphics object consisting of 5 graphics primitives
sage: list(p)
[Text '$0$' at the point (0.0,0.0),
 Arrow from (0.0,0.0) to (0.5,0.866024518389),
 Text '$1$' at the point (0.525,0.909325744308),
 Arrow from (0.0,0.0) to (-0.5,0.866024518389),
 Text '$2$' at the point (-0.525,0.909325744308)]
```

finalize(*G*)

Finalize a root system plot.

INPUT:

- G – a root system plot or 0

This sets the aspect ratio to 1 and remove the axes. This should be called by all the user-level plotting methods of root systems. This will become mostly obsolete when customization options won't be lost anymore upon addition of graphics objects and there will be a proper empty object for 2D and 3D plots.

EXAMPLES:

```
sage: L = RootSystem(["B",2,1]).ambient_space()
sage: options = L.plot_parse_options()
sage: p = L.plot_roots(plot_options=options)
sage: p += L.plot_coroots(plot_options=options)
sage: p.axes()
True
sage: p = options.finalize(p)
sage: p.axes()
False
sage: p.aspect_ratio()
1.0

sage: options = L.plot_parse_options(affine=False)
sage: p = L.plot_roots(plot_options=options)
sage: p += point([[1,1,0]])
sage: p = options.finalize(p)
sage: p.aspect_ratio()
[1.0, 1.0, 1.0]
```

If the input is 0, this returns an empty graphics object:

```
sage: type(options.finalize(0))
<class 'sage.plot.plot3d.base.Graphics3dGroup'>

sage: options = L.plot_parse_options()
sage: type(options.finalize(0))
<class 'sage.plot.graphics.Graphics'>
sage: list(options.finalize(0))
[]
```

in_bounding_box(*x*)

Return whether x is in the bounding box.

INPUT:

- x – an element of the root lattice realization

This method is currently one of the bottlenecks, and therefore cached.

EXAMPLES:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: options = L.plot_parse_options()
sage: alpha = L.simple_roots()
sage: options.in_bounding_box(alpha[1])
True
sage: options.in_bounding_box(3*alpha[1])
False
```

index_of_object(*i*)

Try to return the node of the Dynkin diagram indexing the object i.

OUTPUT: a node of the Dynkin diagram or `None`

EXAMPLES:

```
sage: L = RootSystem(["A",3]).root_lattice()
sage: alpha = L.simple_roots()
sage: omega = RootSystem(["A",3]).weight_lattice().fundamental_weights()
sage: options = L.plot_parse_options(labels=False)
sage: options.index_of_object(3)
3
sage: options.index_of_object(alpha[1])
1
sage: options.index_of_object(omega[2])
2
sage: options.index_of_object(omega[2]+omega[3])
sage: options.index_of_object(30)
sage: options.index_of_object("bla")
```

intersection_at_level_1(*x*)

Return `x` scaled at the appropriate level, if level is set; otherwise return `x`.

INPUT:

- `x` – an element of the root lattice realization

EXAMPLES:

```
sage: L = RootSystem(["A",2,1]).weight_space()
sage: options = L.plot_parse_options()
sage: options.intersection_at_level_1(L.rho())
1/3*Lambda[0] + 1/3*Lambda[1] + 1/3*Lambda[2]

sage: options = L.plot_parse_options(affine=False, level=2)
sage: options.intersection_at_level_1(L.rho())
2/3*Lambda[0] + 2/3*Lambda[1] + 2/3*Lambda[2]
```

When `level` is not set, `x` is returned:

```
sage: options = L.plot_parse_options(affine=False)
sage: options.intersection_at_level_1(L.rho())
Lambda[0] + Lambda[1] + Lambda[2]
```

projection(*v*)

Return the projection of `v`.

INPUT:

- `x` – an element of the root lattice realization

OUTPUT:

An immutable vector with integer or rational coefficients.

EXAMPLES:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: options = L.plot_parse_options()
sage: options.projection(L.rho())
(0, 989/571)

sage: options = L.plot_parse_options(projection=False)
```

```
sage: options.projection(L.rho())
(2, 1, 0)
```

reflection_hyperplane(*coroot*, *as_polyhedron=False*)

Return a plot of the reflection hyperplane indexed by this coroot.

- `coroot` – a coroot

EXAMPLES:

```
sage: L = RootSystem(["B",2]).weight_space()
sage: alphacheck = L.simple_coroots()
sage: options = L.plot_parse_options()
sage: H = options.reflection_hyperplane(alphacheck[1]); H
Graphics object consisting of 2 graphics primitives
```

TESTS:

```
sage: print H.description()
Text '$H_{\alpha^\vee_{1}}$' at the point (0.0,3.15)
Line defined by 2 points: [(0.0, 3.0), (0.0, -3.0)]

sage: L = RootSystem(["A",3,1]).ambient_space()
sage: alphacheck = L.simple_coroots()
sage: options = L.plot_parse_options()
sage: H = options.reflection_hyperplane(alphacheck[1], as_polyhedron=True); H
A 2-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex and 2 lines
sage: H.lines()
(A line in the direction (0, 0, 1), A line in the direction (0, 1, 0))
sage: H.vertices()
(A vertex at (0, 0, 0),)

sage: all(options.reflection_hyperplane(c, as_polyhedron=True).dim() == 2
...       for c in alphacheck)
True
```

Todo

Display the periodic orientation by adding a $+$ and a $-$ sign close to the label. Typically by using the associated root to shift a bit from the vertex upon which the hyperplane label is attached.

---

text(*label*, *position*, *rgbcolor=(0, 0, 0)*)

Return text widget with label `label` at position `position`

INPUT:

- `label` – a string, or a Sage object upon which latex will be called
- `position` – a position
- `rgbcolor` – the color as an RGB tuple

EXAMPLES:

```
sage: L = RootSystem(["A",2]).root_lattice()
sage: options = L.plot_parse_options()
sage: list(options.text("coucou", [0,1]))
[Text 'coucou' at the point (0.0,1.0)]
sage: list(options.text(L.simple_root(1), [0,1]))
[Text '$\alpha_{1}$' at the point (0.0,1.0)]
sage: list(options.text(L.simple_root(2), [1,0], rgbcolor=(1,0.5,0)))
```

```
[Text '$\alpha_{2}$' at the point (1.0,0.0)]

sage: options = RootSystem(["A",2]).root_lattice().plot_parse_options(labels=False)
sage: options.text("coucou", [0,1])
0

sage: options = RootSystem(["B",3]).root_lattice().plot_parse_options()
sage: print options.text("coucou", [0,1,2]).x3d_str()
<Transform translation='0 1 2'>
<Shape><Text string='coucou' solid='true'/><Appearance><Material diffuseColor='0.0 0.0 0.0'

</Transform>
```

thickness(*i*)

Return the thickness to be used for lines indexed by i.

INPUT:

- `i` – an index

See also:

`index_of_object()`

EXAMPLES:

```
sage: L = RootSystem(["A",2,1]).root_lattice()
sage: options = L.plot_parse_options(labels=False)
sage: alpha = L.simple_roots()
sage: options.thickness(0)
2
sage: options.thickness(1)
1
sage: options.thickness(2)
1
sage: for alpha in L.simple_roots():
....:     print alpha, options.thickness(alpha)
alpha[0] 2
alpha[1] 1
alpha[2] 1
```

sage.combinat.root_system.plot.**barycentric_projection_matrix**(*n*, *angle=0*)

Returns a family of $n+1$ vectors evenly spaced in a real vector space of dimension n

Those vectors are of norm 1, the scalar product between any two vector is $1/n$, thus the distance between two tips is constant.

The family is built recursively and uniquely determined by the following property: the last vector is $(0, \ldots, 0, -1)$, and the projection of the first n vectors in dimension $n-1$, after appropriate rescaling to norm 1, retrieves the family for $n-1$.

OUTPUT:

A matrix with $n+1$ columns of height n with rational or symbolic coefficients.

EXAMPLES:

One vector in dimension 0:

```
sage: from sage.combinat.root_system.root_lattice_realizations import barycentric_projection_mat
sage: m = barycentric_projection_matrix(0); m
[]
```

```
sage: matrix(QQ,0,1).nrows()
0
sage: matrix(QQ,0,1).ncols()
1
```

Two vectors in dimension 1:

```
sage: barycentric_projection_matrix(1)
[ 1 -1]
```

Three vectors in dimension 2:

```
sage: barycentric_projection_matrix(2)
[ 1/2*sqrt(3) -1/2*sqrt(3)            0]
[         1/2          1/2           -1]
```

Four vectors in dimension 3:

```
sage: m = barycentric_projection_matrix(3); m
[ 1/3*sqrt(3)*sqrt(2) -1/3*sqrt(3)*sqrt(2)                    0                    0]
[         1/3*sqrt(2)          1/3*sqrt(2)         -2/3*sqrt(2)                    0]
[                1/3                  1/3                  1/3                   -1]
```

The columns give four vectors that sum up to zero:

```
sage: sum(m.columns())
(0, 0, 0)
```

and have regular mutual angles:

```
sage: m.transpose()*m
[   1 -1/3 -1/3 -1/3]
[-1/3    1 -1/3 -1/3]
[-1/3 -1/3    1 -1/3]
[-1/3 -1/3 -1/3    1]
```

Here is a plot of them:

```
sage: sum(arrow((0,0,0),x) for x in m.columns())
Graphics3d Object
```

For 2D drawings of root systems, it is desirable to rotate the result to match with the usual conventions:

```
sage: barycentric_projection_matrix(2, angle=2*pi/3)
[         1/2          -1         1/2]
[ 1/2*sqrt(3)           0 -1/2*sqrt(3)]
```

TESTS:

```
sage: for n in range(1, 7):
...       m = barycentric_projection_matrix(n)
...       assert sum(m.columns()).is_zero()
...       assert matrix(QQ, n+1,n+1, lambda i,j: 1 if i==j else -1/n) == m.transpose()*m
```

5.1.207 Group algebras of root lattice realizations

class sage.combinat.root_system.root_lattice_realization_algebras.**Algebras**(*category*, **args*)

Bases: sage.categories.algebra_functor.AlgebrasCategory

The category of group algebras of root lattice realizations.

This includes typically weight rings (group algebras of weight lattices).

TESTS:

```
sage: for ct in CartanType.samples(crystallographic=True): # long time
....:     TestSuite(RootSystem(ct).root_lattice().algebra(QQ)).run()
```

class ElementMethods

acted_upon(*w*)

Implements the action of `w` on `self`.

INPUT:

- `w` – an element of the Weyl group acting on the underlying weight lattice realization

EXAMPLES:

```
sage: L = RootSystem(["A",3]).ambient_space()
sage: W = L.weyl_group()
sage: M = L.algebra(QQ['q','t'])
sage: m = M.an_element(); m  # TODO: investigate why we don't get something more interes
B[(2, 2, 3, 0)]
sage: m = (m+1)^2; m
B[(0, 0, 0, 0)] + 2*B[(2, 2, 3, 0)] + B[(4, 4, 6, 0)]
sage: w = W.an_element(); w.reduced_word()
[1, 2, 3]
sage: m.acted_upon(w)
B[(0, 0, 0, 0)] + 2*B[(0, 2, 2, 3)] + B[(0, 4, 4, 6)]
```

expand(*alphabet*)

Expand `self` into variables in the `alphabet`.

INPUT:

- `alphabet` – a non empty list/tuple of (invertible) variables in a ring to expand in

EXAMPLES:

```
sage: L = RootSystem(["A",2]).ambient_lattice()
sage: KL = L.algebra(QQ)
sage: p = KL.an_element() + KL.sum_of_monomials(L.some_elements()); p
B[(1, 0, 0)] + B[(1, -1, 0)] + B[(1, 1, 0)] + 2*B[(2, 2, 3)] + B[(0, 1, -1)]
sage: F = LaurentPolynomialRing(QQ, 'x,y,z')
sage: p.expand(F.gens())
2*x^2*y^2*z^3 + x*y + x + y*z^-1 + x*y^-1
```

TESTS:

```
sage: type(p.expand(F.gens()))
<type 'sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair'>

sage: p = KL.zero()
sage: p.expand(F.gens())
0
sage: type(p.expand(F.gens()))
<type 'sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair'>
```

class `Algebras`**.ParentMethods**

T0_check_on_basis(*q1*, *q2*, *convention='antidominant'*)

Return the $T_0^\vee$ operator acting on the basis.

This implements the formula for $T_{0'}$ in Section 6.12 of [Haiman06].

REFERENCES:

Warning: The current implementation probably returns just nonsense, if the convention is not "dominant".

EXAMPLES:

```
sage: K = QQ['q1,q2'].fraction_field()
sage: q1,q2 = K.gens()

sage: L = RootSystem(["A",1,1]).ambient_space()
sage: L0 = L.classical()
sage: KL = L.algebra(K)
sage: some_weights = L.fundamental_weights()
sage: f = KL.T0_check_on_basis(q1,q2, convention="dominant")
sage: f(L0.zero())
(q1+q2)*B[(0, 0)] + q1*B[(1, -1)]

sage: L = RootSystem(["A",3,1]).ambient_space()
sage: L0 = L.classical()
sage: KL = L.algebra(K)
sage: some_weights = L0.fundamental_weights()
sage: f = KL.T0_check_on_basis(q1,q2, convention="dominant")
sage: f(L0.zero())       # not checked
(q1+q2)*B[(0, 0, 0, 0)] + q1^3/q2^2*B[(1, 0, 0, -1)]
```

The following results have not been checked:

```
sage: for x in some_weights:
....:     print x, ": ", f(x)
(1, 0, 0, 0) :  q1*B[(1, 0, 0, 0)]
(1, 1, 0, 0) :  q1*B[(1, 1, 0, 0)]
(1, 1, 1, 0) :  q1*B[(1, 1, 1, 0)]
```

Some examples for type $B_2^{(1)}$ dual:

```
sage: L = RootSystem("B2~*").ambient_space()
sage: L0 = L.classical()
sage: e = L.basis()
sage: K = QQ['q,u'].fraction_field()
sage: q,u = K.gens()
sage: q1 = u
sage: q2 = -1/u
sage: KL = L.algebra(K)
sage: KL0 = KL.classical()
sage: f = KL.T0_check_on_basis(q1,q2, convention="dominant")
sage: T = KL.twisted_demazure_lusztig_operators(q1,q2, convention="dominant")
```

Direct calculation:

```
sage: T.Tw(0)(KL0.monomial(L0([0,0])))
((u^2-1)/u)*B[(0, 0)] + u^3*B[(1, 1)]
sage: KL.T0_check_on_basis(q1,q2, convention="dominant")(L0([0,0]))
((u^2-1)/u)*B[(0, 0)] + u^3*B[(1, 1)]
```

Step by step calculation, comparing by hand with Mark Shimozono:

```
sage: res = T.Tw(2)(KL0.monomial(L0([0,0]))); res
u*B[(0, 0)]
sage: res = res * KL0.monomial(L0([-1,1])); res
u*B[(-1, 1)]
sage: res = T.Tw_inverse(1)(res); res
(u^2-1)*B[(0, 0)] + u^2*B[(1, -1)]
```

```
sage: res = T.Tw_inverse(2)(res); res
((u^2-1)/u)*B[(0, 0)] + u^3*B[(1, 1)]
```

cartan_type()

Return the Cartan type of `self`.

EXAMPLES:

```
sage: A = RootSystem(["A",2,1]).ambient_space().algebra(QQ)
sage: A.cartan_type()
['A', 2, 1]
sage: A = RootSystem(["B",2]).weight_space().algebra(QQ)
sage: A.cartan_type()
['B', 2]
```

classical()

Return the group algebra of the corresponding classical lattice.

EXAMPLES:

```
sage: KL = RootSystem(["A",2,1]).ambient_space().algebra(QQ)
sage: KL.classical()
Group algebra of the Ambient space of the Root system of type ['A', 2] over Rational Fie
```

demazure_lusztig_operator_on_basis(*weight*, *i*, *q1*, *q2*, *convention='antidominant'*)

Return the result of applying the i-th Demazure-Lusztig operator on `weight`.

INPUT:

- `weight` – an element λ of the weight lattice
- `i` – an element of the index set
- `q1, q2` – two elements of the ground ring
- `convention` – "antidominant", "bar", or "dominant" (default: "antidominant")

See `demazure_lusztig_operators()` for the details.

EXAMPLES:

```
sage: L = RootSystem(["A",1]).ambient_space()
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KL = L.algebra(K)
sage: KL.demazure_lusztig_operator_on_basis(L((2,2)), 1, q1, q2)
q1*B[(2, 2)]
sage: KL.demazure_lusztig_operator_on_basis(L((3,0)), 1, q1, q2)
(q1+q2)*B[(1, 2)] + (q1+q2)*B[(2, 1)] + (q1+q2)*B[(3, 0)] + q1*B[(0, 3)]
sage: KL.demazure_lusztig_operator_on_basis(L((0,3)), 1, q1, q2)
(-q1-q2)*B[(1, 2)] + (-q1-q2)*B[(2, 1)] + (-q2)*B[(3, 0)]
```

At $q_1 = 1$ and $q_2 = 0$ we recover the action of the isobaric divided differences π_i:

```
sage: KL.demazure_lusztig_operator_on_basis(L((2,2)), 1, 1, 0)
B[(2, 2)]
sage: KL.demazure_lusztig_operator_on_basis(L((3,0)), 1, 1, 0)
B[(1, 2)] + B[(2, 1)] + B[(3, 0)] + B[(0, 3)]
sage: KL.demazure_lusztig_operator_on_basis(L((0,3)), 1, 1, 0)
-B[(1, 2)] - B[(2, 1)]
```

Or $1 - \pi_i$ for `bar=True`:

```
sage: KL.demazure_lusztig_operator_on_basis(L((2,2)), 1, 1, 0, convention="bar")
0
sage: KL.demazure_lusztig_operator_on_basis(L((3,0)), 1, 1, 0, convention="bar")
-B[(1, 2)] - B[(2, 1)] - B[(0, 3)]
sage: KL.demazure_lusztig_operator_on_basis(L((0,3)), 1, 1, 0, convention="bar")
```

```
B[(1, 2)] + B[(2, 1)] + B[(0, 3)]
```

At $q_1 = 1$ and $q_2 = -1$ we recover the action of the simple reflection s_i:

```
sage: KL.demazure_lusztig_operator_on_basis(L((2,2)), 1, 1, -1)
B[(2, 2)]
sage: KL.demazure_lusztig_operator_on_basis(L((3,0)), 1, 1, -1)
B[(0, 3)]
sage: KL.demazure_lusztig_operator_on_basis(L((0,3)), 1, 1, -1)
B[(3, 0)]
```

demazure_lusztig_operator_on_classical_on_basis(*weight*, *i*, *q*, *q1*, *q2*, *convention='antidominant'*)

Return the result of applying the i-th Demazure-Lusztig operator on the classical weight `weight` embedded at level 0.

INPUT:

- `weight` – a classical weight λ
- `i` – an element of the index set
- `q1, q2` – two elements of the ground ring
- `convention` – "antidominant", "bar", or "dominant" (default: "antidominant")

See `demazure_lusztig_operators()` for the details.

Todo

- Optimize the code to only do the embedding/projection for T_0
- Add an option to specify at which level one wants to work. Currently this is level 0.

EXAMPLES:

```
sage: L = RootSystem(["A",1,1]).ambient_space()
sage: L0 = L.classical()
sage: K = QQ['q,q1,q2']
sage: q, q1, q2 = K.gens()
sage: KL = L.algebra(K)
sage: KL0 = L0.algebra(K)
```

These operators coincide with the usual Demazure-Lusztig operators:

```
sage: KL.demazure_lusztig_operator_on_classical_on_basis(L0((2,2)), 1, q, q1, q2)
q1*B[(2, 2)]
sage: KL0.demazure_lusztig_operator_on_basis(L0((2,2)), 1, q1, q2)
q1*B[(2, 2)]

sage: KL.demazure_lusztig_operator_on_classical_on_basis(L0((3,0)), 1, q, q1, q2)
(q1+q2)*B[(1, 2)] + (q1+q2)*B[(2, 1)] + (q1+q2)*B[(3, 0)] + q1*B[(0, 3)]
sage: KL0.demazure_lusztig_operator_on_basis(L0((3,0)), 1, q1, q2)
(q1+q2)*B[(1, 2)] + (q1+q2)*B[(2, 1)] + (q1+q2)*B[(3, 0)] + q1*B[(0, 3)]
```

except that we now have an action of T_0, which introduces some q s:

```
sage: KL.demazure_lusztig_operator_on_classical_on_basis(L0((2,2)), 0, q, q1, q2)
q1*B[(2, 2)]
sage: KL.demazure_lusztig_operator_on_classical_on_basis(L0((3,0)), 0, q, q1, q2)
(-q^2*q1-q^2*q2)*B[(1, 2)] + (-q*q1-q*q2)*B[(2, 1)] + (-q^3*q2)*B[(0, 3)]
```

demazure_lusztig_operators(*q1*, *q2*, *convention='antidominant'*)

Return the Demazure-Lusztig operators acting on `self`.

INPUT:

- `q1, q2` – two elements of the ground ring
- `convention` – "antidominant", "bar", or "dominant" (default: "antidominant")

If R is the parent weight ring, the Demazure-Lusztig operator T_i is the linear map $R \to R$ obtained by interpolating between the isobaric divided difference operator π_i (see `isobaric_divided_difference_on_basis()`) and the simple reflection s_i.

$$(q_1 + q_2)\pi_i - q_2 s_i$$

The Demazure-Lusztig operators give the usual representation of the operator T_i of the (affine) Hecke algebra with eigenvalues q_1 and q_2 associated to the Weyl group.

Several variants are available to match with various conventions used in the literature:

- "bar" replaces π_i in the formula above by $\overline{\pi}_i = (1 - \pi_i)$.
- "dominant" conjugates the operator by $x^\lambda \mapsto x^- \lambda$.

The names dominant and antidominant for the conventions were chosen with regards to the nonsymmetric Macdonald polynomials. The Y operators for the Macdonald polynomials in the "dominant" convention satisfy $Y_\lambda = T_{t_\lambda}$ for λ dominant. This is also the convention used in [Haiman06]. For the "antidominant" convention, $Y_\lambda = T_{t_\lambda}$ with λ antidominant.

See also:

- `demazure_lusztig_operator_on_basis()`.
- `NonSymmetricMacdonaldPolynomials`.

REFERENCES:

EXAMPLES:

```
sage: L = RootSystem(["A",1]).ambient_space()
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KL = L.algebra(K)
sage: T = KL.demazure_lusztig_operators(q1, q2)
sage: Tbar = KL.demazure_lusztig_operators(q1, q2, convention="bar")
sage: Tdominant = KL.demazure_lusztig_operators(q1, q2, convention="dominant")
sage: x = KL.monomial(L((3,0)))
sage: T[1](x)
(q1+q2)*B[(1, 2)] + (q1+q2)*B[(2, 1)] + (q1+q2)*B[(3, 0)] + q1*B[(0, 3)]
sage: Tbar[1](x)
(-q1-q2)*B[(1, 2)] + (-q1-q2)*B[(2, 1)] + (-q1-2*q2)*B[(0, 3)]
sage: Tbar[1](x) + T[1](x)
(q1+q2)*B[(3, 0)] + (-2*q2)*B[(0, 3)]
sage: Tdominant[1](x)
(-q1-q2)*B[(1, 2)] + (-q1-q2)*B[(2, 1)] + (-q2)*B[(0, 3)]

sage: Tdominant.Tw_inverse(1)(KL.monomial(-L.simple_root(1)))
((-q1-q2)/(q1*q2))*B[(0, 0)] - 1/q2*B[(1, -1)]
```

We repeat similar computation in the affine setting:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KL = L.algebra(K)
sage: T = KL.demazure_lusztig_operators(q1, q2)
sage: Tbar = KL.demazure_lusztig_operators(q1, q2, convention="bar")
sage: Tdominant = KL.demazure_lusztig_operators(q1, q2, convention="dominant")
sage: e = L.basis()
sage: x = KL.monomial(3*e[0])
sage: T[1](x)
(q1+q2)*B[e[0] + 2*e[1]] + (q1+q2)*B[2*e[0] + e[1]] + (q1+q2)*B[3*e[0]] + q1*B[3*e[1]]
sage: Tbar[1](x)
(-q1-q2)*B[e[0] + 2*e[1]] + (-q1-q2)*B[2*e[0] + e[1]] + (-q1-2*q2)*B[3*e[1]]
```

```
sage: Tbar[1](x) + T[1](x)
(q1+q2)*B[3*e[0]] + (-2*q2)*B[3*e[1]]
sage: Tdominant[1](x)
(-q1-q2)*B[e[0] + 2*e[1]] + (-q1-q2)*B[2*e[0] + e[1]] + (-q2)*B[3*e[1]]
sage: Tdominant.Tw_inverse(1)(KL.monomial(-L.simple_root(1)))
((-q1-q2)/(q1*q2))*B[0] - 1/q2*B[e[0] - e[1]]
```

One can obtain iterated operators by passing a reduced word or an element of the Weyl group:

```
sage: T[1,2](x)
(q1^2+2*q1*q2+q2^2)*B[e[0] + e[1] + e[2]] +
(q1^2+2*q1*q2+q2^2)*B[e[0] + 2*e[1]] +
(q1^2+q1*q2)*B[e[0] + 2*e[2]] + (q1^2+2*q1*q2+q2^2)*B[2*e[0] + e[1]] +
(q1^2+q1*q2)*B[2*e[0] + e[2]] + (q1^2+q1*q2)*B[3*e[0]] +
(q1^2+q1*q2)*B[e[1] + 2*e[2]] + (q1^2+q1*q2)*B[2*e[1] + e[2]] +
(q1^2+q1*q2)*B[3*e[1]] + q1^2*B[3*e[2]]
```

and use that to check, for example, the braid relations:

```
sage: T[1,2,1](x) - T[2,1,2](x)
0
```

The operators satisfy the relations of the affine Hecke algebra with parameters q_1, q_2:

```
sage: T._test_relations()
sage: Tdominant._test_relations()
sage: Tbar._test_relations() #-q2,q1+2*q2   # todo: not implemented: set the appropriate
```

And the $\bar{T}$ are basically the inverses of the T s:

```
sage: Tinv = KL.demazure_lusztig_operators(2/q1+1/q2,-1/q1,convention="bar")
sage: [Tinv[1](T[1](x))-x for x in KL.some_elements()]
[0, 0, 0, 0, 0, 0, 0]
```

We check that $\Lambda_1 - \Lambda_0$ is an eigenvector for the Y s in affine type:

```
sage: K = QQ['q,q1,q2'].fraction_field()
sage: q,q1,q2=K.gens()
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: L0 = L.classical()
sage: Lambda = L.fundamental_weights()
sage: alphacheck = L0.simple_coroots()
sage: KL = L.algebra(K)
sage: T = KL.demazure_lusztig_operators(q1, q2, convention="dominant")
sage: Y = T.Y()
sage: alphacheck = Y.keys().alpha() # alpha of coroot lattice is alphacheck
sage: alphacheck
Finite family {0: alphacheck[0], 1: alphacheck[1], 2: alphacheck[2]}
sage: x = KL.monomial(Lambda[1]-Lambda[0]); x
B[e[0]]
```

In fact it is not exactly an eigenvector, but the extra 'delta' term is to be interpreted as a q parameter:

```
sage: Y[alphacheck[0]](KL.one())
q2^2/q1^2*B[0]
sage: Y[alphacheck[1]](x)
((-q2^2)/(-q1^2))*B[e[0] - e['delta']]
sage: Y[alphacheck[2]](x)
(q1/(-q2))*B[e[0]]
sage: KL.q_project(Y[alphacheck[1]](x),q)
((-q2^2)/(-q*q1^2))*B[(1, 0, 0)]

sage: KL.q_project(x, q)
B[(1, 0, 0)]
```

```
sage: KL.q_project(Y[alphacheck[0]](x),q)
((-q*q1)/q2)*B[(1, 0, 0)]
sage: KL.q_project(Y[alphacheck[1]](x),q)
((-q2^2)/(-q*q1^2))*B[(1, 0, 0)]
sage: KL.q_project(Y[alphacheck[2]](x),q)
(q1/(-q2))*B[(1, 0, 0)]
```

We now check systematically that the Demazure-Lusztig operators satisfy the relations of the Iwahori-Hecke algebra:

```
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: for cartan_type in CartanType.samples(crystallographic=True): # long time 12s
....:     L = RootSystem(cartan_type).root_lattice()
....:     KL = L.algebra(K)
....:     T = KL.demazure_lusztig_operators(q1,q2)
....:     T._test_relations()

sage: for cartan_type in CartanType.samples(crystallographic=True): # long time 12s
....:     L = RootSystem(cartan_type).weight_lattice()
....:     KL = L.algebra(K)
....:     T = KL.demazure_lusztig_operators(q1,q2)
....:     T._test_relations()
```

Recall that the Demazure-Lusztig operators are only defined when all monomials belong to the weight lattice. Thus, in the group algebra of the ambient space, we need to specify explicitly the elements on which to run the tests:

```
sage: for cartan_type in CartanType.samples(crystallographic=True): # long time 12s
....:     L = RootSystem(cartan_type).ambient_space()
....:     KL = L.algebra(K)
....:     weight_lattice = RootSystem(cartan_type).weight_lattice(extended=L.is_extended(
....:     elements = [ KL.monomial(L(weight)) for weight in weight_lattice.some_elements(
....:     T = KL.demazure_lusztig_operators(q1,q2)
....:     T._test_relations(elements=elements)
```

demazure_lusztig_operators_on_classical(*q*, *q1*, *q2*, *convention='antidominant'*)

Return the Demazure-Lusztig operators acting at level 1 on `self.classical()`.

INPUT:

- `q, q1, q2` – three elements of the ground ring
- `convention` – "antidominant", "bar", or "dominant" (default: "antidominant")

Let KL be the group algebra of an affine weight lattice realization L. The Demazure-Lusztig operators for KL act on the group algebra of the corresponding classical weight lattice by embedding it at level 1, and projecting back.

See also:

- `demazure_lusztig_operators()`.
- `demazure_lusztig_operator_on_classical_on_basis()`.
- `q_project()`

EXAMPLES:

```
sage: L = RootSystem(["A",1,1]).ambient_space()
sage: K = QQ['q,q1,q2'].fraction_field()
sage: q, q1, q2 = K.gens()
sage: KL = L.algebra(K)
sage: KL0 = KL.classical()
sage: L0 = KL0.basis().keys()
sage: T = KL.demazure_lusztig_operators_on_classical(q, q1, q2)
```

```
sage: x = KL0.monomial(L0((3,0))); x
B[(3, 0)]
```

For $T_1, \ldots$ we recover the usual Demazure-Lusztig operators:

```
sage: T[1](x)
(q1+q2)*B[(1, 2)] + (q1+q2)*B[(2, 1)] + (q1+q2)*B[(3, 0)] + q1*B[(0, 3)]
```

For T_0, we can note that, in the projection, δ is mapped to q:

```
sage: T[0](x)
(-q^2*q1-q^2*q2)*B[(1, 2)] + (-q*q1-q*q2)*B[(2, 1)] + (-q^3*q2)*B[(0, 3)]
```

Note that there is no translation part, and in particular 1 is an eigenvector for all T_i's:

```
sage: T[0](KL0.one())
q1*B[(0, 0)]
sage: T[1](KL0.one())
q1*B[(0, 0)]

sage: Y = T.Y()
sage: alphacheck=Y.keys().simple_roots()
sage: Y[alphacheck[0]](KL0.one())
((-q2)/(q*q1))*B[(0, 0)]
```

Matching with Ion Bogdan's hand calculations from 3/15/2013:

```
sage: L = RootSystem(["A",1,1]).weight_space(extended=True)
sage: K = QQ['q,u'].fraction_field()
sage: q, u = K.gens()
sage: KL = L.algebra(K)
sage: KL0 = KL.classical()
sage: L0 = KL0.basis().keys()
sage: omega = L0.fundamental_weights()
sage: T = KL.demazure_lusztig_operators_on_classical(q, u, -1/u, convention="dominant")
sage: Y = T.Y()
sage: alphacheck = Y.keys().simple_roots()

sage: Ydelta = Y[Y.keys().null_root()]
sage: Ydelta.word, Ydelta.signs, Ydelta.scalar
((), (), 1/q)

sage: Y1 = Y[alphacheck[1]]
sage: Y1.word, Y1.signs, Y1.scalar # This is T_0 T_1 (T_1 acts first, then T_0); Ion get
((1, 0), (1, 1), 1)

sage: Y0 = Y[alphacheck[0]]
sage: Y0.word, Y0.signs, Y0.scalar # This is 1/q T_1^-1 T_0^-1
((0, 1), (-1, -1), 1/q)
```

Note that the following computations use the "dominant" convention:

```
sage: T0 = T.Tw(0)
sage: T0(KL0.monomial(omega[1]))
q*u*B[-Lambda[1]] + ((u^2-1)/u)*B[Lambda[1]]
sage: T0(KL0.monomial(2*omega[1]))
((q*u^2-q)/u)*B[0] + q^2*u*B[-2*Lambda[1]] + ((u^2-1)/u)*B[2*Lambda[1]]

sage: T0(KL0.monomial(-omega[1]))
1/(q*u)*B[Lambda[1]]
sage: T0(KL0.monomial(-2*omega[1]))
((-u^2+1)/(q*u))*B[0] + 1/(q^2*u)*B[2*Lambda[1]]
```

demazure_operators()

Return the Demazure operators acting on `self`.

The i-th Demazure operator is defined by:

$$\pi_i = \frac{1 - e^{-\alpha_i} s_i}{1 - e^{-\alpha_i}}$$

It acts on e^λ, for λ a weight, by:

$$\pi_i e^\lambda = \frac{e^\lambda - e^{-\alpha_i + s_i \lambda}}{1 - e^{-\alpha_i}}$$

This matches with Lascoux' definition [Lascoux2003] of π_i, and with the i-th Demazure operator of [Kumar1987], which also works for general Kac-Moody types.

REFERENCES:

EXAMPLES:

We compute some Schur functions, as images of dominant monomials under the action of the maximal isobaric divided difference Δ_{w_0}:

```
sage: L = RootSystem(["A",2]).ambient_lattice()
sage: KL = L.algebra(QQ)
sage: w0 = tuple(L.weyl_group().long_element().reduced_word())
sage: pi = KL.demazure_operators()
sage: pi0 = pi[w0]
sage: pi0(KL.monomial(L((2,1))))
2*B[(1, 1, 1)] + B[(1, 2, 0)] + B[(1, 0, 2)] + B[(2, 1, 0)] + B[(2, 0, 1)] + B[(0, 1, 2)
```

Let us make the result into an actual polynomial:

```
sage: P = QQ['x,y,z']
sage: pi0(KL.monomial(L((2,1)))).expand(P.gens())
x^2*y + x*y^2 + x^2*z + 2*x*y*z + y^2*z + x*z^2 + y*z^2
```

This is indeed a Schur function:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s[2,1].expand(3, P.gens())
x^2*y + x*y^2 + x^2*z + 2*x*y*z + y^2*z + x*z^2 + y*z^2
```

Let us check this systematically on Schur functions of degree 6:

```
sage: for p in Partitions(6, max_length=3).list():
....:     assert s.monomial(p).expand(3, P.gens()) == pi0(KL.monomial(L(tuple(p)))).expa
```

We check systematically that these operators satisfy the Iwahori-Hecke algebra relations:

```
sage: for cartan_type in CartanType.samples(crystallographic=True): # long time 12s
....:     L = RootSystem(cartan_type).weight_lattice()
....:     KL = L.algebra(QQ)
....:     T = KL.demazure_operators()
....:     T._test_relations()

sage: L = RootSystem(['A',1,1]).weight_lattice()
sage: KL = L.algebra(QQ)
sage: T = KL.demazure_operators()
sage: T._test_relations()
```

Warning: The Demazure operators are only defined if all the elements in the support have integral scalar products with the coroots (basically, they are in the weight lattice). Otherwise an error is raised:

```
sage: L = RootSystem(CartanType(["G",2]).dual()).ambient_space()
sage: KL = L.algebra(QQ)
sage: pi = KL.demazure_operators()
sage: pi[1](KL.monomial(L([0,0,1])))
Traceback (most recent call last):
...
ValueError: the weight does not have an integral scalar product with the coroot
```

divided_difference_on_basis(*weight*, *i*)

Return the result of applying the i-th divided difference on `weight`.

INPUT:

- `weight` – a weight
- `i` – an element of the index set

Todo

type free definition (Viviane's definition uses that we are in the ambient space)

EXAMPLES:

```
sage: L = RootSystem(["A",1]).ambient_space()
sage: KL = L.algebra(QQ)
sage: KL.divided_difference_on_basis(L((2,2)), 1) # todo: not implemented
0
sage: KL.divided_difference_on_basis(L((3,0)), 1) # todo: not implemented
B[(2, 0)] + B[(1, 1)] + B[(0, 2)]
sage: KL.divided_difference_on_basis(L((0,3)), 1) # todo: not implemented
-B[(2, 0)] - B[(1, 1)] - B[(0, 2)]
```

In type A and in the ambient lattice, we recover the usual action of divided differences polynomials:

```
sage: x,y = QQ['x,y'].gens()
sage: d = lambda p: (p - p(y,x)) / (x-y)
sage: d(x^2*y^2)
0
sage: d(x^3)
x^2 + x*y + y^2
sage: d(y^3)
-x^2 - x*y - y^2
```

from_polynomial(*p*)

Construct an element of `self` from a polynomial p.

INPUT:

- `p` – a polynomial

EXAMPLES:

```
sage: L = RootSystem(["A",2]).ambient_lattice()
sage: KL = L.algebra(QQ)
sage: x,y,z = QQ['x,y,z'].gens()
sage: KL.from_polynomial(x)
B[(1, 0, 0)]
sage: KL.from_polynomial(x^2*y + 2*y - z)
B[(2, 1, 0)] + 2*B[(0, 1, 0)] - B[(0, 0, 1)]
```

TESTS:

```
sage: KL.from_polynomial(x).leading_support().parent() is L
True
sage: KL.from_polynomial(x-x)
0
sage: KL.from_polynomial(x-x).parent() is KL
True
```

Todo

make this work for Laurent polynomials too

---

isobaric_divided_difference_on_basis(*weight*, *i*)

Return the result of applying the i-th isobaric divided difference on `weight`.

INPUT:

- `weight` – a weight
- `i` – an element of the index set

See also:

`demazure_operators()`

EXAMPLES:

```
sage: L = RootSystem(["A",1]).ambient_space()
sage: KL = L.algebra(QQ)
sage: KL.isobaric_divided_difference_on_basis(L((2,2)), 1)
B[(2, 2)]
sage: KL.isobaric_divided_difference_on_basis(L((3,0)), 1)
B[(1, 2)] + B[(2, 1)] + B[(3, 0)] + B[(0, 3)]
sage: KL.isobaric_divided_difference_on_basis(L((0,3)), 1)
-B[(1, 2)] - B[(2, 1)]
```

In type A and in the ambient lattice, we recover the usual action of divided differences on polynomials:

```
sage: x,y = QQ['x,y'].gens()
sage: d = lambda p: (x*p - (x*p)(y,x)) / (x-y)
sage: d(x^2*y^2)
x^2*y^2
sage: d(x^3)
x^3 + x^2*y + x*y^2 + y^3
sage: d(y^3)
-x^2*y - x*y^2
```

REFERENCES:

q_project(*x*, *q*)

Implement the q-projection morphism from `self` to the group algebra of the classical space.

INPUT:

- `x` – an element of the group algebra of `self`
- `q` – an element of the ground ring

This is an algebra morphism mapping δ to q and X^b to its classical counterpart for the other elements b of the basis of the realization.

EXAMPLES:

```
sage: K = QQ['q'].fraction_field()
sage: q = K.gen()
sage: KL = RootSystem(["A",2,1]).ambient_space().algebra(K)
sage: L = KL.basis().keys()
sage: e = L.basis()
```

```
sage: x = KL.an_element() + KL.monomial(4*e[1] + 3*e[2] + e['deltacheck'] - 2*e['delta'])
B[2*e[0] + 2*e[1] + 3*e[2]] + B[4*e[1] + 3*e[2] - 2*e['delta'] + e['deltacheck']]
sage: KL.q_project(x, q)
B[(2, 2, 3)] + 1/q^2*B[(0, 4, 3)]

sage: KL = RootSystem(["BC",3,2]).ambient_space().algebra(K)
sage: L = KL.basis().keys()
sage: e = L.basis()
sage: x = KL.an_element() + KL.monomial(4*e[1] + 3*e[2] + e['deltacheck'] - 2*e['delta'])
B[2*e[0] + 2*e[1] + 3*e[2]] + B[4*e[1] + 3*e[2] - 2*e['delta'] + e['deltacheck']]
sage: KL.q_project(x, q)
B[(2, 2, 3)] + 1/q^2*B[(0, 4, 3)]
```

Warning: Recall that the null root, usually denoted δ, is in fact `a[0]\delta` in Sage's notation, in order to avoid half integer coefficients (this only makes a difference in type BC). Similarly, what's usually denoted q is in fact `q^a[0]` in Sage's notations, to avoid manipulating square roots:

```
sage: KL.q_project(KL.monomial(L.null_root()),q)
q^2*B[(0, 0, 0)]
```

q_project_on_basis(l, q)

Return the monomial $c * cl(l)$ in the group algebra of the classical lattice.

INPUT:

- `l` – an element of the root lattice realization
- `q` – an element of the ground ring

Here, $cl(l)$ is the projection of l in the classical lattice, and c is the coefficient of l in δ.

See also:

`q_project_on_basis()`

EXAMPLES:

```
sage: K = QQ['q'].fraction_field()
sage: q = K.gen()
sage: KL = RootSystem(["A",2,1]).ambient_space().algebra(K)
sage: L = KL.basis().keys()
sage: e = L.basis()
sage: KL.q_project_on_basis( 4*e[1] + 3*e[2] + e['deltacheck'] - 2*e['delta'], q)
1/q^2*B[(0, 4, 3)]
```

some_elements()

Return some elements of the algebra `self`.

EXAMPLES:

```
sage: A = RootSystem(["A",2,1]).ambient_space().algebra(QQ)
sage: A.some_elements()
[B[2*e[0] + 2*e[1] + 3*e[2]],
B[-e[0] + e[2] + e['delta']],
B[e[0] - e[1]],
B[e[1] - e[2]],
B[e['deltacheck']],
B[e[0] + e['deltacheck']],
B[e[0] + e[1] + e['deltacheck']]]

sage: A = RootSystem(["B",2]).weight_space().algebra(QQ)
```

```
sage: A.some_elements()
[B[2*Lambda[1] + 2*Lambda[2]],
B[2*Lambda[1] - 2*Lambda[2]],
B[-Lambda[1] + 2*Lambda[2]],
B[Lambda[1]],
B[Lambda[2]]]
```

twisted_demazure_lusztig_operator_on_basis(*weight*, *i*, *q1*, *q2*, *convention='antidominant'*)

Return the twisted Demazure-Lusztig operator acting on the basis.

INPUT:

- `weight` – an element λ of the weight lattice
- `i` – an element of the index set
- `q1, q2` – two elements of the ground ring
- `convention` – "antidominant", "bar", or "dominant" (default: "antidominant")

See also:

`twisted_demazure_lusztig_operators()`

EXAMPLES:

```
sage: L = RootSystem(["A",3,1]).ambient_space()
sage: e = L.basis()
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KL = L.algebra(K)
sage: Lambda = L.classical().fundamental_weights()
sage: KL.twisted_demazure_lusztig_operator_on_basis(Lambda[1]+2*Lambda[2], 1, q1, q2, co
(-q2)*B[(2, 3, 0, 0)]
sage: KL.twisted_demazure_lusztig_operator_on_basis(Lambda[1]+2*Lambda[2], 2, q1, q2, co
(-q1-q2)*B[(3, 1, 1, 0)] + (-q2)*B[(3, 0, 2, 0)]
sage: KL.twisted_demazure_lusztig_operator_on_basis(Lambda[1]+2*Lambda[2], 3, q1, q2, co
q1*B[(3, 2, 0, 0)]
sage: KL.twisted_demazure_lusztig_operator_on_basis(Lambda[1]+2*Lambda[2], 0, q1, q2, co
((q1*q2+q2^2)/q1)*B[(1, 2, 1, 1)] + ((q1*q2+q2^2)/q1)*B[(1, 2, 2, 0)] + q2^2/q1*B[(1, 2,
+ ((q1^2+2*q1*q2+q2^2)/q1)*B[(2, 1, 1, 1)] + ((q1^2+2*q1*q2+q2^2)/q1)*B[(2, 1, 2, 0)]
+ ((q1*q2+q2^2)/q1)*B[(2, 1, 0, 2)] + ((q1^2+2*q1*q2+q2^2)/q1)*B[(2, 2, 1, 0)] + ((q1*q2
```

twisted_demazure_lusztig_operators(*q1*, *q2*, *convention='antidominant'*)

Return the twisted Demazure-Lusztig operators acting on `self`.

INPUT:

- `q1, q2` – two elements of the ground ring
- `convention` – "antidominant", "bar", or "dominant" (default: "antidominant")

Warning:

- the code is currently only tested for $q_1 q_2 = -1$
- only the "dominant" convention is functional for $i = 0$

For $T_1, \ldots, T_n$, these operators are the usual Demazure-Lusztig operators. On the other hand, the operator T_0 is twisted:

```
sage: L = RootSystem(["A",3,1]).ambient_space()
sage: e = L.basis()
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KL = L.algebra(K)
sage: T = KL.twisted_demazure_lusztig_operators(q1, q2, convention="dominant")
sage: T._test_relations()
```

TESTS:

The following computations were checked with Mark Shimozono for type $A_1^{(1)}$:

```
sage: L = RootSystem(["A",1,1]).ambient_space()
sage: e = L.basis()
sage: K = QQ['q1,q2'].fraction_field()
sage: q1,q2 = K.gens()
sage: KL = L.algebra(K)
sage: T = KL.twisted_demazure_lusztig_operators(q1, q2, convention="dominant")
sage: T._test_relations()
sage: L0 = L.classical()
sage: alpha = L0.simple_roots()
sage: T.Ti_on_basis(L0.zero(), 1)
q1*B[(0, 0)]
sage: T.Ti_inverse_on_basis(L0.zero(), 1)
1/q1*B[(0, 0)]
sage: T.Ti_on_basis(alpha[1], 1)
(-q1-q2)*B[(0, 0)] + (-q2)*B[(-1, 1)]
sage: T.Ti_inverse_on_basis(alpha[1], 1)
((q1+q2)/(q1*q2))*B[(0, 0)] + 1/q1*B[(-1, 1)] + ((q1+q2)/(q1*q2))*B[(1, -1)]
sage: T.Ti_on_basis(L0.zero(), 0)
(q1+q2)*B[(0, 0)] + q1*B[(1, -1)]
```

The next computations were checked with Mark Shimozono for type $A_2^{(1)}$:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: e = L.basis()
sage: K = QQ['u'].fraction_field()
sage: u = K.gen()
sage: KL = L.algebra(K)
sage: T = KL.twisted_demazure_lusztig_operators(u, ~u, convention="dominant")
sage: T._test_relations()
sage: L0 = L.classical()
sage: KL0 = L0.algebra(K)
sage: alpha = L0.simple_roots()

sage: phi = L0.highest_root(); phi
(1, 0, -1)
sage: phi.to_simple_root(reduced_word=True)
(2, (1,))
sage: res = T.Ti_on_basis(L0([1,0,1]), 1); res
1/u*B[(0, 1, 1)]
sage: res = res * KL0.monomial(-alpha[2]); res
1/u*B[(0, 0, 2)]
sage: res = T.Tw_inverse(2)(res); res
((u^2-1)/u^2)*B[(0, 1, 1)] + B[(0, 2, 0)]
sage: res = T.Tw_inverse(1)(res); res
((u^2-1)/u)*B[(1, 1, 0)] + ((u^2-1)/u)*B[(1, 0, 1)] + u*B[(2, 0, 0)]
```

Todo

Choose a good set of Cartan Type to run on. Rank >4 is too big. But C_1 and B_1 are boring.

We now check systematically that those operators satisfy the relations of the Iwahori-Hecke algebra:

```
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: for cartan_type in CartanType.samples(affine=True, crystallographic=True): # long
....:     if cartan_type.rank() > 4: continue
....:     if cartan_type.type() == 'BC': continue
```

```
....:     KL = RootSystem(cartan_type).weight_lattice().algebra(K)
....:     T = KL.twisted_demazure_lusztig_operators(q1, q2, convention="dominant")
....:     T._test_relations()
```

Todo

Investigate why $T_0^\vee$ currently does not satisfy the quadratic relation in type BC. This should hopefuly be fixed when $T_0^\vee$ will have a more uniform implementation:

```
sage: cartan_type = CartanType(["BC",1,2])
sage: KL = RootSystem(cartan_type).weight_lattice().algebra(K)
sage: T = KL.twisted_demazure_lusztig_operators(q1,q2, convention="dominant")
sage: T._test_relations()
... tester.assert_(Ti(Ti(x,i,-q2),i,-q1).is_zero()) ...
AssertionError: False is not true
Traceback (most recent call last):
```

Comparison with T0:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: e = L.basis()
sage: K = QQ['t,q'].fraction_field()
sage: t,q = K.gens()
sage: q1 = t
sage: q2 = -1
sage: KL = L.algebra(K)
sage: L0 = L.classical()
sage: T = KL.demazure_lusztig_operators(q1,q2, convention="dominant")
sage: def T0(*l0): return KL.q_project(T[0].on_basis()(L.embed_at_level(L0(l0), 1)), q)
sage: T0_check_on_basis = KL.T0_check_on_basis(q1, q2, convention="dominant")
sage: def T0c(*l0): return T0_check_on_basis(L0(l0))

sage: T0(0,0,1)                                    # not double checked
((-t+1)/q)*B[(1, 0, 0)] + 1/q^2*B[(2, 0, -1)]
sage: T0c(0,0,1)
(t^2-t)*B[(1, 0, 0)] + (t^2-t)*B[(1, 1, -1)] + t^2*B[(2, 0, -1)] + (t-1)*B[(0, 0, 1)]
```

5.1.208 Root lattice realizations

class sage.combinat.root_system.root_lattice_realizations.**RootLatticeRealizations**(*base*, *name=None*)

Bases: sage.categories.category_types.Category_over_base_ring

The category of root lattice realizations over a given base ring

A *root lattice realization* L over a base ring R is a free module (or vector space if R is a field) endowed with an embedding of the root lattice of some root system.

Typical root lattice realizations over $\mathbf{Z}$ include the root lattice, weight lattice, and ambient lattice. Typical root lattice realizations over $\mathbf{Q}$ include the root space, weight space, and ambient space.

To describe the embedding, a root lattice realization must implement a method `simple_root()` returning for each i in the index set the image of the simple root α_i under the embedding.

A root lattice realization must further implement a method on elements `scalar()`, computing the scalar product with elements of the coroot lattice or coroot space.

Using those, this category provides tools for reflections, roots, the Weyl group and its action, ...

See also:

- RootSystem
- WeightLatticeRealizations
- RootSpace
- WeightSpace
- AmbientSpace

EXAMPLES:

Here, we consider the root system of type A_7, and embed the root lattice element $x = \alpha_2 + 2\alpha_6$ in several root lattice realizations:

```
sage: R = RootSystem(["A",7])
sage: alpha = R.root_lattice().simple_roots()
sage: x = alpha[2] + 2 * alpha[5]

sage: L = R.root_space()
sage: L(x)
alpha[2] + 2*alpha[5]

sage: L = R.weight_lattice()
sage: L(x)
-Lambda[1] + 2*Lambda[2] - Lambda[3] - 2*Lambda[4] + 4*Lambda[5] - 2*Lambda[6]

sage: L = R.ambient_space()
sage: L(x)
(0, 1, -1, 0, 2, -2, 0, 0)
```

We embed the root space element $x = \alpha_2 + 1/2\alpha_6$ in several root lattice realizations:

```
sage: alpha = R.root_space().simple_roots()
sage: x = alpha[2] + 1/2 * alpha[5]

sage: L = R.weight_space()
sage: L(x)
-Lambda[1] + 2*Lambda[2] - Lambda[3] - 1/2*Lambda[4] + Lambda[5] - 1/2*Lambda[6]

sage: L = R.ambient_space()
sage: L(x)
(0, 1, -1, 0, 1/2, -1/2, 0, 0)
```

Of course, one can't embed the root space in the weight lattice:

```
sage: L = R.weight_lattice()
sage: L(x)
Traceback (most recent call last):
...
TypeError: do not know how to make x (= alpha[2] + 1/2*alpha[5]) an element of self (=Weight lat
```

If K_1 is a subring of K_2, then one could in theory have an embedding from the root space over K_1 to any root lattice realization over K_2; this is not implemented:

```
sage: K1 = QQ
sage: K2 = QQ['q']
sage: L = R.weight_space(K2)

sage: alpha = R.root_space(K2).simple_roots()
sage: L(alpha[1])
2*Lambda[1] - Lambda[2]
```

```
sage: alpha = R.root_space(K1).simple_roots()
sage: L(alpha[1])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= alpha[1]) an element of self (=Weight space over the Uni
```

By a slight abuse, the embedding of the root lattice is not actually required to be faithful. Typically for an affine root system, the null root of the root lattice is killed in the non extended weight lattice:

```
sage: R = RootSystem(["A", 3, 1])
sage: delta = R.root_lattice().null_root()
sage: L = R.weight_lattice()
sage: L(delta)
0
```

TESTS:

```
sage: TestSuite(L).run()
```

Algebras
alias of Algebras

class ElementMethods

associated_coroot()
Returns the coroot associated to this root

EXAMPLES:

```
sage: alpha = RootSystem(["A", 3]).root_space().simple_roots()
sage: alpha[1].associated_coroot()
alphacheck[1]
```

associated_reflection()
Given a positive root `self`, returns a reduced word for the reflection orthogonal to `self`.

Since the answer is cached, it is a tuple instead of a list.

EXAMPLES:

```
sage: RootSystem(['C',3]).root_lattice().simple_root(3).weyl_action([1,2]).associated_re
(1, 2, 3, 2, 1)
sage: RootSystem(['C',3]).root_lattice().simple_root(2).associated_reflection()
(2,)
```

descents(*index_set=None*, *positive=False*)
Returns the descents of pt

EXAMPLES:

```
sage: space=RootSystem(['A',5]).weight_space()
sage: alpha=space.simple_roots()
sage: (alpha[1]+alpha[2]+alpha[4]).descents()
[3, 5]
```

extraspecial_pair()
Return the extraspecial pair of `self` under the ordering defined by `positive_roots_by_height()`.

The *extraspecial pair* of a positive root γ with some total ordering $<$ of the root lattice that respects height is the pair of positive roots (α, β) such that $\gamma = \alpha + \beta$ and α is as small as possible.

EXAMPLES:

```
sage: Q = RootSystem(['G', 2]).root_lattice()
sage: Q.highest_root().extraspecial_pair()
(alpha[2], 3*alpha[1] + alpha[2])
```

first_descent(*index_set=None*, *positive=False*)

Returns the first descent of pt

One can use the index_set option to restrict to the parabolic subgroup indexed by index_set.

EXAMPLES:

```
sage: space=RootSystem(['A',5]).weight_space()
sage: alpha=space.simple_roots()
sage: (alpha[1]+alpha[2]+alpha[4]).first_descent()
3
sage: (alpha[1]+alpha[2]+alpha[4]).first_descent([1,2,5])
5
sage: (alpha[1]+alpha[2]+alpha[4]).first_descent([1,2,5,3,4])
5
```

greater()

Returns the elements in the orbit of self which are greater than self in the weak order.

EXAMPLES:

```
sage: L = RootSystem(['A',3]).ambient_lattice()
sage: e = L.basis()
sage: e[2].greater()
[(0, 0, 1, 0), (0, 0, 0, 1)]
sage: len(L.rho().greater())
24
sage: len((-L.rho()).greater())
1
sage: sorted([len(x.greater()) for x in L.rho().orbit()])
[1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 6, 6, 6, 8, 8, 8, 8, 12, 12, 12, 24]
```

has_descent(*i*, *positive=False*)

Test if self has a descent at position i, that is if self is on the strict negative side of the i^{th} simple reflection hyperplane.

If positive if True, tests if it is on the strict positive side instead.

EXAMPLES:

```
sage: space=RootSystem(['A',5]).weight_space()
sage: alpha=RootSystem(['A',5]).weight_space().simple_roots()
sage: [alpha[i].has_descent(1) for i in space.index_set()]
[False, True, False, False, False]
sage: [(-alpha[i]).has_descent(1) for i in space.index_set()]
[True, False, False, False, False]
sage: [alpha[i].has_descent(1, True) for i in space.index_set()]
[True, False, False, False, False]
sage: [(-alpha[i]).has_descent(1, True) for i in space.index_set()]
[False, True, False, False, False]
sage: (alpha[1]+alpha[2]+alpha[4]).has_descent(3)
True
sage: (alpha[1]+alpha[2]+alpha[4]).has_descent(1)
False
sage: (alpha[1]+alpha[2]+alpha[4]).has_descent(1, True)
True
```

height()

Return the height of `self`.

The height of a root $\alpha = \sum_i a_i \alpha_i$ is defined to be $h(\alpha) := \sum_i a_i$.

EXAMPLES:

```
sage: Q = RootSystem(['G', 2]).root_lattice()
sage: Q.highest_root().height()
5
```

is_dominant (*index_set=None*, *positive=True*)

Returns whether self is dominant.

This is done with respect to the subrootsystem indicated by the subset of Dynkin nodes index_set. If index_set is None then the entire Dynkin node set is used. If positive is False then the dominance condition is replaced by antidominance.

EXAMPLES:

```
sage: L = RootSystem(['A',2]).ambient_lattice()
sage: Lambda = L.fundamental_weights()
sage: [x.is_dominant() for x in Lambda]
[True, True]
sage: [x.is_dominant(positive=False) for x in Lambda]
[False, False]
sage: (Lambda[1]-Lambda[2]).is_dominant()
False
sage: (-Lambda[1]+Lambda[2]).is_dominant()
False
sage: (Lambda[1]-Lambda[2]).is_dominant([1])
True
sage: (Lambda[1]-Lambda[2]).is_dominant([2])
False
sage: [x.is_dominant() for x in L.roots()]
[False, True, False, False, False, False]
sage: [x.is_dominant(positive=False) for x in L.roots()]
[False, False, False, False, True, False]
```

is_dominant_weight ()

Tests whether `self` is a dominant element of the weight lattice

EXAMPLES:

```
sage: L = RootSystem(['A',2]).ambient_lattice()
sage: Lambda = L.fundamental_weights()
sage: [x.is_dominant() for x in Lambda]
[True, True]
sage: (3*Lambda[1]+Lambda[2]).is_dominant()
True
sage: (Lambda[1]-Lambda[2]).is_dominant()
False
sage: (-Lambda[1]+Lambda[2]).is_dominant()
False
```

Tests that the scalar products with the coroots are all nonnegative integers. For example, if x is the sum of a dominant element of the weight lattice plus some other element orthogonal to all coroots, then the implementation correctly reports x to be a dominant weight:

```
sage: x = Lambda[1] + L([-1,-1,-1])
sage: x.is_dominant_weight()
True
```

is_imaginary_root ()

Return `True` if `self` is an imaginary root.

A root α is imaginary if it is not W conjugate to a simple root where W is the corresponding Weyl group.

EXAMPLES:

```
sage: Q = RootSystem(['B',2,1]).root_lattice()
sage: alpha = Q.simple_roots()
sage: alpha[0].is_imaginary_root()
False
sage: elt = alpha[0] + alpha[1] + 2*alpha[2]
sage: elt.is_imaginary_root()
True
```

is_long_root()

Return `True` if `self` is a long (real) root.

EXAMPLES:

```
sage: Q = RootSystem(['B',2,1]).root_lattice()
sage: alpha = Q.simple_roots()
sage: alpha[0].is_long_root()
True
sage: alpha[1].is_long_root()
True
sage: alpha[2].is_long_root()
False
```

is_parabolic_root(*index_set*)

Supposing that `self` is a root, is it in the parabolic subsystem with Dynkin nodes `index_set`?

INPUT:

- `index_set` – the Dynkin node set of the parabolic subsystem.

Todo

This implementation is only valid in the root or weight lattice

---

EXAMPLES:

```
sage: alpha = RootSystem(['A',3]).root_lattice().from_vector(vector([1,1,0]))
sage: alpha.is_parabolic_root([1,3])
False
sage: alpha.is_parabolic_root([1,2])
True
sage: alpha.is_parabolic_root([2])
False
```

is_real_root()

Return `True` if `self` is a real root.

A root α is real if it is W conjugate to a simple root where W is the corresponding Weyl group.

EXAMPLES:

```
sage: Q = RootSystem(['B',2,1]).root_lattice()
sage: alpha = Q.simple_roots()
sage: alpha[0].is_real_root()
True
sage: elt = alpha[0] + alpha[1] + 2*alpha[2]
sage: elt.is_real_root()
False
```

is_short_root()

Return `True` if `self` is a short (real) root.

Returns False unless the parent is an irreducible root system of finite type having two root lengths and `self` is of the shorter length. There is no check of whether `self` is actually a root.

EXAMPLES:

```
sage: Q = RootSystem(['C',2]).root_lattice()
sage: al = Q.simple_root(1).weyl_action([1,2]); al
alpha[1] + alpha[2]
sage: al.is_short_root()
True
sage: bt = Q.simple_root(2).weyl_action([2,1,2]); bt
-2*alpha[1] - alpha[2]
sage: bt.is_short_root()
False
sage: RootSystem(['A',2]).root_lattice().simple_root(1).is_short_root()
False
```

An example in affine type:

```
sage: Q = RootSystem(['B',2,1]).root_lattice()
sage: alpha = Q.simple_roots()
sage: alpha[0].is_short_root()
False
sage: alpha[1].is_short_root()
False
sage: alpha[2].is_short_root()
True
```

level()

EXAMPLES:

```
sage: L = RootSystem(['A',2,1]).weight_lattice()
sage: L.rho().level()
3
```

norm_squared()

Return the norm squared of `self` with respect to the symmetric form.

EXAMPLES:

```
sage: Q = RootSystem(['B',2,1]).root_lattice()
sage: alpha = Q.simple_roots()
sage: alpha[1].norm_squared()
4
sage: alpha[2].norm_squared()
2
sage: elt = alpha[0] - 3*alpha[1] + alpha[2]
sage: elt.norm_squared()
50
sage: elt = alpha[0] + alpha[1] + 2*alpha[2]
sage: elt.norm_squared()
0
sage: Q = RootSystem(CartanType(['A',4,2]).dual()).root_lattice()
sage: Qc = RootSystem(['A',4,2]).coroot_lattice()
sage: alpha = Q.simple_roots()
sage: alphac = Qc.simple_roots()
sage: elt = alpha[0] + 2*alpha[1] + 2*alpha[2]
sage: eltc = alphac[0] + 2*alphac[1] + 2*alphac[2]
sage: elt.norm_squared()
0
sage: eltc.norm_squared()
0
```

orbit()

The orbit of self under the action of the Weyl group

EXAMPLES:

ρ is a regular element whose orbit is in bijection with the Weyl group. In particular, it as 6 elements for the symmetric group S_3:

```
sage: L = RootSystem(["A", 2]).ambient_lattice()
sage: sorted(L.rho().orbit())                # the output order is not specified
[(1, 2, 0), (1, 0, 2), (2, 1, 0), (2, 0, 1), (0, 1, 2), (0, 2, 1)]

sage: L = RootSystem(["A", 3]).weight_lattice()
sage: len(L.rho().orbit())
24
sage: len(L.fundamental_weights()[1].orbit())
4
sage: len(L.fundamental_weights()[2].orbit())
6
```

pred(*index_set=None*)

Return the immediate predecessors of `self` for the weak order.

INPUT:

- `index_set` - a subset (as a list or iterable) of the nodes of the dynkin diagram; (default: `None` for all of them)

If `index_set` is specified, the successors for the corresponding parabolic subsystem are returned.

EXAMPLES:

```
sage: L = RootSystem(['A',3]).weight_lattice()
sage: Lambda = L.fundamental_weights()
sage: Lambda[1].pred()
[]
sage: L.rho().pred()
[]
sage: (-L.rho()).pred()
[Lambda[1] - 2*Lambda[2] - Lambda[3], -2*Lambda[1] + Lambda[2] - 2*Lambda[3], -Lambda[1]
sage: (-L.rho()).pred(index_set=[1])
[Lambda[1] - 2*Lambda[2] - Lambda[3]]
```

reduced_word(*index_set=None*, *positive=True*)

Returns a reduced word for the inverse of the shortest Weyl group element that sends the vector `self` into the dominant chamber.

With the `index_set` optional parameter, this is done with respect to the corresponding parabolic subgroup.

If `positive` is False, use the antidominant chamber instead.

EXAMPLES:

```
sage: space=RootSystem(['A',5]).weight_space()
sage: alpha=RootSystem(['A',5]).weight_space().simple_roots()
sage: alpha[1].reduced_word()
[2, 3, 4, 5]
sage: alpha[1].reduced_word([1,2])
[2]
```

reflection(*root*, *use_coroot=False*)

Reflects `self` across the hyperplane orthogonal to `root`.

If `use_coroot` is True, `root` is interpreted as a coroot.

EXAMPLES:

```
sage: R = RootSystem(['C',4])
sage: weight_lattice = R.weight_lattice()
sage: mu = weight_lattice.from_vector(vector([0,0,1,2]))
sage: coroot_lattice = R.coroot_lattice()
sage: alphavee = coroot_lattice.from_vector(vector([0,0,1,1]))
sage: mu.reflection(alphavee, use_coroot=True)
6*Lambda[2] - 5*Lambda[3] + 2*Lambda[4]
sage: root_lattice = R.root_lattice()
sage: beta = root_lattice.from_vector(vector([0,1,1,0]))
sage: mu.reflection(beta)
Lambda[1] - Lambda[2] + 3*Lambda[4]
```

scalar(*lambdacheck*)

Implement the natural pairing with the coroot lattice.

INPUT:

- `self` – an element of a root lattice realization
- `lambdacheck` – an element of the coroot lattice or coroot space

OUTPUT: the scalar product of `self` and `lambdacheck`

EXAMPLES:

```
sage: L = RootSystem(['A',4]).root_lattice()
sage: alpha      = L.simple_roots()
sage: alphacheck = L.simple_coroots()
sage: alpha[1].scalar(alphacheck[1])
2
sage: alpha[1].scalar(alphacheck[2])
-1
sage: matrix([ [ alpha[i].scalar(alphacheck[j])
...              for i in L.index_set() ]
...            for j in L.index_set() ])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]
```

TESTS:

```
sage: super(sage.combinat.root_system.root_space.RootSpaceElement,alpha[1]).scalar(alpha
Traceback (most recent call last):
...
NotImplementedError: <abstract method scalar at ...>
```

simple_reflection(*i*)

Returns the image of `self` by the i-th simple reflection.

EXAMPLES:

```
sage: alpha = RootSystem(["A", 3]).root_lattice().alpha()
sage: alpha[1].simple_reflection(2)
alpha[1] + alpha[2]

sage: Q = RootSystem(['A', 3, 1]).weight_lattice(extended = True)
sage: Lambda = Q.fundamental_weights()
sage: L = Lambda[0] + Q.null_root()
sage: L.simple_reflection(0)
-Lambda[0] + Lambda[1] + Lambda[3]
```

simple_reflections()

The images of self by all the simple reflections

EXAMPLES:

```
sage: alpha = RootSystem(["A", 3]).root_lattice().alpha()
sage: alpha[1].simple_reflections()
[-alpha[1], alpha[1] + alpha[2], alpha[1]]
```

smaller()

Returns the elements in the orbit of self which are smaller than self in the weak order.

EXAMPLES:

```
sage: L = RootSystem(['A',3]).ambient_lattice()
sage: e = L.basis()
sage: e[2].smaller()
[(0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0)]
sage: len(L.rho().smaller())
1
sage: len((-L.rho()).smaller())
24
sage: sorted([len(x.smaller()) for x in L.rho().orbit()])
[1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 6, 6, 6, 8, 8, 8, 8, 12, 12, 12, 24]
```

succ(*index_set=None*)

Return the immediate successors of `self` for the weak order.

INPUT:

- `index_set` - a subset (as a list or iterable) of the nodes of the dynkin diagram; (default: `None` for all of them)

If `index_set` is specified, the successors for the corresponding parabolic subsystem are returned.

EXAMPLES:

```
sage: L = RootSystem(['A',3]).weight_lattice()
sage: Lambda = L.fundamental_weights()
sage: Lambda[1].succ()
[-Lambda[1] + Lambda[2]]
sage: L.rho().succ()
[-Lambda[1] + 2*Lambda[2] + Lambda[3], 2*Lambda[1] - Lambda[2] + 2*Lambda[3], Lambda[1] +
sage: (-L.rho()).succ()
[]
sage: L.rho().succ(index_set=[1])
[-Lambda[1] + 2*Lambda[2] + Lambda[3]]
sage: L.rho().succ(index_set=[2])
[2*Lambda[1] - Lambda[2] + 2*Lambda[3]]
```

symmetric_form(*alpha*)

Return the symmetric form of `self` with `alpha`.

Consider the simple roots α_i and let $(b_{ij})_{ij}$ denote the symmetrized Cartan matrix $(a_{ij})_{ij}$, we have

$$(\alpha_i|\alpha_j) = b_{ij}$$

and extended bilinearly. See Chapter 6 in Kac, Infinite Dimensional Lie Algebras for more details.

EXAMPLES:

```
sage: Q = RootSystem(['B',2,1]).root_lattice()
sage: alpha = Q.simple_roots()
sage: alpha[1].symmetric_form(alpha[0])
0
sage: alpha[1].symmetric_form(alpha[1])
4
sage: elt = alpha[0] - 3*alpha[1] + alpha[2]
sage: elt.symmetric_form(alpha[1])
```

```
-14
sage: elt.symmetric_form(alpha[0]+2*alpha[2])
14
sage: Q = RootSystem(CartanType(['A',4,2]).dual()).root_lattice()
sage: Qc = RootSystem(['A',4,2]).coroot_lattice()
sage: alpha = Q.simple_roots()
sage: alphac = Qc.simple_roots()
sage: elt = alpha[0] + 2*alpha[1] + 2*alpha[2]
sage: eltc = alphac[0] + 2*alphac[1] + 2*alphac[2]
sage: elt.symmetric_form(alpha[1])
0
sage: eltc.symmetric_form(alphac[1])
0
```

to_ambient()

Map `self` to the ambient space.

EXAMPLES:

```
sage: alpha = CartanType(['B',4]).root_system().root_lattice().an_element(); alpha
2*alpha[1] + 2*alpha[2] + 3*alpha[3]
sage: alpha.to_ambient()
(2, 0, 1, -3)
sage: mu = CartanType(['B',4]).root_system().weight_lattice().an_element(); mu
2*Lambda[1] + 2*Lambda[2] + 3*Lambda[3]
sage: mu.to_ambient()
(7, 5, 3, 0)
sage: v = CartanType(['B',4]).root_system().ambient_space().an_element(); v
(2, 2, 3, 0)
sage: v.to_ambient()
(2, 2, 3, 0)
sage: alphavee = CartanType(['B',4]).root_system().coroot_lattice().an_element(); alphav
2*alphacheck[1] + 2*alphacheck[2] + 3*alphacheck[3]
sage: alphavee.to_ambient()
(2, 0, 1, -3)
```

to_classical()

Map `self` to the classical lattice/space.

Only makes sense for affine type.

EXAMPLES:

```
sage: R = CartanType(['A',3,1]).root_system()
sage: alpha = R.root_lattice().an_element(); alpha
2*alpha[0] + 2*alpha[1] + 3*alpha[2]
sage: alb = alpha.to_classical(); alb
alpha[2] - 2*alpha[3]
sage: alb.parent()
Root lattice of the Root system of type ['A', 3]
sage: v = R.ambient_space().an_element(); v
2*e[0] + 2*e[1] + 3*e[2]
sage: v.to_classical()
(2, 2, 3, 0)
```

to_dominant_chamber(*index_set=None*, *positive=True*, *reduced_word=False*)

Returns the unique dominant element in the Weyl group orbit of the vector `self`.

If `positive` is False, returns the antidominant orbit element.

With the `index_set` optional parameter, this is done with respect to the corresponding parabolic subgroup.

If `reduced_word` is True, returns the 2-tuple (`weight`, `direction`) where `weight` is the (anti)dominant orbit element and `direction` is a reduced word for the Weyl group element sending `weight` to `self`.

Warning: In infinite type, an orbit may not contain a dominant element. In this case the function may go into an infinite loop.
For affine root systems, errors are generated if the orbit does not contain the requested kind of representative. If the input vector is of positive (resp. negative) level, then there is a dominant (resp. antidominant) element in its orbit but not an antidominant (resp. dominant) one. If the vector is of level zero, then there are neither dominant nor antidominant orbit representatives, except for multiples of the null root, which are themselves both dominant and antidominant orbit representatives.

EXAMPLES:

```
sage: space=RootSystem(['A',5]).weight_space()
sage: alpha=RootSystem(['A',5]).weight_space().simple_roots()
sage: alpha[1].to_dominant_chamber()
Lambda[1] + Lambda[5]
sage: alpha[1].to_dominant_chamber([1,2])
Lambda[1] + Lambda[2] - Lambda[3]
sage: wl=RootSystem(['A',2,1]).weight_lattice(extended=True)
sage: mu=wl.from_vector(vector([1,-3,0]))
sage: mu.to_dominant_chamber(positive=False, reduced_word = True)
(-Lambda[1] - Lambda[2] - delta, [0, 2])

sage: R = RootSystem(['A',1,1])
sage: rl = R.root_lattice()
sage: nu = rl.zero()
sage: nu.to_dominant_chamber()
0
sage: nu.to_dominant_chamber(positive=False)
0
sage: mu = rl.from_vector(vector([0,1]))
sage: mu.to_dominant_chamber()
Traceback (most recent call last):
...
ValueError: alpha[1] is not in the orbit of the fundamental chamber
sage: mu.to_dominant_chamber(positive=False)
Traceback (most recent call last):
...
ValueError: alpha[1] is not in the orbit of the negative of the fundamental chamber
```

to_dual_type_cospace()
Map `self` to the dual type cospace.

For example, if `self` is in the root lattice of type $['B', 2]$, send it to the coroot lattice of type $['C', 2]$.

EXAMPLES:

```
sage: v = CartanType(['C',3]).root_system().weight_lattice().an_element(); v
2*Lambda[1] + 2*Lambda[2] + 3*Lambda[3]
sage: w = v.to_dual_type_cospace(); w
2*Lambdacheck[1] + 2*Lambdacheck[2] + 3*Lambdacheck[3]
sage: w.parent()
Coweight lattice of the Root system of type ['B', 3]
```

to_simple_root(*reduced_word=False*)
Return (the index of) a simple root in the orbit of the positive root `self`.

INPUT:

- self – a positive root
- reduced_word – a boolean (default: False)

OUTPUT:

- The index i of a simple root α_i. If reduced_word is True, this returns instead a pair (i, word), where word is a sequence of reflections mapping α_i up the root poset to self.

EXAMPLES:

```
sage: L = RootSystem(["A",3]).root_lattice()
sage: positive_roots = L.positive_roots()
sage: for alpha in positive_roots:
...       print alpha, alpha.to_simple_root()
alpha[1] 1
alpha[2] 2
alpha[3] 3
alpha[1] + alpha[2] 2
alpha[2] + alpha[3] 3
alpha[1] + alpha[2] + alpha[3] 3
sage: for alpha in positive_roots:
...       print alpha, alpha.to_simple_root(reduced_word=True)
alpha[1] (1, ())
alpha[2] (2, ())
alpha[3] (3, ())
alpha[1] + alpha[2] (2, (1,))
alpha[2] + alpha[3] (3, (2,))
alpha[1] + alpha[2] + alpha[3] (3, (1, 2))
```

ALGORITHM:

This method walks from self down to the antidominant chamber by applying successively the simple reflection given by the first descent. Since self is a positive root, each step goes down the root poset, and one must eventually cross a simple root α_i.

See also:

- first_descent()
- to_dominant_chamber()

Warning: The behavior is not specified if the input is not a positive root. For a finite root system, this is currently caught (albeit with a not perfect message):

```
sage: alpha = L.simple_roots()
sage: (2*alpha[1]).to_simple_root()
Traceback (most recent call last):
...
ValueError: -2*alpha[1] - 2*alpha[2] - 2*alpha[3] is not a positive root
```

For an infinite root systems, this method may run into an infinite reccursion if the input is not a positive root.

translation(x)

INPUT:

- self - an element t at level 0
- x - an element of the same space

Returns x translated by t, that is $x + level(x)t$

EXAMPLES:

```
sage: L = RootSystem(['A',2,1]).weight_lattice()
sage: alpha = L.simple_roots()
sage: Lambda = L.fundamental_weights()
sage: t = alpha[2]
```

Let us look at the translation of an element of level 1:

```
sage: Lambda[1].level()
1
sage: t.translation(Lambda[1])
-Lambda[0] + 2*Lambda[2]
sage: Lambda[1] + t
-Lambda[0] + 2*Lambda[2]
```

and of an element of level 0:

```
sage: alpha [1].level()
0
sage: t.translation(alpha [1])
-Lambda[0] + 2*Lambda[1] - Lambda[2]
sage: alpha[1] + 0*t
-Lambda[0] + 2*Lambda[1] - Lambda[2]
```

The arguments are given in this seemingly unnatural order to make it easy to construct the translation function:

```
sage: f = t.translation
sage: f(Lambda[1])
-Lambda[0] + 2*Lambda[2]
```

weyl_action(*element*, *inverse=False*)

Acts on `self` by an element of the Coxeter or Weyl group.

INPUT:

- `element` – an element of a Coxeter or Weyl group of the same Cartan type, or a tuple or a list (such as a reduced word) of elements from the index set.
- `inverse` – a boolean (default: False); whether to act by the inverse element.

EXAMPLES:

```
sage: wl = RootSystem(['A',3]).weight_lattice()
sage: mu = wl.from_vector(vector([1,0,-2]))
sage: mu
Lambda[1] - 2*Lambda[3]
sage: mudom, rw = mu.to_dominant_chamber(positive=False, reduced_word = True)
sage: mudom, rw
(-Lambda[2] - Lambda[3], [1, 2])
```

Acting by a (reduced) word:

```
sage: mudom.weyl_action(rw)
Lambda[1] - 2*Lambda[3]
sage: mu.weyl_action(rw, inverse = True)
-Lambda[2] - Lambda[3]
```

Acting by an element of the Coxeter or Weyl group on a vector in its own lattice of definition (implemented by matrix multiplication on a vector):

```
sage: w = wl.weyl_group().from_reduced_word([1, 2])
sage: mudom.weyl_action(w)
Lambda[1] - 2*Lambda[3]
```

Acting by an element of an isomorphic Coxeter or Weyl group (implemented by the action of a corresponding reduced word):

```
sage: W = WeylGroup(['A',3], prefix="s")
sage: w = W.from_reduced_word([1, 2])
sage: wl.weyl_group() == W
False
sage: mudom.weyl_action(w)
Lambda[1] - 2*Lambda[3]
```

weyl_stabilizer(*index_set=None*)

Returns the subset of Dynkin nodes whose reflections fix `self`.

If `index_set` is not None, only consider nodes in this set. Note that if `self` is dominant or antidominant, then its stabilizer is the parabolic subgroup defined by the returned node set.

EXAMPLES:

```
sage: wl = RootSystem(['A',2,1]).weight_lattice(extended = True)
sage: al = wl.null_root()
sage: al.weyl_stabilizer()
[0, 1, 2]
sage: wl = RootSystem(['A',4]).weight_lattice()
sage: mu = wl.from_vector(vector([1,1,0,0]))
sage: mu.weyl_stabilizer()
[3, 4]
sage: mu.weyl_stabilizer(index_set = [1,2,3])
[3]
```

class RootLatticeRealizations.**ParentMethods**

a_long_simple_root()

Returns a long simple root, corresponding to the highest outgoing edge in the Dynkin diagram.

Caveat: this may be break in affine type $A_{2n}^{(2)}$

Caveat: meaningful/broken for non irreducible?

TODO: implement CartanType.nodes_by_length as in MuPAD-Combinat (using CartanType.symmetrizer), and use it here.

TESTS:

```
sage: X=RootSystem(['A',1]).weight_space()
sage: X.a_long_simple_root()
2*Lambda[1]
sage: X=RootSystem(['A',5]).weight_space()
sage: X.a_long_simple_root()
2*Lambda[1] - Lambda[2]
```

almost_positive_roots()

Returns the almost positive roots of `self`

These are the positive roots together with the simple negative roots.

See also:

`almost_positive_root_decomposition()`, `tau_plus_minus()`

EXAMPLES:

```
sage: L = RootSystem(['A',2]).root_lattice()
sage: L.almost_positive_roots()
[-alpha[1], alpha[1], alpha[1] + alpha[2], -alpha[2], alpha[2]]
```

almost_positive_roots_decomposition()
Returns the decomposition of the almost positive roots of `self`

This is the list of the orbits of the almost positive roots under the action of the dihedral group generated by the operators τ_+ and τ_-.

See also:

- almost_positive_roots()
- tau_plus_minus()

EXAMPLES:

```
sage: RootSystem(['A',2]).root_lattice().almost_positive_roots_decomposition()
[[-alpha[1], alpha[1], alpha[1] + alpha[2], alpha[2], -alpha[2]]]

sage: RootSystem(['B',2]).root_lattice().almost_positive_roots_decomposition()
[[-alpha[1], alpha[1], alpha[1] + 2*alpha[2]], [-alpha[2], alpha[2], alpha[1] + alpha[2]

sage: RootSystem(['D',4]).root_lattice().almost_positive_roots_decomposition()
[[-alpha[1], alpha[1], alpha[1] + alpha[2], alpha[2] + alpha[3] + alpha[4]],
 [-alpha[2], alpha[2], alpha[1] + alpha[2] + alpha[3] + alpha[4], alpha[1] + 2*alpha[2]
 [-alpha[3], alpha[3], alpha[2] + alpha[3], alpha[1] + alpha[2] + alpha[4]],
 [-alpha[4], alpha[4], alpha[2] + alpha[4], alpha[1] + alpha[2] + alpha[3]]]
```

alpha()
Returns the family $(\alpha_i)_{i\in I}$ of the simple roots, with the extra feature that, for simple irreducible root systems, α_0 yields the opposite of the highest root.

EXAMPLES:

```
sage: alpha = RootSystem(["A",2]).root_lattice().alpha()
sage: alpha[1]
alpha[1]
sage: alpha[0]
-alpha[1] - alpha[2]
```

alphacheck()
Returns the family $(\alpha_i^\vee)_{i\in I}$ of the simple coroots, with the extra feature that, for simple irreducible root systems, $\alpha_0^\vee$ yields the coroot associated to the opposite of the highest root (caveat: for non simply laced root systems, this is not the opposite of the highest coroot!)

EXAMPLES:

```
sage: alphacheck = RootSystem(["A",2]).ambient_space().alphacheck()
sage: alphacheck
Finite family {1: (1, -1, 0), 2: (0, 1, -1)}
```

Here is now $\alpha_0^\vee$:
(-1, 0, 1)

Todo

add a non simply laced example

Finaly, here is an affine example:

```
sage: RootSystem(["A",2,1]).weight_space().alphacheck()
Finite family {0: alphacheck[0], 1: alphacheck[1], 2: alphacheck[2]}

sage: RootSystem(["A",3]).ambient_space().alphacheck()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1)}
```

basic_imaginary_roots()
Return the basic imaginary roots of `self`.

The basic imaginary roots δ are the set of imaginary roots in $-C^\vee$ where C is the dominant chamger (i.e., $\langle \beta, \alpha_i^\vee \rangle \leq 0$ for all $i \in I$). All imaginary roots are W-conjugate to a simple imaginary root.

EXAMPLES:

```
sage: RootSystem(['A', 2]).root_lattice().basic_imaginary_roots()
()
sage: Q = RootSystem(['A', 2, 1]).root_lattice()
sage: Q.basic_imaginary_roots()
(alpha[0] + alpha[1] + alpha[2],)
sage: delta = Q.basic_imaginary_roots()[0]
sage: all(delta.scalar(Q.simple_coroot(i)) <= 0 for i in Q.index_set())
True
```

cartan_type()
EXAMPLES:

```
sage: r = RootSystem(['A',4]).root_space()
sage: r.cartan_type()
['A', 4]
```

classical()
Return the corresponding root/weight/ambient lattice/space.

EXAMPLES:

```
sage: RootSystem(["A",4,1]).root_lattice().classical()
Root lattice of the Root system of type ['A', 4]
sage: RootSystem(["A",4,1]).weight_lattice().classical()
Weight lattice of the Root system of type ['A', 4]
sage: RootSystem(["A",4,1]).ambient_space().classical()
Ambient space of the Root system of type ['A', 4]
```

cohighest_root()
Returns the associated coroot of the highest root.

Note: this is usually not the highest coroot.

EXAMPLES:

```
sage: RootSystem(['A', 3]).ambient_space().cohighest_root()
(1, 0, 0, -1)
```

coroot_lattice()
Returns the coroot lattice.

EXAMPLES:

```
sage: RootSystem(['A',2]).root_lattice().coroot_lattice()
Coroot lattice of the Root system of type ['A', 2]
```

coroot_space(*base_ring=Rational Field*)
Returns the coroot space over `base_ring`

INPUT:

- `base_ring` – a ring (default: **Q**)

EXAMPLES:

```
sage: RootSystem(['A',2]).root_lattice().coroot_space()
Coroot space over the Rational Field of the Root system of type ['A', 2]
```

```
sage: RootSystem(['A',2]).root_lattice().coroot_space(QQ['q'])
Coroot space over the Univariate Polynomial Ring in q over Rational Field of the Root sy
```

dual_type_cospace()

Returns the cospace of dual type.

For example, if invoked on the root lattice of type $['B', 2]$, returns the coroot lattice of type $['C', 2]$.

..warning:
Not implemented for ambient spaces.

EXAMPLES:

```
sage: CartanType(['B',2]).root_system().root_lattice().dual_type_cospace()
Coroot lattice of the Root system of type ['C', 2]
sage: CartanType(['F',4]).root_system().coweight_lattice().dual_type_cospace()
Weight lattice of the Root system of type ['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1
```

dynkin_diagram()

EXAMPLES:

```
sage: r = RootSystem(['A',4]).root_space()
sage: r.dynkin_diagram()
O---O---O---O
1   2   3   4
A4
```

fundamental_weights_from_simple_roots()

Return the fundamental weights.

This is computed from the simple roots by using the inverse of the Cartan matrix. This method is therefore only valid for finite types and if this realization of the root lattice is large enough to contain them.

EXAMPLES:

In the root space, we retrieve the inverse of the Cartan matrix:

```
sage: L = RootSystem(["B",3]).root_space()
sage: L.fundamental_weights_from_simple_roots()
Finite family {1:     alpha[1] +   alpha[2] +     alpha[3],
               2:     alpha[1] + 2*alpha[2] +   2*alpha[3],
               3: 1/2*alpha[1] +   alpha[2] + 3/2*alpha[3]}
sage: ~L.cartan_type().cartan_matrix()
[  1   1 1/2]
[  1   2   1]
[  1   2 3/2]
```

In the weight lattice and the ambient space, we retrieve the fundamental weights:

```
sage: L = RootSystem(["B",3]).weight_lattice()
sage: L.fundamental_weights_from_simple_roots()
Finite family {1: Lambda[1], 2: Lambda[2], 3: Lambda[3]}

sage: L = RootSystem(["B",3]).ambient_space()
sage: L.fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
sage: L.fundamental_weights_from_simple_roots()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
```

However the fundamental weights do not belong to the root lattice:

```
sage: L = RootSystem(["B",3]).root_lattice()
sage: L.fundamental_weights_from_simple_roots()
```

```
Traceback (most recent call last):
...
ValueError: The fundamental weights do not live in this realization of the root lattice
```

Beware of the usual GL_n vs SL_n catch in type A:

```
sage: L = RootSystem(["A",3]).ambient_space()
sage: L.fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1, 1, 1, 0)}
sage: L.fundamental_weights_from_simple_roots()
Finite family {1: (3/4, -1/4, -1/4, -1/4), 2: (1/2, 1/2, -1/2, -1/2), 3: (1/4, 1/4, 1/4,

sage: L = RootSystem(["A",3]).ambient_lattice()
sage: L.fundamental_weights_from_simple_roots()
Traceback (most recent call last):
...
ValueError: The fundamental weights do not live in this realization of the root lattice
```

generalized_nonnesting_partition_lattice(*m*, *facade=False*)

Return the lattice of m-nonnesting partitions

This has been defined by Athanasiadis, see chapter 5 of [Arm06].

INPUT:

- m – integer

See also:

`nonnesting_partition_lattice()`

EXAMPLES:

```
sage: R = RootSystem(['A', 2])
sage: RS = R.root_lattice()
sage: P = RS.generalized_nonnesting_partition_lattice(2); P
Finite lattice containing 12 elements
sage: P.coxeter_transformation()**20 == 1
True
```

highest_root()

Returns the highest root (for an irreducible finite root system)

EXAMPLES:

```
sage: RootSystem(['A',4]).ambient_space().highest_root()
(1, 0, 0, 0, -1)

sage: RootSystem(['E',6]).weight_space().highest_root()
Lambda[2]
```

index_set()

EXAMPLES:

```
sage: r = RootSystem(['A',4]).root_space()
sage: r.index_set()
(1, 2, 3, 4)
```

long_roots()

Return a list of the long roots of `self`.

EXAMPLES:

```
sage: L = RootSystem(['B',3]).root_lattice()
sage: sorted(L.long_roots())
[-alpha[1], -alpha[1] - 2*alpha[2] - 2*alpha[3],
```

```
 -alpha[1] - alpha[2], -alpha[1] - alpha[2] - 2*alpha[3],
 alpha[1], alpha[1] + alpha[2],
 alpha[1] + alpha[2] + 2*alpha[3],
 alpha[1] + 2*alpha[2] + 2*alpha[3], -alpha[2],
 -alpha[2] - 2*alpha[3], alpha[2], alpha[2] + 2*alpha[3]]
```

negative_roots()

Returns the negative roots of self.

EXAMPLES:

```
sage: L = RootSystem(['A', 2]).weight_lattice()
sage: sorted(L.negative_roots())
[-2*Lambda[1] + Lambda[2], -Lambda[1] - Lambda[2], Lambda[1] - 2*Lambda[2]]
```

Algorithm: negate the positive roots

nonnesting_partition_lattice(*facade=False*)

Return the lattice of nonnesting partitions

This is the lattice of order ideals of the root poset.

This has been defined by Postnikov, see Remark 2 in [Reiner97].

See also:

`generalized_nonnesting_partition_lattice()`, `root_poset()`

EXAMPLES:

```
sage: R = RootSystem(['A', 3])
sage: RS = R.root_lattice()
sage: P = RS.nonnesting_partition_lattice(); P
Finite lattice containing 14 elements
sage: P.coxeter_transformation()**10 == 1
True

sage: R = RootSystem(['B', 3])
sage: RS = R.root_lattice()
sage: P = RS.nonnesting_partition_lattice(); P
Finite lattice containing 20 elements
sage: P.coxeter_transformation()**7 == 1
True
```

REFERENCES:

nonparabolic_positive_root_sum(*index_set=None*)

Return the sum of positive roots not in a parabolic subsystem.

The conventions for `index_set` are as in `nonparabolic_positive_roots()`.

EXAMPLES:

```
sage: Q = RootSystem(['A',3]).root_lattice()
sage: Q.nonparabolic_positive_root_sum((1,2))
alpha[1] + 2*alpha[2] + 3*alpha[3]
sage: Q.nonparabolic_positive_root_sum()
0
sage: Q.nonparabolic_positive_root_sum(())
3*alpha[1] + 4*alpha[2] + 3*alpha[3]
```

nonparabolic_positive_roots(*index_set=None*)

Return the positive roots of `self` that are not in the parabolic subsystem indicated by `index_set`.

If index_set is None, as in positive_roots() it is assumed to be the entire Dynkin node set. Then the parabolic subsystem consists of all positive roots and the empty list is returned.

EXAMPLES:

```
sage: L = RootSystem(['A',3]).root_lattice()
sage: L.nonparabolic_positive_roots()
[]
sage: sorted(L.nonparabolic_positive_roots((1,2)))
[alpha[1] + alpha[2] + alpha[3], alpha[2] + alpha[3], alpha[3]]
sage: sorted(L.nonparabolic_positive_roots(()))
[alpha[1], alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3], alpha[2], alpha[2] + alpl
```

null_coroot()

Returns the null coroot of self.

The null coroot is the smallest non trivial positive coroot which is orthogonal to all simple roots. It exists for any affine root system.

EXAMPLES:

```
sage: RootSystem(['C',2,1]).root_lattice().null_coroot()
alphacheck[0] + alphacheck[1] + alphacheck[2]
sage: RootSystem(['D',4,1]).root_lattice().null_coroot()
alphacheck[0] + alphacheck[1] + 2*alphacheck[2] + alphacheck[3] + alphacheck[4]
sage: RootSystem(['F',4,1]).root_lattice().null_coroot()
alphacheck[0] + 2*alphacheck[1] + 3*alphacheck[2] + 2*alphacheck[3] + alphacheck[4]
```

null_root()

Returns the null root of self. The null root is the smallest non trivial positive root which is orthogonal to all simple coroots. It exists for any affine root system.

EXAMPLES:

```
sage: RootSystem(['C',2,1]).root_lattice().null_root()
alpha[0] + 2*alpha[1] + alpha[2]
sage: RootSystem(['D',4,1]).root_lattice().null_root()
alpha[0] + alpha[1] + 2*alpha[2] + alpha[3] + alpha[4]
sage: RootSystem(['F',4,1]).root_lattice().null_root()
alpha[0] + 2*alpha[1] + 3*alpha[2] + 4*alpha[3] + 2*alpha[4]
```

plot(*roots='simple'*, *coroots=False*, *reflection_hyperplanes='simple'*, *fundamental_weights=None*, *fundamental_chamber=None*, *alcoves=None*, *alcove_labels=False*, *alcove_walk=None*, ***options*)

Return a picture of this root lattice realization.

INPUT:

- roots – which roots to display, if any. Can be one of the following:
  - "simple" – The simple roots (the default)
  - "classical" – Not yet implemented
  - "all" – Only works in the finite case
  - A list or tuple of roots
  - False
- coroots – which coroots to display, if any. Can be one of the following:
  - "simple" – The simple coroots (the default)
  - "classical" – Not yet implemented
  - "all" – Only works in the finite case
  - A list or tuple of coroots
  - False
- fundamental_weights – a boolean or None (default: None) whether to display the fundamental weights. If None, the fundamental weights are drawn if available.

- `reflection_hyperplanes` – which reflection hyperplanes to display, if any. Can be one of the following:
  - `"simple"` – The simple roots
  - `"classical"` – Not yet implemented
  - `"all"` – Only works in the finite case
  - A list or tuple of roots
  - `False` (the default)
- `fundamental_chamber` – whether and how to draw the fundamental chamber. Can be one of the following:
  - A boolean – Set to `True` to draw the fundamental chamber
  - `"classical"` – Draw the classical fundamental chamber
  - `None` – (the default) The fundamental chamber is drawn except in the root lattice where this is not yet implemented. For affine types the classical fundamental chamber is drawn instead.
- `alcoves` – one of the following (default: `True`):
  - A boolean – Whether to display the alcoves
  - A list of alcoves – The alcoves to be drawn. Each alcove is specified by the coordinates of its center in the root lattice (affine type only). Otherwise the alcoves that intersect the bounding box are drawn.
- `alcove_labels` – one of the following (default: `False`):
  - A boolean – Whether to display the elements of the Weyl group indexing the alcoves. This currently requires to also set the `alcoves` option.
  - A number l – The label is drawn at level l (affine type only), which only makes sense if `affine` is `False`.
- `bounding_box` – a rational number or a list of pairs thereof (default: 3)

  Specifies a bounding box, in the coordinate system for this plot, in which to plot alcoves and other infinite objects. If the bounding box is a number a, then the bounding box is of the form $[-a, a]$ in all directions. Beware that there can be some border effects and the returned graphic is not necessarily strictly contained in the bounding box.
- `alcove_walk` – an alcove walk or `None` (default: `None`)

  The alcove walk is described by a list (or iterable) of vertices of the Dynkin diagram which specifies which wall is crossed at each step, starting from the fundamental alcove.
- `projection` – one of the following (default: `True`):
  - `True` – The default projection for the root lattice realization is used.
  - `False` – No projection is used.
  - `barycentric` – A barycentric projection is used.
  - A function – If a function is specified, it should implement a linear (or affine) map taking as input an element of this root lattice realization and returning its desired coordinates in the plot, as a vector with rational coordinates.
- `color` – a function mapping vertices of the Dynkin diagram to colors (default: `"black"` for 0, `"blue"` for 1, `"red"` for 2, `"green"` for 3)

  This is used to set the color for the simple roots, fundamental weights, reflection hyperplanes, alcove facets, etc. If the color is `None`, the object is not drawn.
- `labels` – a boolean (default: `True`) whether to display labels on the simple roots, fundamental weights, etc.

EXAMPLES:

```
sage: L = RootSystem(["A",2,1]).ambient_space().plot()     # long time
```

See also:

- `plot_parse_options()`
- `plot_roots()`, `plot_coroots()`
- `plot_fundamental_weights()`
- `plot_fundamental_chamber()`

- plot_reflection_hyperplanes()
- plot_alcoves()
- plot_alcove_walk()
- plot_ls_paths()
- plot_crystal()

plot_alcove_walk(*word*, *start=None*, *foldings=None*, *color='orange'*, ***options*)

Plot an alcove walk.

INPUT:

- word – a list of elements of the index set
- foldings – a list of booleans or None (default: None)
- start – an element of this space (default: None for ρ)
- **options – plotting options

See also:

- plot() for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

An alcove walk of type $A_2^{(1)}$:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: w1 = [0,2,1,2,0,2,1,0,2,1,2,1,2,0,2,0,1,2,0]
sage: p = L.plot_alcoves(bounding_box=5)           # long time (5s)
sage: p += L.plot_alcove_walk(w1)                  # long time
sage: p                                            # long time
Graphics object consisting of 375 graphics primitives
```

The same plot with another alcove walk:

```
sage: w2 = [2,1,2,0,2,0,2,1,2,0,1,2,1,2,1,0,1,2,0,2,0,1,2,0,2]
sage: p += L.plot_alcove_walk(w2, color="orange")  # long time
```

And another with some foldings:

```
sage: L.plot_alcoves(bounding_box=3) + \
....:   L.plot_alcove_walk([0,1,2,0,2,0,1,2,0,1],
....:                      foldings = [False, False, True, False, False, False, True, Fa
....:                      color="green")            # long time (3s)
Graphics object consisting of 155 graphics primitives
```

TESTS:

```
sage: L = RootSystem(["A",2,1]).weight_space()
sage: p = L.plot_alcove_walk([0,1,2,0,2,0,1,2,0,1],
...                          foldings = [False, False, True, False, False, False, True,
...                          color="green",
...                          start=L.rho())
sage: print p.description()
Line defined by 2 points: [(-1.0, 8.0), (-1.5, 9.0)]
Line defined by 2 points: [(1.0, 4.0), (1.5, 4.5)]
Line defined by 2 points: [(1.0, 7.0), (1.5, 6.0)]
Arrow from (-1.0,5.0) to (-2.0,7.0)
Arrow from (-1.0,8.0) to (1.0,7.0)
Arrow from (-1.5,9.0) to (-1.0,8.0)
Arrow from (-2.0,7.0) to (-1.0,8.0)
Arrow from (1.0,1.0) to (2.0,2.0)
Arrow from (1.0,4.0) to (-1.0,5.0)
Arrow from (1.0,7.0) to (2.0,8.0)
Arrow from (1.5,4.5) to (1.0,4.0)
```

```
Arrow from (1.5,6.0) to (1.0,7.0)
Arrow from (2.0,2.0) to (1.0,4.0)
```

plot_alcoves(*alcoves=True*, *alcove_labels=False*, *wireframe=False*, ***options*)

Plot the alcoves and optionaly their labels.

INPUT:

- `alcoves` – a list of alcoves or `True` (default: `True`)
- `alcove_labels` – a boolean or a number specifying at which level to put the label (default: `False`)
- `**options` – Plotting options

See also:

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting, and in particular how the alcoves can be specified.

EXAMPLES:

2D plots:

```
sage: RootSystem(["B",2,1]).ambient_space().plot_alcoves()                    # long t
Graphics object consisting of 228 graphics primitives
```

3D plots:

```
sage: RootSystem(["A",2,1]).weight_space() .plot_alcoves(affine=False)        # long t
Graphics3d Object
sage: RootSystem(["G",2,1]).ambient_space().plot_alcoves(affine=False, level=1) # long t
Graphics3d Object
```

Here we plot a single alcove:

```
sage: L = RootSystem(["A",3,1]).ambient_space()
sage: W = L.weyl_group()
sage: L.plot(alcoves=[W.one()], reflection_hyperplanes=False, bounding_box=2)
Graphics3d Object
```

TESTS:

```
sage: L = RootSystem(["A",2,1]).weight_space()
sage: p = L.plot_alcoves(alcoves=[[0,0]])
sage: print p.description()
Line defined by 2 points: [(-1.0, 0.0), (0.0, -1.0)]
Line defined by 2 points: [(-1.0, 1.0), (-1.0, 0.0)]
Line defined by 2 points: [(-1.0, 1.0), (0.0, 0.0)]
Line defined by 2 points: [(0.0, 0.0), (-1.0, 0.0)]
Line defined by 2 points: [(0.0, 0.0), (0.0, -1.0)]
Line defined by 2 points: [(0.0, 0.0), (1.0, -1.0)]
Line defined by 2 points: [(0.0, 1.0), (-1.0, 1.0)]
Line defined by 2 points: [(0.0, 1.0), (0.0, 0.0)]
Line defined by 2 points: [(0.0, 1.0), (1.0, 0.0)]
Line defined by 2 points: [(1.0, -1.0), (0.0, -1.0)]
Line defined by 2 points: [(1.0, 0.0), (0.0, 0.0)]
Line defined by 2 points: [(1.0, 0.0), (1.0, -1.0)]
sage: sorted((line.options()['rgbcolor'], line.options()['thickness']) for line in p)
[('black', 2), ('black', 2), ('black', 2),
 ('black', 2), ('black', 2), ('black', 2),
 ('blue', 1), ('blue', 1), ('blue', 1),
 ('red', 1), ('red', 1), ('red', 1)]
```

plot_bounding_box(***options*)

Plot the bounding box.

INPUT:

- `**options` – Plotting options

This is mostly for testing purposes.

See also:

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: L.plot_bounding_box()
Graphics object consisting of 1 graphics primitive
```

TESTS:

```
sage: list(L.plot_bounding_box())
[Polygon defined by 4 points]
```

plot_coroots(*collection='simple'*, ***options*)

Plot the (simple/classical) coroots of this root lattice.

INPUT:

- `collection` – which coroots to display. Can be one of the following:
  - `"simple"` (the default)
  - `"classical"`
  - `"all"`
- `**options` – Plotting options

See also:

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: RootSystem(["B",3]).ambient_space().plot_coroots()
Graphics3d Object
```

TESTS:

```
sage: list(RootSystem(["B",2]).ambient_space().plot_coroots())
[Arrow from (0.0,0.0) to (1.0,-1.0),
 Text '$\alpha^\vee_{1}$' at the point (1.05,-1.05),
 Arrow from (0.0,0.0) to (0.0,2.0),
 Text '$\alpha^\vee_{2}$' at the point (0.0,2.1)]
```

plot_crystal(*crystal*, *plot_labels=True*, *label_color='black'*, *edge_labels=False*, *circle_size=0.06*, *circle_thickness=1.6*, ***options*)

Plot a finite crystal.

INPUT:

- `crystal` – the finite crystal to plot
- `plot_labels` – (default: `True`) can be one of the following:
  - `True` - use the latex labels
  - `'circles'` - use circles for multiplicity up to 4; if the multiplicity is larger, then it uses the multiplicity
  - `'multiplicities'` - use the multiplicities
- `label_color` – (default: `'black'`) the color of the labels
- `edge_labels` – (default: `False`) if `True`, then draw in the edge label
- `circle_size` – (default: 0.06) the size of the circles
- `circle_thickness` – (default: 1.6) the thinkness of the extra rings of circles
- `**options` – plotting options

See also:

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: L = RootSystem(['A',2]).ambient_space()
sage: C = crystals.Tableaux(['A',2], shape=[2,1])
sage: L.plot_crystal(C)
Graphics object consisting of 16 graphics primitives
sage: C = crystals.Tableaux(['A',2], shape=[8,4])
sage: p = L.plot_crystal(C, plot_labels='circles')
sage: p.show(figsize=15)
```

A 3-dimensional example:

```
sage: L = RootSystem(['B',3]).ambient_space()
sage: C = crystals.Tableaux(['B',3], shape=[2,1])
sage: L.plot_crystal(C, plot_labels='circles', edge_labels=True) # long time
Graphics3d Object
```

plot_fundamental_chamber(*style='normal'*, ***options*)

Plot the (classical) fundamental chamber.

INPUT:

- `style` – `"normal"` or `"classical"` (default: `"normal"`)
- `**options` – Plotting options

See also:

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

2D plots:

```
sage: RootSystem(["B",2]).ambient_space().plot_fundamental_chamber()
Graphics object consisting of 1 graphics primitive
sage: RootSystem(["B",2,1]).ambient_space().plot_fundamental_chamber()
Graphics object consisting of 1 graphics primitive
sage: RootSystem(["B",2,1]).ambient_space().plot_fundamental_chamber("classical")
Graphics object consisting of 1 graphics primitive
```

3D plots:

```
sage: RootSystem(["A",3,1]).weight_space() .plot_fundamental_chamber()
Graphics3d Object
sage: RootSystem(["B",3,1]).ambient_space().plot_fundamental_chamber()
Graphics3d Object
```

This feature is currently not available in the root lattice/space:

```
sage: list(RootSystem(["A",2]).root_lattice().plot_fundamental_chamber())
Traceback (most recent call last):
...
TypeError: classical fundamental chamber not yet available in the root lattice
```

TESTS:

```
sage: L = RootSystem(["B",2,1]).ambient_space()
sage: print L.plot_fundamental_chamber().description()
Polygon defined by 3 points:     [(0.5, 0.5), (1.0, 0.0), (0.0, 0.0)]
```

```
sage: print L.plot_fundamental_chamber(style="classical").description()
Polygon defined by 3 points:     [(0.0, 0.0), (3.0, 3.0), (3.0, 0.0)]
```

plot_fundamental_weights(**options*)

Plot the fundamental weights of this root lattice.

INPUT:

- `**options` – Plotting options

See also:

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: RootSystem(["B",3]).ambient_space().plot_fundamental_weights()
Graphics3d Object
```

TESTS:

```
sage: sorted(RootSystem(["A",2]).weight_lattice().plot_fundamental_weights(), key=str)
[Arrow from (0.0,0.0) to (0.0,1.0),
 Arrow from (0.0,0.0) to (1.0,0.0),
 Text '$\Lambda_{1}$' at the point (1.05,0.0),
 Text '$\Lambda_{2}$' at the point (0.0,1.05)]

 sage: sorted(RootSystem(["A",2]).ambient_lattice().plot_fundamental_weights(), key=str)
 [Arrow from (0.0,0.0) to (-0.5,0.866024518389),
  Arrow from (0.0,0.0) to (0.5,0.866024518389),
  Text '$\Lambda_{1}$' at the point (0.525,0.909325744308),
  Text '$\Lambda_{2}$' at the point (-0.525,0.909325744308)]
```

plot_hedron(**options*)

Plot the polyhedron whose vertices are given by the orbit of ρ.

In type A, this is the usual permutohedron.

See also:

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: RootSystem(["A",2]).ambient_space().plot_hedron()
Graphics object consisting of 8 graphics primitives
sage: RootSystem(["A",3]).ambient_space().plot_hedron()
Graphics3d Object
sage: RootSystem(["B",3]).ambient_space().plot_hedron()
Graphics3d Object
sage: RootSystem(["C",3]).ambient_space().plot_hedron()
Graphics3d Object
sage: RootSystem(["D",3]).ambient_space().plot_hedron()
Graphics3d Object
```

Surprise: polyhedrons of large dimension know how to project themselves nicely:

```
sage: RootSystem(["F",4]).ambient_space().plot_hedron() # long time
Graphics3d Object
```

TESTS:

```
sage: L = RootSystem(["B",2]).ambient_space()
sage: print L.plot_hedron().description()
```

```
Polygon defined by 8 points: [(1.5, 0.5), (0.5, 1.5), (-0.5, 1.5), (-1.5, 0.5), (-1.5, -
Line defined by 2 points: [(-0.5, -1.5), (0.5, -1.5)]
Line defined by 2 points: [(-0.5, 1.5), (0.5, 1.5)]
Line defined by 2 points: [(-1.5, -0.5), (-0.5, -1.5)]
Line defined by 2 points: [(-1.5, -0.5), (-1.5, 0.5)]
Line defined by 2 points: [(-1.5, 0.5), (-0.5, 1.5)]
Line defined by 2 points: [(0.5, -1.5), (1.5, -0.5)]
Line defined by 2 points: [(0.5, 1.5), (1.5, 0.5)]
Line defined by 2 points: [(1.5, -0.5), (1.5, 0.5)]
Point set defined by 8 point(s): [(-1.5, -0.5), (-1.5, 0.5), (-0.5, -1.5), (-0.5, 1.5),
```

plot_ls_paths(*paths*, *plot_labels=None*, *colored_labels=True*, ***options*)

Plot LS paths.

INPUT:

- `paths` – a finite crystal or list of LS paths
- `plot_labels` – (default: `None`) the distance to plot the LS labels from the endpoint of the path; set to `None` to not display the labels
- `colored_labels` – (default: `True`) if `True`, then color the labels the same color as the LS path
- `**options` – plotting options

See also:

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: B = crystals.LSPaths(['A',2], [1,1])
sage: L = RootSystem(['A',2]).ambient_space()
sage: L.plot_fundamental_weights() + L.plot_ls_paths(B)
Graphics object consisting of 14 graphics primitives
```

This also works in 3 dimensions:

```
sage: B = crystals.LSPaths(['B',3], [2,0,0])
sage: L = RootSystem(['B',3]).ambient_space()
sage: L.plot_ls_paths(B)
Graphics3d Object
```

plot_parse_options(***args*)

Return an option object to be used for root system plotting.

EXAMPLES:

```
sage: L = RootSystem(["A",2,1]).ambient_space()
sage: options = L.plot_parse_options()
sage: options
<sage.combinat.root_system.plot.PlotOptions instance at ...>
```

See also:

- `plot()` for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

plot_reflection_hyperplanes(*collection='simple'*, ***options*)

Plot the simple reflection hyperplanes.

INPUT:

- `collection` – which reflection hyperplanes to display. Can be one of the following:
  - `"simple"` (the default)
  - `"classical"`

- "all"
- **options – Plotting options

See also:

- plot() for a description of the plotting options
- *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: RootSystem(["A",2,1]).ambient_space().plot_reflection_hyperplanes()
Graphics object consisting of 6 graphics primitives
sage: RootSystem(["G",2,1]).ambient_space().plot_reflection_hyperplanes()
Graphics object consisting of 6 graphics primitives
sage: RootSystem(["A",3]).weight_space().plot_reflection_hyperplanes()
Graphics3d Object
sage: RootSystem(["B",3]).ambient_space().plot_reflection_hyperplanes()
Graphics3d Object
sage: RootSystem(["A",3,1]).weight_space().plot_reflection_hyperplanes()
Graphics3d Object
sage: RootSystem(["B",3,1]).ambient_space().plot_reflection_hyperplanes()
Graphics3d Object
sage: RootSystem(["A",2,1]).weight_space().plot_reflection_hyperplanes(affine=False, lev
Graphics3d Object
sage: RootSystem(["A",2]).root_lattice().plot_reflection_hyperplanes()
Graphics object consisting of 4 graphics primitives
```

TESTS:

```
sage: L = RootSystem(["A",2]).ambient_space()
sage: print L.plot_reflection_hyperplanes().description()
Text '$H_{\alpha^\vee_{1}}$' at the point (-1.81...,3.15)
Text '$H_{\alpha^\vee_{2}}$' at the point (1.81...,3.15)
Line defined by 2 points: [(-1.73..., 3.0), (1.73..., -3.0)]
Line defined by 2 points: [(1.73..., 3.0), (-1.73..., -3.0)]

sage: print L.plot_reflection_hyperplanes("all").description()
Text '$H_{\alpha^\vee_{1} + \alpha^\vee_{2}}$' at the point (3.15,0.0)
Text '$H_{\alpha^\vee_{1}}$' at the point (-1.81...,3.15)
Text '$H_{\alpha^\vee_{2}}$' at the point (1.81...,3.15)
Line defined by 2 points: [(-1.73..., 3.0), (1.73..., -3.0)]
Line defined by 2 points: [(1.73..., 3.0), (-1.73..., -3.0)]
Line defined by 2 points: [(3.0, 0.0), (-3.0, 0.0)]

sage: L = RootSystem(["A",2,1]).ambient_space()
sage: print L.plot_reflection_hyperplanes().description()
Text '$H_{\alpha^\vee_{0}}$' at the point (3.15,0.90...)
Text '$H_{\alpha^\vee_{1}}$' at the point (-1.81...,3.15)
Text '$H_{\alpha^\vee_{2}}$' at the point (1.81...,3.15)
Line defined by 2 points: [(-1.73..., 3.0), (1.73..., -3.0)]
Line defined by 2 points: [(1.73..., 3.0), (-1.73..., -3.0)]
Line defined by 2 points: [(3.0, 0.86...), (-3.0, 0.86...)]
```

Todo

Provide an option for transparency?

---

plot_roots(*collection='simple'*, ***options*)

Plot the (simple/classical) roots of this root lattice.

INPUT:

- collection – which roots to display can be one of the following:

- "simple" (the default)
- "classical"
- "all"

• **options – Plotting options

See also:

• plot() for a description of the plotting options
• *Tutorial: visualizing root systems* for a tutorial on root system plotting

EXAMPLES:

```
sage: RootSystem(["B",3]).ambient_space().plot_roots()
Graphics3d Object
sage: RootSystem(["B",3]).ambient_space().plot_roots("all")
Graphics3d Object
```

TESTS:

```
sage: list(RootSystem(["A",2]).root_lattice().plot_roots())
[Arrow from (0.0,0.0) to (1.0,0.0),
 Text '$\alpha_{1}$' at the point (1.05,0.0),
 Arrow from (0.0,0.0) to (0.0,1.0),
 Text '$\alpha_{2}$' at the point (0.0,1.05)]

sage: list(RootSystem(["A",2]).weight_lattice().plot_roots(labels=False))
[Arrow from (0.0,0.0) to (2.0,-1.0),
 Arrow from (0.0,0.0) to (-1.0,2.0)]

 sage: list(RootSystem(["A",2]).ambient_lattice().plot_roots())
 [Arrow from (0.0,0.0) to (1.5,0.86...),
  Text '$\alpha_{1}$' at the point (1.575,0.90...),
  Arrow from (0.0,0.0) to (-1.5,0.86...),
  Text '$\alpha_{2}$' at the point (-1.575,0.90...)]

 sage: list(RootSystem(["B",2]).ambient_space().plot_roots())
 [Arrow from (0.0,0.0) to (1.0,-1.0),
  Text '$\alpha_{1}$' at the point (1.05,-1.05),
  Arrow from (0.0,0.0) to (0.0,1.0),
  Text '$\alpha_{2}$' at the point (0.0,1.05)]

sage: list(RootSystem(["A",2]).root_lattice().plot_roots("all"))
[Arrow from (0.0,0.0) to (1.0,0.0),
 Text '$\alpha_{1}$' at the point (1.05,0.0),
 Arrow from (0.0,0.0) to (0.0,1.0),
 Text '$\alpha_{2}$' at the point (0.0,1.05),
 Arrow from (0.0,0.0) to (1.0,1.0),
 Text '$\alpha_{1} + \alpha_{2}$' at the point (1.05,1.05),
 Arrow from (0.0,0.0) to (-1.0,0.0),
 Text '$-\alpha_{1}$' at the point (-1.05,0.0),
 Arrow from (0.0,0.0) to (0.0,-1.0),
 Text '$-\alpha_{2}$' at the point (0.0,-1.05),
 Arrow from (0.0,0.0) to (-1.0,-1.0),
 Text '$-\alpha_{1} - \alpha_{2}$' at the point (-1.05,-1.05)]
```

positive_imaginary_roots()

Return the positive imaginary roots of self.

EXAMPLES:

```
sage: L = RootSystem(['A',3]).root_lattice()
sage: L.positive_imaginary_roots()
()
```

```
sage: L = RootSystem(['A',3,1]).root_lattice()
sage: PIR = L.positive_imaginary_roots(); PIR
Positive imaginary roots of type ['A', 3, 1]
sage: [PIR.unrank(i) for i in range(5)]
[alpha[0] + alpha[1] + alpha[2] + alpha[3],
 2*alpha[0] + 2*alpha[1] + 2*alpha[2] + 2*alpha[3],
 3*alpha[0] + 3*alpha[1] + 3*alpha[2] + 3*alpha[3],
 4*alpha[0] + 4*alpha[1] + 4*alpha[2] + 4*alpha[3],
 5*alpha[0] + 5*alpha[1] + 5*alpha[2] + 5*alpha[3]]
```

positive_real_roots()

Return the positive real roots of `self`.

EXAMPLES:

```
sage: L = RootSystem(['A',3]).root_lattice()
sage: sorted(L.positive_real_roots())
[alpha[1], alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3],
 alpha[2], alpha[2] + alpha[3], alpha[3]]

sage: L = RootSystem(['A',3,1]).root_lattice()
sage: PRR = L.positive_real_roots(); PRR
Positive real roots of type ['A', 3, 1]
sage: [PRR.unrank(i) for i in range(10)]
[alpha[1],
 alpha[2],
 alpha[3],
 alpha[1] + alpha[2],
 alpha[2] + alpha[3],
 alpha[1] + alpha[2] + alpha[3],
 alpha[0] + 2*alpha[1] + alpha[2] + alpha[3],
 alpha[0] + alpha[1] + 2*alpha[2] + alpha[3],
 alpha[0] + alpha[1] + alpha[2] + 2*alpha[3],
 alpha[0] + 2*alpha[1] + 2*alpha[2] + alpha[3]]

sage: Q = RootSystem(['A',4,2]).root_lattice()
sage: PR = Q.positive_roots()
sage: [PR.unrank(i) for i in range(5)]
[alpha[1],
 alpha[2],
 2*alpha[1] + alpha[2],
 alpha[1] + alpha[2],
 alpha[0] + alpha[1] + alpha[2]]

sage: Q = RootSystem(['D',3,2]).root_lattice()
sage: PR = Q.positive_roots()
sage: [PR.unrank(i) for i in range(5)]
[alpha[1],
 alpha[2],
 alpha[1] + 2*alpha[2],
 alpha[1] + alpha[2],
 alpha[0] + alpha[1] + 2*alpha[2]]
```

positive_roots(*index_set=None*)

Return the positive roots of `self`.

If `index_set` is not `None`, returns the positive roots of the parabolic subsystem with simple roots in `index_set`.

Algorithm for finite type: generate them from the simple roots by applying successive reflections

toward the positive chamber.

EXAMPLES:

```
sage: L = RootSystem(['A',3]).root_lattice()
sage: sorted(L.positive_roots())
[alpha[1], alpha[1] + alpha[2],
 alpha[1] + alpha[2] + alpha[3], alpha[2],
 alpha[2] + alpha[3], alpha[3]]
sage: sorted(L.positive_roots((1,2)))
[alpha[1], alpha[1] + alpha[2], alpha[2]]
sage: sorted(L.positive_roots(()))
[]

sage: L = RootSystem(['A',3,1]).root_lattice()
sage: PR = L.positive_roots(); PR
Disjoint union of Family (Positive real roots of type ['A', 3, 1],
    Positive imaginary roots of type ['A', 3, 1])
sage: [PR.unrank(i) for i in range(10)]
[alpha[1],
 alpha[2],
 alpha[3],
 alpha[1] + alpha[2],
 alpha[2] + alpha[3],
 alpha[1] + alpha[2] + alpha[3],
 alpha[0] + 2*alpha[1] + alpha[2] + alpha[3],
 alpha[0] + alpha[1] + 2*alpha[2] + alpha[3],
 alpha[0] + alpha[1] + alpha[2] + 2*alpha[3],
 alpha[0] + 2*alpha[1] + 2*alpha[2] + alpha[3]]
```

positive_roots_by_height(*increasing=True*)

Returns a list of positive roots in increasing order by height.

If `increasing` is False, returns them in decreasing order.

Warning: Raise an error if the Cartan type is not finite.

EXAMPLES:

```
sage: L = RootSystem(['C',2]).root_lattice()
sage: L.positive_roots_by_height()
[alpha[2], alpha[1], alpha[1] + alpha[2], 2*alpha[1] + alpha[2]]
sage: L.positive_roots_by_height(increasing = False)
[2*alpha[1] + alpha[2], alpha[1] + alpha[2], alpha[2], alpha[1]]

sage: L = RootSystem(['A',2,1]).root_lattice()
sage: L.positive_roots_by_height()
Traceback (most recent call last):
...
NotImplementedError: Only implemented for finite Cartan type
```

positive_roots_nonparabolic(*index_set=None*)

Returns the set of positive roots outside the parabolic subsystem with Dynkin node set `index_set`.

INPUT:

- `index_set` – (default:None) the Dynkin node set of the parabolic subsystem. It should be a tuple. The default value implies the entire Dynkin node set

EXAMPLES:

```
sage: lattice =  RootSystem(['A',3]).root_lattice()
sage: sorted(lattice.positive_roots_nonparabolic((1,3)), key=str)
```

```
[alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3], alpha[2], alpha[2] + alpha[3]]
sage: sorted(lattice.positive_roots_nonparabolic((2,3)), key=str)
[alpha[1], alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3]]
sage: lattice.positive_roots_nonparabolic()
[]
sage: lattice.positive_roots_nonparabolic((1,2,3))
[]
```

Warning: This returns an error if the Cartan type is not finite.

positive_roots_nonparabolic_sum(*index_set=None*)

Returns the sum of positive roots outside the parabolic subsystem with Dynkin node set `index_set`.

INPUT:

- `index_set` – (default:None) the Dynkin node set of the parabolic subsystem. It should be a tuple. The default value implies the entire Dynkin node set

EXAMPLES:

```
sage: lattice = RootSystem(['A',3]).root_lattice()
sage: lattice.positive_roots_nonparabolic_sum((1,3))
2*alpha[1] + 4*alpha[2] + 2*alpha[3]
sage: lattice.positive_roots_nonparabolic_sum((2,3))
3*alpha[1] + 2*alpha[2] + alpha[3]
sage: lattice.positive_roots_nonparabolic_sum(())
3*alpha[1] + 4*alpha[2] + 3*alpha[3]
sage: lattice.positive_roots_nonparabolic_sum()
0
sage: lattice.positive_roots_nonparabolic_sum((1,2,3))
0
```

Warning: This returns an error if the Cartan type is not finite.

positive_roots_parabolic(*index_set=None*)

Return the set of positive roots for the parabolic subsystem with Dynkin node set `index_set`.

INPUT:

- `index_set` – (default:None) the Dynkin node set of the parabolic subsystem. It should be a tuple. The default value implies the entire Dynkin node set

EXAMPLES:

```
sage: lattice = RootSystem(['A',3]).root_lattice()
sage: sorted(lattice.positive_roots_parabolic((1,3)), key=str)
[alpha[1], alpha[3]]
sage: sorted(lattice.positive_roots_parabolic((2,3)), key=str)
[alpha[2], alpha[2] + alpha[3], alpha[3]]
sage: sorted(lattice.positive_roots_parabolic(), key=str)
[alpha[1], alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3], alpha[2], alpha[2] + alpl
```

Warning: This returns an error if the Cartan type is not finite.

projection(*root*, *coroot=None*, *to_negative=True*)

Returns the projection along the root, and across the hyperplane define by coroot, as a function π from self to self. π is a half-linear map which stabilizes the negative half space, and acts by reflection on the positive half space.

If to_negative is False, then this project onto the positive half space instead.

EXAMPLES:

```
sage: space = RootSystem(['A',2]).weight_lattice()
sage: x=space.simple_roots()[1]
sage: y=space.simple_coroots()[1]
sage: pi = space.projection(x,y)
sage: x
2*Lambda[1] - Lambda[2]
sage: pi(x)
-2*Lambda[1] + Lambda[2]
sage: pi(-x)
-2*Lambda[1] + Lambda[2]
sage: pi = space.projection(x,y,False)
sage: pi(-x)
2*Lambda[1] - Lambda[2]
```

reflection(*root*, *coroot=None*)

Returns the reflection along the root, and across the hyperplane define by coroot, as a function from self to self.

EXAMPLES:

```
sage: space = RootSystem(['A',2]).weight_lattice()
sage: x=space.simple_roots()[1]
sage: y=space.simple_coroots()[1]
sage: s = space.reflection(x,y)
sage: x
2*Lambda[1] - Lambda[2]
sage: s(x)
-2*Lambda[1] + Lambda[2]
sage: s(-x)
2*Lambda[1] - Lambda[2]
```

root_poset(*restricted=False*, *facade=False*)

Returns the (restricted) root poset associated to `self`.

The elements are given by the positive roots (resp. non-simple, positive roots), and $\alpha \leq \beta$ iff $\beta - \alpha$ is a non-negative linear combination of simple roots.

INPUT:

- `restricted` – (default:False) if True, only non-simple roots are considered.
- `facade` – (default:False) passes facade option to the poset generator.

EXAMPLES:

```
sage: Phi = RootSystem(['A',1]).root_poset(); Phi
Finite poset containing 1 elements
sage: Phi.cover_relations()
[]

sage: Phi = RootSystem(['A',2]).root_poset(); Phi
Finite poset containing 3 elements

sage: sorted(Phi.cover_relations(), key=str)
[[alpha[1], alpha[1] + alpha[2]], [alpha[2], alpha[1] + alpha[2]]]

sage: Phi = RootSystem(['A',3]).root_poset(restricted=True); Phi
Finite poset containing 3 elements
sage: sorted(Phi.cover_relations(), key=str)
[[alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3]], [alpha[2] + alpha[3], alpha[1] +

sage: Phi = RootSystem(['B',2]).root_poset(); Phi
Finite poset containing 4 elements
```

```
sage: sorted(Phi.cover_relations(), key=str)
[[alpha[1] + alpha[2], alpha[1] + 2*alpha[2]],
 [alpha[1], alpha[1] + alpha[2]],
 [alpha[2], alpha[1] + alpha[2]]]
```

TESTS:

Check that trac ticket #17982 is fixed:

```
sage: RootSystem(['A', 2]).ambient_space().root_poset()
Finite poset containing 3 elements
```

roots()

Return the roots of `self`.

EXAMPLES:

```
sage: RootSystem(['A',2]).ambient_lattice().roots()
[(1, -1, 0), (1, 0, -1), (0, 1, -1), (-1, 1, 0), (-1, 0, 1), (0, -1, 1)]
```

This matches with Wikipedia article Root_systems:

```
sage: for T in CartanType.samples(finite = True, crystallographic = True):
...       print "%s %3s %3s"%(T, len(RootSystem(T).root_lattice().roots()), len(RootSyst
['A', 1]   2   2
['A', 5]  30  30
['B', 1]   2   2
['B', 5]  50  50
['C', 1]   2   2
['C', 5]  50  50
['D', 2]   4   4
['D', 3]  12  12
['D', 5]  40  40
['E', 6]  72  72
['E', 7] 126 126
['E', 8] 240 240
['F', 4]  48  48
['G', 2]  12  12
```

Todo

The result should be an enumerated set, and handle infinite root systems.

---

s()

Returns the family $(s_i)_{i \in I}$ of the simple reflections of this root system.

EXAMPLES:

```
sage: r = RootSystem(["A", 2]).root_lattice()
sage: s = r.simple_reflections()
sage: s[1]( r.simple_root(1) )
-alpha[1]
```

TEST:

```
sage: s
simple reflections
```

short_roots()

Return a list of the short roots of `self`.

EXAMPLES:

```
sage: L = RootSystem(['B',3]).root_lattice()
sage: sorted(L.short_roots())
[-alpha[1] - alpha[2] - alpha[3],
 alpha[1] + alpha[2] + alpha[3],
 -alpha[2] - alpha[3],
 alpha[2] + alpha[3],
 -alpha[3],
 alpha[3]]
```

simple_coroot(*i*)

Returns the i^{th} simple coroot.

EXAMPLES:

```
sage: RootSystem(['A',2]).root_lattice().simple_coroot(1)
alphacheck[1]
```

simple_coroots()

Returns the family $(\alpha_i^\vee)_{i\in I}$ of the simple coroots.

EXAMPLES:

```
sage: alphacheck = RootSystem(['A',3]).root_lattice().simple_coroots()
sage: [alphacheck[i] for i in [1, 2, 3]]
[alphacheck[1], alphacheck[2], alphacheck[3]]
```

simple_projection(*i*, *to_negative=True*)

Returns the projection along the i^{th} simple root, and across the hyperplane define by the i^{th} simple coroot, as a function from self to self.

INPUT:

- `i` - i is in self's index set

EXAMPLES:

```
sage: space = RootSystem(['A',2]).weight_lattice()
sage: x = space.simple_roots()[1]
sage: pi = space.simple_projection(1)
sage: x
2*Lambda[1] - Lambda[2]
sage: pi(x)
-2*Lambda[1] + Lambda[2]
sage: pi(-x)
-2*Lambda[1] + Lambda[2]
sage: pi = space.simple_projection(1,False)
sage: pi(-x)
2*Lambda[1] - Lambda[2]
```

simple_projections(*to_negative=True*)

Returns the family $(s_i)_{i\in I}$ of the simple projections of this root system

EXAMPLES:

```
sage: space = RootSystem(['A',2]).weight_lattice()
sage: pi = space.simple_projections()
sage: x = space.simple_roots()
sage: pi[1](x[2])
-Lambda[1] + 2*Lambda[2]
```

TESTS: sage: pi pi

simple_reflection(*i*)

Returns the i^{th} simple reflection, as a function from self to self.

INPUT:

- i - i is in self's index set

EXAMPLES:

```
sage: space = RootSystem(['A',2]).ambient_lattice()
sage: s = space.simple_reflection(1)
sage: x = space.simple_roots()[1]
sage: x
(1, -1, 0)
sage: s(x)
(-1, 1, 0)
```

simple_reflections()

Returns the family $(s_i)_{i \in I}$ of the simple reflections of this root system.

EXAMPLES:

```
sage: r = RootSystem(["A", 2]).root_lattice()
sage: s = r.simple_reflections()
sage: s[1]( r.simple_root(1) )
-alpha[1]
```

TEST:

```
sage: s
simple reflections
```

simple_root(i)

Returns the i^{th} simple root.

This should be overridden by any subclass, and typically implemented as a cached method for efficiency.

EXAMPLES:

```
sage: r = RootSystem(["A",3]).root_lattice()
sage: r.simple_root(1)
alpha[1]
```

TESTS:

```
sage: super(sage.combinat.root_system.root_space.RootSpace, r).simple_root(1)
Traceback (most recent call last):
...
NotImplementedError: <abstract method simple_root at ...>
```

simple_roots()

Returns the family $(\alpha_i)_{i \in I}$ of the simple roots.

EXAMPLES:

```
sage: alpha = RootSystem(["A",3]).root_lattice().simple_roots()
sage: [alpha[i] for i in [1,2,3]]
[alpha[1], alpha[2], alpha[3]]
```

simple_roots_tilde()

Return the family $(\tilde{\alpha}_i)_{i \in I}$ of the simple roots.

INPUT:

- self – an affine root lattice realization

The $\tilde{\alpha}_i$ give the embedding of the root lattice of the other affinization of the same classical root lattice into this root lattice (space?).

This uses the fact that $\alpha_i = \tilde{\alpha}_i$ for i not a special node, and that

$$\delta = \sum a_i \alpha_i = \sum b_i \tilde{\alpha}_i$$

EXAMPLES:

In simply laced cases, this is boring:

```
sage: RootSystem(["A",3, 1]).root_lattice().simple_roots_tilde()
Finite family {0: alpha[0], 1: alpha[1], 2: alpha[2], 3: alpha[3]}
```

This was checked by hand:

```
sage: RootSystem(["C",2,1]).coroot_lattice().simple_roots_tilde()
Finite family {0: alphacheck[0] - alphacheck[2], 1: alphacheck[1], 2: alphacheck[2]}
sage: RootSystem(["B",2,1]).coroot_lattice().simple_roots_tilde()
Finite family {0: alphacheck[0] - alphacheck[1], 1: alphacheck[1], 2: alphacheck[2]}
```

What about type BC?

some_elements()

Return some elements of this root lattice realization

EXAMPLES:

```
sage: L = RootSystem(["A",2]).weight_lattice()
sage: L.some_elements()
[2*Lambda[1] + 2*Lambda[2], 2*Lambda[1] - Lambda[2], -Lambda[1] + 2*Lambda[2], Lambda[1]
sage: L = RootSystem(["A",2]).root_lattice()
sage: L.some_elements()
[2*alpha[1] + 2*alpha[2], alpha[1], alpha[2]]
```

tau_epsilon_operator_on_almost_positive_roots(J)

The τ_ϵ operator on almost positive roots

Given a subset J of non adjacent vertices of the Dynkin diagram, this constructs the operator on the almost positive roots which fixes the negative simple roots α_i for i not in J, and acts otherwise by:

$$\tau_+(\beta) = (\prod_{i \in J} s_i)(\beta)$$

See Equation (1.2) of [CFZ].

EXAMPLES:

```
sage: L = RootSystem(['A',4]).root_lattice()
sage: tau = L.tau_epsilon_operator_on_almost_positive_roots([1,3])
sage: alpha = L.simple_roots()
```

The action on a negative simple root not in J:

```
sage: tau(-alpha[2])
-alpha[2]
```

The action on a negative simple root in J:

```
sage: tau(-alpha[1])
alpha[1]
```

The action on all almost positive roots:

```
sage: for root in L.almost_positive_roots():
...       print 'tau({:<41}) ='.format(root), tau(root)
tau(-alpha[1]                                ) = alpha[1]
tau(alpha[1]                                 ) = -alpha[1]
tau(alpha[1] + alpha[2]                      ) = alpha[2] + alpha[3]
tau(alpha[1] + alpha[2] + alpha[3]           ) = alpha[2]
tau(alpha[1] + alpha[2] + alpha[3] + alpha[4]) = alpha[2] + alpha[3] + alpha[4]
tau(-alpha[2]                                ) = -alpha[2]
tau(alpha[2]                                 ) = alpha[1] + alpha[2] + alpha[3]
```

```
tau(alpha[2] + alpha[3]                          ) = alpha[1] + alpha[2]
tau(alpha[2] + alpha[3] + alpha[4]               ) = alpha[1] + alpha[2] + alpha[3] + alpha[
tau(-alpha[3]                                    ) = alpha[3]
tau(alpha[3]                                     ) = -alpha[3]
tau(alpha[3] + alpha[4]                          ) = alpha[4]
tau(-alpha[4]                                    ) = -alpha[4]
tau(alpha[4]                                     ) = alpha[3] + alpha[4]
```

This method works on any root lattice realization:

```
sage: L = RootSystem(['B',3]).ambient_space()
sage: tau = L.tau_epsilon_operator_on_almost_positive_roots([1,3])
sage: for root in L.almost_positive_roots():
...       print 'tau({:<41}) ='.format(root), tau(root)
tau((-1, 1, 0)                                   ) = (1, -1, 0)
tau((1, 0, 0)                                    ) = (0, 1, 0)
tau((1, -1, 0)                                   ) = (-1, 1, 0)
tau((1, 1, 0)                                    ) = (1, 1, 0)
tau((1, 0, -1)                                   ) = (0, 1, 1)
tau((1, 0, 1)                                    ) = (0, 1, -1)
tau((0, -1, 1)                                   ) = (0, -1, 1)
tau((0, 1, 0)                                    ) = (1, 0, 0)
tau((0, 1, -1)                                   ) = (1, 0, 1)
tau((0, 1, 1)                                    ) = (1, 0, -1)
tau((0, 0, -1)                                   ) = (0, 0, 1)
tau((0, 0, 1)                                    ) = (0, 0, -1)
```

See also:

`tau_plus_minus()`

REFERENCES:

tau_plus_minus()

Returns the τ^+ and τ^- piecewise linear operators on `self`

Those operators are induced by the bipartition $\{L, R\}$ of the simple roots of `self`, and stabilize the almost positive roots. Namely, τ_+ fixes the negative simple roots α_i for i in R, and acts otherwise by:

$$\tau_+(\beta) = (\prod_{i \in L} s_i)(\beta)$$

τ_- acts analogously, with L and R interchanged.

Those operators are used to construct the associahedron, a polytopal realization of the cluster complex (see `Associahedron`).

See also:

`tau_epsilon_operator_on_almost_positive_roots()`

EXAMPLES:

We explore the example of [CFZ] Eq.(1.3):

```
sage: S = RootSystem(['A',2]).root_lattice()
sage: taup, taum = S.tau_plus_minus()
sage: for beta in S.almost_positive_roots(): print beta, ",", taup(beta), ",", taum(beta
-alpha[1] , alpha[1] , -alpha[1]
alpha[1] , -alpha[1] , alpha[1] + alpha[2]
alpha[1] + alpha[2] , alpha[2] , alpha[1]
-alpha[2] , -alpha[2] , alpha[2]
alpha[2] , alpha[1] + alpha[2] , -alpha[2]
```

to_ambient_space_morphism()
Return the morphism to the ambient space.

EXAMPLES:

```
sage: CartanType(['B',2]).root_system().root_lattice().to_ambient_space_morphism()
Generic morphism:
From: Root lattice of the Root system of type ['B', 2]
To:   Ambient space of the Root system of type ['B', 2]
sage: CartanType(['B',2]).root_system().coroot_lattice().to_ambient_space_morphism()
Generic morphism:
From: Coroot lattice of the Root system of type ['B', 2]
To:   Ambient space of the Root system of type ['B', 2]
sage: CartanType(['B',2]).root_system().weight_lattice().to_ambient_space_morphism()
Generic morphism:
From: Weight lattice of the Root system of type ['B', 2]
To:   Ambient space of the Root system of type ['B', 2]
```

weyl_group(*prefix=None*)
Returns the Weyl group associated to self.

EXAMPLES:

```
sage: RootSystem(['F',4]).ambient_space().weyl_group()
Weyl Group of type ['F', 4] (as a matrix group acting on the ambient space)
sage: RootSystem(['F',4]).root_space().weyl_group()
Weyl Group of type ['F', 4] (as a matrix group acting on the root space)
```

RootLatticeRealizations.**super_categories**()
EXAMPLES:

```
sage: from sage.combinat.root_system.root_lattice_realizations import RootLatticeRealization
sage: RootLatticeRealizations(QQ).super_categories()
[Category of vector spaces with basis over Rational Field]
```

5.1.209 Root lattices and root spaces

class sage.combinat.root_system.root_space.**RootSpace**(*root_system*, *base_ring*)
Bases: sage.misc.cachefunc.ClearCacheOnPickle, sage.combinat.free_module.CombinatorialFree

The root space of a root system over a given base ring

INPUT:

- root_system - a root system
- base_ring: a ring R

The *root space* (or lattice if base_ring is $\mathbf{Z}$) of a root system is the formal free module $\bigoplus_i R\alpha_i$ generated by the simple roots $(\alpha_i)_{i\in I}$ of the root system.

This class is also used for coroot spaces (or lattices).

See also:

- RootSystem()
- RootSystem.root_lattice() and RootSystem.root_space()
- RootLatticeRealizations()

Todo: standardize the variable used for the root space in the examples (P?)

TESTS:

```
sage: for ct in CartanType.samples(crystallographic=True)+[CartanType(["A",2],["C",5,1])]:
...       TestSuite(ct.root_system().root_lattice()).run()
...       TestSuite(ct.root_system().root_space()).run()
sage: r = RootSystem(['A',4]).root_lattice()
sage: r.simple_root(1)
alpha[1]
sage: latex(r.simple_root(1))
\alpha_{1}
```

Element
alias of RootSpaceElement

simple_root()
Return the basis element indexed by i.

INPUT:

- i – an element of the index set

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.monomial('a')
B['a']
```

F.monomial is in fact (almost) a map:

```
sage: F.monomial
Term map from {'a', 'b', 'c'} to Free module generated by {'a', 'b', 'c'} over Rational Fiel
```

to_ambient_space_morphism()
The morphism from self to its associated ambient space.

EXAMPLES:

```
sage: CartanType(['A',2]).root_system().root_lattice().to_ambient_space_morphism()
Generic morphism:
From: Root lattice of the Root system of type ['A', 2]
To:   Ambient space of the Root system of type ['A', 2]
```

to_coroot_space_morphism()
Returns the nu map to the coroot space over the same base ring, using the symmetrizer of the Cartan matrix

It does not map the root lattice to the coroot lattice, but has the property that any root is mapped to some scalar multiple of its associated coroot.

EXAMPLES:

```
sage: R = RootSystem(['A',3]).root_space()
sage: alpha = R.simple_roots()
sage: f = R.to_coroot_space_morphism()
sage: f(alpha[1])
alphacheck[1]
sage: f(alpha[1]+alpha[2])
alphacheck[1] + alphacheck[2]

sage: R = RootSystem(['A',3]).root_lattice()
sage: alpha = R.simple_roots()
sage: f = R.to_coroot_space_morphism()
sage: f(alpha[1])
```

```
alphacheck[1]
sage: f(alpha[1]+alpha[2])
alphacheck[1] + alphacheck[2]

sage: S = RootSystem(['G',2]).root_space()
sage: alpha = S.simple_roots()
sage: f = S.to_coroot_space_morphism()
sage: f(alpha[1])
alphacheck[1]
sage: f(alpha[1]+alpha[2])
alphacheck[1] + 3*alphacheck[2]
```

class sage.combinat.root_system.root_space.**RootSpaceElement**(M, x)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

associated_coroot()

Returns the coroot associated to this root

OUTPUT:

An element of the coroot space over the same base ring; in particular the result is in the coroot lattice whenever self is in the root lattice.

EXAMPLES:

```
sage: L = RootSystem(["B", 3]).root_space()
sage: alpha = L.simple_roots()
sage: alpha[1].associated_coroot()
alphacheck[1]
sage: alpha[1].associated_coroot().parent()
Coroot space over the Rational Field of the Root system of type ['B', 3]

sage: L.highest_root()
alpha[1] + 2*alpha[2] + 2*alpha[3]
sage: L.highest_root().associated_coroot()
alphacheck[1] + 2*alphacheck[2] + alphacheck[3]

sage: alpha = RootSystem(["B", 3]).root_lattice().simple_roots()
sage: alpha[1].associated_coroot()
alphacheck[1]
sage: alpha[1].associated_coroot().parent()
Coroot lattice of the Root system of type ['B', 3]
```

is_positive_root()

Checks whether an element in the root space lies in the nonnegative cone spanned by the simple roots.

EXAMPLES:

```
sage: R=RootSystem(['A',3,1]).root_space()
sage: B=R.basis()
sage: w=B[0]+B[3]
sage: w.is_positive_root()
True
sage: w=B[1]-B[2]
sage: w.is_positive_root()
False
```

max_coroot_le()

Returns the highest positive coroot whose associated root is less than or equal to `self`.

INPUT:

- `self` – an element of the nonnegative integer span of simple roots.

Returns None for the zero element.

Really `self` is an element of a coroot lattice and this method returns the highest root whose associated coroot is <= `self`.

Warning: This implementation only handles finite Cartan types

EXAMPLES:

```
sage: root_lattice = RootSystem(['C',2]).root_lattice()
sage: root_lattice.from_vector(vector([1,1])).max_coroot_le()
alphacheck[1] + 2*alphacheck[2]
sage: root_lattice.from_vector(vector([2,1])).max_coroot_le()
alphacheck[1] + 2*alphacheck[2]
sage: root_lattice = RootSystem(['B',2]).root_lattice()
sage: root_lattice.from_vector(vector([1,1])).max_coroot_le()
2*alphacheck[1] + alphacheck[2]
sage: root_lattice.from_vector(vector([1,2])).max_coroot_le()
2*alphacheck[1] + alphacheck[2]

sage: root_lattice.zero().max_coroot_le() is None
True
sage: root_lattice.from_vector(vector([-1,0])).max_coroot_le()
Traceback (most recent call last):
...
ValueError: -alpha[1] is not in the positive cone of roots
sage: root_lattice = RootSystem(['A',2,1]).root_lattice()
sage: root_lattice.simple_root(1).max_coroot_le()
Traceback (most recent call last):
...
NotImplementedError: Only implemented for finite Cartan type
```

max_quantum_element()

Returns a reduced word for the longest element of the Weyl group whose shortest path in the quantum Bruhat graph to the identity, has sum of quantum coroots at most `self`.

INPUT:

- `self` – an element of the nonnegative integer span of simple roots.

Really `self` is an element of a coroot lattice.

Warning: This implementation only handles finite Cartan types

EXAMPLES:

```
sage: Qvee = RootSystem(['C',2]).coroot_lattice()
sage: Qvee.from_vector(vector([1,2])).max_quantum_element()
[2, 1, 2, 1]
sage: Qvee.from_vector(vector([1,1])).max_quantum_element()
[1, 2, 1]
sage: Qvee.from_vector(vector([0,2])).max_quantum_element()
[2]
```

quantum_root()

Returns True if `self` is a quantum root and False otherwise.

INPUT:

- `self` – an element of the nonnegative integer span of simple roots.

A root α is a quantum root if $\ell(s_\alpha) = \langle 2\rho, \alpha^\vee \rangle - 1$ where ℓ is the length function, s_α is the reflection across the hyperplane orthogonal to α, and 2ρ is the sum of positive roots.

Warning: This implementation only handles finite Cartan types and assumes that `self` is a root.

Todo

Rename to is_quantum_root

EXAMPLES:

```
sage: Q = RootSystem(['C',2]).root_lattice()
sage: positive_roots = Q.positive_roots()
sage: for x in positive_roots:
....:     print x, x.quantum_root()
alpha[1] True
alpha[2] True
2*alpha[1] + alpha[2] True
alpha[1] + alpha[2] False
```

scalar(*lambdacheck*)

The scalar product between the root lattice and the coroot lattice.

EXAMPLES:

```
sage: L = RootSystem(['B',4]).root_lattice()
sage: alpha      = L.simple_roots()
sage: alphacheck = L.simple_coroots()
sage: alpha[1].scalar(alphacheck[1])
2
sage: alpha[1].scalar(alphacheck[2])
-1
```

The scalar products between the roots and coroots are given by the Cartan matrix:

```
sage: matrix([ [ alpha[i].scalar(alphacheck[j])
...              for i in L.index_set() ]
...            for j in L.index_set() ])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -2  2]
```

```
sage: L.cartan_type().cartan_matrix()
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -2  2]
```

to_ambient()

Map `self` to the ambient space.

EXAMPLES:

```
sage: alpha = CartanType(['B',2]).root_system().root_lattice().an_element(); alpha
2*alpha[1] + 2*alpha[2]
sage: alpha.to_ambient()
(2, 0)
sage: alphavee = CartanType(['B',2]).root_system().coroot_lattice().an_element(); alphavee
2*alphacheck[1] + 2*alphacheck[2]
sage: alphavee.to_ambient()
(2, 2)
```

5.1.210 Root systems

See *Root Systems* for an overview.

class sage.combinat.root_system.root_system.**RootSystem**(*cartan_type*, *as_dual_of=None*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.sage_object.SageObject

A class for root systems.

EXAMPLES:

We construct the root system for type B_3:

```
sage: R=RootSystem(['B',3]); R
Root system of type ['B', 3]
```

`R` models the root system abstractly. It comes equipped with various realizations of the root and weight lattices, where all computations take place. Let us play first with the root lattice:

```
sage: space = R.root_lattice()
sage: space
Root lattice of the Root system of type ['B', 3]
```

This is the free $\mathbf{Z}$-module $\bigoplus_i \mathbf{Z}.\alpha_i$ spanned by the simple roots:

```
sage: space.base_ring()
Integer Ring
sage: list(space.basis())
[alpha[1], alpha[2], alpha[3]]
```

Let us do some computations with the simple roots:

```
sage: alpha = space.simple_roots()
sage: alpha[1] + alpha[2]
alpha[1] + alpha[2]
```

There is a canonical pairing between the root lattice and the coroot lattice:

```
sage: R.coroot_lattice()
Coroot lattice of the Root system of type ['B', 3]
```

We construct the simple coroots, and do some computations (see comments about duality below for some caveat):

```
sage: alphacheck = space.simple_coroots()
sage: list(alphacheck)
[alphacheck[1], alphacheck[2], alphacheck[3]]
```

We can carry over the same computations in any of the other realizations of the root lattice, like the root space $\bigoplus_i \mathbf{Q}.\alpha_i$, the weight lattice $\bigoplus_i \mathbf{Z}.\Lambda_i$, the weight space $\bigoplus_i \mathbf{Q}.\Lambda_i$. For example:

```
sage: space = R.weight_space()
sage: space
Weight space over the Rational Field of the Root system of type ['B', 3]

sage: space.base_ring()
Rational Field
sage: list(space.basis())
[Lambda[1], Lambda[2], Lambda[3]]

sage: alpha = space.simple_roots()
sage: alpha[1] + alpha[2]
Lambda[1] + Lambda[2] - 2*Lambda[3]
```

The fundamental weights are the dual basis of the coroots:

```
sage: Lambda = space.fundamental_weights()
sage: Lambda[1]
Lambda[1]

sage: alphacheck = space.simple_coroots()
sage: list(alphacheck)
[alphacheck[1], alphacheck[2], alphacheck[3]]

sage: [Lambda[i].scalar(alphacheck[1]) for i in space.index_set()]
[1, 0, 0]
sage: [Lambda[i].scalar(alphacheck[2]) for i in space.index_set()]
[0, 1, 0]
sage: [Lambda[i].scalar(alphacheck[3]) for i in space.index_set()]
[0, 0, 1]
```

Let us use the simple reflections. In the weight space, they work as in the *number game*: firing the node i on an element x adds c times the simple root α_i, where c is the coefficient of i in x:

```
sage: s = space.simple_reflections()
sage: Lambda[1].simple_reflection(1)
-Lambda[1] + Lambda[2]
sage: Lambda[2].simple_reflection(1)
Lambda[2]
sage: Lambda[3].simple_reflection(1)
Lambda[3]
sage: (-2*Lambda[1] + Lambda[2] + Lambda[3]).simple_reflection(1)
2*Lambda[1] - Lambda[2] + Lambda[3]
```

It can be convenient to manipulate the simple reflections themselves:

```
sage: s = space.simple_reflections()
sage: s[1](Lambda[1])
-Lambda[1] + Lambda[2]
sage: s[1](Lambda[2])
Lambda[2]
sage: s[1](Lambda[3])
Lambda[3]
```

Ambient spaces

The root system may also come equipped with an ambient space. This is a $\mathbf{Q}$-module, endowed with its canonical Euclidean scalar product, which admits simultaneous embeddings of the (extended) weight and the (extended) coweight lattice, and therefore the root and the coroot lattice. This is implemented on a type by type basis for the finite crystallographic root systems following Bourbaki's conventions and is extended to the affine cases. Coefficients permitting, this is also available as an ambient lattice.

See also:

`ambient_space()` and `ambient_lattice()` for details

In finite type A, we recover the natural representation of the symmetric group as group of permutation matrices:

```
sage: RootSystem(["A",2]).ambient_space().weyl_group().simple_reflections()
Finite family {1: [0 1 0]
                  [1 0 0]
                  [0 0 1],
               2: [1 0 0]
                  [0 0 1]
                  [0 1 0]}
```

In type B, C, and D, we recover the natural representation of the Weyl group as groups of signed permutation matrices:

```
sage: RootSystem(["B",3]).ambient_space().weyl_group().simple_reflections()
Finite family {1: [0 1 0]
                  [1 0 0]
                  [0 0 1],
               2: [1 0 0]
                  [0 0 1]
                  [0 1 0],
               3: [ 1  0  0]
                  [ 0  1  0]
                  [ 0  0 -1]}
```

In (untwisted) affine types A, ..., D, one can recover from the ambient space the affine permutation representation, in window notation. Let us consider the ambient space for affine type A:

```
sage: L = RootSystem(["A",2,1]).ambient_space(); L
Ambient space of the Root system of type ['A', 2, 1]
```

Define the "identity" by an appropriate vector at level -3:

```
sage: e = L.basis(); Lambda = L.fundamental_weights()
sage: id = e[0] + 2*e[1] + 3*e[2]  - 3*Lambda[0]
```

The corresponding permutation is obtained by projecting it onto the classical ambient space:

```
sage: L.classical()
Ambient space of the Root system of type ['A', 2]
sage: L.classical()(id)
(1, 2, 3)
```

Here is the orbit of the identity under the action of the finite group:

```
sage: W = L.weyl_group()
sage: S3 = [ w.action(id) for w in W.classical() ]
sage: [L.classical()(x) for x in S3]
[(1, 2, 3), (3, 2, 1), (3, 1, 2), (2, 1, 3), (2, 3, 1), (1, 3, 2)]
```

And the action of s_0 on these yields:

```
sage: s = W.simple_reflections()
sage: [L.classical()(s[0].action(x)) for x in S3]
[(0, 2, 4), (-2, 2, 6), (-1, 1, 6), (0, 1, 5), (-2, 3, 5), (-1, 3, 4)]
```

We can also plot various components of the ambient spaces:

```
sage: L = RootSystem(['A',2]).ambient_space()
sage: L.plot()
Graphics object consisting of 13 graphics primitives
```

For more on plotting, see *Tutorial: visualizing root systems*.

Dual root systems

The root system is aware of its dual root system:

```
sage: R.dual
Dual of root system of type ['B', 3]
```

`R.dual` is really the root system of type C_3:

```
sage: R.dual.cartan_type()
['C', 3]
```

And the coroot lattice that we have been manipulating before is really implemented as the root lattice of the dual root system:

```
sage: R.dual.root_lattice()
Coroot lattice of the Root system of type ['B', 3]
```

In particular, the coroots for the root lattice are in fact the roots of the coroot lattice:

```
sage: list(R.root_lattice().simple_coroots())
[alphacheck[1], alphacheck[2], alphacheck[3]]
sage: list(R.coroot_lattice().simple_roots())
[alphacheck[1], alphacheck[2], alphacheck[3]]
sage: list(R.dual.root_lattice().simple_roots())
[alphacheck[1], alphacheck[2], alphacheck[3]]
```

The coweight lattice and space are defined similarly. Note that, to limit confusion, all the output have been tweaked appropriately.

See also:

- `sage.combinat.root_system`

- RootSpace
- WeightSpace
- AmbientSpace
- RootLatticeRealizations
- WeightLatticeRealizations

TESTS:

```
sage: R = RootSystem(['C',3])
sage: TestSuite(R).run()
sage: L = R.ambient_space()
sage: s = L.simple_reflections() # this used to break the testsuite below due to caching an unpi
sage: s = L.simple_projections() # todo: not implemented
sage: TestSuite(L).run()
sage: L = R.root_space()
sage: s = L.simple_reflections()
sage: TestSuite(L).run()

sage: for T in CartanType.samples(crystallographic=True):  # long time (13s on sage.math, 2012)
...       TestSuite(RootSystem(T)).run()
```

ambient_lattice()

Return the ambient lattice for this root_system.

This is the ambient space, over **Z**.

See also:

- ambient_space()
- root_lattice()
- weight_lattice()

EXAMPLES:

```
sage: RootSystem(['A',4]).ambient_lattice()
Ambient lattice of the Root system of type ['A', 4]
sage: RootSystem(['A',4,1]).ambient_lattice()
Ambient lattice of the Root system of type ['A', 4, 1]
```

Except in type A, only an ambient space can be realized:

```
sage: RootSystem(['B',4]).ambient_lattice()
sage: RootSystem(['C',4]).ambient_lattice()
sage: RootSystem(['D',4]).ambient_lattice()
sage: RootSystem(['E',6]).ambient_lattice()
sage: RootSystem(['F',4]).ambient_lattice()
sage: RootSystem(['G',2]).ambient_lattice()
```

ambient_space(*base_ring=Rational Field*)

Return the usual ambient space for this root_system.

INPUT:

- base_ring – a base ring (default: **Q**)

This is a base_ring-module, endowed with its canonical Euclidean scalar product, which admits simultaneous embeddings into the weight and the coweight lattice, and therefore the root and the coroot lattice,

and preserves scalar products between elements of the coroot lattice and elements of the root or weight lattice (and dually).

There is no mechanical way to define the ambient space just from the Cartan matrix. Instead is is constructed from hard coded type by type data, according to the usual Bourbaki conventions. Such data is provided for all the finite (crystallographic) types. From this data, ambient spaces can be built as well for dual types, reducible types and affine types. When no data is available, or if the base ring is not large enough, None is returned.

Warning: for affine types

See also:

- The section on ambient spaces in RootSystem
- ambient_lattice()
- AmbientSpace
- AmbientSpace
- root_space()
- weight:space()

EXAMPLES:

```
sage: RootSystem(['A',4]).ambient_space()
Ambient space of the Root system of type ['A', 4]

sage: RootSystem(['B',4]).ambient_space()
Ambient space of the Root system of type ['B', 4]

sage: RootSystem(['C',4]).ambient_space()
Ambient space of the Root system of type ['C', 4]

sage: RootSystem(['D',4]).ambient_space()
Ambient space of the Root system of type ['D', 4]

sage: RootSystem(['E',6]).ambient_space()
Ambient space of the Root system of type ['E', 6]

sage: RootSystem(['F',4]).ambient_space()
Ambient space of the Root system of type ['F', 4]

sage: RootSystem(['G',2]).ambient_space()
Ambient space of the Root system of type ['G', 2]
```

An alternative base ring can be provided as an option:

```
sage: e = RootSystem(['B',3]).ambient_space(RR)
sage: TestSuite(e).run()
```

It should contain the smallest ring over which the ambient space can be defined ($\mathbf{Z}$ in type A or $\mathbf{Q}$ otherwise). Otherwise None is returned:

```
sage: RootSystem(['B',2]).ambient_space(ZZ)
```

The base ring should also be totally ordered. In practice, only $\mathbf{Z}$ and $\mathbf{Q}$ are really supported at this point, but you are welcome to experiment:

```
sage: e = RootSystem(['G',2]).ambient_space(RR)
sage: TestSuite(e).run()
Failure in _test_root_lattice_realization:
Traceback (most recent call last):
...
AssertionError: 2.00000000000000 != 2.00000000000000
----------------------------------------------------------------------
The following tests failed: _test_root_lattice_realization
```

cartan_matrix()

EXAMPLES:

```
sage: RootSystem(['A',3]).cartan_matrix()
[ 2 -1  0]
[-1  2 -1]
[ 0 -1  2]
```

cartan_type()

Returns the Cartan type of the root system.

EXAMPLES:

```
sage: R = RootSystem(['A',3])
sage: R.cartan_type()
['A', 3]
```

coambient_space(*base_ring=Rational Field*)

Return the coambient space for this root system.

This is the ambient space of the dual root system.

See also:

- ambient_space()

EXAMPLES:

```
sage: L = RootSystem(["B",2]).ambient_space(); L
Ambient space of the Root system of type ['B', 2]
sage: coL = RootSystem(["B",2]).coambient_space(); coL
Coambient space of the Root system of type ['B', 2]
```

The roots and coroots are interchanged:

```
sage: coL.simple_roots()
Finite family {1: (1, -1), 2: (0, 2)}
sage: L.simple_coroots()
Finite family {1: (1, -1), 2: (0, 2)}

sage: coL.simple_coroots()
Finite family {1: (1, -1), 2: (0, 1)}
sage: L.simple_roots()
Finite family {1: (1, -1), 2: (0, 1)}
```

coroot_lattice()

Returns the coroot lattice associated to self.

EXAMPLES:

```
sage: RootSystem(['A',3]).coroot_lattice()
Coroot lattice of the Root system of type ['A', 3]
```

coroot_space(*base_ring=Rational Field*)
Returns the coroot space associated to self.

EXAMPLES:

```
sage: RootSystem(['A',3]).coroot_space()
Coroot space over the Rational Field of the Root system of type ['A', 3]
```

coweight_lattice(*extended=False*)
Returns the coweight lattice associated to self.

This is the weight lattice of the dual root system.

EXAMPLES:

```
sage: RootSystem(['A',3]).coweight_lattice()
Coweight lattice of the Root system of type ['A', 3]

sage: RootSystem(['A',3,1]).coweight_lattice(extended = True)
Extended coweight lattice of the Root system of type ['A', 3, 1]
```

coweight_space(*base_ring=Rational Field*, *extended=False*)
Returns the coweight space associated to self.

This is the weight space of the dual root system.

EXAMPLES:

```
sage: RootSystem(['A',3]).coweight_space()
Coweight space over the Rational Field of the Root system of type ['A', 3]

sage: RootSystem(['A',3,1]).coweight_space(extended=True)
Extended coweight space over the Rational Field of the Root system of type ['A', 3, 1]
```

dynkin_diagram()
Returns the Dynkin diagram of the root system.

EXAMPLES:

```
sage: R = RootSystem(['A',3])
sage: R.dynkin_diagram()
O---O---O
1   2   3
A3
```

index_set()
EXAMPLES:

```
sage: RootSystem(['A',3]).index_set()
(1, 2, 3)
```

is_finite()
Returns True if self is a finite root system.

EXAMPLES:

```
sage: RootSystem(["A",3]).is_finite()
True
```

```
sage: RootSystem(["A",3,1]).is_finite()
False
```

is_irreducible()

Returns True if self is an irreducible root system.

EXAMPLES:

```
sage: RootSystem(['A', 3]).is_irreducible()
True
sage: RootSystem("A2xB2").is_irreducible()
False
```

root_lattice()

Returns the root lattice associated to self.

EXAMPLES:

```
sage: RootSystem(['A',3]).root_lattice()
Root lattice of the Root system of type ['A', 3]
```

root_poset(*restricted=False*, *facade=False*)

Returns the (restricted) root poset associated to `self`.

The elements are given by the positive roots (resp. non-simple, positive roots), and $\alpha \leq \beta$ iff $\beta - \alpha$ is a non-negative linear combination of simple roots.

INPUT:

- `restricted` – (default:False) if True, only non-simple roots are considered.
- `facade` – (default:False) passes facade option to the poset generator.

EXAMPLES:

```
sage: Phi = RootSystem(['A',2]).root_poset(); Phi
Finite poset containing 3 elements
sage: sorted(Phi.cover_relations(), key=str)
[[alpha[1], alpha[1] + alpha[2]], [alpha[2], alpha[1] + alpha[2]]]

sage: Phi = RootSystem(['A',3]).root_poset(restricted=True); Phi
Finite poset containing 3 elements
sage: sorted(Phi.cover_relations(), key=str)
[[alpha[1] + alpha[2], alpha[1] + alpha[2] + alpha[3]], [alpha[2] + alpha[3], alpha[1] + alp

sage: Phi = RootSystem(['B',2]).root_poset(); Phi
Finite poset containing 4 elements
sage: Phi.cover_relations()
[[alpha[2], alpha[1] + alpha[2]], [alpha[1], alpha[1] + alpha[2]], [alpha[1] + alpha[2], alp
```

root_space(*base_ring=Rational Field*)

Returns the root space associated to self.

EXAMPLES:

```
sage: RootSystem(['A',3]).root_space()
Root space over the Rational Field of the Root system of type ['A', 3]
```

weight_lattice(*extended=False*)

Returns the weight lattice associated to self.

EXAMPLES:

```
sage: RootSystem(['A',3]).weight_lattice()
Weight lattice of the Root system of type ['A', 3]

sage: RootSystem(['A',3,1]).weight_space(extended = True)
Extended weight space over the Rational Field of the Root system of type ['A', 3, 1]
```

weight_space(*base_ring=Rational Field*, *extended=False*)

Returns the weight space associated to self.

EXAMPLES:

```
sage: RootSystem(['A',3]).weight_space()
Weight space over the Rational Field of the Root system of type ['A', 3]

sage: RootSystem(['A',3,1]).weight_space(extended = True)
Extended weight space over the Rational Field of the Root system of type ['A', 3, 1]
```

sage.combinat.root_system.root_system.**WeylDim**(*ct*, *coeffs*)

The Weyl Dimension Formula.

INPUT:

- `type` - a Cartan type
- `coeffs` - a list of nonnegative integers

The length of the list must equal the rank type[1]. A dominant weight hwv is constructed by summing the fundamental weights with coefficients from this list. The dimension of the irreducible representation of the semisimple complex Lie algebra with highest weight vector hwv is returned.

EXAMPLES:

For $SO(7)$, the Cartan type is B_3, so:

```
sage: WeylDim(['B',3],[1,0,0]) # standard representation of SO(7)
7
sage: WeylDim(['B',3],[0,1,0]) # exterior square
21
sage: WeylDim(['B',3],[0,0,1]) # spin representation of spin(7)
8
sage: WeylDim(['B',3],[1,0,1]) # sum of the first and third fundamental weights
48
sage: [WeylDim(['F',4],x) for x in [1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]]
[52, 1274, 273, 26]
sage: [WeylDim(['E', 6], x) for x in [0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1],
[1, 78, 27, 351, 351, 351, 27, 650, 351]
```

5.1.211 Root system data for type A

class sage.combinat.root_system.type_A.**AmbientSpace**(*root_system*, *base_ring*)

Bases: sage.combinat.root_system.ambient_space.AmbientSpace

EXAMPLES:

```
sage: R = RootSystem(["A",3])
sage: e = R.ambient_space(); e
Ambient space of the Root system of type ['A', 3]
sage: TestSuite(e).run()
```

By default, this ambient space uses the barycentric projection for plotting:

```
sage: L = RootSystem(["A",2]).ambient_space()
sage: e = L.basis()
sage: L._plot_projection(e[0])
(1/2, 989/1142)
sage: L._plot_projection(e[1])
(-1, 0)
sage: L._plot_projection(e[2])
(1/2, -989/1142)
sage: L = RootSystem(["A",3]).ambient_space()
sage: l = L.an_element(); l
(2, 2, 3, 0)
sage: L._plot_projection(l)
(0, -1121/1189, 7/3)
```

See also:

- sage.combinat.root_system.root_lattice_realizations.RootLatticeRealizations.ParentM

det(*k=1*)

returns the vector (1, ... ,1) which in the ['A',r] weight lattice, interpreted as a weight of GL(r+1,CC) is the determinant. If the optional parameter k is given, returns (k, ... ,k), the k-th power of the determinant.

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_space()
sage: e.det(1/2)
(1/2, 1/2, 1/2, 1/2)
```

dimension()

EXAMPLES:

```
sage: e = RootSystem(["A",3]).ambient_space()
sage: e.dimension()
4
```

fundamental_weight(*i*)

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: e.fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1, 1, 1, 0)}
```

highest_root()

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: e.highest_root()
(1, 0, 0, -1)
```

negative_roots()

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: e.negative_roots()
[(-1, 1, 0, 0),
 (-1, 0, 1, 0),
 (-1, 0, 0, 1),
 (0, -1, 1, 0),
```

```
 (0, -1, 0, 1),
 (0, 0, -1, 1)]
```

positive_roots()

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: e.positive_roots()
[(1, -1, 0, 0),
 (1, 0, -1, 0),
 (0, 1, -1, 0),
 (1, 0, 0, -1),
 (0, 1, 0, -1),
 (0, 0, 1, -1)]
```

root(i, j)

Note that indexing starts at 0.

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: e.root(0,1)
(1, -1, 0, 0)
```

simple_root(i)

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: e.simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1)}
```

classmethod smallest_base_ring(*cartan_type=None*)

Returns the smallest base ring the ambient space can be defined upon

See also:

smallest_base_ring()

EXAMPLES:

```
sage: e = RootSystem(["A",3]).ambient_space()
sage: e.smallest_base_ring()
Integer Ring
```

class sage.combinat.root_system.type_A.**CartanType**(n)

Bases: sage.combinat.root_system.cartan_type.CartanType_standard_finite, sage.combinat.root_system.cartan_type.CartanType_simply_laced, sage.combinat.root_system.cartan_type.CartanType_simple

Cartan Type A_n

See also:

CartanType()

AmbientSpace

alias of AmbientSpace

PieriFactors

alias of PieriFactors_type_A

ascii_art(*label=<function <lambda>>*, *node=None*)
Return an ascii art representation of the Dynkin diagram.

EXAMPLES:

```
sage: print CartanType(['A',0]).ascii_art()
sage: print CartanType(['A',1]).ascii_art()
O
1
sage: print CartanType(['A',3]).ascii_art()
O---O---O
1   2   3
sage: print CartanType(['A',12]).ascii_art()
O---O---O---O---O---O---O---O---O---O---O---O
1   2   3   4   5   6   7   8   9   10  11  12
sage: print CartanType(['A',5]).ascii_art(label = lambda x: x+2)
O---O---O---O---O
3   4   5   6   7
sage: print CartanType(['A',5]).ascii_art(label = lambda x: x-2)
O---O---O---O---O
-1  0   1   2   3
```

coxeter_number()
Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['A',4]).coxeter_number()
5
```

dual_coxeter_number()
Return the dual Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['A',4]).dual_coxeter_number()
5
```

dynkin_diagram()
Returns the Dynkin diagram of type A.

EXAMPLES:

```
sage: a = CartanType(['A',3]).dynkin_diagram()
sage: a
O---O---O
1   2   3
A3
sage: sorted(a.edges())
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 1)]
```

TEST:

```
sage: a = DynkinDiagram(['A',1])
sage: a
O
1
A1
sage: a.vertices(), a.edges()
([1], [])
```

5.1.212 Root system data for (untwisted) type A affine

class sage.combinat.root_system.type_A_affine.**CartanType**(*n*)

Bases: sage.combinat.root_system.cartan_type.CartanType_standard_untwisted_affine

EXAMPLES:

```
sage: ct = CartanType(['A',4,1])
sage: ct
['A', 4, 1]
sage: ct._repr_(compact = True)
'A4~'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_untwisted_affine()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
True
sage: ct.classical()
['A', 4]
sage: ct.dual()
['A', 4, 1]

sage: ct = CartanType(['A', 1, 1])
sage: ct.is_simply_laced()
False
sage: ct.dual()
['A', 1, 1]
```

TESTS:

```
sage: TestSuite(ct).run()
```

PieriFactors

alias of PieriFactors_type_A_affine

ascii_art(*label=<function <lambda>>*, *node=None*)

Return an ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print CartanType(['A',3,1]).ascii_art()
0
O-------+
|       |
|       |
O---O---O
1   2   3

sage: print CartanType(['A',5,1]).ascii_art(label = lambda x: x+2)
2
O---------------+
|               |
|               |
```

```
O---O---O---O---O
3   4   5   6   7

sage: print CartanType(['A',1,1]).ascii_art()
O<=>O
0   1

sage: print CartanType(['A',1,1]).ascii_art(label = lambda x: x+2)
O<=>O
2   3
```

dual()

Type A_1^1 is self dual despite not being simply laced.

EXAMPLES:

```
sage: CartanType(['A',1,1]).dual()
['A', 1, 1]
```

dynkin_diagram()

Returns the extended Dynkin diagram for affine type A.

EXAMPLES:

```
sage: a = CartanType(['A',3,1]).dynkin_diagram()
sage: a
 0
 O-------+
 |       |
 |       |
 O---O---O
 1   2   3
 A3~
sage: sorted(a.edges())
[(0, 1, 1),
 (0, 3, 1),
 (1, 0, 1),
 (1, 2, 1),
 (2, 1, 1),
 (2, 3, 1),
 (3, 0, 1),
 (3, 2, 1)]

sage: a = DynkinDiagram(['A',1,1])
sage: a
O<=>O
0   1
A1~
sage: sorted(a.edges())
[(0, 1, 2), (1, 0, 2)]
```

5.1.213 Root system data for type B

class sage.combinat.root_system.type_B.**AmbientSpace**(*root_system*, *base_ring*)

Bases: sage.combinat.root_system.ambient_space.AmbientSpace

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: s = e.simple_reflections()

sage: L = RootSystem(['A',3]).coroot_lattice()
sage: e.has_coerce_map_from(L)
True
sage: e(L.simple_root(1))
(1, -1, 0, 0)
```

dimension()

EXAMPLES:

```
sage: e = RootSystem(['B',3]).ambient_space()
sage: e.dimension()
3
```

fundamental_weight(*i*)

EXAMPLES:

```
sage: RootSystem(['B',3]).ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
```

negative_roots()

EXAMPLES:

```
sage: RootSystem(['B',3]).ambient_space().negative_roots()
[(-1, 1, 0),
 (-1, -1, 0),
 (-1, 0, 1),
 (-1, 0, -1),
 (0, -1, 1),
 (0, -1, -1),
 (-1, 0, 0),
 (0, -1, 0),
 (0, 0, -1)]
```

positive_roots()

EXAMPLES:

```
sage: RootSystem(['B',3]).ambient_space().positive_roots()
[(1, -1, 0),
 (1, 1, 0),
 (1, 0, -1),
 (1, 0, 1),
 (0, 1, -1),
 (0, 1, 1),
 (1, 0, 0),
 (0, 1, 0),
 (0, 0, 1)]
```

root(*i*, *j*)

Note that indexing starts at 0.

EXAMPLES:

```
sage: e = RootSystem(['B',3]).ambient_space()
sage: e.root(0,1)
(1, -1, 0)
```

simple_root(*i*)

EXAMPLES:

```
sage: e = RootSystem(['B',4]).ambient_space()
sage: e.simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1), 4: (0, 0, 0, 1)}
sage: e.positive_roots()
[(1, -1, 0, 0),
(1, 1, 0, 0),
(1, 0, -1, 0),
(1, 0, 1, 0),
(1, 0, 0, -1),
(1, 0, 0, 1),
(0, 1, -1, 0),
(0, 1, 1, 0),
(0, 1, 0, -1),
(0, 1, 0, 1),
(0, 0, 1, -1),
(0, 0, 1, 1),
(1, 0, 0, 0),
(0, 1, 0, 0),
(0, 0, 1, 0),
(0, 0, 0, 1)]
sage: e.fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1, 1, 1, 0), 4: (1/2, 1/2, 1/2, 1/2)}
```

class sage.combinat.root_system.type_B.**CartanType**(*n*)

Bases: sage.combinat.root_system.cartan_type.CartanType_standard_finite, sage.combinat.root_system.cartan_type.CartanType_simple, sage.combinat.root_system.cartan_type.CartanType_crystallographic

EXAMPLES:

```
sage: ct = CartanType(['B',4])
sage: ct
['B', 4]
sage: ct._repr_(compact = True)
'B4'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
True
sage: ct.is_affine()
False
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.affine()
['B', 4, 1]
sage: ct.dual()
['C', 4]

sage: ct = CartanType(['B',1])
sage: ct.is_simply_laced()
True
sage: ct.affine()
['B', 1, 1]
```

TESTS:

```
sage: TestSuite(ct).run()
```

AmbientSpace
alias of AmbientSpace

PieriFactors
alias of PieriFactors_type_B

ascii_art(*label=<function <lambda>>*, *node=None*)
Return an ascii art representation of the Dynkin diagram.

EXAMPLES:

```
sage: print CartanType(['B',1]).ascii_art()
O
1
sage: print CartanType(['B',2]).ascii_art()
O=>=O
1   2
sage: print CartanType(['B',5]).ascii_art(label = lambda x: x+2)
O---O---O---O=>=O
3   4   5   6   7
```

coxeter_number()
Return the Coxeter number associated with self.

EXAMPLES:

```
sage: CartanType(['B',4]).coxeter_number()
8
```

dual()
Types B and C are in duality:

EXAMPLES:

```
sage: CartanType(["C", 3]).dual()
['B', 3]
```

dual_coxeter_number()
Return the dual Coxeter number associated with self.

EXAMPLES:

```
sage: CartanType(['B',4]).dual_coxeter_number()
7
```

dynkin_diagram()
Returns a Dynkin diagram for type B.

EXAMPLES:

```
sage: b = CartanType(['B',3]).dynkin_diagram()
sage: b
O---O=>=O
1   2   3
B3
sage: sorted(b.edges())
[(1, 2, 1), (2, 1, 1), (2, 3, 2), (3, 2, 1)]

sage: b = CartanType(['B',1]).dynkin_diagram()
sage: b
```

```
O
1
B1
sage: sorted(b.edges())
[]
```

5.1.214 Root system data for type BC affine

class sage.combinat.root_system.type_BC_affine.**CartanType**(*n*)

Bases: sage.combinat.root_system.cartan_type.CartanType_standard_affine

EXAMPLES:

```
sage: ct = CartanType(['BC',4,2])
sage: ct
['BC', 4, 2]
sage: ct._repr_(compact = True)
'BC4~'
sage: ct.dynkin_diagram()
O=<=O---O---O=<=O
0   1   2   3   4
BC4~

sage: ct.is_irreducible()
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.classical()
['C', 4]

sage: dual = ct.dual()
sage: dual.dynkin_diagram()
O=>=O---O---O=>=O
0   1   2   3   4
BC4~*

sage: dual.special_node()
0
sage: dual.classical().dynkin_diagram()
O---O---O=>=O
1   2   3   4
B4

sage: CartanType(['BC',1,2]).dynkin_diagram()
  4
O=<=O
0   1
BC1~
```

TESTS:

```
sage: TestSuite(ct).run()
```

ascii_art(*label=<function <lambda>>*, *node=None*)

Return a ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print CartanType(['BC',2,2]).ascii_art()
O=<=O=<=O
0   1   2
sage: print CartanType(['BC',3,2]).ascii_art()
O=<=O---O=<=O
0   1   2   3
sage: print CartanType(['BC',5,2]).ascii_art(label = lambda x: x+2)
O=<=O---O---O---O=<=O
2   3   4   5   6   7

sage: print CartanType(['BC',1,2]).ascii_art(label = lambda x: x+2)
  4
O=<=O
2   3
```

basic_untwisted()

Return the basic untwisted Cartan type associated with this affine Cartan type.

Given an affine type $X_n^{(r)}$, the basic untwisted type is X_n. In other words, it is the classical Cartan type that is twisted to obtain `self`.

EXAMPLES:

```
sage: CartanType(['A', 2, 2]).basic_untwisted()
['A', 2]
sage: CartanType(['A', 4, 2]).basic_untwisted()
['A', 4]
sage: CartanType(['BC', 4, 2]).basic_untwisted()
['A', 8]
```

classical()

Returns the classical Cartan type associated with self

sage: CartanType([”BC”, 3, 2]).classical() [’C’, 3]

dynkin_diagram()

Returns the extended Dynkin diagram for affine type BC.

EXAMPLES:

```
sage: c = CartanType(['BC',3,2]).dynkin_diagram()
sage: c
O=<=O---O=<=O
0   1   2   3
BC3~
sage: sorted(c.edges())
[(0, 1, 1), (1, 0, 2), (1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]

sage: c = CartanType(["A", 6, 2]).dynkin_diagram() # should be the same as above; did fail a
sage: c
O=<=O---O=<=O
0   1   2   3
BC3~
sage: sorted(c.edges())
```

```
[(0, 1, 1), (1, 0, 2), (1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]

sage: c = CartanType(['BC',2,2]).dynkin_diagram()
sage: c
O=<=O=<=O
0   1   2
BC2~
sage: sorted(c.edges())
[(0, 1, 1), (1, 0, 2), (1, 2, 1), (2, 1, 2)]

sage: c = CartanType(['BC',1,2]).dynkin_diagram()
sage: c
  4
O=<=O
0   1
BC1~
sage: sorted(c.edges())
[(0, 1, 1), (1, 0, 4)]
```

5.1.215 Root system data for (untwisted) type B affine

class sage.combinat.root_system.type_B_affine.**CartanType**(*n*)

Bases: sage.combinat.root_system.cartan_type.CartanType_standard_untwisted_affine

EXAMPLES:

```
sage: ct = CartanType(['B',4,1])
sage: ct
['B', 4, 1]
sage: ct._repr_(compact = True)
'B4~'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_untwisted_affine()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.classical()
['B', 4]
sage: ct.dual()
['B', 4, 1]^*
sage: ct.dual().is_untwisted_affine()
False
```

TESTS:

```
sage: TestSuite(ct).run()
```

PieriFactors

alias of PieriFactors_type_B_affine

ascii_art(*label=<function <lambda>>*, *node=None*)

Return an ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print CartanType(['B',3,1]).ascii_art()
    O 0
    |
    |
O---O=>=O
1   2   3

sage: print CartanType(['B',5,1]).ascii_art(label = lambda x: x+2)
    O 2
    |
    |
O---O---O---O=>=O
3   4   5   6   7

sage: print CartanType(['B',2,1]).ascii_art(label = lambda x: x+2)
O=>=O=<=O
2   4   3
sage: print CartanType(['B',1,1]).ascii_art(label = lambda x: x+2)
O<=>O
2   3
```

dynkin_diagram()

Return the extended Dynkin diagram for affine type B.

EXAMPLES:

```
sage: b = CartanType(['B',3,1]).dynkin_diagram()
sage: b
    O 0
    |
    |
O---O=>=O
1   2   3
B3~
sage: sorted(b.edges())
[(0, 2, 1), (1, 2, 1), (2, 0, 1), (2, 1, 1), (2, 3, 2), (3, 2, 1)]

sage: b = CartanType(['B',2,1]).dynkin_diagram(); b
O=>=O=<=O
0   2   1
B2~
sage: sorted(b.edges())
[(0, 2, 2), (1, 2, 2), (2, 0, 1), (2, 1, 1)]

sage: b = CartanType(['B',1,1]).dynkin_diagram(); b
O<=>O
0   1
B1~
sage: sorted(b.edges())
[(0, 1, 2), (1, 0, 2)]
```

5.1.216 Root system data for type C

class sage.combinat.root_system.type_C.**AmbientSpace**(*root_system*, *base_ring*)
Bases: sage.combinat.root_system.ambient_space.AmbientSpace

EXAMPLES:

```
sage: e = RootSystem(['C',2]).ambient_space(); e
Ambient space of the Root system of type ['C', 2]
```

One cannot construct the ambient lattice because the fundamental coweights have rational coefficients:

```
sage: e.smallest_base_ring()
Rational Field

sage: RootSystem(['B',2]).ambient_space().fundamental_weights()
Finite family {1: (1, 0), 2: (1/2, 1/2)}
```

TESTS:

```
sage: TestSuite(e).run()
```

dimension()
EXAMPLES:

```
sage: e = RootSystem(['C',3]).ambient_space()
sage: e.dimension()
3
```

fundamental_weight(*i*)
EXAMPLES:

```
sage: RootSystem(['C',3]).ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1, 1, 1)}
```

negative_roots()
EXAMPLES:

```
sage: RootSystem(['C',3]).ambient_space().negative_roots()
[(-1, 1, 0),
 (-1, 0, 1),
 (0, -1, 1),
 (-1, -1, 0),
 (-1, 0, -1),
 (0, -1, -1),
 (-2, 0, 0),
 (0, -2, 0),
 (0, 0, -2)]
```

positive_roots()
EXAMPLES:

```
sage: RootSystem(['C',3]).ambient_space().positive_roots()
[(1, 1, 0),
 (1, 0, 1),
 (0, 1, 1),
 (1, -1, 0),
 (1, 0, -1),
 (0, 1, -1),
 (2, 0, 0),
 (0, 2, 0),
```

```
(0, 0, 2)]
```

root(*i*, *j*, *p1*, *p2*)

Note that indexing starts at 0.

EXAMPLES:

```
sage: e = RootSystem(['C',3]).ambient_space()
sage: e.root(0, 1, 1, 1)
(-1, -1, 0)
```

simple_root(*i*)

EXAMPLES:

```
sage: RootSystem(['C',3]).ambient_space().simple_roots()
Finite family {1: (1, -1, 0), 2: (0, 1, -1), 3: (0, 0, 2)}
```

class sage.combinat.root_system.type_C.**CartanType**(*n*)

Bases: sage.combinat.root_system.cartan_type.CartanType_standard_finite, sage.combinat.root_system.cartan_type.CartanType_simple, sage.combinat.root_system.cartan_type.CartanType_crystallographic

EXAMPLES:

```
sage: ct = CartanType(['C',4])
sage: ct
['C', 4]
sage: ct._repr_(compact = True)
'C4'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.affine()
['C', 4, 1]
sage: ct.dual()
['B', 4]

sage: ct = CartanType(['C',1])
sage: ct.is_simply_laced()
True
sage: ct.affine()
['C', 1, 1]
```

TESTS:

```
sage: TestSuite(ct).run()
```

AmbientSpace

alias of AmbientSpace

ascii_art(*label=<function <lambda>>*, *node=None*)

Return a ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print CartanType(['C',1]).ascii_art()
O
1
sage: print CartanType(['C',2]).ascii_art()
O=<=O
1   2
sage: print CartanType(['C',3]).ascii_art()
O---O=<=O
1   2   3
sage: print CartanType(['C',5]).ascii_art(label = lambda x: x+2)
O---O---O---O=<=O
3   4   5   6   7
```

coxeter_number()

Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['C',4]).coxeter_number()
8
```

dual()

Types B and C are in duality:

EXAMPLES:

```
sage: CartanType(["C", 3]).dual()
['B', 3]
```

dual_coxeter_number()

Return the dual Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['C',4]).dual_coxeter_number()
5
```

dynkin_diagram()

Returns a Dynkin diagram for type C.

EXAMPLES:

```
sage: c = CartanType(['C',3]).dynkin_diagram()
sage: c
O---O=<=O
1   2   3
C3
sage: e = c.edges(); e.sort(); e
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]

 sage: b = CartanType(['C',1]).dynkin_diagram()
 sage: b
 O
 1
 C1
 sage: sorted(b.edges())
 []
```

5.1.217 Root system data for (untwisted) type C affine

class sage.combinat.root_system.type_C_affine.**CartanType**(*n*)

Bases: sage.combinat.root_system.cartan_type.CartanType_standard_untwisted_affine

EXAMPLES:

```
sage: ct = CartanType(['C',4,1])
sage: ct
['C', 4, 1]
sage: ct._repr_(compact = True)
'C4~'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_untwisted_affine()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.classical()
['C', 4]
sage: ct.dual()
['C', 4, 1]^*
sage: ct.dual().is_untwisted_affine()
False
```

TESTS:

```
sage: TestSuite(ct).run()
```

PieriFactors

alias of PieriFactors_type_C_affine

ascii_art(*label=<function <lambda>>*, *node=None*)

Return a ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print CartanType(['C',5,1]).ascii_art(label = lambda x: x+2)
O=>=O---O---O---O=<=O
2   3   4   5   6   7

sage: print CartanType(['C',3,1]).ascii_art()
O=>=O---O=<=O
0   1   2   3

sage: print CartanType(['C',2,1]).ascii_art()
O=>=O=<=O
0   1   2

sage: print CartanType(['C',1,1]).ascii_art()
O<=>O
0   1
```

dynkin_diagram()

Returns the extended Dynkin diagram for affine type C.

EXAMPLES:

```
sage: c = CartanType(['C',3,1]).dynkin_diagram()
sage: c
 O=>=O---O=<=O
 0   1   2   3
 C3~
sage: sorted(c.edges())
[(0, 1, 2), (1, 0, 1), (1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]
```

5.1.218 Root system data for type D

class sage.combinat.root_system.type_D.**AmbientSpace**(*root_system*, *base_ring*)

Bases: sage.combinat.root_system.ambient_space.AmbientSpace

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: s = e.simple_reflections()

sage: L = RootSystem(['A',3]).coroot_lattice()
sage: e.has_coerce_map_from(L)
True
sage: e(L.simple_root(1))
(1, -1, 0, 0)
```

dimension()

EXAMPLES:

```
sage: e = RootSystem(['D',3]).ambient_space()
sage: e.dimension()
3
```

fundamental_weight(*i*)

EXAMPLES:

```
sage: RootSystem(['D',4]).ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1/2, 1/2, 1/2, -1/2), 4: (1/2, 1/2, 1/2
```

negative_roots()

EXAMPLES:

```
sage: RootSystem(['D',4]).ambient_space().negative_roots()
[(-1, 1, 0, 0),
 (-1, 0, 1, 0),
 (0, -1, 1, 0),
 (-1, 0, 0, 1),
 (0, -1, 0, 1),
 (0, 0, -1, 1),
 (-1, -1, 0, 0),
 (-1, 0, -1, 0),
 (0, -1, -1, 0),
 (-1, 0, 0, -1),
 (0, -1, 0, -1),
 (0, 0, -1, -1)]
```

positive_roots()

EXAMPLES:

```
sage: RootSystem(['D',4]).ambient_space().positive_roots()
[(1, 1, 0, 0),
 (1, 0, 1, 0),
 (0, 1, 1, 0),
 (1, 0, 0, 1),
 (0, 1, 0, 1),
 (0, 0, 1, 1),
 (1, -1, 0, 0),
 (1, 0, -1, 0),
 (0, 1, -1, 0),
 (1, 0, 0, -1),
 (0, 1, 0, -1),
 (0, 0, 1, -1)]
```

root(*i*, *j*, *p1*, *p2*)

Note that indexing starts at 0.

EXAMPLES:

```
sage: e = RootSystem(['D',3]).ambient_space()
sage: e.root(0, 1, 1, 1)
(-1, -1, 0)
sage: e.root(0, 0, 1, 1)
(-1, 0, 0)
```

simple_root(*i*)

EXAMPLES:

```
sage: RootSystem(['D',4]).ambient_space().simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1), 4: (0, 0, 1, 1)}
```

class sage.combinat.root_system.type_D.**CartanType**(*n*)

Bases: sage.combinat.root_system.cartan_type.CartanType_standard_finite, sage.combinat.root_system.cartan_type.CartanType_simply_laced

EXAMPLES:

```
sage: ct = CartanType(['D',4])
sage: ct
['D', 4]
sage: ct._repr_(compact = True)
'D4'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
True
sage: ct.dual()
['D', 4]
sage: ct.affine()
['D', 4, 1]

sage: ct = CartanType(['D',2])
sage: ct.is_irreducible()
False
```

```
sage: ct.dual()
['D', 2]
sage: ct.affine()
Traceback (most recent call last):
...
ValueError: ['D', 2, 1] is not a valid Cartan type
```

TESTS:

```
sage: TestSuite(ct).run()
```

AmbientSpace
alias of AmbientSpace

ascii_art(*label=<function <lambda>>*, *node=None*)
Return a ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print CartanType(['D',3]).ascii_art()
O 3
|
|
O---O
1   2
sage: print CartanType(['D',4]).ascii_art()
    O 4
    |
    |
O---O---O
1   2   3
sage: print CartanType(['D',4]).ascii_art(label = lambda x: x+2)
    O 6
    |
    |
O---O---O
3   4   5
sage: print CartanType(['D',6]).ascii_art(label = lambda x: x+2)
            O 8
            |
            |
O---O---O---O---O
3   4   5   6   7
```

coxeter_number()
Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['D',4]).coxeter_number()
6
```

dual_coxeter_number()
Return the dual Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['D',4]).dual_coxeter_number()
6
```

dynkin_diagram()
Returns a Dynkin diagram for type D.

EXAMPLES:

```
sage: d = CartanType(['D',5]).dynkin_diagram(); d
        O 5
        |
        |
O---O---O---O
1   2   3   4
D5
sage: sorted(d.edges())
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 1), (3, 4, 1), (3, 5, 1), (4, 3, 1), (5, 3, 1)]

sage: d = CartanType(['D',4]).dynkin_diagram(); d
    O 4
    |
    |
O---O---O
1   2   3
D4
sage: sorted(d.edges())
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (2, 4, 1), (3, 2, 1), (4, 2, 1)]

sage: d = CartanType(['D',3]).dynkin_diagram(); d
O 3
|
|
O---O
1   2
D3
sage: sorted(d.edges())
[(1, 2, 1), (1, 3, 1), (2, 1, 1), (3, 1, 1)]


sage: d = CartanType(['D',2]).dynkin_diagram(); d
O   O
1   2
D2
sage: sorted(d.edges())
[]
```

is_atomic()
Implements `CartanType_abstract.is_atomic()`

D_2 is atomic, like all D_n, despite being non irreducible.

EXAMPLES:

```
sage: CartanType(["D",2]).is_atomic()
True
sage: CartanType(["D",2]).is_irreducible()
False
```

5.1.219 Root system data for (untwisted) type D affine

class sage.combinat.root_system.type_D_affine.**CartanType**(*n*)
Bases: sage.combinat.root_system.cartan_type.CartanType_standard_untwisted_affine,

sage.combinat.root_system.cartan_type.CartanType_simply_laced

EXAMPLES:

```
sage: ct = CartanType(['D',4,1])
sage: ct
['D', 4, 1]
sage: ct._repr_(compact = True)
'D4~'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_untwisted_affine()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
True
sage: ct.classical()
['D', 4]
sage: ct.dual()
['D', 4, 1]
```

TESTS:

```
sage: TestSuite(ct).run()
```

PieriFactors

alias of PieriFactors_type_D_affine

ascii_art(*label=<function <lambda>>*, *node=None*)

Return an ascii art representation of the extended Dynkin diagram.

TESTS:

```
sage: print CartanType(['D',6,1]).ascii_art(label = lambda x: x+2)
  2 O       O 8
    |       |
    |       |
O---O---O---O---O
3   4   5   6   7

sage: print CartanType(['D',4,1]).ascii_art(label = lambda x: x+2)
    O 6
    |
    |
O---O---O
3   |4  5
    |
    O 2

sage: print CartanType(['D',3,1]).ascii_art(label = lambda x: x+2)
2
O-------+
|       |
|       |
O---O---O
```

```
5   3   4
```

dynkin_diagram()

Returns the extended Dynkin diagram for affine type D.

EXAMPLES:

```
sage: d = CartanType(['D', 6, 1]).dynkin_diagram()
sage: d
   0 O       O 6
     |       |
     |       |
 O---O---O---O---O
 1   2   3   4   5
 D6~
sage: sorted(d.edges())
[(0, 2, 1), (1, 2, 1), (2, 0, 1), (2, 1, 1), (2, 3, 1),
 (3, 2, 1), (3, 4, 1), (4, 3, 1), (4, 5, 1), (4, 6, 1), (5, 4, 1), (6, 4, 1)]

sage: d = CartanType(['D', 4, 1]).dynkin_diagram()
sage: d
    O 4
    |
    |
O---O---O
1   |2  3
    |
    O 0
D4~
sage: sorted(d.edges())
[(0, 2, 1),
 (1, 2, 1),
 (2, 0, 1),
 (2, 1, 1),
 (2, 3, 1),
 (2, 4, 1),
 (3, 2, 1),
 (4, 2, 1)]

sage: d = CartanType(['D', 3, 1]).dynkin_diagram()
sage: d
0
O-------+
|       |
|       |
O---O---O
3   1   2
D3~
sage: sorted(d.edges())
[(0, 2, 1), (0, 3, 1), (1, 2, 1), (1, 3, 1), (2, 0, 1), (2, 1, 1), (3, 0, 1), (3, 1, 1)]
```

5.1.220 Root system data for type E

class sage.combinat.root_system.type_E.**AmbientSpace**(*root_system*, *baseRing*)

Bases: sage.combinat.root_system.ambient_space.AmbientSpace

The lattice behind E6, E7, or E8. The computations are based on Bourbaki, Groupes et Algebres de Lie, Ch.

4,5,6 (planche V-VII).

dimension()

EXAMPLES:

```
sage: e = RootSystem(['E',6]).ambient_space()
sage: e.dimension()
8
```

fundamental_weights()

EXAMPLES:

```
sage: e = RootSystem(['E',6]).ambient_space()
sage: e.fundamental_weights()
Finite family {1: (0, 0, 0, 0, 0, -2/3, -2/3, 2/3), 2: (1/2, 1/2, 1/2, 1/2, 1/2, -1/2, -1/2,
```

negative_roots()

The negative negative roots.

EXAMPLES:

```
sage: e = RootSystem(['E',6]).ambient_space()
sage: e.negative_roots()
[(-1, -1, 0, 0, 0, 0, 0, 0),
 (-1, 0, -1, 0, 0, 0, 0, 0),
 (-1, 0, 0, -1, 0, 0, 0, 0),
 (-1, 0, 0, 0, -1, 0, 0, 0),
 (0, -1, -1, 0, 0, 0, 0, 0),
 (0, -1, 0, -1, 0, 0, 0, 0),
 (0, -1, 0, 0, -1, 0, 0, 0),
 (0, 0, -1, -1, 0, 0, 0, 0),
 (0, 0, -1, 0, -1, 0, 0, 0),
 (0, 0, 0, -1, -1, 0, 0, 0),
 (1, -1, 0, 0, 0, 0, 0, 0),
 (1, 0, -1, 0, 0, 0, 0, 0),
 (1, 0, 0, -1, 0, 0, 0, 0),
 (1, 0, 0, 0, -1, 0, 0, 0),
 (0, 1, -1, 0, 0, 0, 0, 0),
 (0, 1, 0, -1, 0, 0, 0, 0),
 (0, 1, 0, 0, -1, 0, 0, 0),
 (0, 0, 1, -1, 0, 0, 0, 0),
 (0, 0, 1, 0, -1, 0, 0, 0),
 (0, 0, 0, 1, -1, 0, 0, 0),
 (-1/2, -1/2, -1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
 (-1/2, -1/2, -1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (-1/2, -1/2, 1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (-1/2, -1/2, 1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
 (-1/2, 1/2, -1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (-1/2, 1/2, -1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
 (-1/2, 1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
 (-1/2, 1/2, 1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, -1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, -1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, 1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, 1/2, -1/2, 1/2, 1/2, -1/2)]
```

positive_roots()

These are the roots positive w.r. to lexicographic ordering of the basis elements (e1<...<e4).

EXAMPLES:

```
sage: e = RootSystem(['E',6]).ambient_space()
sage: e.positive_roots()
[(1, 1, 0, 0, 0, 0, 0, 0),
 (1, 0, 1, 0, 0, 0, 0, 0),
 (1, 0, 0, 1, 0, 0, 0, 0),
 (1, 0, 0, 0, 1, 0, 0, 0),
 (0, 1, 1, 0, 0, 0, 0, 0),
 (0, 1, 0, 1, 0, 0, 0, 0),
 (0, 1, 0, 0, 1, 0, 0, 0),
 (0, 0, 1, 1, 0, 0, 0, 0),
 (0, 0, 1, 0, 1, 0, 0, 0),
 (0, 0, 0, 1, 1, 0, 0, 0),
 (-1, 1, 0, 0, 0, 0, 0, 0),
 (-1, 0, 1, 0, 0, 0, 0, 0),
 (-1, 0, 0, 1, 0, 0, 0, 0),
 (-1, 0, 0, 0, 1, 0, 0, 0),
 (0, -1, 1, 0, 0, 0, 0, 0),
 (0, -1, 0, 1, 0, 0, 0, 0),
 (0, -1, 0, 0, 1, 0, 0, 0),
 (0, 0, -1, 1, 0, 0, 0, 0),
 (0, 0, -1, 0, 1, 0, 0, 0),
 (0, 0, 0, -1, 1, 0, 0, 0),
 (1/2, 1/2, 1/2, 1/2, 1/2, -1/2, -1/2, 1/2),
 (1/2, 1/2, 1/2, -1/2, -1/2, -1/2, -1/2, 1/2),
 (1/2, 1/2, -1/2, 1/2, -1/2, -1/2, -1/2, 1/2),
 (1/2, 1/2, -1/2, -1/2, 1/2, -1/2, -1/2, 1/2),
 (1/2, -1/2, 1/2, 1/2, -1/2, -1/2, -1/2, 1/2),
 (1/2, -1/2, 1/2, -1/2, 1/2, -1/2, -1/2, 1/2),
 (1/2, -1/2, -1/2, 1/2, 1/2, -1/2, -1/2, 1/2),
 (1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 1/2),
 (-1/2, 1/2, 1/2, 1/2, -1/2, -1/2, -1/2, 1/2),
 (-1/2, 1/2, 1/2, -1/2, 1/2, -1/2, -1/2, 1/2),
 (-1/2, 1/2, -1/2, 1/2, 1/2, -1/2, -1/2, 1/2),
 (-1/2, 1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 1/2),
 (-1/2, -1/2, 1/2, 1/2, 1/2, -1/2, -1/2, 1/2),
 (-1/2, -1/2, 1/2, -1/2, -1/2, -1/2, -1/2, 1/2),
 (-1/2, -1/2, -1/2, 1/2, -1/2, -1/2, -1/2, 1/2),
 (-1/2, -1/2, -1/2, -1/2, 1/2, -1/2, -1/2, 1/2)]
sage: e.rho()
(0, 1, 2, 3, 4, -4, -4, 4)
sage: E8 = RootSystem(['E',8])
sage: e = E8.ambient_space()
sage: e.negative_roots()
[(-1, -1, 0, 0, 0, 0, 0, 0),
 (-1, 0, -1, 0, 0, 0, 0, 0),
 (-1, 0, 0, -1, 0, 0, 0, 0),
 (-1, 0, 0, 0, -1, 0, 0, 0),
 (-1, 0, 0, 0, 0, -1, 0, 0),
 (-1, 0, 0, 0, 0, 0, -1, 0),
 (-1, 0, 0, 0, 0, 0, 0, -1),
 (0, -1, -1, 0, 0, 0, 0, 0),
 (0, -1, 0, -1, 0, 0, 0, 0),
 (0, -1, 0, 0, -1, 0, 0, 0),
 (0, -1, 0, 0, 0, -1, 0, 0),
 (0, -1, 0, 0, 0, 0, -1, 0),
```

```
(0, -1, 0, 0, 0, 0, 0, -1),
(0, 0, -1, -1, 0, 0, 0, 0),
(0, 0, -1, 0, -1, 0, 0, 0),
(0, 0, -1, 0, 0, -1, 0, 0),
(0, 0, -1, 0, 0, 0, -1, 0),
(0, 0, -1, 0, 0, 0, 0, -1),
(0, 0, 0, -1, -1, 0, 0, 0),
(0, 0, 0, -1, 0, -1, 0, 0),
(0, 0, 0, -1, 0, 0, -1, 0),
(0, 0, 0, -1, 0, 0, 0, -1),
(0, 0, 0, 0, -1, -1, 0, 0),
(0, 0, 0, 0, -1, 0, -1, 0),
(0, 0, 0, 0, -1, 0, 0, -1),
(0, 0, 0, 0, 0, -1, -1, 0),
(0, 0, 0, 0, 0, -1, 0, -1),
(0, 0, 0, 0, 0, 0, -1, -1),
(1, -1, 0, 0, 0, 0, 0, 0),
(1, 0, -1, 0, 0, 0, 0, 0),
(1, 0, 0, -1, 0, 0, 0, 0),
(1, 0, 0, 0, -1, 0, 0, 0),
(1, 0, 0, 0, 0, -1, 0, 0),
(1, 0, 0, 0, 0, 0, -1, 0),
(1, 0, 0, 0, 0, 0, 0, -1),
(0, 1, -1, 0, 0, 0, 0, 0),
(0, 1, 0, -1, 0, 0, 0, 0),
(0, 1, 0, 0, -1, 0, 0, 0),
(0, 1, 0, 0, 0, -1, 0, 0),
(0, 1, 0, 0, 0, 0, -1, 0),
(0, 1, 0, 0, 0, 0, 0, -1),
(0, 0, 1, -1, 0, 0, 0, 0),
(0, 0, 1, 0, -1, 0, 0, 0),
(0, 0, 1, 0, 0, -1, 0, 0),
(0, 0, 1, 0, 0, 0, -1, 0),
(0, 0, 1, 0, 0, 0, 0, -1),
(0, 0, 0, 1, -1, 0, 0, 0),
(0, 0, 0, 1, 0, -1, 0, 0),
(0, 0, 0, 1, 0, 0, -1, 0),
(0, 0, 0, 1, 0, 0, 0, -1),
(0, 0, 0, 0, 1, -1, 0, 0),
(0, 0, 0, 0, 1, 0, -1, 0),
(0, 0, 0, 0, 1, 0, 0, -1),
(0, 0, 0, 0, 0, 1, -1, 0),
(0, 0, 0, 0, 0, 1, 0, -1),
(0, 0, 0, 0, 0, 0, 1, -1),
(-1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2),
(-1/2, -1/2, -1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
(-1/2, -1/2, -1/2, -1/2, 1/2, -1/2, 1/2, -1/2),
(-1/2, -1/2, -1/2, -1/2, 1/2, 1/2, -1/2, -1/2),
(-1/2, -1/2, -1/2, 1/2, -1/2, -1/2, 1/2, -1/2),
(-1/2, -1/2, -1/2, 1/2, -1/2, 1/2, -1/2, -1/2),
(-1/2, -1/2, -1/2, 1/2, 1/2, -1/2, -1/2, -1/2),
(-1/2, -1/2, -1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, -1/2, -1/2, -1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, -1/2, -1/2, 1/2, -1/2, -1/2),
(-1/2, -1/2, 1/2, -1/2, 1/2, -1/2, -1/2, -1/2),
(-1/2, -1/2, 1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, 1/2, -1/2, -1/2, -1/2, -1/2),
(-1/2, -1/2, 1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
```

```
 (-1/2, -1/2, 1/2, 1/2, 1/2, -1/2, 1/2, -1/2),
 (-1/2, -1/2, 1/2, 1/2, 1/2, 1/2, -1/2, -1/2),
 (-1/2, 1/2, -1/2, -1/2, -1/2, -1/2, 1/2, -1/2),
 (-1/2, 1/2, -1/2, -1/2, -1/2, 1/2, -1/2, -1/2),
 (-1/2, 1/2, -1/2, -1/2, 1/2, -1/2, -1/2, -1/2),
 (-1/2, 1/2, -1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (-1/2, 1/2, -1/2, 1/2, -1/2, -1/2, -1/2, -1/2),
 (-1/2, 1/2, -1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
 (-1/2, 1/2, -1/2, 1/2, 1/2, -1/2, 1/2, -1/2),
 (-1/2, 1/2, -1/2, 1/2, 1/2, 1/2, -1/2, -1/2),
 (-1/2, 1/2, 1/2, -1/2, -1/2, -1/2, -1/2, -1/2),
 (-1/2, 1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
 (-1/2, 1/2, 1/2, -1/2, 1/2, -1/2, 1/2, -1/2),
 (-1/2, 1/2, 1/2, -1/2, 1/2, 1/2, -1/2, -1/2),
 (-1/2, 1/2, 1/2, 1/2, -1/2, -1/2, 1/2, -1/2),
 (-1/2, 1/2, 1/2, 1/2, -1/2, 1/2, -1/2, -1/2),
 (-1/2, 1/2, 1/2, 1/2, 1/2, -1/2, -1/2, -1/2),
 (-1/2, 1/2, 1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 1/2, -1/2),
 (1/2, -1/2, -1/2, -1/2, -1/2, 1/2, -1/2, -1/2),
 (1/2, -1/2, -1/2, -1/2, 1/2, -1/2, -1/2, -1/2),
 (1/2, -1/2, -1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, -1/2, 1/2, -1/2, -1/2, -1/2, -1/2),
 (1/2, -1/2, -1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, -1/2, 1/2, 1/2, -1/2, 1/2, -1/2),
 (1/2, -1/2, -1/2, 1/2, 1/2, 1/2, -1/2, -1/2),
 (1/2, -1/2, 1/2, -1/2, -1/2, -1/2, -1/2, -1/2),
 (1/2, -1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, -1/2, 1/2, -1/2, 1/2, -1/2, 1/2, -1/2),
 (1/2, -1/2, 1/2, -1/2, 1/2, 1/2, -1/2, -1/2),
 (1/2, -1/2, 1/2, 1/2, -1/2, -1/2, 1/2, -1/2),
 (1/2, -1/2, 1/2, 1/2, -1/2, 1/2, -1/2, -1/2),
 (1/2, -1/2, 1/2, 1/2, 1/2, -1/2, -1/2, -1/2),
 (1/2, -1/2, 1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2),
 (1/2, 1/2, -1/2, -1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, -1/2, 1/2, -1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, -1/2, 1/2, 1/2, -1/2, -1/2),
 (1/2, 1/2, -1/2, 1/2, -1/2, -1/2, 1/2, -1/2),
 (1/2, 1/2, -1/2, 1/2, -1/2, 1/2, -1/2, -1/2),
 (1/2, 1/2, -1/2, 1/2, 1/2, -1/2, -1/2, -1/2),
 (1/2, 1/2, -1/2, 1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, -1/2, -1/2, -1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, -1/2, -1/2, 1/2, -1/2, -1/2),
 (1/2, 1/2, 1/2, -1/2, 1/2, -1/2, -1/2, -1/2),
 (1/2, 1/2, 1/2, -1/2, 1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, 1/2, -1/2, -1/2, -1/2, -1/2),
 (1/2, 1/2, 1/2, 1/2, -1/2, 1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, 1/2, 1/2, -1/2, 1/2, -1/2),
 (1/2, 1/2, 1/2, 1/2, 1/2, 1/2, -1/2, -1/2)]
sage: e.rho()
(0, 1, 2, 3, 4, 5, 6, 23)
```

root(*i1*, *i2=None*, *i3=None*, *i4=None*, *i5=None*, *i6=None*, *i7=None*, *i8=None*, *p1=0*, *p2=0*, *p3=0*, *p4=0*, *p5=0*, *p6=0*, *p7=0*, *p8=0*)

Compute an element of the underlying lattice, using the specified elements of the standard basis, with signs dictated by the corresponding 'pi' arguments. We rely on the caller to provide the correct arguments. This is typically used to generate roots, although the generated elements need not be roots themselves. We

assume that if one of the indices is not given, the rest are not as well. This should work for E6, E7, E8.

EXAMPLES:

```
sage: e = RootSystem(['E',6]).ambient_space()
sage: [ e.root(i, j, p3=1) for i in xrange(e.n) for j in xrange(i+1, e.n) ]
[(1, 1, 0, 0, 0, 0, 0, 0),
 (1, 0, 1, 0, 0, 0, 0, 0),
 (1, 0, 0, 1, 0, 0, 0, 0),
 (1, 0, 0, 0, 1, 0, 0, 0),
 (1, 0, 0, 0, 0, 1, 0, 0),
 (1, 0, 0, 0, 0, 0, 1, 0),
 (1, 0, 0, 0, 0, 0, 0, 1),
 (0, 1, 1, 0, 0, 0, 0, 0),
 (0, 1, 0, 1, 0, 0, 0, 0),
 (0, 1, 0, 0, 1, 0, 0, 0),
 (0, 1, 0, 0, 0, 1, 0, 0),
 (0, 1, 0, 0, 0, 0, 1, 0),
 (0, 1, 0, 0, 0, 0, 0, 1),
 (0, 0, 1, 1, 0, 0, 0, 0),
 (0, 0, 1, 0, 1, 0, 0, 0),
 (0, 0, 1, 0, 0, 1, 0, 0),
 (0, 0, 1, 0, 0, 0, 1, 0),
 (0, 0, 1, 0, 0, 0, 0, 1),
 (0, 0, 0, 1, 1, 0, 0, 0),
 (0, 0, 0, 1, 0, 1, 0, 0),
 (0, 0, 0, 1, 0, 0, 1, 0),
 (0, 0, 0, 1, 0, 0, 0, 1),
 (0, 0, 0, 0, 1, 1, 0, 0),
 (0, 0, 0, 0, 1, 0, 1, 0),
 (0, 0, 0, 0, 1, 0, 0, 1),
 (0, 0, 0, 0, 0, 1, 1, 0),
 (0, 0, 0, 0, 0, 1, 0, 1),
 (0, 0, 0, 0, 0, 0, 1, 1)]
```

simple_root(*i*)

There are computed as what Bourbaki calls the Base: a1 = e2-e3, a2 = e3-e4, a3 = e4, a4 = 1/2*(e1-e2-e3-e4)

EXAMPLES:

```
sage: LE6 = RootSystem(['E',6]).ambient_space()
sage: LE6.simple_roots()
Finite family {1: (1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 1/2), 2: (1, 1, 0, 0, 0, 0, 0, 0
```

class sage.combinat.root_system.type_E.**CartanType**(*n*)

Bases: sage.combinat.root_system.cartan_type.CartanType_standard_finite, sage.combinat.root_system.cartan_type.CartanType_simple, sage.combinat.root_system.cartan_type.CartanType_simply_laced

EXAMPLES:

```
sage: ct = CartanType(['E',6])
sage: ct
['E', 6]
sage: ct._repr_(compact = True)
'E6'
sage: ct.is_irreducible()
True
sage: ct.is_finite()
```

```
True
sage: ct.is_affine()
False
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
True
sage: ct.affine()
['E', 6, 1]
sage: ct.dual()
['E', 6]
```

TESTS:

```
sage: TestSuite(ct).run()
```

AmbientSpace
alias of AmbientSpace

ascii_art(*label=<function <lambda>>*, *node=None*)
Return a ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print CartanType(['E',6]).ascii_art(label = lambda x: x+2)
        O 4
        |
        |
O---O---O---O---O
3   5   6   7   8
sage: print CartanType(['E',7]).ascii_art(label = lambda x: x+2)
        O 4
        |
        |
O---O---O---O---O---O
3   5   6   7   8   9
sage: print CartanType(['E',8]).ascii_art(label = lambda x: x+1)
        O 3
        |
        |
O---O---O---O---O---O---O
2   4   5   6   7   8   9
```

coxeter_number()
Return the Coxeter number associated with self.

EXAMPLES:

```
sage: CartanType(['E',6]).coxeter_number()
12
sage: CartanType(['E',7]).coxeter_number()
18
sage: CartanType(['E',8]).coxeter_number()
30
```

dual_coxeter_number()
Return the dual Coxeter number associated with self.

EXAMPLES:

```
sage: CartanType(['E',6]).dual_coxeter_number()
12
sage: CartanType(['E',7]).dual_coxeter_number()
18
sage: CartanType(['E',8]).dual_coxeter_number()
30
```

dynkin_diagram()

Returns a Dynkin diagram for type E.

EXAMPLES:

```
sage: e = CartanType(['E',6]).dynkin_diagram()
sage: e
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6
sage: sorted(e.edges())
[(1, 3, 1), (2, 4, 1), (3, 1, 1), (3, 4, 1), (4, 2, 1), (4, 3, 1), (4, 5, 1), (5, 4, 1), (5,
sage: e = CartanType(['E',7]).dynkin_diagram()
sage: e
        O 2
        |
        |
O---O---O---O---O---O
1   3   4   5   6   7
E7
sage: sorted(e.edges())
[(1, 3, 1), (2, 4, 1), (3, 1, 1), (3, 4, 1), (4, 2, 1),
 (4, 3, 1), (4, 5, 1), (5, 4, 1), (5, 6, 1), (6, 5, 1),
 (6, 7, 1), (7, 6, 1)]
sage: e = CartanType(['E',8]).dynkin_diagram()
sage: e
        O 2
        |
        |
O---O---O---O---O---O---O
1   3   4   5   6   7   8
E8
sage: sorted(e.edges())
[(1, 3, 1), (2, 4, 1), (3, 1, 1), (3, 4, 1), (4, 2, 1),
 (4, 3, 1), (4, 5, 1), (5, 4, 1), (5, 6, 1), (6, 5, 1),
 (6, 7, 1), (7, 6, 1), (7, 8, 1), (8, 7, 1)]
```

5.1.221 Root system data for (untwisted) type E affine

class sage.combinat.root_system.type_E_affine.**CartanType**(*n*)

Bases: sage.combinat.root_system.cartan_type.CartanType_standard_untwisted_affine, sage.combinat.root_system.cartan_type.CartanType_simply_laced

EXAMPLES:

```
sage: ct = CartanType(['E',6,1])
sage: ct
['E', 6, 1]
```

```
sage: ct._repr_(compact = True)
'E6~'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_untwisted_affine()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
True
sage: ct.classical()
['E', 6]
sage: ct.dual()
['E', 6, 1]
```

TESTS:

```
sage: TestSuite(ct).run()
```

ascii_art(*label=<function <lambda>>*, *node=None*)

Return an ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print CartanType(['E',6,1]).ascii_art(label = lambda x: x+2)
        O 2
        |
        |
        O 4
        |
        |
O---O---O---O---O---O
3   5   6   7   8   9
sage: print CartanType(['E',7,1]).ascii_art(label = lambda x: x+2)
            O 4
            |
            |
O---O---O---O---O---O---O---O
2   3   5   6   7   8   9   10
sage: print CartanType(['E',8,1]).ascii_art(label = lambda x: x-3)
        O -1
        |
        |
O---O---O---O---O---O---O---O---O
-2  0   1   2   3   4   5   6   -3
```

dynkin_diagram()

Returns the extended Dynkin diagram for affine type E.

EXAMPLES:

```
sage: e = CartanType(['E', 6, 1]).dynkin_diagram()
sage: e
        O 0
        |
```

```
        |
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6~
sage: sorted(e.edges())
[(0, 2, 1),
 (1, 3, 1),
 (2, 0, 1),
 (2, 4, 1),
 (3, 1, 1),
 (3, 4, 1),
 (4, 2, 1),
 (4, 3, 1),
 (4, 5, 1),
 (5, 4, 1),
 (5, 6, 1),
 (6, 5, 1)]

sage: e = CartanType(['E', 7, 1]).dynkin_diagram()
sage: e
            O 2
            |
            |
O---O---O---O---O---O---O
0   1   3   4   5   6   7
E7~
sage: sorted(e.edges())
[(0, 1, 1), (1, 0, 1), (1, 3, 1), (2, 4, 1), (3, 1, 1), (3, 4, 1),
 (4, 2, 1), (4, 3, 1), (4, 5, 1), (5, 4, 1), (5, 6, 1),
 (6, 5, 1), (6, 7, 1), (7, 6, 1)]
sage: e = CartanType(['E', 8, 1]).dynkin_diagram()
sage: e
        O 2
        |
        |
O---O---O---O---O---O---O---O
1   3   4   5   6   7   8   0
E8~
sage: sorted(e.edges())
[(0, 8, 1), (1, 3, 1), (2, 4, 1), (3, 1, 1), (3, 4, 1),
 (4, 2, 1), (4, 3, 1), (4, 5, 1), (5, 4, 1), (5, 6, 1),
 (6, 5, 1), (6, 7, 1), (7, 6, 1), (7, 8, 1), (8, 0, 1), (8, 7, 1)]
```

5.1.222 Root system data for type F

class sage.combinat.root_system.type_F.**AmbientSpace**(*root_system*, *base_ring*)

Bases: sage.combinat.root_system.ambient_space.AmbientSpace

The lattice behind F_4. The computations are based on Bourbaki, Groupes et Algebres de Lie, Ch. 4,5,6 (planche VIII).

dimension()

Return the dimension of self.

EXAMPLES:

```
sage: e = RootSystem(['F',4]).ambient_space()
sage: e.dimension()
4
```

fundamental_weights()

Return the fundamental weights of `self`.

EXAMPLES:

```
sage: e =  RootSystem(['F',4]).ambient_space()
sage: e.fundamental_weights()
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0), 3: (3/2, 1/2, 1/2, 1/2), 4: (1, 0, 0, 0)}
```

negative_roots()

Return the negative roots.

EXAMPLES:

```
sage: e = RootSystem(['F',4]).ambient_space()
sage: e.negative_roots()
[(-1, 0, 0, 0),
(0, -1, 0, 0),
(0, 0, -1, 0),
(0, 0, 0, -1),
(-1, -1, 0, 0),
(-1, 0, -1, 0),
(-1, 0, 0, -1),
(0, -1, -1, 0),
(0, -1, 0, -1),
(0, 0, -1, -1),
(-1, 1, 0, 0),
(-1, 0, 1, 0),
(-1, 0, 0, 1),
(0, -1, 1, 0),
(0, -1, 0, 1),
(0, 0, -1, 1),
(-1/2, -1/2, -1/2, -1/2),
(-1/2, -1/2, -1/2, 1/2),
(-1/2, -1/2, 1/2, -1/2),
(-1/2, -1/2, 1/2, 1/2),
(-1/2, 1/2, -1/2, -1/2),
(-1/2, 1/2, -1/2, 1/2),
(-1/2, 1/2, 1/2, -1/2),
(-1/2, 1/2, 1/2, 1/2)]
```

positive_roots()

Return the positive roots.

These are the roots which are positive with respect to the lexicographic ordering of the basis elements ($\epsilon_1 < \epsilon_2 < \epsilon_3 < \epsilon_4$).

EXAMPLES:

```
sage: e = RootSystem(['F',4]).ambient_space()
sage: e.positive_roots()
[(1, 0, 0, 0),
(0, 1, 0, 0),
(0, 0, 1, 0),
(0, 0, 0, 1),
(1, 1, 0, 0),
```

```
(1, 0, 1, 0),
(1, 0, 0, 1),
(0, 1, 1, 0),
(0, 1, 0, 1),
(0, 0, 1, 1),
(1, -1, 0, 0),
(1, 0, -1, 0),
(1, 0, 0, -1),
(0, 1, -1, 0),
(0, 1, 0, -1),
(0, 0, 1, -1),
(1/2, 1/2, 1/2, 1/2),
(1/2, 1/2, 1/2, -1/2),
(1/2, 1/2, -1/2, 1/2),
(1/2, 1/2, -1/2, -1/2),
(1/2, -1/2, 1/2, 1/2),
(1/2, -1/2, 1/2, -1/2),
(1/2, -1/2, -1/2, 1/2),
(1/2, -1/2, -1/2, -1/2)]
sage: e.rho()
(11/2, 5/2, 3/2, 1/2)
```

root (*i*, *j=None*, *k=None*, *l=None*, *p1=0*, *p2=0*, *p3=0*, *p4=0*)

Compute a root from base elements of the underlying lattice. The arguments specify the basis elements and the signs. Sadly, the base elements are indexed zero-based. We assume that if one of the indices is not given, the rest are not as well.

EXAMPLES:

```
sage: e = RootSystem(['F',4]).ambient_space()
sage: [ e.root(i,j,p2=1) for i in xrange(e.n) for j in xrange(i+1,e.n) ]
[(1, -1, 0, 0), (1, 0, -1, 0), (1, 0, 0, -1), (0, 1, -1, 0), (0, 1, 0, -1), (0, 0, 1, -1)]
```

simple_root (*i*)

Return the i-th simple root.

It is computed according to what Bourbaki calls the Base:

$$\alpha_1 = \epsilon_2 - \epsilon_3, \alpha_2 = \epsilon_3 - \epsilon_4, \alpha_3 = \epsilon_4, \alpha_4 = \frac{1}{2}(\epsilon_1 - \epsilon_2 - \epsilon_3 - \epsilon_4).$$

EXAMPLES:

```
sage: e = RootSystem(['F',4]).ambient_space()
sage: e.simple_roots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 1), 4: (1/2, -1/2, -1/2, -1/
```

class sage.combinat.root_system.type_F.**CartanType**

Bases: sage.combinat.root_system.cartan_type.CartanType_standard_finite, sage.combinat.root_system.cartan_type.CartanType_simple, sage.combinat.root_system.cartan_type.CartanType_crystallographic

EXAMPLES:

```
sage: ct = CartanType(['F',4])
sage: ct
['F', 4]
sage: ct._repr_(compact = True)
'F4'
```

```
sage: ct.is_irreducible()
True
sage: ct.is_finite()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.dual()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
sage: ct.affine()
['F', 4, 1]
```

TESTS:

```
sage: TestSuite(ct).run()
```

AmbientSpace
alias of AmbientSpace

ascii_art(*label=<function <lambda>>*, *node=None*)
Return an ascii art representation of the extended Dynkin diagram.

EXAMPLES:

```
sage: print CartanType(['F',4]).ascii_art(label = lambda x: x+2)
O---O=>=O---O
3   4   5   6
sage: print CartanType(['F',4]).ascii_art(label = lambda x: x-2)
O---O=>=O---O
-1  0   1   2
```

coxeter_number()
Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['F',4]).coxeter_number()
12
```

dual()
Return the dual Cartan type.

This uses that F_4 is self-dual up to relabelling.

EXAMPLES:

```
sage: F4 = CartanType(['F',4])
sage: F4.dual()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}

sage: F4.dynkin_diagram()
O---O=>=O---O
1   2   3   4
F4
sage: F4.dual().dynkin_diagram()
O---O=>=O---O
4   3   2   1
F4 relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

dual_coxeter_number()
Return the dual Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['F',4]).dual_coxeter_number()
9
```

dynkin_diagram()
Returns a Dynkin diagram for type F.

EXAMPLES:

```
sage: f = CartanType(['F',4]).dynkin_diagram()
sage: f
O---O=>=O---O
1   2   3   4
F4
sage: sorted(f.edges())
[(1, 2, 1), (2, 1, 1), (2, 3, 2), (3, 2, 1), (3, 4, 1), (4, 3, 1)]
```

5.1.223 Root system data for (untwisted) type F affine

class sage.combinat.root_system.type_F_affine.**CartanType**
Bases: sage.combinat.root_system.cartan_type.CartanType_standard_untwisted_affine

EXAMPLES:

```
sage: ct = CartanType(['F',4,1])
sage: ct
['F', 4, 1]
sage: ct._repr_(compact = True)
'F4~'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_untwisted_affine()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.classical()
['F', 4]
sage: ct.dual()
['F', 4, 1]^*
sage: ct.dual().is_untwisted_affine()
False
```

TESTS:

```
sage: TestSuite(ct).run()
```

ascii_art(*label=<function <lambda>>*, *node=None*)
Returns a ascii art representation of the extended Dynkin diagram

EXAMPLES:

```
sage: print CartanType(['F',4,1]).ascii_art(label = lambda x: x+2)
O---O---O=>=O---O
2   3   4   5   6
```

dynkin_diagram()

Returns the extended Dynkin diagram for affine type F.

EXAMPLES:

```
sage: f = CartanType(['F', 4, 1]).dynkin_diagram()
sage: f
O---O---O=>=O---O
0   1   2   3   4
F4~
sage: sorted(f.edges())
[(0, 1, 1), (1, 0, 1), (1, 2, 1), (2, 1, 1), (2, 3, 2), (3, 2, 1), (3, 4, 1), (4, 3, 1)]
```

5.1.224 Root system data for type G

class sage.combinat.root_system.type_G.**AmbientSpace**(*root_system*, *base_ring*)

Bases: sage.combinat.root_system.ambient_space.AmbientSpace

EXAMPLES:

```
sage: e = RootSystem(['G',2]).ambient_space(); e
Ambient space of the Root system of type ['G', 2]
```

One can not construct the ambient lattice because the simple coroots have rational coefficients:

```
sage: e.simple_coroots()
Finite family {1: (0, 1, -1), 2: (1/3, -2/3, 1/3)}
sage: e.smallest_base_ring()
Rational Field
```

TESTS:

```
sage: TestSuite(e).run()
sage: [WeylDim(['G',2],[a,b]) for a,b in [[0,0], [1,0], [0,1], [1,1]]] # indirect doctest
[1, 7, 14, 64]
```

By default, this ambient space uses the barycentric projection for plotting:

```
sage: L = RootSystem(["G",2]).ambient_space()
sage: e = L.basis()
sage: L._plot_projection(e[0])
(1/2, 989/1142)
sage: L._plot_projection(e[1])
(-1, 0)
sage: L._plot_projection(e[2])
(1/2, -989/1142)
sage: L = RootSystem(["A",3]).ambient_space()
sage: l = L.an_element(); l
(2, 2, 3, 0)
sage: L._plot_projection(l)
(0, -1121/1189, 7/3)
```

See also:

- sage.combinat.root_system.root_lattice_realizations.RootLatticeRealizations.ParentM

dimension()

EXAMPLES:

```
sage: e = RootSystem(['G',2]).ambient_space()
sage: e.dimension()
3
```

fundamental_weights()

EXAMPLES:

```
sage: CartanType(['G',2]).root_system().ambient_space().fundamental_weights()
Finite family {1: (1, 0, -1), 2: (2, -1, -1)}
```

negative_roots()

EXAMPLES:

```
sage: CartanType(['G',2]).root_system().ambient_space().negative_roots()
[(0, -1, 1), (-1, 2, -1), (-1, 1, 0), (-1, 0, 1), (-1, -1, 2), (-2, 1, 1)]
```

positive_roots()

EXAMPLES:

```
sage: CartanType(['G',2]).root_system().ambient_space().positive_roots()
[(0, 1, -1), (1, -2, 1), (1, -1, 0), (1, 0, -1), (1, 1, -2), (2, -1, -1)]
```

simple_root(*i*)

EXAMPLES:

```
sage: CartanType(['G',2]).root_system().ambient_space().simple_roots()
Finite family {1: (0, 1, -1), 2: (1, -2, 1)}
```

class sage.combinat.root_system.type_G.**CartanType**

Bases: sage.combinat.root_system.cartan_type.CartanType_standard_finite, sage.combinat.root_system.cartan_type.CartanType_simple, sage.combinat.root_system.cartan_type.CartanType_crystallographic

EXAMPLES:

```
sage: ct = CartanType(['G',2])
sage: ct
['G', 2]
sage: ct._repr_(compact = True)
'G2'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.dual()
['G', 2] relabelled by {1: 2, 2: 1}
sage: ct.affine()
['G', 2, 1]
```

TESTS:

```
sage: TestSuite(ct).run()
```

AmbientSpace
alias of AmbientSpace

ascii_art(*label=<function <lambda>>*, *node=None*)
Return an ascii art representation of the Dynkin diagram.

EXAMPLES:

```
sage: print CartanType(['G',2]).ascii_art(label=lambda x: x+2)
  3
O=<=O
3   4
```

coxeter_number()
Return the Coxeter number associated with self.

EXAMPLES:

```
sage: CartanType(['G',2]).coxeter_number()
6
```

dual()
Return the dual Cartan type.

This uses that G_2 is self-dual up to relabelling.

EXAMPLES:

```
sage: G2 = CartanType(['G',2])
sage: G2.dual()
['G', 2] relabelled by {1: 2, 2: 1}

sage: G2.dynkin_diagram()
  3
O=<=O
1   2
G2
sage: G2.dual().dynkin_diagram()
  3
O=<=O
2   1
G2 relabelled by {1: 2, 2: 1}
```

dual_coxeter_number()
Return the dual Coxeter number associated with self.

EXAMPLES:

```
sage: CartanType(['G',2]).dual_coxeter_number()
4
```

dynkin_diagram()
Returns a Dynkin diagram for type G.

EXAMPLES:

```
sage: g = CartanType(['G',2]).dynkin_diagram()
sage: g
  3
O=<=O
```

```
1   2
G2
sage: sorted(g.edges())
[(1, 2, 1), (2, 1, 3)]
```

5.1.225 Root system data for (untwisted) type G affine

class sage.combinat.root_system.type_G_affine.**CartanType**

Bases: sage.combinat.root_system.cartan_type.CartanType_standard_untwisted_affine

EXAMPLES:

```
sage: ct = CartanType(['G',2,1])
sage: ct
['G', 2, 1]
sage: ct._repr_(compact = True)
'G2~'

sage: ct.is_irreducible()
True
sage: ct.is_finite()
False
sage: ct.is_affine()
True
sage: ct.is_untwisted_affine()
True
sage: ct.is_crystallographic()
True
sage: ct.is_simply_laced()
False
sage: ct.classical()
['G', 2]
sage: ct.dual()
['G', 2, 1]^*
sage: ct.dual().is_untwisted_affine()
False
```

TESTS:

```
sage: TestSuite(ct).run()
```

ascii_art(*label=<function <lambda>>*, *node=None*)

Returns an ascii art representation of the Dynkin diagram

EXAMPLES:

```
sage: print CartanType(['G',2,1]).ascii_art(label = lambda x: x+2)
  3
O=<=O---O
3   4   2
```

dynkin_diagram()

Returns the extended Dynkin diagram for type G.

EXAMPLES:

```
sage: g = CartanType(['G',2,1]).dynkin_diagram()
sage: g
  3
```

```
O=<=O---O
1   2   0
G2~
sage: sorted(g.edges())
[(0, 2, 1), (1, 2, 1), (2, 0, 1), (2, 1, 3)]
```

5.1.226 Root system data for type H

class sage.combinat.root_system.type_H.**CartanType**(*n*)

Bases: sage.combinat.root_system.cartan_type.CartanType_standard_finite, sage.combinat.root_system.cartan_type.CartanType_simple

EXAMPLES:

```
sage: ct = CartanType(['H',3])
sage: ct
['H', 3]
sage: ct._repr_(compact = True)
'H3'
sage: ct.rank()
3

sage: ct.is_irreducible()
True
sage: ct.is_finite()
True
sage: ct.is_affine()
False
sage: ct.is_crystallographic()
False
sage: ct.is_simply_laced()
False
```

TESTS:

```
sage: TestSuite(ct).run()
```

coxeter_diagram()

Returns a Coxeter diagram for type H.

EXAMPLES:

```
sage: ct = CartanType(['H',3])
sage: ct.coxeter_diagram()
Graph on 3 vertices
sage: sorted(ct.coxeter_diagram().edges())
[(1, 2, 3), (2, 3, 5)]
sage: ct.coxeter_matrix()
[1 3 2]
[3 1 5]
[2 5 1]

sage: ct = CartanType(['H',4])
sage: ct.coxeter_diagram()
Graph on 4 vertices
sage: sorted(ct.coxeter_diagram().edges())
[(1, 2, 3), (2, 3, 3), (3, 4, 5)]
sage: ct.coxeter_matrix()
```

```
[1 3 2 2]
[3 1 3 2]
[2 3 1 5]
[2 2 5 1]
```

coxeter_number()

Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['H',3]).coxeter_number()
10
sage: CartanType(['H',4]).coxeter_number()
30
```

5.1.227 Root system data for type I

class sage.combinat.root_system.type_I.**CartanType**(*n*)

Bases: sage.combinat.root_system.cartan_type.CartanType_standard_finite, sage.combinat.root_system.cartan_type.CartanType_simple

EXAMPLES:

```
sage: ct = CartanType(['I',5])
sage: ct
['I', 5]
sage: ct._repr_(compact = True)
'I5'
sage: ct.rank()
2
sage: ct.index_set()
(1, 2)

sage: ct.is_irreducible()
True
sage: ct.is_finite()
True
sage: ct.is_affine()
False
sage: ct.is_crystallographic()
False
sage: ct.is_simply_laced()
False
```

TESTS:

```
sage: TestSuite(ct).run()
```

coxeter_diagram()

Returns the Coxeter matrix for this type.

EXAMPLES:

```
sage: ct = CartanType(['I', 4])
sage: ct.coxeter_diagram()
Graph on 2 vertices
sage: ct.coxeter_diagram().edges()
[(1, 2, 4)]
sage: ct.coxeter_matrix()
```

```
[1 4]
[4 1]
```

coxeter_number()
Return the Coxeter number associated with `self`.

EXAMPLES:

```
sage: CartanType(['I',3]).coxeter_number()
3
sage: CartanType(['I',12]).coxeter_number()
12
```

index_set()
Type I_p is of rank 2

EXAMPLES:

```
sage: CartanType(['I', 5]).index_set()
(1, 2)
```

rank()
Type I_p is of rank 2

EXAMPLES:

```
sage: CartanType(['I', 5]).rank()
2
```

5.1.228 Root system data for affine Cartan types

class sage.combinat.root_system.type_affine.**AmbientSpace**(*root_system*, *base_ring*)
Bases: sage.combinat.free_module.CombinatorialFreeModule

Ambient space for affine types.

This is constructed from the data in the corresponding classical ambient space. Namely, this space is obtained by adding two elements δ and $\delta^\vee$ to the basis of the classical ambient space, and by endowing it with the canonical scalar product.

The coefficient of an element in $\delta^\vee$, thus its scalar product with $\delta^\vee$ gives its level, and dualy for the colevel. The canonical projection onto the classical ambient space (by killing δ and $\delta^\vee$) maps the simple roots (except α_0) onto the corresponding classical simple roots, and similarly for the coroots, fundamental weights, ... Altogether, this uniquely determines the embedding of the root, coroot, weight, and coweight lattices. See `simple_root()` and `fundamental_weight()` for the details.

Warning: In type BC, the null root is in fact:

```
sage: R = RootSystem(["BC",3,2]).ambient_space()
sage: R.null_root()
2*e['delta']
```

Warning: In the literature one often considers a larger affine ambient space obtained from the classical ambient space by adding four dimensions, namely for the fundamental weight Λ_0 the fundamental coweight $\Lambda_0^\vee$, the null root δ, and the null coroot c (aka central element). In this larger ambient space, the scalar product is degenerate: $\langle \delta, \delta \rangle = 0$ and similarly for the null coroot.
In the current implementation, Λ_0 and the null coroot are identified:
sage: L = RootSystem(["A",3,1]).ambient_space() sage: Lambda = L.fundamental_weights()
sage: Lambda[0] e['deltacheck'] sage: L.null_coroot() e['deltacheck']
Therefore the scalar product of the null coroot with itself differs from the larger ambient space:

```
sage: L.null_coroot().scalar(L.null_coroot())
1
```

In general, scalar products between two elements that do not live on "opposite sides" won't necessarily match.

EXAMPLES:

```
sage: R = RootSystem(["A",3,1])
sage: e = R.ambient_space(); e
Ambient space of the Root system of type ['A', 3, 1]
sage: TestSuite(e).run()
```

Systematic checks on all affine types:

```
sage: for ct in CartanType.samples(affine=True, crystallographic=True):
...       if ct.classical().root_system().ambient_space() is not None:
...           print ct
...           L = ct.root_system().ambient_space()
...           assert L
...           TestSuite(L).run()
['A', 1, 1]
['A', 5, 1]
['B', 1, 1]
['B', 5, 1]
['C', 1, 1]
['C', 5, 1]
['D', 3, 1]
['D', 5, 1]
['E', 6, 1]
['E', 7, 1]
['E', 8, 1]
['F', 4, 1]
['G', 2, 1]
['BC', 1, 2]
['BC', 5, 2]
['B', 5, 1]^*
['C', 4, 1]^*
['F', 4, 1]^*
['G', 2, 1]^*
['BC', 1, 2]^*
['BC', 5, 2]^*
```

class **Element**(M, x)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

associated_coroot()

Return the coroot associated to `self`.

INPUT:

- `self` – a root

EXAMPLES:

```
sage: alpha = RootSystem(['C',2,1]).ambient_space().simple_roots()
sage: alpha
Finite family {0: -2*e[0] + e['delta'], 1: e[0] - e[1], 2: 2*e[1]}
sage: alpha[0].associated_coroot()
-e[0] + e['deltacheck']
sage: alpha[1].associated_coroot()
e[0] - e[1]
sage: alpha[2].associated_coroot()
e[1]
```

inner_product(*other*)

Implement the canonical inner product of `self` with `other`.

EXAMPLES:

```
sage: e = RootSystem(['B',3,1]).ambient_space()
sage: B = e.basis()
sage: matrix([[x.inner_product(y) for x in B] for y in B])
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
sage: x = e.an_element(); x
2*e[0] + 2*e[1] + 3*e[2]
sage: x.inner_product(x)
17
```

`scalar()` is an alias for this method:

```
sage: x.scalar(x)
17
```

Todo

Lift to CombinatorialFreeModule.Element as canonical_inner_product

---

scalar(*other*)

Implement the canonical inner product of `self` with `other`.

EXAMPLES:

```
sage: e = RootSystem(['B',3,1]).ambient_space()
sage: B = e.basis()
sage: matrix([[x.inner_product(y) for x in B] for y in B])
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
```

```
[0 0 0 1 0]
[0 0 0 0 1]
sage: x = e.an_element(); x
2*e[0] + 2*e[1] + 3*e[2]
sage: x.inner_product(x)
17
```

`scalar()` is an alias for this method:

```
sage: x.scalar(x)
17
```

Todo

Lift to CombinatorialFreeModule.Element as canonical_inner_product

AmbientSpace.**coroot_lattice**()

EXAMPLES:

```
sage: RootSystem(["A",3,1]).ambient_lattice().coroot_lattice()
Ambient lattice of the Root system of type ['A', 3, 1]
```

Todo

Factor out this code with the classical ambient space.

AmbientSpace.**fundamental_weight**(i)

Return the fundamental weight Λ_i in this ambient space.

It is constructed by taking the corresponding fundamental weight of the classical ambient space (or 0 for Λ_0) and raising it to the appropriate level by adding a suitable multiple of $\delta^\vee$.

EXAMPLES:

```
sage: RootSystem(['A',3,1]).ambient_space().fundamental_weight(2)
e[0] + e[1] + e['deltacheck']
sage: RootSystem(['A',3,1]).ambient_space().fundamental_weights()
Finite family {0: e['deltacheck'], 1: e[0] + e['deltacheck'],
               2: e[0] + e[1] + e['deltacheck'], 3: e[0] + e[1] + e[2] + e['deltacheck']
sage: RootSystem(['A',3]).ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1, 1, 1, 0)}
sage: RootSystem(['A',3,1]).weight_lattice().fundamental_weights().map(attrcall("level"))
Finite family {0: 1, 1: 1, 2: 1, 3: 1}

sage: RootSystem(['B',3,1]).ambient_space().fundamental_weights()
Finite family {0: e['deltacheck'], 1: e[0] + e['deltacheck'],
               2: e[0] + e[1] + 2*e['deltacheck'], 3: 1/2*e[0] + 1/2*e[1] + 1/2*e[2] + e
sage: RootSystem(['B',3]).ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
sage: RootSystem(['B',3,1]).weight_lattice().fundamental_weights().map(attrcall("level"))
Finite family {0: 1, 1: 1, 2: 2, 3: 1}
```

In type BC dual, the coefficient of 'delta^vee' is the level divided by 2 to take into account that the null coroot is $2\delta^\vee$:

```
sage: R = CartanType(['BC',3,2]).dual().root_system()
sage: R.ambient_space().fundamental_weights()
Finite family {0: e['deltacheck'], 1: e[0] + e['deltacheck'],
```

```
                    2: e[0] + e[1] + e['deltacheck'],
                    3: 1/2*e[0] + 1/2*e[1] + 1/2*e[2] + 1/2*e['deltacheck']}
    sage: R.weight_lattice().fundamental_weights().map(attrcall("level"))
    Finite family {0: 2, 1: 2, 2: 2, 3: 1}
    sage: R.ambient_space().null_coroot()
    2*e['deltacheck']
```

By a slight naming abuse this function also accepts "delta" as input so that it can be used to implement the embedding from the extended weight lattice::

```
    sage: RootSystem(['A',3,1]).ambient_space().fundamental_weight("delta")
    e['delta']
```

AmbientSpace.**is_extended**()

Return whether this is a realization of the extended weight lattice: yes!

See also:

- sage.combinat.root_system.weight_space.WeightSpace
- sage.combinat.root_sytem.weight_lattice_realizations.WeightLatticeRealizations.

EXAMPLES:

```
sage: RootSystem(['A',3,1]).ambient_space().is_extended()
True
```

AmbientSpace.**simple_coroot**(i)

Return the i-th simple coroot $\alpha_i^\vee$ of this affine ambient space.

EXAMPLES:

```
sage: RootSystem(["A",3,1]).ambient_space().simple_coroot(1)
e[0] - e[1]
```

It is built as the coroot associated to the simple root α_i:

```
sage: RootSystem(["B",3,1]).ambient_space().simple_roots()
Finite family {0: -e[0] - e[1] + e['delta'], 1: e[0] - e[1], 2: e[1] - e[2], 3: e[2]}
sage: RootSystem(["B",3,1]).ambient_space().simple_coroots()
Finite family {0: -e[0] - e[1] + e['deltacheck'], 1: e[0] - e[1], 2: e[1] - e[2], 3: 2*e[2]}
```

Todo

Factor out this code with the classical ambient space.

AmbientSpace.**simple_root**(i)

Return the i-th simple root of this affine ambient space.

EXAMPLES:

It is built straightforwardly from the corresponding simple root α_i in the classical ambient space:

```
sage: RootSystem(["A",3,1]).ambient_space().simple_root(1)
e[0] - e[1]
```

For the special node (typically $i = 0$), α_0 is built from the other simple roots using the column annihilator of the Cartan matrix and adding δ, where δ is the null root:

```
sage: RootSystem(["A",3]).ambient_space().simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1)}
sage: RootSystem(["A",3,1]).ambient_space().simple_roots()
Finite family {0: -e[0] + e[3] + e['delta'], 1: e[0] - e[1], 2: e[1] - e[2], 3: e[2] - e[3]}
```

Here is a twisted affine example:

```
sage: RootSystem(CartanType(["B",3,1]).dual()).ambient_space().simple_roots()
Finite family {0: -e[0] - e[1] + e['delta'], 1: e[0] - e[1], 2: e[1] - e[2], 3: 2*e[2]}
```

In fact δ is really $1/a_0$ times the null root (see the discussion in WeightSpace) but this only makes a difference in type BC:

```
sage: L = RootSystem(CartanType(["BC",3,2])).ambient_space()
sage: L.simple_roots()
Finite family {0: -e[0] + e['delta'], 1: e[0] - e[1], 2: e[1] - e[2], 3: 2*e[2]}
sage: L.null_root()
2*e['delta']
```

Note: An alternative would have been to use the default implementation of the simple roots as linear combinations of the fundamental weights. However, as in type A_n it is preferable to take a slight variant to avoid rational coefficient (the usual GL_n vs SL_n issue).

See also:

- simple_root()
- WeightSpace
- CartanType.col_annihilator()
- null_root()

classmethod AmbientSpace.**smallest_base_ring**(*cartan_type*)

Return the smallest base ring the ambient space can be defined on.

This is the smallest base ring for the associated classical ambient space.

See also:

smallest_base_ring()

EXAMPLES:

```
sage: cartan_type = CartanType(["A",3,1])
sage: cartan_type.AmbientSpace.smallest_base_ring(cartan_type)
Integer Ring
sage: cartan_type = CartanType(["B",3,1])
sage: cartan_type.AmbientSpace.smallest_base_ring(cartan_type)
Rational Field
```

5.1.229 Root system data for dual Cartan types

class sage.combinat.root_system.type_dual.**AmbientSpace**(*root_system*, *base_ring*)

Bases: sage.combinat.root_system.ambient_space.AmbientSpace

Ambient space for a dual finite Cartan type.

It is constructed in the canonical way from the ambient space of the original Cartan type by switching the roles of simple roots, fundamental weights, etc.

Note: Recall that, for any finite Cartan type, and in particular the a simply laced one, the dual Cartan type is constructed as another preexisting Cartan type. Furthermore the ambient space for an affine type is constructed from the ambient space for its classical type. Thus this code is not actually currently used.

It is kept for cross-checking and for reference in case it could become useful, e.g., for dual of general Kac-Moody types.

For the doctests, we need to explicitly create a dual type. Subsequently, since reconstruction of the dual of type F_4 is the relabelled Cartan type, pickling fails on the `TestSuite` run.

EXAMPLES:

```
sage: ct = sage.combinat.root_system.type_dual.CartanType(CartanType(['F',4]))
sage: L = ct.root_system().ambient_space(); L
Ambient space of the Root system of type ['F', 4]^*
sage: TestSuite(L).run(skip=["_test_elements","_test_pickling"])
```

dimension()

Return the dimension of this ambient space.

See also:

`sage.combinat.root_system.ambient_space.AmbientSpace.dimension()`

EXAMPLES:

```
sage: ct = sage.combinat.root_system.type_dual.CartanType(CartanType(['F',4]))
sage: L = ct.root_system().ambient_space()
sage: L.dimension()
4
```

fundamental_weights()

Return the fundamental weights.

They are computed from the simple roots by inverting the Cartan matrix. This is acceptable since this is only about ambient spaces for finite Cartan types. Also, we do not have to worry about the usual GL_n vs SL_n catch because type A is self dual.

An alternative would have been to start from the fundamental coweights in the dual ambient space, but those are not yet implemented.

EXAMPLES:

```
sage: ct = sage.combinat.root_system.type_dual.CartanType(CartanType(['F',4]))
sage: L = ct.root_system().ambient_space()
sage: L.fundamental_weights()
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0), 3: (3, 1, 1, 1), 4: (2, 0, 0, 0)}
```

Note that this ambient space is isomorphic, but not equal, to that obtained by constructing F_4 dual by relabelling:

```
sage: ct = CartanType(['F',4]).dual(); ct
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
sage: ct.root_system().ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (3/2, 1/2, 1/2, 1/2), 3: (2, 1, 1, 0), 4: (1, 1, 0, 0)}
```

simple_root(i)

Return the `i`-th simple root.

It is constructed by looking up the corresponding simple coroot in the ambient space for the dual Cartan type.

EXAMPLES:

```
sage: ct = sage.combinat.root_system.type_dual.CartanType(CartanType(['F',4]))
sage: ct.root_system().ambient_space().simple_root(1)
(0, 1, -1, 0)

sage: ct.root_system().ambient_space().simple_roots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 2), 4: (1, -1, -1, -1)}

sage: ct.dual().root_system().ambient_space().simple_coroots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 2), 4: (1, -1, -1, -1)}
```

Note that this ambient space is isomorphic, but not equal, to that obtained by constructing F_4 dual by relabelling:

```
sage: ct = CartanType(['F',4]).dual(); ct
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
sage: ct.root_system().ambient_space().simple_roots()
Finite family {1: (1/2, -1/2, -1/2, -1/2), 2: (0, 0, 0, 1), 3: (0, 0, 1, -1), 4: (0, 1, -1,
```

class sage.combinat.root_system.type_dual.**CartanType**(*type*)

Bases: sage.combinat.root_system.cartan_type.CartanType_decorator, sage.combinat.root_system.cartan_type.CartanType_crystallographic

A class for dual Cartan types.

The dual of a (crystallographic) Cartan type is a Cartan type with the same index set, but all arrows reversed in the Dynkin diagram (otherwise said, the Cartan matrix is transposed). It shares a lot of properties in common with its dual. In particular, the Weyl group is isomorphic to that of the dual as a Coxeter group.

EXAMPLES:

For all finite Cartan types, and in particular the simply laced ones, the dual Cartan type is given by another preexisting Cartan type:

```
sage: CartanType(['A',4]).dual()
['A', 4]
sage: CartanType(['B',4]).dual()
['C', 4]
sage: CartanType(['C',4]).dual()
['B', 4]
sage: CartanType(['F',4]).dual()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

So to exercise this class we consider some non simply laced affine Cartan types and also create explicitely F_4^* as a dual cartan type:

```
sage: from sage.combinat.root_system.type_dual import CartanType as CartanTypeDual
sage: F4d = CartanTypeDual(CartanType(['F',4])); F4d
['F', 4]^*
sage: G21d = CartanType(['G',2,1]).dual(); G21d
['G', 2, 1]^*
```

They share many properties with their original Cartan types:

```
sage: F4d.is_irreducible()
True
sage: F4d.is_crystallographic()
True
```

```
sage: F4d.is_simply_laced()
False
sage: F4d.is_finite()
True
sage: G21d.is_finite()
False
sage: F4d.is_affine()
False
sage: G21d.is_affine()
True
```

TESTS:

```
sage: TestSuite(F4d).run(skip=["_test_pickling"])
sage: TestSuite(G21d).run()
```

Note: F4d is pickled by construction as F4.dual() hence the above failure.

---

ascii_art(*label=<function <lambda>>*, *node=None*)

Return an ascii art representation of this Cartan type

(by hacking the ascii art representation of the dual Cartan type)

EXAMPLES:

```
sage: print CartanType(["B", 3, 1]).dual().ascii_art()
    O 0
    |
    |
O---O=<=O
1   2   3
sage: print CartanType(["C", 4, 1]).dual().ascii_art()
O=<=O---O---O=>=O
0   1   2   3   4
sage: print CartanType(["G", 2, 1]).dual().ascii_art()
  3
O=>=O---O
1   2   0
sage: print CartanType(["F", 4, 1]).dual().ascii_art()
O---O---O=<=O---O
0   1   2   3   4
sage: print CartanType(["BC", 4, 2]).dual().ascii_art()
O=>=O---O---O=>=O
0   1   2   3   4
```

dual()

EXAMPLES:

```
sage: ct = CartanType(['F', 4, 1]).dual()
sage: ct.dual()
['F', 4, 1]
```

dynkin_diagram()

EXAMPLES:

```
sage: ct = CartanType(['F', 4, 1]).dual()
sage: ct.dynkin_diagram()
O---O---O=<=O---O
0   1   2   3   4
```

```
F4~*
```

class sage.combinat.root_system.type_dual.**CartanType_affine**(*type*)

Bases: sage.combinat.root_system.type_dual.CartanType, sage.combinat.root_system.cartan_type.CartanType_affine

INPUT:

- type – a Cartan type

EXAMPLES:

```
sage: ct = CartanType(['F',4,1]).dual()
sage: TestSuite(ct).run()
```

TESTS:

```
sage: ct1 = CartanType(['B',3,1]).dual()
sage: ct2 = CartanType(['B',3,1]).dual()
sage: ct3 = CartanType(['D',4,1]).dual()
sage: ct1 == ct2
True
sage: ct1 == ct3
False
```

Test that the produced Cartan type is in the appropriate abstract classes (see trac ticket #13724):

```
sage: from sage.combinat.root_system import cartan_type
sage: ct = CartanType(['B',3,1]).dual()
sage: TestSuite(ct).run()
sage: isinstance(ct, cartan_type.CartanType_simple)
True
sage: isinstance(ct, cartan_type.CartanType_finite)
False
sage: isinstance(ct, cartan_type.CartanType_affine)
True
sage: isinstance(ct, cartan_type.CartanType_crystallographic)
True
sage: isinstance(ct, cartan_type.CartanType_simply_laced)
False
```

By default, the dual of a reducible and finite type is not constructed as such:

```
sage: ct = CartanType([['B',4],['A',2]]).dual(); ct
C4xA2
```

In order to exercise the dual infrastructure we force the construction as a dual:

```
sage: from sage.combinat.root_system import type_dual
sage: ct = type_dual.CartanType(CartanType([['B',4],['A',2]])); ct
B4xA2^*
sage: isinstance(ct, type_dual.CartanType)
True
sage: TestSuite(ct).run(skip=["_test_pickling"])
sage: isinstance(ct, cartan_type.CartanType_finite)
True
sage: isinstance(ct, cartan_type.CartanType_simple)
False
sage: isinstance(ct, cartan_type.CartanType_affine)
False
sage: isinstance(ct, cartan_type.CartanType_crystallographic)
```

```
True
sage: isinstance(ct, cartan_type.CartanType_simply_laced)
False
```

basic_untwisted()

Return the basic untwisted Cartan type associated with this affine Cartan type.

Given an affine type $X_n^{(r)}$, the basic untwisted type is X_n. In other words, it is the classical Cartan type that is twisted to obtain `self`.

EXAMPLES:

```
sage: CartanType(['A', 7, 2]).basic_untwisted()
['A', 7]
sage: CartanType(['E', 6, 2]).basic_untwisted()
['E', 6]
sage: CartanType(['D', 4, 3]).basic_untwisted()
['D', 4]
```

classical()

Return the classical Cartan type associated with self (which should be affine).

EXAMPLES:

```
sage: CartanType(['A',3,1]).dual().classical()
['A', 3]
sage: CartanType(['B',3,1]).dual().classical()
['C', 3]
sage: CartanType(['F',4,1]).dual().classical()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
sage: CartanType(['BC',4,2]).dual().classical()
['B', 4]
```

special_node()

Implement `CartanType_affine.special_node()`

The special node of the dual of an affine type T is the special node of T.

EXAMPLES:

```
sage: CartanType(['A',3,1]).dual().special_node()
0
sage: CartanType(['B',3,1]).dual().special_node()
0
sage: CartanType(['F',4,1]).dual().special_node()
0
sage: CartanType(['BC',4,2]).dual().special_node()
0
```

class sage.combinat.root_system.type_dual.**CartanType_finite**(*type*)

Bases: sage.combinat.root_system.type_dual.CartanType, sage.combinat.root_system.cartan_type.CartanType_finite

INPUT:

- `type` – a Cartan type

EXAMPLES:

```
sage: ct = CartanType(['F',4,1]).dual()
sage: TestSuite(ct).run()
```

TESTS:

```
sage: ct1 = CartanType(['B',3,1]).dual()
sage: ct2 = CartanType(['B',3,1]).dual()
sage: ct3 = CartanType(['D',4,1]).dual()
sage: ct1 == ct2
True
sage: ct1 == ct3
False
```

Test that the produced Cartan type is in the appropriate abstract classes (see trac ticket #13724):

```
sage: from sage.combinat.root_system import cartan_type
sage: ct = CartanType(['B',3,1]).dual()
sage: TestSuite(ct).run()
sage: isinstance(ct, cartan_type.CartanType_simple)
True
sage: isinstance(ct, cartan_type.CartanType_finite)
False
sage: isinstance(ct, cartan_type.CartanType_affine)
True
sage: isinstance(ct, cartan_type.CartanType_crystallographic)
True
sage: isinstance(ct, cartan_type.CartanType_simply_laced)
False
```

By default, the dual of a reducible and finite type is not constructed as such:

```
sage: ct = CartanType([['B',4],['A',2]]).dual(); ct
C4xA2
```

In order to exercise the dual infrastructure we force the construction as a dual:

```
sage: from sage.combinat.root_system import type_dual
sage: ct = type_dual.CartanType(CartanType([['B',4],['A',2]])); ct
B4xA2^*
sage: isinstance(ct, type_dual.CartanType)
True
sage: TestSuite(ct).run(skip=["_test_pickling"])
sage: isinstance(ct, cartan_type.CartanType_finite)
True
sage: isinstance(ct, cartan_type.CartanType_simple)
False
sage: isinstance(ct, cartan_type.CartanType_affine)
False
sage: isinstance(ct, cartan_type.CartanType_crystallographic)
True
sage: isinstance(ct, cartan_type.CartanType_simply_laced)
False
```

AmbientSpace
alias of AmbientSpace

5.1.230 Extended Affine Weyl Groups

AUTHORS:

- Daniel Bump (2012): initial version
- Daniel Orr (2012): initial version

- Anne Schilling (2012): initial version
- Mark Shimozono (2012): initial version
- Nicolas M. Thiery (2012): initial version
- Mark Shimozono (2013): twisted affine root systems, multiple realizations, GL_n

sage.combinat.root_system.extended_affine_weyl_group.**ExtendedAffineWeylGroup**(*cartan_type*, *general_linear=None*, ***print_options*)

The extended affine Weyl group.

INPUT:

- `cartan_type` – An affine or finite Cartan type (a finite Cartan type is an abbreviation for its untwisted affinization)
- `general_linear` – (default: None) If True and `cartan_type` indicates untwisted type A, returns the universal central extension
- `print_options` – Special instructions for printing elements (see below)

Mnemonics

- "P" – subgroup of translations
- "Pv" – subgroup of translations in a dual form
- "W0" – classical Weyl group
- "W" – affine Weyl group
- "F" – fundamental group of length zero elements

There are currently six realizations: "PW0", "W0P, "WF", "FW", "PvW0", and "W0Pv".

"PW0" means the semidirect product of "P" with "W0" acting from the right. "W0P" is similar but with "W0" acting from the left. "WF" is the semidirect product of "W" with "F" acting from the right, etc.

Recognized arguments for `print_options` are:

- `print_tuple` – True or False (default: False) If True, elements are printed (a, b), otherwise as $a * b$
- `affine` – Prefix for simple reflections in the affine Weyl group
- `classical` – Prefix for simple reflections in the classical Weyl group
- `translation` – Prefix for the translation elements
- `fundamental` – Prefix for the elements of the fundamental group

These options are not mutable.

The *extended affine Weyl group* was introduced in the following references.

REFERENCES:

Notation

- R – An irreducible affine root system
- I – Set of nodes of the Dynkin diagram of R

- R_0 – The classical subsystem of R
- I_0 – Set of nodes of the Dynkin diagram of R_0
- E – Extended affine Weyl group of type R
- W – Affine Weyl group of type R
- W_0 – finite (classical) Weyl group (of type R_0)
- M – translation lattice for W
- L – translation lattice for E
- F – Fundamental subgroup of E (the length zero elements)
- P – Finite weight lattice
- Q – Finite root lattice
- $P^\vee$ – Finite coweight lattice
- $Q^\vee$ – Finite coroot lattice

Translation lattices

The styles "PW0" and "W0P" use the following lattices:

- Untwisted affine: $L = P^\vee$, $M = Q^\vee$
- Dual of untwisted affine: $L = P$, $M = Q$
- BC_n ($A_{2n}^{(2)}$): $L = M = P$
- Dual of BC_n ($A_{2n}^{(2)\dagger}$): $L = M = P^\vee$

The styles "PvW0" and "W0Pv" use the following lattices:

- Untwisted affine: The weight lattice of the dual finite Cartan type.
- Dual untwisted affine: The same as for "PW0" and "W0P".

For mixed affine type ($A_{2n}^{(2)}$, aka $\tilde{BC}_n$, and their affine duals) the styles "PvW0" and "W0Pv" are not implemented.

Finite and affine Weyl groups W_0 and W

The finite Weyl group W_0 is generated by the simple reflections s_i for $i \in I_0$ where s_i is the reflection across a suitable hyperplane H_i through the origin in the real span V of the lattice M.

R specifies another (affine) hyperplane H_0. The affine Weyl group W is generated by W_0 and the reflection S_0 across H_0.

Extended affine Weyl group E

The complement in V of the set H of hyperplanes obtained from the H_i by the action of W, has connected components called alcoves. W acts freely and transitively on the set of alcoves. After the choice of a certain alcove (the fundamental alcove), there is an induced bijection from W to the set of alcoves under which the identity in W maps to the fundamental alcove.

Then L is the largest sublattice of V, whose translations stabilize the set of alcoves.

There are isomorphisms

$$W \cong M \rtimes W_0 \cong W_0 \ltimes M$$
$$E \cong L \rtimes W_0 \cong W_0 \ltimes L$$

Fundamental group of affine Dynkin automorphisms

Since L acts on the set of alcoves, the group $F = L/M$ may be viewed as a subgroup of the symmetries of the fundamental alcove or equivalently the symmetries of the affine Dynkin diagram. F acts on the set of alcoves and hence on W. Conjugation by an element of F acts on W by permuting the indices of simple reflections.

There are isomorphisms

$$E \cong F \ltimes W \cong W \rtimes F$$

An affine Dynkin node is *special* if it is conjugate to the zero node under some affine Dynkin automorphism.

There is a bijection $i \mapsto \pi_i$ from the set of special nodes to the group F, where π_i is the unique element of F that sends 0 to i. When $L = P$ (resp. $L = P^\vee$) the element π_i is induced (under the isomorphism $F \cong L/M$) by addition of the coset of the i-th fundamental weight (resp. coweight).

The length function of the Coxeter group W may be extended to E by $\ell(w\pi) = \ell(w)$ where $w \in W$ and $\pi \in F$. This is the number of hyperplanes in H separating the fundamental alcove from its image by $w\pi$ (or equivalently w).

It is known that if G is the compact Lie group of adjoint type with root system R_0 then F is isomorphic to the fundamental group of G, or to the center of its simply-connected covering group. That is why we call F the *fundamental group*.

In the future we may want to build an element of the group from an appropriate linear map f on some of the root lattice realizations for this cartan type: W.from_endomorphism(f).

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(["A",2,1]); E
Extended affine Weyl group of type ['A', 2, 1]
sage: type(E)
<class 'sage.combinat.root_system.extended_affine_weyl_group.ExtendedAffineWeylGroup_Class_with_

sage: PW0=E.PW0(); PW0
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Multiplicative

sage: W0P = E.W0P(); W0P
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Weyl Group of t

sage: PvW0 = E.PvW0(); PvW0
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Multiplicative

sage: W0Pv = E.W0Pv(); W0Pv
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Weyl Group of t

sage: WF = E.WF(); WF
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Weyl Group of t

sage: FW = E.FW(); FW
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Fundamental gro
```

When the realizations are constructed from each other as above, there are built-in coercions between them.

```
sage: F = E.fundamental_group()
sage: x = WF.from_reduced_word([0,1,2]) * WF(F(2)); x
S0*S1*S2 * pi[2]
sage: FW(x)
pi[2] * S1*S2*S0
sage: W0P(x)
s1*s2*s1 * t[-2*Lambdacheck[1] - Lambdacheck[2]]
sage: PW0(x)
t[Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2*s1
sage: PvW0(x)
t[Lambda[1] + 2*Lambda[2]] * s1*s2*s1
```

The translation lattice and its distinguished basis are obtained from `E`:

```
sage: L = E.lattice(); L
Coweight lattice of the Root system of type ['A', 2]
sage: b = E.lattice_basis(); b
Finite family {1: Lambdacheck[1], 2: Lambdacheck[2]}
```

Translation lattice elements can be coerced into any realization:

```
sage: PW0(b[1]-b[2])
t[Lambdacheck[1] - Lambdacheck[2]]
sage: FW(b[1]-b[2])
pi[2] * S0*S1
```

The dual form of the translation lattice and its basis are similarly obtained:

```
sage: Lv = E.dual_lattice(); Lv
Weight lattice of the Root system of type ['A', 2]
sage: bv = E.dual_lattice_basis(); bv
Finite family {1: Lambda[1], 2: Lambda[2]}
sage: FW(bv[1]-bv[2])
pi[2] * S0*S1
```

The abstract fundamental group is accessed from `E`:

```
sage: F = E.fundamental_group(); F
Fundamental group of type ['A', 2, 1]
```

Its elements are indexed by the set of special nodes of the affine Dynkin diagram:

```
sage: E.cartan_type().special_nodes()
(0, 1, 2)
sage: F.special_nodes()
(0, 1, 2)
sage: [F(i) for i in F.special_nodes()]
[pi[0], pi[1], pi[2]]
```

There is a coercion from the fundamental group into each realization:

```
sage: F(2)
pi[2]
sage: WF(F(2))
pi[2]
sage: W0P(F(2))
s2*s1 * t[-Lambdacheck[1]]
sage: W0Pv(F(2))
s2*s1 * t[-Lambda[1]]
```

Using E one may access the classical and affine Weyl groups and their morphisms into each realization:

```
sage: W0 = E.classical_weyl(); W0
Weyl Group of type ['A', 2] (as a matrix group acting on the coweight lattice)
sage: v = W0.from_reduced_word([1,2,1]); v
s1*s2*s1
sage: PW0(v)
s1*s2*s1
sage: WF(v)
S1*S2*S1
sage: W = E.affine_weyl(); W
Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root lattice)
sage: w = W.from_reduced_word([2,1,0]); w
S2*S1*S0
sage: WF(w)
S2*S1*S0
sage: PW0(w)
t[Lambdacheck[1] - 2*Lambdacheck[2]] * s1
```

Note that for untwisted affine type the dual form of the classical Weyl group is isomorphic to the usual one, but acts on a different lattice and is therefore different to sage:

```
sage: W0v = E.dual_classical_weyl(); W0v
Weyl Group of type ['A', 2] (as a matrix group acting on the weight lattice)
sage: v = W0v.from_reduced_word([1,2])
sage: x = PvW0(v); x
s1*s2
sage: y = PW0(v); y
s1*s2
sage: x == y
False
sage: x.parent() == y.parent()
False
```

An element can be created directly from a reduced word:

```
sage: PW0.from_reduced_word([2,1,0])
t[Lambdacheck[1] - 2*Lambdacheck[2]] * s1
```

Here is a demonstration of the printing options:

```
sage: E = ExtendedAffineWeylGroup(["A",2,1], affine="sx", classical="Sx",translation="x",fundame
sage: PW0 = E.PW0()
sage: y = PW0(E.lattice_basis()[1])
sage: y
x[Lambdacheck[1]]
sage: FW = E.FW()
sage: FW(y)
pix[1] * sx2*sx1
sage: PW0.an_element()
x[2*Lambdacheck[1] + 2*Lambdacheck[2]] * Sx1*Sx2
```

Todo

- Implement a "slow" action of E on any affine root or weight lattice realization.
- Implement the level m actions of E and W on the lattices of finite type.
- Implement the relevant methods from the usual affine weyl group
- Implementation by matrices: style "M".

•Use case: implement the Hecke algebra on top of this

The semidirect product construction in sage currently only admits multiplicative groups. Therefore for the styles involving "P" and "Pv", one must convert the additive group of translations L into a multiplicative group by applying the `sage.groups.group_exp.GroupExp` functor.

The general linear case

The general linear group is not semisimple. Sage can build its extended affine Weyl group:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1], general_linear=True); E
Extended affine Weyl group of GL(3)
```

If the Cartan type is `['A', n-1, 1]` and the parameter `general_linear` is not True, the extended affine Weyl group that is built will be for SL_n, not GL_n. But if `general_linear` is True, let W_a and W_e be the affine and extended affine Weyl groups. We make the following nonstandard definition: the extended affine Weyl group $W_e(GL_n)$ is defined by

$$W_e(GL_n) = P(GL_n) \rtimes W$$

where W is the finite Weyl group (the symmetric group S_n) and $P(GL_n)$ is the weight lattice of GL_n, which is usually identified with the lattice $\mathbf{Z}^n$ of n-tuples of integers:

```
sage: PW0 = E.PW0(); PW0
Extended affine Weyl group of GL(3) realized by Semidirect product of Multiplicative form of Amb
sage: PW0.an_element()
t[(2, 2, 3)] * s1*s2
```

There is an isomorphism

$$W_e(GL_n) = \mathbf{Z} \ltimes W_a$$

where the group of integers $\mathbf{Z}$ (with generator π) acts on W_a by

$$\pi\, s_i\, \pi^{-1} = s_{i+1}$$

and the indices of the simple reflections are taken modulo n:

```
sage: FW = E.FW(); FW
Extended affine Weyl group of GL(3) realized by Semidirect product of Fundamental group of GL(3)
sage: FW.an_element()
pi[5] * S0*S1*S2
```

We regard $\mathbf{Z}$ as the fundamental group of affine type GL_n:

```
sage: F = E.fundamental_group(); F
Fundamental group of GL(3)
sage: F.special_nodes()
Integer Ring

sage: x = FW.from_fundamental(F(10)); x
pi[10]
sage: x*x
pi[20]
sage: E.PvW0()(x*x)
t[(7, 7, 6)] * s2*s1
```

class sage.combinat.root_system.extended_affine_weyl_group.**ExtendedAffineWeylGroup_Class**(*carta gen- eral_ **pri*

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The parent-with-realization class of an extended affine Weyl group.

class ExtendedAffineWeylGroupFW(*E*)

Bases: sage.groups.group_semidirect_product.GroupSemidirectProduct, sage.misc.bindable_class.BindableClass

Extended affine Weyl group, realized as the semidirect product of the affine Weyl group by the fundamental group.

INPUT:

- *E* – A parent with realization in ExtendedAffineWeylGroup_Class

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).FW()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Fundamental
```

from_affine_weyl(*w*)

Return the image of *w* under the map of the affine Weyl group into the right (affine Weyl group) factor in the "FW" style.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1],print_tuple=True)
sage: E.FW().from_affine_weyl(E.affine_weyl().from_reduced_word([0,2,1]))
(pi[0], S0*S2*S1)
```

from_fundamental(*f*)

Return the image of the fundamental group element *f* into self.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1],print_tuple=True)
sage: E.FW().from_fundamental(E.fundamental_group()(2))
(pi[2], 1)
```

simple_reflections()

Return the family of simple reflections of self.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1],print_tuple=True).FW().simple_reflections()
Finite family {0: (pi[0], S0), 1: (pi[0], S1), 2: (pi[0], S2)}
```

class ExtendedAffineWeylGroup_Class.**ExtendedAffineWeylGroupFWElement**

Bases: sage.groups.group_semidirect_product.GroupSemidirectProductElement

The element class for the "FW" realization.

action_on_affine_roots(*beta*)

Act by self on the affine root lattice element beta.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1],affine="s")
sage: x = E.FW().an_element(); x
pi[2] * s0*s1*s2
```

```
sage: v = RootSystem(['A',2,1]).root_lattice().an_element(); v
2*alpha[0] + 2*alpha[1] + 3*alpha[2]
sage: x.action_on_affine_roots(v)
alpha[0] + alpha[1]
```

has_descent(*i*, *side='right'*, *positive=False*)

Return whether `self` has descent at i.

INPUT:

- i – an affine Dynkin index.

OPTIONAL:

- `side` – 'left' or 'right' (default: 'right')
- `positive` – True or False (default: False)

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.FW().an_element(); x
pi[2] * S0*S1*S2
sage: [(i, x.has_descent(i)) for i in E.cartan_type().index_set()]
[(0, False), (1, False), (2, True)]
```

to_affine_weyl_right()

Project `self` to the right (affine Weyl group) factor in the "FW" style.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.FW().from_translation(E.lattice_basis()[1]); x
pi[1] * S2*S1
sage: x.to_affine_weyl_right()
S2*S1
```

to_fundamental_group()

Return the projection of `self` to the fundamental group in the "FW" style.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.FW().from_translation(E.lattice_basis()[2]); x
pi[2] * S1*S2
sage: x.to_fundamental_group()
pi[2]
```

class ExtendedAffineWeylGroup_Class.**ExtendedAffineWeylGroupPW0**(*E*)

Bases: sage.groups.group_semidirect_product.GroupSemidirectProduct, sage.misc.bindable_class.BindableClass

Extended affine Weyl group, realized as the semidirect product of the translation lattice by the finite Weyl group.

INPUT:

- E – A parent with realization in `ExtendedAffineWeylGroup_Class`

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).PW0()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Multiplicat
```

S0()

Return the affine simple reflection.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['B',2]).PW0().S0()
t[Lambdacheck[2]] * s2*s1*s2
```

from_classical_weyl(*w*)
Return the image of w under the homomorphism of the classical Weyl group into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup("A3",print_tuple=True)
sage: E.PW0().from_classical_weyl(E.classical_weyl().from_reduced_word([1,2]))
(t[0], s1*s2)
```

from_translation(*la*)
Map the translation lattice element `la` into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1], translation="tau", print_tuple = True)
sage: la = E.lattice().an_element(); la
2*Lambdacheck[1] + 2*Lambdacheck[2]
sage: E.PW0().from_translation(la)
(tau[2*Lambdacheck[1] + 2*Lambdacheck[2]], 1)
```

simple_reflection(*i*)
Return the i-th simple reflection in `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup("G2")
sage: [(i, E.PW0().simple_reflection(i)) for i in E.cartan_type().index_set()]
[(0, t[Lambdacheck[2]] * s2*s1*s2*s1*s2), (1, s1), (2, s2)]
```

simple_reflections()
Return a family for the simple reflections of `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup("A3").PW0().simple_reflections()
Finite family {0: t[Lambdacheck[1] + Lambdacheck[3]] * s1*s2*s3*s2*s1, 1: s1, 2: s2, 3: s
```

class ExtendedAffineWeylGroup_Class.**ExtendedAffineWeylGroupPW0Element**
Bases: sage.groups.group_semidirect_product.GroupSemidirectProductElement

The element class for the "PW0" realization.

action(*la*)
Return the action of `self` on an element `la` of the translation lattice.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1]); PW0=E.PW0()
sage: x = PW0.an_element(); x
t[2*Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2
sage: la = E.lattice().an_element(); la
2*Lambdacheck[1] + 2*Lambdacheck[2]
sage: x.action(la)
-2*Lambdacheck[1] + 4*Lambdacheck[2]
```

has_descent(*i*, *side='right'*, *positive=False*)
Return whether `self` has i as a descent.

INPUT:

- i – an affine Dynkin node

OPTIONAL:

- `side` – 'left' or 'right' (default: 'right')

- positive – True or False (default: False)

EXAMPLES:

```
sage: w = ExtendedAffineWeylGroup(['A',4,2]).PW0().from_reduced_word([0,1]); w
t[Lambda[1]] * s1*s2
sage: w.has_descent(0, side='left')
True
```

to_classical_weyl()

Return the image of self under the homomorphism that projects to the classical Weyl group factor after rewriting it in either style "PW0" or "W0P".

EXAMPLES:

```
sage: s = ExtendedAffineWeylGroup(['A',2,1]).PW0().S0(); s
t[Lambdacheck[1] + Lambdacheck[2]] * s1*s2*s1
sage: s.to_classical_weyl()
s1*s2*s1
```

to_translation_left()

The image of self under the map that projects to the translation lattice factor after factoring it to the left as in style "PW0".

EXAMPLES:

```
sage: s = ExtendedAffineWeylGroup(['A',2,1]).PW0().S0(); s
t[Lambdacheck[1] + Lambdacheck[2]] * s1*s2*s1
sage: s.to_translation_left()
Lambdacheck[1] + Lambdacheck[2]
```

class ExtendedAffineWeylGroup_Class.**ExtendedAffineWeylGroupPvW0**(*E*)

Bases: sage.groups.group_semidirect_product.GroupSemidirectProduct, sage.misc.bindable_class.BindableClass

Extended affine Weyl group, realized as the semidirect product of the dual form of the translation lattice by the finite Weyl group.

INPUT:

- *E* – A parent with realization in ExtendedAffineWeylGroup_Class

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).PvW0()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Multiplicat
```

from_dual_classical_weyl(*w*)

Return the image of *w* under the homomorphism of the dual form of the classical Weyl group into self.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1],print_tuple=True)
sage: E.PvW0().from_dual_classical_weyl(E.dual_classical_weyl().from_reduced_word([1,2]))
(t[0], s1*s2)
```

from_dual_translation(*la*)

Map the dual translation lattice element la into self.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1], translation="tau", print_tuple = True)
sage: la = E.dual_lattice().an_element(); la
2*Lambda[1] + 2*Lambda[2]
sage: E.PvW0().from_dual_translation(la)
(tau[2*Lambda[1] + 2*Lambda[2]], 1)
```

simple_reflections()
Return a family for the simple reflections of self.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).PvW0().simple_reflections()
Finite family {0: t[Lambda[1] + Lambda[3]] * s1*s2*s3*s2*s1, 1: s1, 2: s2, 3: s3}
```

class ExtendedAffineWeylGroup_Class.**ExtendedAffineWeylGroupPvW0Element**
Bases: sage.groups.group_semidirect_product.GroupSemidirectProductElement

The element class for the "PvW0" realization.

dual_action(*la*)
Return the action of self on an element la of the dual version of the translation lattice.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.PvW0().an_element(); x
t[2*Lambda[1] + 2*Lambda[2]] * s1*s2
sage: la = E.dual_lattice().an_element(); la
2*Lambda[1] + 2*Lambda[2]
sage: x.dual_action(la)
-2*Lambda[1] + 4*Lambda[2]
```

has_descent(*i*, *side='right'*, *positive=False*)
Return whether self has i as a descent.

INPUT:

- i - an affine Dynkin index

OPTIONAL:

- side – 'left' or 'right' (default: 'right')
- positive – True or False (default: False)

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',4,2])
sage: w = E.PvW0().from_reduced_word([0,1]); w
t[Lambda[1]] * s1*s2
sage: [(i, w.has_descent(i, side='left')) for i in E.cartan_type().index_set()]
[(0, True), (1, False), (2, False)]
```

to_dual_classical_weyl()
Return the image of self under the homomorphism that projects to the dual classical Weyl group factor after rewriting it in either style "PvW0" or "W0Pv".

EXAMPLES:

```
sage: s = ExtendedAffineWeylGroup(['A',2,1]).PvW0().simple_reflection(0); s
t[Lambda[1] + Lambda[2]] * s1*s2*s1
sage: s.to_dual_classical_weyl()
s1*s2*s1
```

to_dual_translation_left()
The image of self under the map that projects to the dual translation lattice factor after factoring it to the left as in style "PvW0".

EXAMPLES:

```
sage: s = ExtendedAffineWeylGroup(['A',2,1]).PvW0().simple_reflection(0); s
t[Lambda[1] + Lambda[2]] * s1*s2*s1
sage: s.to_dual_translation_left()
Lambda[1] + Lambda[2]
```

class ExtendedAffineWeylGroup_Class.**ExtendedAffineWeylGroupW0P**(*E*)
Bases: sage.groups.group_semidirect_product.GroupSemidirectProduct, sage.misc.bindable_class.BindableClass

Extended affine Weyl group, realized as the semidirect product of the finite Weyl group by the translation lattice.

INPUT:

- *E* – A parent with realization in ExtendedAffineWeylGroup_Class

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).W0P()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Weyl Group
```

S0()
Return the zero-th simple reflection in style "W0P".

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(["A",3,1]).W0P().S0()
s1*s2*s3*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[3]]
```

from_classical_weyl(*w*)
Return the image of the classical Weyl group element *w* in self.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1],print_tuple=True)
sage: E.W0P().from_classical_weyl(E.classical_weyl().from_reduced_word([2,1]))
(s2*s1, t[0])
```

from_translation(*la*)
Return the image of the lattice element la in self.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1],print_tuple=True)
sage: E.W0P().from_translation(E.lattice().an_element())
(1, t[2*Lambdacheck[1] + 2*Lambdacheck[2]])
```

simple_reflection(*i*)
Return the *i*-th simple reflection in self.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); W0P = E.W0P()
sage: [(i, W0P.simple_reflection(i)) for i in E.cartan_type().index_set()]
[(0, s1*s2*s3*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[3]]), (1, s1), (2, s2), (3, s3)]
```

simple_reflections()
Return the family of simple reflections.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(["A",3,1]).W0P().simple_reflections()
Finite family {0: s1*s2*s3*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[3]], 1: s1, 2: s2, 3:
```

class ExtendedAffineWeylGroup_Class.**ExtendedAffineWeylGroupW0PElement**
Bases: sage.groups.group_semidirect_product.GroupSemidirectProductElement

The element class for the W0P realization.

has_descent(*i*, *side='right'*, *positive=False*)
Return whether `self` has i as a descent.

INPUT:
- i - an index.

OPTIONAL:
- `side` - 'left' or 'right' (default: 'right')
- `positive` - True or False (default: False)

EXAMPLES:
```
sage: W0P = ExtendedAffineWeylGroup(['A',4,2]).W0P()
sage: w = W0P.from_reduced_word([0,1]); w
s1*s2 * t[Lambda[1] - Lambda[2]]
sage: w.has_descent(0, side='left')
True
```

to_classical_weyl()
Project `self` into the classical Weyl group.

EXAMPLES:
```
sage: x = ExtendedAffineWeylGroup(['A',2,1]).W0P().simple_reflection(0); x;
s1*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[2]]
sage: x.to_classical_weyl()
s1*s2*s1
```

to_translation_right()
Project onto the right (translation) factor in the "W0P" style.

EXAMPLES:
```
sage: x = ExtendedAffineWeylGroup(['A',2,1]).W0P().simple_reflection(0); x
s1*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[2]]
sage: x.to_translation_right()
-Lambdacheck[1] - Lambdacheck[2]
```

class ExtendedAffineWeylGroup_Class.**ExtendedAffineWeylGroupW0Pv**(*E*)
Bases: sage.groups.group_semidirect_product.GroupSemidirectProduct, sage.misc.bindable_class.BindableClass

Extended affine Weyl group, realized as the semidirect product of the finite Weyl group, acting on the dual form of the translation lattice.

INPUT:
- E – A parent with realization in ExtendedAffineWeylGroup_Class

EXAMPLES:
```
sage: ExtendedAffineWeylGroup(['A',2,1]).W0Pv()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Weyl Group
```

from_dual_classical_weyl(*w*)
Return the image of w under the homomorphism of the dual form of the classical Weyl group into `self`.

EXAMPLES:
```
sage: E = ExtendedAffineWeylGroup(['A',3,1],print_tuple=True)
sage: E.W0Pv().from_dual_classical_weyl(E.dual_classical_weyl().from_reduced_word([1,2]))
(s1*s2, t[0])
```

from_dual_translation(*la*)
Map the dual translation lattice element `la` into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1], translation="tau", print_tuple = True)
sage: la = E.dual_lattice().an_element(); la
2*Lambda[1] + 2*Lambda[2]
sage: E.W0Pv().from_dual_translation(la)
(1, tau[2*Lambda[1] + 2*Lambda[2]])
```

simple_reflections()

Return a family for the simple reflections of `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).W0Pv().simple_reflections()
Finite family {0: s1*s2*s3*s2*s1 * t[-Lambda[1] - Lambda[3]], 1: s1, 2: s2, 3: s3}
```

class ExtendedAffineWeylGroup_Class.**ExtendedAffineWeylGroupW0PvElement**

Bases: sage.groups.group_semidirect_product.GroupSemidirectProductElement

The element class for the "W0Pv" realization.

dual_action(*la*)

Return the action of `self` on an element `la` of the dual version of the translation lattice.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.W0Pv().an_element(); x
s1*s2 * t[2*Lambda[1] + 2*Lambda[2]]
sage: la = E.dual_lattice().an_element(); la
2*Lambda[1] + 2*Lambda[2]
sage: x.dual_action(la)
-8*Lambda[1] + 4*Lambda[2]
```

has_descent(*i*, *side='right'*, *positive=False*)

Return whether `self` has i as a descent.

INPUT:

- i - an affine Dynkin index

OPTIONAL:

- `side` - 'left' or 'right' (default: 'right')
- `positive` - True or False (default: False)

EXAMPLES:

```
sage: w = ExtendedAffineWeylGroup(['A',4,2]).W0Pv().from_reduced_word([0,1]); w
s1*s2 * t[Lambda[1] - Lambda[2]]
sage: w.has_descent(0, side='left')
True
```

to_dual_classical_weyl()

Return the image of `self` under the homomorphism that projects to the dual classical Weyl group factor after rewriting it in either style "PvW0" or "W0Pv".

EXAMPLES:

```
sage: s = ExtendedAffineWeylGroup(['A',2,1]).W0Pv().simple_reflection(0); s
s1*s2*s1 * t[-Lambda[1] - Lambda[2]]
sage: s.to_dual_classical_weyl()
s1*s2*s1
```

to_dual_translation_right()

The image of `self` under the map that projects to the dual translation lattice factor after factoring it to the right as in style "W0Pv".

EXAMPLES:

```
sage: s = ExtendedAffineWeylGroup(['A',2,1]).W0Pv().simple_reflection(0); s
s1*s2*s1 * t[-Lambda[1] - Lambda[2]]
sage: s.to_dual_translation_right()
-Lambda[1] - Lambda[2]
```

class ExtendedAffineWeylGroup_Class.**ExtendedAffineWeylGroupWF**(E)
Bases: sage.groups.group_semidirect_product.GroupSemidirectProduct, sage.misc.bindable_class.BindableClass

Extended affine Weyl group, realized as the semidirect product of the affine Weyl group by the fundamental group.

INPUT:

- E – A parent with realization in ExtendedAffineWeylGroup_Class

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).WF()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Weyl Group
```

from_affine_weyl(w)
Return the image of the affine Weyl group element w in self.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['C',2,1],print_tuple=True)
sage: E.WF().from_affine_weyl(E.affine_weyl().from_reduced_word([1,2,1,0]))
(S1*S2*S1*S0, pi[0])
```

from_fundamental(f)
Return the image of f under the homomorphism from the fundamental group into the right (fundamental group) factor in "WF" style.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['E',6,1],print_tuple=True); WF = E.WF(); F = E.fundam
sage: [(x,WF.from_fundamental(x)) for x in F]
[(pi[0], (1, pi[0])), (pi[1], (1, pi[1])), (pi[6], (1, pi[6]))]
```

simple_reflections()
Return the family of simple reflections.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(["A",3,1],affine="r").WF().simple_reflections()
Finite family {0: r0, 1: r1, 2: r2, 3: r3}
```

class ExtendedAffineWeylGroup_Class.**ExtendedAffineWeylGroupWFElement**
Bases: sage.groups.group_semidirect_product.GroupSemidirectProductElement

Element class for the "WF" realization.

bruhat_le(x)
Return whether self is less than or equal to x in the Bruhat order.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1],affine="s", print_tuple=True); WF=E.WF()
sage: r = E.affine_weyl().from_reduced_word
sage: v = r([1,0])
sage: w = r([1,2,0])
sage: v.bruhat_le(w)
True
sage: vv = WF.from_affine_weyl(v); vv
```

```
(s1*s0, pi[0])
sage: ww = WF.from_affine_weyl(w); ww
(s1*s2*s0, pi[0])
sage: vv.bruhat_le(ww)
True
sage: f = E.fundamental_group()(2); f
pi[2]
sage: ff = WF.from_fundamental(f); ff
(1, pi[2])
sage: vv.bruhat_le(ww*ff)
False
sage: (vv*ff).bruhat_le(ww*ff)
True
```

has_descent(*i*, *side='right'*, *positive=False*)

Return whether `self` has i as a descent.

INPUT:

- i – an affine Dynkin index

OPTIONAL:

- `side` – 'left' or 'right' (default: 'right')
- `positive` – True or False (default: False)

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.WF().an_element(); x
S0*S1*S2 * pi[2]
sage: [(i, x.has_descent(i)) for i in E.cartan_type().index_set()]
[(0, True), (1, False), (2, False)]
```

to_affine_weyl_left()

Project `self` to the left (affine Weyl group) factor in the "WF" style.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.WF().from_translation(E.lattice_basis()[1]); x
S0*S2 * pi[1]
sage: x.to_affine_weyl_left()
S0*S2
```

to_fundamental_group()

Project `self` to the right (fundamental group) factor in the "WF" style.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: x = E.WF().from_translation(E.lattice_basis()[1]); x
S0*S2 * pi[1]
sage: x.to_fundamental_group()
pi[1]
```

ExtendedAffineWeylGroup_Class.**FW**()

Realizes `self` in "FW"-style.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).FW()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Fundamental
```

ExtendedAffineWeylGroup_Class.**PW0**()

Realizes `self` in "PW0"-style.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).PW0()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Multiplicat
```

ExtendedAffineWeylGroup_Class.**PW0_to_WF_func**(*x*)

Implements coercion from style "PW0" to "WF".

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(["A", 2, 1])
sage: x = E.PW0().an_element(); x
t[2*Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2
sage: E.PW0_to_WF_func(x)
S0*S1*S2*S0*S1*S0
```

Warning: This function cannot use coercion, because it is used to define the coercion maps.

ExtendedAffineWeylGroup_Class.**PvW0**()

Realizes `self` in "PvW0"-style.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).PvW0()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Multiplicat
```

class ExtendedAffineWeylGroup_Class.**Realizations**(*parent_with_realization*)

Bases: `sage.categories.realizations.Category_realization_of_parent`

The category of the realizations of an extended affine Weyl group

class ElementMethods

action(*la*)

Action of `self` on a lattice element `la`.

INPUT:

- `self` – an element of the extended affine Weyl group
- `la` – an element of the translation lattice of the extended affine Weyl group, the lattice denoted by the mnemonic "P" in the documentation for `ExtendedAffineWeylGroup()`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1],affine="s")
sage: x = E.FW().an_element(); x
pi[2] * s0*s1*s2
sage: la = E.lattice().an_element(); la
2*Lambdacheck[1] + 2*Lambdacheck[2]
sage: x.action(la)
5*Lambdacheck[1] - 3*Lambdacheck[2]
sage: E = ExtendedAffineWeylGroup(['C',2,1],affine="s")
sage: x = E.PW0().from_translation(E.lattice_basis()[1])
sage: x.action(E.lattice_basis()[2])
Lambdacheck[1] + Lambdacheck[2]
```

Warning: Must be implemented by style "PW0".

action_on_affine_roots(*beta*)

Act by `self` on the affine root lattice element `beta`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: beta = E.cartan_type().root_system().root_lattice().an_element(); beta
2*alpha[0] + 2*alpha[1] + 3*alpha[2]
sage: x = E.FW().an_element(); x
pi[2] * S0*S1*S2
sage: x.action_on_affine_roots(beta)
alpha[0] + alpha[1]
```

Warning: Must be implemented by style "FW".

alcove_walk_signs()

Return a signed alcove walk for `self`.

INPUT:

•An element `self` of the extended affine Weyl group.

OUTPUT:

•A 3-tuple (g, `rw`, `signs`).

ALGORITHM:

The element `self` can be uniquely written `self` = g * w where g has length zero and w is an element of the nonextended affine Weyl group. Let w have reduced word `rw`. Starting with g and applying simple reflections from `rw`, one obtains a sequence of extended affine Weyl group elements (that is, alcoves) and simple roots. The signs give the sequence of sides on which the alcoves lie, relative to the face indicated by the simple roots.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); FW=E.FW()
sage: w = FW.from_reduced_word([0,2,1,3,0])*FW.from_fundamental(1); w
pi[1] * S3*S1*S2*S0*S3
sage: w.alcove_walk_signs()
(pi[1], [3, 1, 2, 0, 3], [-1, 1, -1, -1, 1])
```

apply_simple_projection(*i*, *side='right'*, *length_increasing=True*)

Return the product of `self` by the simple reflection s_i if that product is of greater length than `self` and otherwise return `self`.

INPUT:

•`self` – an element of the extended affine Weyl group

•i – a Dynkin node (index of a simple reflection s_i)

•`side` – 'right' or 'left' (default: 'right') according to which side of `self` the reflection s_i should be multiplied

•`length_increasing` – True or False (default True). If False do the above with the word "greater" replaced by "less".

EXAMPLES:

```
sage: x = ExtendedAffineWeylGroup(['A',3,1]).WF().an_element(); x
S0*S1*S2*S3 * pi[3]
sage: x.apply_simple_projection(1)
S0*S1*S2*S3*S0 * pi[3]
sage: x.apply_simple_projection(1, length_increasing=False)
S0*S1*S2*S3 * pi[3]
```

apply_simple_reflection(*i*, *side='right'*)

Apply the i-th simple reflection to `self`.

EXAMPLES:

```
sage: x = ExtendedAffineWeylGroup(['A',3,1]).WF().an_element(); x
S0*S1*S2*S3 * pi[3]
sage: x.apply_simple_reflection(1)
S0*S1*S2*S3*S0 * pi[3]
sage: x.apply_simple_reflection(0, side='left')
S1*S2*S3 * pi[3]
```

bruhat_le(x)

Return whether `self` <= x in Bruhat order.

INPUT:

- `self` – an element of the extended affine Weyl group
- x – another element with the same parent as `self`

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1],print_tuple=True); WF=E.WF()
sage: W = E.affine_weyl()
sage: v = W.from_reduced_word([2,1,0])
sage: w = W.from_reduced_word([2,0,1,0])
sage: v.bruhat_le(w)
True
sage: vx = WF.from_affine_weyl(v); vx
(S2*S1*S0, pi[0])
sage: wx = WF.from_affine_weyl(w); wx
(S2*S0*S1*S0, pi[0])
sage: vx.bruhat_le(wx)
True
sage: F = E.fundamental_group()
sage: f = WF.from_fundamental(F(2))
sage: vx.bruhat_le(wx*f)
False
sage: (vx*f).bruhat_le(wx*f)
True
```

Warning: Must be implemented by "WF".

coset_representative(*index_set*, *side='right'*)

Return the minimum length representative in the coset of `self` with respect to the subgroup generated by the reflections given by `index_set`.

INPUT:

- `self` – an element of the extended affine Weyl group
- `index_set` – a subset of the set of Dynkin nodes
- `side` – 'right' or 'left' (default: 'right') the side on which the subgroup acts

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); WF = E.WF()
sage: b = E.lattice_basis()
sage: I0 = E.cartan_type().classical().index_set()
sage: [WF.from_translation(x).coset_representative(index_set=I0) for x in b]
[pi[1], pi[2], pi[3]]
```

dual_action(*la*)

Action of `self` on a dual lattice element `la`.

INPUT:

- `self` – an element of the extended affine Weyl group
- `la` – an element of the dual translation lattice of the extended affine Weyl group, the lattice denoted by the mnemonic "Pv" in the documentation for `ExtendedAffineWeylGroup()`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1],affine="s")
sage: x = E.FW().an_element(); x
pi[2] * s0*s1*s2
sage: la = E.dual_lattice().an_element(); la
2*Lambda[1] + 2*Lambda[2]
sage: x.dual_action(la)
5*Lambda[1] - 3*Lambda[2]
sage: E = ExtendedAffineWeylGroup(['C',2,1],affine="s")
sage: x = E.PvW0().from_dual_translation(E.dual_lattice_basis()[1])
sage: x.dual_action(E.dual_lattice_basis()[2])
Lambda[1] + Lambda[2]
```

Warning: Must be implemented by style "PvW0".

face_data(*i*)

Return a description of one of the bounding hyperplanes of the alcove of an extended affine Weyl group element.

INPUT:

- `self` – An element of the extended affine Weyl group
- i – an affine Dynkin node

OUTPUT:

- A 2-tuple (m, β) defined as follows.

ALGORITHM:

Each element of the extended affine Weyl group corresponds to an alcove, and each alcove has a face for each affine Dynkin node. Given the data of `self` and i, let the extended affine Weyl group element `self` act on the affine simple root α_i, yielding a real affine root, which can be expressed uniquely as

$$"self" \cdot \alpha_i = m\delta + \beta$$

where m is an integer (the height of the i-th bounding hyperplane of the alcove of `self`) and β is a classical root (the normal vector for the hyperplane which points towards the alcove).

EXAMPLES:

```
sage: x = ExtendedAffineWeylGroup(['A',2,1]).PW0().an_element(); x
t[2*Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2
sage: x.face_data(0)
(-1, alpha[1])
```

first_descent(*side='right'*, *positive=False*, *index_set=None*)

Return the first descent of `self`.

INPUT:

- `side` – 'left' or 'right' (default: 'right')
- `positive` – True or False (default: False)
- `index_set` – an optional subset of Dynkin nodes

If `index_set` is not None, then the descent must be in the `index_set`.

EXAMPLES:

```
sage: x = ExtendedAffineWeylGroup(['A',3,1]).WF().an_element(); x
S0*S1*S2*S3 * pi[3]
sage: x.first_descent()
0
sage: x.first_descent(side='left')
0
```

```
sage: x.first_descent(positive=True)
1
sage: x.first_descent(side='left',positive=True)
1
```

has_descent(*i*, *side='right'*, *positive=False*)

Return whether `self` * s_i < `self` where s_i is the i-th simple reflection in the realized group.

INPUT:

- `i` – an affine Dynkin index

OPTIONAL:

- `side` – 'right' or 'left' (default: 'right')
- `positive` – True or False (default: False)

If `side``='left'` then the reflection acts on the left. If ``positive` = True then the inequality is reversed.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); WF=E.WF()
sage: F = E.fundamental_group()
sage: x = WF.an_element(); x
S0*S1*S2*S3 * pi[3]
sage: I = E.cartan_type().index_set()
sage: [(i, x.has_descent(i)) for i in I]
[(0, True), (1, False), (2, False), (3, False)]
sage: [(i, x.has_descent(i,side='left')) for i in I]
[(0, True), (1, False), (2, False), (3, False)]
sage: [(i, x.has_descent(i,positive=True)) for i in I]
[(0, False), (1, True), (2, True), (3, True)]
```

Warning: This method is abstract because it is used in the recursive coercions between "PW0" and "WF" and other methods use this coercion.

is_affine_grassmannian()

Return whether `self` is affine Grassmannian.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1]); PW0=E.PW0()
sage: F = E.fundamental_group()
sage: [(x,PW0.from_fundamental(x).is_affine_grassmannian()) for x in F]
[(pi[0], True), (pi[1], True), (pi[2], True)]
sage: b = E.lattice_basis()
sage: [(-x,PW0.from_translation(-x).is_affine_grassmannian()) for x in b]
[(-Lambdacheck[1], True), (-Lambdacheck[2], True)]
```

is_grassmannian(*index_set*, *side='right'*)

Return whether `self` is of minimum length in its coset with respect to the subgroup generated by the reflections of `index_set`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); PW0=E.PW0()
sage: x = PW0.from_translation(E.lattice_basis()[1]); x
t[Lambdacheck[1]]
sage: I = E.cartan_type().index_set()
sage: [(i, x.is_grassmannian(index_set=[i])) for i in I]
[(0, True), (1, False), (2, True), (3, True)]
sage: [(i, x.is_grassmannian(index_set=[i], side='left')) for i in I]
[(0, False), (1, True), (2, True), (3, True)]
```

is_translation()
Return whether `self` is a translation element or not.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1]); FW=E.FW()
sage: F = E.fundamental_group()
sage: FW.from_affine_weyl(E.affine_weyl().from_reduced_word([1,2,1,0])).is_translatio
True
sage: FW.from_translation(E.lattice_basis()[1]).is_translation()
True
sage: FW.simple_reflection(0).is_translation()
False
```

length()
Return the length of `self` in the Coxeter group sense.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); PW0=E.PW0()
sage: I0 = E.cartan_type().classical().index_set()
sage: [PW0.from_translation(E.lattice_basis()[i]).length() for i in I0]
[3, 4, 3]
```

to_affine_grassmannian()
Return the unique affine Grassmannian element in the same coset of `self` with respect to the finite Weyl group acting on the right.

EXAMPLES:

```
sage: elts = ExtendedAffineWeylGroup(['A',2,1]).PW0().some_elements()
sage: [(x, x.to_affine_grassmannian()) for x in elts]
[(t[2*Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2, t[2*Lambdacheck[1] + 2*Lambdacheck[
```

to_affine_weyl_left()
Return the projection of `self` to the affine Weyl group on the left, after factorizing using the style "WF".

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); PW0 = E.PW0()
sage: b = E.lattice_basis()
sage: [(x,PW0.from_translation(x).to_affine_weyl_left()) for x in b]
[(Lambdacheck[1], S0*S3*S2), (Lambdacheck[2], S0*S3*S1*S0), (Lambdacheck[3], S0*S1*S2
```

Warning: Must be implemented in style "WF".

to_affine_weyl_right()
Return the projection of `self` to the affine Weyl group on the right, after factorizing using the style "FW".

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); PW0=E.PW0()
sage: b = E.lattice_basis()
sage: [(x,PW0.from_translation(x).to_affine_weyl_right()) for x in b]
[(Lambdacheck[1], S3*S2*S1), (Lambdacheck[2], S2*S3*S1*S2), (Lambdacheck[3], S1*S2*S3
```

Warning: Must be implemented in style "FW".

to_classical_weyl()
Return the image of `self` under the homomorphism to the classical Weyl group.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).WF().simple_reflection(0).to_classical_weyl(
s1*s2*s3*s2*s1
```

Warning: Must be implemented in style "PW0".

to_dual_classical_weyl()

Return the image of `self` under the homomorphism to the dual form of the classical Weyl group.

EXAMPLES:

```
sage: x = ExtendedAffineWeylGroup(['A',3,1]).WF().simple_reflection(0).to_dual_classi
s1*s2*s3*s2*s1
sage: x.parent()
Weyl Group of type ['A', 3] (as a matrix group acting on the weight lattice)
```

Warning: Must be implemented in style "PvW0".

to_dual_translation_left()

Return the projection of `self` to the dual translation lattice after factorizing it to the left using the style "PvW0".

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).PvW0().simple_reflection(0).to_dual_translat
Lambda[1] + Lambda[3]
```

Warning: Must be implemented in style "PvW0".

to_dual_translation_right()

Return the projection of `self` to the dual translation lattice after factorizing it to the right using the style "W0Pv".

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).PW0().simple_reflection(0).to_dual_translati
-Lambda[1] - Lambda[3]
```

Warning: Must be implemented in style "W0Pv".

to_fundamental_group()

Return the image of `self` under the homomorphism to the fundamental group.

EXAMPLES:

```
sage: PW0 = ExtendedAffineWeylGroup(['A',3,1]).PW0()
sage: b = PW0.realization_of().lattice_basis()
sage: [(x, PW0.from_translation(x).to_fundamental_group()) for x in b]
[(Lambdacheck[1], pi[1]), (Lambdacheck[2], pi[2]), (Lambdacheck[3], pi[3])]
```

Warning: Must be implemented in style "WF".

to_translation_left()

Return the projection of `self` to the translation lattice after factorizing it to the left using the style "PW0".

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).PW0().simple_reflection(0).to_translation_le
Lambdacheck[1] + Lambdacheck[3]
```

Warning: Must be implemented in style "PW0".

to_translation_right()
Return the projection of `self` to the translation lattice after factorizing it to the right using the style "W0P".

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).PW0().simple_reflection(0).to_translation_ri
-Lambdacheck[1] - Lambdacheck[3]
```

Warning: Must be implemented in style "W0P".

class ExtendedAffineWeylGroup_Class.Realizations.**ParentMethods**

from_affine_weyl(w)
Return the image of w under the homomorphism from the affine Weyl group into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); PW0=E.PW0()
sage: W = E.affine_weyl()
sage: w = W.from_reduced_word([2,1,3,0])
sage: x = PW0.from_affine_weyl(w); x
t[Lambdacheck[1] - 2*Lambdacheck[2] + Lambdacheck[3]] * s3*s1
sage: FW = E.FW()
sage: y = FW.from_affine_weyl(w); y
S2*S3*S1*S0
sage: FW(x) == y
True
```

Warning: Must be implemented in style "WF" and "FW".

from_classical_weyl(w)
Return the image of w from the finite Weyl group into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); PW0=E.PW0()
sage: W0 = E.classical_weyl()
sage: w = W0.from_reduced_word([2,1,3])
sage: y = PW0.from_classical_weyl(w); y
s2*s3*s1
sage: y.parent() == PW0
True
sage: y.to_classical_weyl() == w
True
sage: W0P = E.W0P()
sage: z = W0P.from_classical_weyl(w); z
s2*s3*s1
sage: z.parent() == W0P
True
sage: W0P(y) == z
True
sage: FW = E.FW()
```

```
sage: x = FW.from_classical_weyl(w); x
S2*S3*S1
sage: x.parent() == FW
True
sage: FW(y) == x
True
sage: FW(z) == x
True
```

Warning: Must be implemented in style "PW0" and "W0P".

from_dual_classical_weyl(*w*)

Return the image of w from the finite Weyl group of dual form into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1]); PvW0 = E.PvW0()
sage: W0v = E.dual_classical_weyl()
sage: w = W0v.from_reduced_word([2,1,3])
sage: y = PvW0.from_dual_classical_weyl(w); y
s2*s3*s1
sage: y.parent() == PvW0
True
sage: y.to_dual_classical_weyl() == w
True
sage: x = E.FW().from_dual_classical_weyl(w); x
S2*S3*S1
sage: PvW0(x) == y
True
```

Warning: Must be implemented in style "PvW0" and "W0Pv".

from_dual_translation(*la*)

Return the image of `la` under the homomorphism of the dual version of the translation lattice into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1]); PvW0 = E.PvW0()
sage: bv = E.dual_lattice_basis(); bv
Finite family {1: Lambda[1], 2: Lambda[2]}
sage: x = PvW0.from_dual_translation(2*bv[1]-bv[2]); x
t[2*Lambda[1] - Lambda[2]]
sage: FW = E.FW()
sage: y = FW.from_dual_translation(2*bv[1]-bv[2]); y
S0*S2*S0*S1
sage: FW(x) == y
True
```

from_fundamental(*x*)

Return the image of x under the homomorphism from the fundamental group into `self`.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',3,1])
sage: PW0=E.PW0()
sage: F = E.fundamental_group()
sage: Is = F.special_nodes()
sage: [(i, PW0.from_fundamental(F(i))) for i in Is]
[(0, 1), (1, t[Lambdacheck[1]] * s1*s2*s3), (2, t[Lambdacheck[2]] * s2*s3*s1*s2), (3,
```

```
sage: [(i, E.W0P().from_fundamental((F(i)))) for i in Is]
[(0, 1), (1, s1*s2*s3 * t[-Lambdacheck[3]]), (2, s2*s3*s1*s2 * t[-Lambdacheck[2]]), (
sage: [(i, E.WF().from_fundamental(F(i))) for i in Is]
[(0, 1), (1, pi[1]), (2, pi[2]), (3, pi[3])]
```

Warning: This method must be implemented by the "WF" and "FW" realizations.

from_reduced_word(*word*)

Converts an affine or finite reduced word into a group element.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).PW0().from_reduced_word([1,0,1,2])
t[-Lambdacheck[1] + 2*Lambdacheck[2]]
```

from_translation(*la*)

Return the element of translation by `la` in `self`.

INPUT:

- `self` – a realization of the extended affine Weyl group
- `la` – an element of the translation lattice

In the notation of the documentation for `ExtendedAffineWeylGroup()`, `la` must be an element of "P".

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1]); PW0=E.PW0()
sage: b = E.lattice_basis(); b
Finite family {1: Lambdacheck[1], 2: Lambdacheck[2]}
sage: x = PW0.from_translation(2*b[1]-b[2]); x
t[2*Lambdacheck[1] - Lambdacheck[2]]
sage: FW = E.FW()
sage: y = FW.from_translation(2*b[1]-b[2]); y
S0*S2*S0*S1
sage: FW(x) == y
True
```

Since the implementation as a semidirect product requires wrapping the lattice group to make it multiplicative, we cannot declare that this map is a morphism for sage `Groups()`.

Warning: This method must be implemented by the "PW0" and "W0P" realizations.

simple_reflection(*i*)

Return the i-th simple reflection in `self`.

INPUT:

- `self` – a realization of the extended affine Weyl group
- i – An affine Dynkin node

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).PW0().simple_reflection(0)
t[Lambdacheck[1] + Lambdacheck[3]] * s1*s2*s3*s2*s1
sage: ExtendedAffineWeylGroup(['C',2,1]).WF().simple_reflection(0)
S0
sage: ExtendedAffineWeylGroup(['D',3,2]).PvW0().simple_reflection(1)
s1
```

simple_reflections()

Return a family from the set of affine Dynkin nodes to the simple reflections in the realization of the extended affine Weyl group.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',3,1]).W0P().simple_reflections()
Finite family {0: s1*s2*s3*s2*s1 * t[-Lambdacheck[1] - Lambdacheck[3]], 1: s1, 2: s2,
sage: ExtendedAffineWeylGroup(['A',3,1]).WF().simple_reflections()
Finite family {0: S0, 1: S1, 2: S2, 3: S3}
sage: ExtendedAffineWeylGroup(['A',3,1], print_tuple=True).FW().simple_reflections()
Finite family {0: (pi[0], S0), 1: (pi[0], S1), 2: (pi[0], S2), 3: (pi[0], S3)}
sage: ExtendedAffineWeylGroup(['A',3,1],fundamental="f",print_tuple=True).FW().simple
Finite family {0: (f[0], S0), 1: (f[0], S1), 2: (f[0], S2), 3: (f[0], S3)}
sage: ExtendedAffineWeylGroup(['A',3,1]).PvW0().simple_reflections()
Finite family {0: t[Lambda[1] + Lambda[3]] * s1*s2*s3*s2*s1, 1: s1, 2: s2, 3: s3}
```

ExtendedAffineWeylGroup_Class.Realizations.**super_categories**()

EXAMPLES:

```
sage: R = ExtendedAffineWeylGroup(['A',2,1]).Realizations(); R
Category of realizations of Extended affine Weyl group of type ['A', 2, 1]
sage: R.super_categories()
[Category of associative inverse realizations of unital magmas]
```

ExtendedAffineWeylGroup_Class.**W0P**()

Realizes `self` in "W0P"-style.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).W0P()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Weyl Group
```

ExtendedAffineWeylGroup_Class.**W0Pv**()

Realizes `self` in "W0Pv"-style.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).W0Pv()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Weyl Group
```

ExtendedAffineWeylGroup_Class.**WF**()

Realizes `self` in "WF"-style.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).WF()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Weyl Group
```

ExtendedAffineWeylGroup_Class.**WF_to_PW0_func**(*x*)

Coercion from style "WF" to "PW0".

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(["A", 2, 1])
sage: x = E.WF().an_element(); x
S0*S1*S2 * pi[2]
sage: E.WF_to_PW0_func(x)
t[Lambdacheck[1] + 2*Lambdacheck[2]] * s1*s2*s1
```

Warning: Since this is used to define some coercion maps it cannot itself use coercion.

ExtendedAffineWeylGroup_Class.**a_realization**()

Return the default realization of an extended affine Weyl group.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).a_realization()
Extended affine Weyl group of type ['A', 2, 1] realized by Semidirect product of Multiplicat
```

ExtendedAffineWeylGroup_Class.**affine_weyl**()

Return the affine Weyl group of `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).affine_weyl()
Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root lattice)
sage: ExtendedAffineWeylGroup(['A',5,2]).affine_weyl()
Weyl Group of type ['B', 3, 1]^* (as a matrix group acting on the root lattice)
sage: ExtendedAffineWeylGroup(['A',4,2]).affine_weyl()
Weyl Group of type ['BC', 2, 2] (as a matrix group acting on the root lattice)
sage: ExtendedAffineWeylGroup(CartanType(['A',4,2]).dual()).affine_weyl()
Weyl Group of type ['BC', 2, 2]^* (as a matrix group acting on the root lattice)
```

ExtendedAffineWeylGroup_Class.**cartan_type**()

The Cartan type of `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(["D",3,2]).cartan_type()
['C', 2, 1]^*
```

ExtendedAffineWeylGroup_Class.**classical_weyl**()

Return the classical Weyl group of `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).classical_weyl()
Weyl Group of type ['A', 2] (as a matrix group acting on the coweight lattice)
sage: ExtendedAffineWeylGroup(['A',5,2]).classical_weyl()
Weyl Group of type ['C', 3] (as a matrix group acting on the weight lattice)
sage: ExtendedAffineWeylGroup(['A',4,2]).classical_weyl()
Weyl Group of type ['C', 2] (as a matrix group acting on the weight lattice)
sage: ExtendedAffineWeylGroup(CartanType(['A',4,2]).dual()).classical_weyl()
Weyl Group of type ['C', 2] (as a matrix group acting on the coweight lattice)
```

ExtendedAffineWeylGroup_Class.**classical_weyl_to_affine**(w)

The image of w under the homomorphism from the classical Weyl group into the affine Weyl group.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: W0 = E.classical_weyl()
sage: w = W0.from_reduced_word([1,2]); w
s1*s2
sage: v = E.classical_weyl_to_affine(w); v
S1*S2
```

ExtendedAffineWeylGroup_Class.**dual_classical_weyl**()

Return the dual version of the classical Weyl group of `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).dual_classical_weyl()
Weyl Group of type ['A', 2] (as a matrix group acting on the weight lattice)
sage: ExtendedAffineWeylGroup(['A',5,2]).dual_classical_weyl()
Weyl Group of type ['C', 3] (as a matrix group acting on the weight lattice)
```

ExtendedAffineWeylGroup_Class.**dual_classical_weyl_to_affine**(*w*)

The image of w under the homomorphism from the dual version of the classical Weyl group into the affine Weyl group.

EXAMPLES:

```
sage: E = ExtendedAffineWeylGroup(['A',2,1])
sage: W0v = E.dual_classical_weyl()
sage: w = W0v.from_reduced_word([1,2]); w
s1*s2
sage: v = E.dual_classical_weyl_to_affine(w); v
S1*S2
```

ExtendedAffineWeylGroup_Class.**dual_lattice**()

Return the dual version of the translation lattice for `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).dual_lattice()
Weight lattice of the Root system of type ['A', 2]
sage: ExtendedAffineWeylGroup(['A',5,2]).dual_lattice()
Weight lattice of the Root system of type ['C', 3]
```

ExtendedAffineWeylGroup_Class.**dual_lattice_basis**()

Return the distinguished basis of the dual version of the translation lattice for `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).dual_lattice_basis()
Finite family {1: Lambda[1], 2: Lambda[2]}
sage: ExtendedAffineWeylGroup(['A',5,2]).dual_lattice_basis()
Finite family {1: Lambda[1], 2: Lambda[2], 3: Lambda[3]}
```

ExtendedAffineWeylGroup_Class.**exp_dual_lattice**()

Return the multiplicative version of the dual version of the translation lattice for `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).exp_dual_lattice()
Multiplicative form of Weight lattice of the Root system of type ['A', 2]
```

ExtendedAffineWeylGroup_Class.**exp_lattice**()

Return the multiplicative version of the translation lattice for `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).exp_lattice()
Multiplicative form of Coweight lattice of the Root system of type ['A', 2]
```

ExtendedAffineWeylGroup_Class.**fundamental_group**()

Return the abstract fundamental group.

EXAMPLES:

```
sage: F = ExtendedAffineWeylGroup(['D',5,1]).fundamental_group(); F
Fundamental group of type ['D', 5, 1]
sage: [a for a in F]
[pi[0], pi[1], pi[4], pi[5]]
```

ExtendedAffineWeylGroup_Class.**group_generators**()

Return a set of generators for the default realization of `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).group_generators()
(t[Lambdacheck[1]], t[Lambdacheck[2]], s1, s2)
```

ExtendedAffineWeylGroup_Class.**lattice**()

Return the translation lattice for `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).lattice()
Coweight lattice of the Root system of type ['A', 2]
sage: ExtendedAffineWeylGroup(['A',5,2]).lattice()
Weight lattice of the Root system of type ['C', 3]
sage: ExtendedAffineWeylGroup(['A',4,2]).lattice()
Weight lattice of the Root system of type ['C', 2]
sage: ExtendedAffineWeylGroup(CartanType(['A',4,2]).dual()).lattice()
Coweight lattice of the Root system of type ['B', 2]
sage: ExtendedAffineWeylGroup(CartanType(['A',2,1]), general_linear=True).lattice()
Ambient space of the Root system of type ['A', 2]
```

ExtendedAffineWeylGroup_Class.**lattice_basis**()

Return the distinguished basis of the translation lattice for `self`.

EXAMPLES:

```
sage: ExtendedAffineWeylGroup(['A',2,1]).lattice_basis()
Finite family {1: Lambdacheck[1], 2: Lambdacheck[2]}
sage: ExtendedAffineWeylGroup(['A',5,2]).lattice_basis()
Finite family {1: Lambda[1], 2: Lambda[2], 3: Lambda[3]}
sage: ExtendedAffineWeylGroup(['A',4,2]).lattice_basis()
Finite family {1: Lambda[1], 2: Lambda[2]}
sage: ExtendedAffineWeylGroup(CartanType(['A',4,2]).dual()).lattice_basis()
Finite family {1: Lambdacheck[1], 2: Lambdacheck[2]}
```

5.1.231 Fundamental Group of an Extended Affine Weyl Group

AUTHORS:

- Mark Shimozono (2013) initial version

class sage.combinat.root_system.fundamental_group.**FundamentalGroupElement**(*parent*, *x*)

Bases: `sage.structure.element.MultiplicativeGroupElement`

This should not be called directly

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffineWe
sage: x = FundamentalGroupOfExtendedAffineWeylGroup(['A',4,1], prefix="f").an_element()
sage: TestSuite(x).run()
```

act_on_affine_lattice(*wt*)

Act by `self` on the element `wt` of an affine root/weight lattice realization.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',3,1])
sage: wt = RootSystem(F.cartan_type()).weight_lattice().an_element(); wt
2*Lambda[0] + 2*Lambda[1] + 3*Lambda[2]
```

```
sage: F(3).act_on_affine_lattice(wt)
2*Lambda[0] + 3*Lambda[1] + 2*Lambda[3]
```

Warning: Doesn't work on ambient spaces.

act_on_affine_weyl(*w*)

Act by `self` on the element w of the affine Weyl group.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',3,1])
sage: W = WeylGroup(F.cartan_type(),prefix="s")
sage: w = W.from_reduced_word([2,3,0])
sage: F(1).act_on_affine_weyl(w).reduced_word()
[3, 0, 1]
```

inverse()

Return the inverse element of `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',3,1])
sage: F(1).inverse()
pi[3]
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['E',6,1], prefix="f")
sage: F(1).inverse()
f[6]
```

value()

Return the special node which indexes the special automorphism `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',4,1], prefix="f")
sage: F.special_nodes()
(0, 1, 2, 3, 4)
sage: x = F(4); x
f[4]
sage: x.value()
4
```

class sage.combinat.root_system.fundamental_group.**FundamentalGroupGL**(*cartan_type*, *prefix='pi'*)

Bases: sage.combinat.root_system.fundamental_group.FundamentalGroupOfExtendedAffineWeylGr

Fundamental group of GL_n. It is just the integers with extra privileges.

Element

alias of FundamentalGroupGLElement

action(*i*)

The action of the i-th automorphism on the affine Dynkin node set.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_linear=True)
sage: F.action(4)(2)
0
sage: F.action(-4)(2)
1
```

an_element()
An element of `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_linear=True).an_element()
pi[5]
```

dual_node(i)
The node whose special automorphism is inverse to that of i.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_linear=True)
sage: F.dual_node(2)
-2
```

family()
The family associated with the set of special nodes.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: fam = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_linear=True).family
sage: fam
Lazy family (<lambda>(i))_{i in Integer Ring}
sage: fam[-3]
-3
```

group_generators()
Return group generators for `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_linear=True).group_genera
(pi[1],)
```

one()
Return the identity element of the fundamental group.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_linear=True).one()
pi[0]
```

product(x, y)
Return the product of x and y.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_linear=True)
sage: F.special_nodes()
Integer Ring
sage: F(2)*F(3)
pi[5]
sage: F(1)*F(3)^(-1)
pi[-2]
```

reduced_word(*i*)

A reduced word for the finite permutation part of the special automorphism indexed by i.

More precisely, return a reduced word for the finite Weyl group element u where i-th automorphism (expressed in the extended affine Weyl group) has the form tu where t is a translation element.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_linear=True)
sage: F.reduced_word(10)
(1, 2)
```

some_elements()

Return some elements of `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_linear=True).some_element
[pi[-2], pi[2], pi[5]]
```

class sage.combinat.root_system.fundamental_group.**FundamentalGroupGLElement**(*parent*, *x*)

Bases: sage.combinat.root_system.fundamental_group.FundamentalGroupElement

This should not be called directly

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffineWe
sage: x = FundamentalGroupOfExtendedAffineWeylGroup(['A',4,1], prefix="f").an_element()
sage: TestSuite(x).run()
```

act_on_classical_ambient(*wt*)

Act by `self` on the classical ambient weight vector `wt`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_linear=True)
sage: f = F.an_element(); f
pi[5]
sage: la = F.cartan_type().classical().root_system().ambient_space().an_element(); la
(2, 2, 3)
sage: f.act_on_classical_ambient(la)
(2, 3, 2)
```

sage.combinat.root_system.fundamental_group.**FundamentalGroupOfExtendedAffineWeylGroup**(*cartan pre- fix='pi gen- eral_li*

Factory for the fundamental group of an extended affine Weyl group.

INPUT:

- `cartan_type` – a Cartan type that is either affine or finite, with the latter being a shorthand for the untwisted affinization
- `prefix` (default: 'pi') – string that labels the elements of the group
- `general_linear` – (default: None, meaning False) In untwisted type A, if True, use the universal central extension

Fundamental group

Associated to each affine Cartan type $\tilde{X}$ is an extended affine Weyl group E. Its subgroup of length-zero elements is called the fundamental group F. The group F can be identified with a subgroup of the group of automorphisms of the affine Dynkin diagram. As such, every element of F can be viewed as a permutation of the set I of affine Dynkin nodes.

Let $0 \in I$ be the distinguished affine node; it is the one whose removal produces the associated finite Cartan type (call it X). A node $i \in I$ is called *special* if some automorphism of the affine Dynkin diagram, sends 0 to i. The node 0 is always special due to the identity automorphism. There is a bijection of the set of special nodes with the fundamental group. We denote the image of i by π_i. The structure of F is determined as follows.

- $\tilde{X}$ is untwisted – F is isomorphic to $P^\vee/Q^\vee$ where $P^\vee$ and $Q^\vee$ are the coweight and coroot lattices of type X. The group $P^\vee/Q^\vee$ consists of the cosets $\omega_i^\vee + Q^\vee$ for special nodes i, where $\omega_0^\vee = 0$ by convention. In this case the special nodes i are the *cominuscule* nodes, the ones such that $\omega_i^\vee(\alpha_j)$ is 0 or 1 for all $j \in I_0 = I \setminus \{0\}$. For i special, addition by $\omega_i^\vee + Q^\vee$ permutes $P^\vee/Q^\vee$ and therefore permutes the set of special nodes. This permutation extends uniquely to an automorphism of the affine Dynkin diagram.
- $\tilde{X}$ is dual untwisted – (that is, the dual of $\tilde{X}$ is untwisted) F is isomorphic to P/Q where P and Q are the weight and root lattices of type X. The group P/Q consists of the cosets $\omega_i + Q$ for special nodes i, where $\omega_0 = 0$ by convention. In this case the special nodes i are the *minuscule* nodes, the ones such that $\alpha_j^\vee(\omega_i)$ is 0 or 1 for all $j \in I_0$. For i special, addition by $\omega_i + Q$ permutes P/Q and therefore permutes the set of special nodes. This permutation extends uniquely to an automorphism of the affine Dynkin diagram.
- $\tilde{X}$ is mixed – (that is, not of the above two types) F is the trivial group.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffineWe
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',3,1]); F
Fundamental group of type ['A', 3, 1]
sage: F.cartan_type().dynkin_diagram()
0
O-------+
|       |
|       |
O---O---O
1   2   3
A3~
sage: F.special_nodes()
(0, 1, 2, 3)
sage: F(1)^2
```

```
pi[2]
sage: F(1)*F(2)
pi[3]
sage: F(3)^(-1)
pi[1]

sage: F = FundamentalGroupOfExtendedAffineWeylGroup("B3"); F
Fundamental group of type ['B', 3, 1]
sage: F.cartan_type().dynkin_diagram()
    O 0
    |
    |
O---O=>=O
1   2   3
B3~
sage: F.special_nodes()
(0, 1)

sage: F = FundamentalGroupOfExtendedAffineWeylGroup("C2"); F
Fundamental group of type ['C', 2, 1]
sage: F.cartan_type().dynkin_diagram()
O=>=O=<=O
0   1   2
C2~
sage: F.special_nodes()
(0, 2)

sage: F = FundamentalGroupOfExtendedAffineWeylGroup("D4"); F
Fundamental group of type ['D', 4, 1]
sage: F.cartan_type().dynkin_diagram()
    O 4
    |
    |
O---O---O
1   |2  3
    |
    O 0
D4~
sage: F.special_nodes()
(0, 1, 3, 4)
sage: (F(4), F(4)^2)
(pi[4], pi[0])

sage: F = FundamentalGroupOfExtendedAffineWeylGroup("D5"); F
Fundamental group of type ['D', 5, 1]
sage: F.cartan_type().dynkin_diagram()
  0 O   O 5
    |   |
    |   |
O---O---O---O
1   2   3   4
D5~
sage: F.special_nodes()
(0, 1, 4, 5)
sage: (F(5), F(5)^2, F(5)^3, F(5)^4)
(pi[5], pi[1], pi[4], pi[0])
sage: F = FundamentalGroupOfExtendedAffineWeylGroup("E6"); F
Fundamental group of type ['E', 6, 1]
```

```
sage: F.cartan_type().dynkin_diagram()
        O 0
        |
        |
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6~
sage: F.special_nodes()
(0, 1, 6)
sage: F(1)^2
pi[6]

sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['D',4,2]); F
Fundamental group of type ['C', 3, 1]^*
sage: F.cartan_type().dynkin_diagram()
O=<=O---O=>=O
0   1   2   3
C3~*
sage: F.special_nodes()
(0, 3)
```

We also implement a fundamental group for GL_n. It is defined to be the group of integers, which is the covering group of the fundamental group Z/nZ for affine SL_n:

```
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_linear=True); F
Fundamental group of GL(3)
sage: x = F.an_element(); x
pi[5]
sage: x*x
pi[10]
sage: x.inverse()
pi[-5]
sage: wt = F.cartan_type().classical().root_system().ambient_space().an_element(); wt
(2, 2, 3)
sage: x.act_on_classical_ambient(wt)
(2, 3, 2)
sage: w = WeylGroup(F.cartan_type(),prefix="s").an_element(); w
s0*s1*s2
sage: x.act_on_affine_weyl(w)
s2*s0*s1
```

class sage.combinat.root_system.fundamental_group.**FundamentalGroupOfExtendedAffineWeylGroup_C**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The group of length zero elements in the extended affine Weyl group.

Element
alias of FundamentalGroupElement

action(*i*)
Return a function which permutes the affine Dynkin node set by the i-th special automorphism.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1])
sage: [[(i, j, F.action(i)(j)) for j in F.index_set()] for i in F.special_nodes()]
[[(0, 0, 0), (0, 1, 1), (0, 2, 2)], [(1, 0, 1), (1, 1, 2), (1, 2, 0)], [(2, 0, 2), (2, 1, 0)
sage: G = FundamentalGroupOfExtendedAffineWeylGroup(['D',4,1])
sage: [[(i, j, G.action(i)(j)) for j in G.index_set()] for i in G.special_nodes()]
[[(0, 0, 0), (0, 1, 1), (0, 2, 2), (0, 3, 3), (0, 4, 4)], [(1, 0, 1), (1, 1, 0), (1, 2, 2),
```

an_element()

Return an element of `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',4,1],prefix="f").an_element()
f[4]
```

cartan_type()

The Cartan type of `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',3,1]).cartan_type()
['A', 3, 1]
```

dual_node(i)

Return the node that indexes the inverse of the i-th element.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',4,1])
sage: [(i, F.dual_node(i)) for i in F.special_nodes()]
[(0, 0), (1, 4), (2, 3), (3, 2), (4, 1)]
sage: G = FundamentalGroupOfExtendedAffineWeylGroup(['E',6,1])
sage: [(i, G.dual_node(i)) for i in G.special_nodes()]
[(0, 0), (1, 6), (6, 1)]
sage: H = FundamentalGroupOfExtendedAffineWeylGroup(['D',5,1])
sage: [(i, H.dual_node(i)) for i in H.special_nodes()]
[(0, 0), (1, 1), (4, 5), (5, 4)]
```

group_generators()

Return a tuple of generators of the fundamental group.

Warning: This returns the entire group, a necessary behavior because it is used in `__iter__()`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: FundamentalGroupOfExtendedAffineWeylGroup(['E',6,1],prefix="f").group_generators()
Finite family {0: f[0], 1: f[1], 6: f[6]}
```

index_set()

The node set of the affine Cartan type of `self`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1]).index_set()
(0, 1, 2)
```

one()

Return the identity element of the fundamental group.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',3,1])
sage: F.one()
pi[0]
```

product(x, y)

Return the product of x and y.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',3,1])
sage: F.special_nodes()
(0, 1, 2, 3)
sage: F(2)*F(3)
pi[1]
sage: F(1)*F(3)^(-1)
pi[2]
```

reduced_word(i)

Return a reduced word for the finite Weyl group element associated with the i-th special automorphism.

More precisely, for each special node i, `self.reduced_word(i)` is a reduced word for the element v in the finite Weyl group such that in the extended affine Weyl group, the i-th special automorphism is equal to tv where t is a translation element.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: F = FundamentalGroupOfExtendedAffineWeylGroup(['A',3,1])
sage: [(i, F.reduced_word(i)) for i in F.special_nodes()]
[(0, ()), (1, (1, 2, 3)), (2, (2, 1, 3, 2)), (3, (3, 2, 1))]
```

special_nodes()

Return the special nodes of `self`.

See `sage.combinat.root_system.cartan_type.special_nodes()`.

EXAMPLES:

```
sage: from sage.combinat.root_system.fundamental_group import FundamentalGroupOfExtendedAffi
sage: FundamentalGroupOfExtendedAffineWeylGroup(['D',4,1]).special_nodes()
(0, 1, 3, 4)
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1]).special_nodes()
(0, 1, 2)
sage: FundamentalGroupOfExtendedAffineWeylGroup(['C',3,1]).special_nodes()
(0, 3)
sage: FundamentalGroupOfExtendedAffineWeylGroup(['D',4,2]).special_nodes()
(0, 3)
sage: FundamentalGroupOfExtendedAffineWeylGroup(['A',2,1], general_linear=True).special_node
Integer Ring
```

5.1.232 Root system data for folded Cartan types

AUTHORS:

- Travis Scrimshaw (2013-01-12) - Initial version

class sage.combinat.root_system.type_folded.**CartanTypeFolded**(*cartan_type*, *folding_of*, *orbit*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.sage_object.SageObject

A Cartan type realized from a (Dynkin) diagram folding.

Given a Cartan type X, we say $\hat{X}$ is a folded Cartan type of X if there exists a diagram folding of the Dynkin diagram of $\hat{X}$ onto X.

A folding of a simply-laced Dynkin diagram D with index set I is an automorphism σ of D where all nodes any orbit of σ are not connected. The resulting Dynkin diagram $\hat{D}$ is induced by I/σ where we identify edges in $\hat{D}$ which are not incident and add a k-edge if we identify k incident edges and the arrow is pointing towards the indicent note. We denote the index set of $\hat{D}$ by $\hat{I}$, and by abuse of notation, we denote the folding by σ.

We also have scaling factors γ_i for $i \in \hat{I}$ and defined as the unique numbers such that the map $\Lambda_j \mapsto \gamma_j \sum_{i \in \sigma^{-1}(j)} \Lambda_i$ is the smallest proper embedding of the weight lattice of X to $\hat{X}$.

If the Cartan type is simply laced, the default folding is the one induced from the identity map on D.

If X is affine type, the default embeddings we consider here are:

$$\begin{aligned} C_n^{(1)}, A_{2n}^{(2)}, A_{2n}^{(2)\dagger}, D_{n+1}^{(2)} &\hookrightarrow A_{2n-1}^{(1)}, \\ A_{2n-1}^{(2)}, B_n^{(1)} &\hookrightarrow D_{n+1}^{(1)}, \\ E_6^{(2)}, F_4^{(1)} &\hookrightarrow E_6^{(1)}, \\ D_4^{(3)}, G_2^{(1)} &\hookrightarrow D_4^{(1)}, \end{aligned}$$

and were chosen based on virtual crystals. In particular, the diagram foldings extend to crystal morphisms and gives a realization of Kirillov-Reshetikhin crystals for non-simply-laced types as simply-laced types. See [OS-Shimo03] and [FOS09] for more details. Here we can compute $\gamma_i = \max(c)/c_i$ where $(c_i)_i$ are the translation factors of the root system. In a more type-dependent way, we can define γ_i as follows:

1. There exists a unique arrow (multiple bond) in X.
   (a) Suppose the arrow points towards 0. Then $\gamma_i = 1$ for all $i \in I$.
   (b) Otherwise γ_i is the order of σ for all i in the connected component of 0 after removing the arrow, else $\gamma_i = 1$.
2. There is not a unique arrow. Thus $\hat{X} = A_{2n-1}^{(1)}$ and $\gamma_i = 1$ for all $1 \leq i \leq n-1$. If $i \in \{0, n\}$, then $\gamma_i = 2$ if the arrow incident to i points away and is 1 otherwise.

We note that γ_i only depends upon X.

If the Cartan type is finite, then we consider the classical foldings/embeddings induced by the above affine foldings/embeddings:

$$\begin{aligned} C_n &\hookrightarrow A_{2n-1}, \\ B_n &\hookrightarrow D_{n+1}, \\ F_4 &\hookrightarrow E_6, \\ G_2 &\hookrightarrow D_4. \end{aligned}$$

For more information on Cartan types, see sage.combinat.root_system.cartan_type.

Other foldings may be constructed by passing in an optional folding_of second argument. See below.

INPUT:

- `cartan_type` – the Cartan type X to create the folded type
- `folding_of` – the Cartan type $\hat{X}$ which X is a folding of
- `orbit` – the orbit of the Dynkin diagram automorphism σ given as a list of lists where the a-th list corresponds to the a-th entry in I or a dictionary with keys in I and values as lists

Note: If X is an affine type, we assume the special node is fixed under σ.

EXAMPLES:

```
sage: fct = CartanType(['C',4,1]).as_folding(); fct
['C', 4, 1] as a folding of ['A', 7, 1]
sage: fct.scaling_factors()
Finite family {0: 2, 1: 1, 2: 1, 3: 1, 4: 2}
sage: fct.folding_orbit()
Finite family {0: (0,), 1: (1, 7), 2: (2, 6), 3: (3, 5), 4: (4,)}
```

A simply laced Cartan type can be considered as a virtual type of itself:

```
sage: fct = CartanType(['A',4,1]).as_folding(); fct
['A', 4, 1] as a folding of ['A', 4, 1]
sage: fct.scaling_factors()
Finite family {0: 1, 1: 1, 2: 1, 3: 1, 4: 1}
sage: fct.folding_orbit()
Finite family {0: (0,), 1: (1,), 2: (2,), 3: (3,), 4: (4,)}
```

Finite types:

```
sage: fct = CartanType(['C',4]).as_folding(); fct
['C', 4] as a folding of ['A', 7]
sage: fct.scaling_factors()
Finite family {1: 1, 2: 1, 3: 1, 4: 2}
sage: fct.folding_orbit()
Finite family {1: (1, 7), 2: (2, 6), 3: (3, 5), 4: (4,)}

sage: fct = CartanType(['F',4]).dual().as_folding(); fct
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1} as a folding of ['E', 6]
sage: fct.scaling_factors()
Finite family {1: 1, 2: 1, 3: 2, 4: 2}
sage: fct.folding_orbit()
Finite family {1: (1, 6), 2: (3, 5), 3: (4,), 4: (2,)}
```

REFERENCES:

- Wikipedia article Dynkin_diagram#Folding

cartan_type()

Return the Cartan type of `self`.

EXAMPLES:

```
sage: fct = CartanType(['C', 4, 1]).as_folding()
sage: fct.cartan_type()
['C', 4, 1]
```

folding_of()

Return the Cartan type of the virtual space.

EXAMPLES:

```
sage: fct = CartanType(['C', 4, 1]).as_folding()
sage: fct.folding_of()
['A', 7, 1]
```

folding_orbit()
Return the orbits under the automorphism σ as a dictionary (of tuples).

EXAMPLES:

```
sage: fct = CartanType(['C', 4, 1]).as_folding()
sage: fct.folding_orbit()
Finite family {0: (0,), 1: (1, 7), 2: (2, 6), 3: (3, 5), 4: (4,)}
```

scaling_factors()
Return the scaling factors of `self`.

EXAMPLES:

```
sage: fct = CartanType(['C', 4, 1]).as_folding()
sage: fct.scaling_factors()
Finite family {0: 2, 1: 1, 2: 1, 3: 1, 4: 2}
sage: fct = CartanType(['BC', 4, 2]).as_folding()
sage: fct.scaling_factors()
Finite family {0: 1, 1: 1, 2: 1, 3: 1, 4: 2}
sage: fct = CartanType(['BC', 4, 2]).dual().as_folding()
sage: fct.scaling_factors()
Finite family {0: 2, 1: 1, 2: 1, 3: 1, 4: 1}
sage: CartanType(['BC', 4, 2]).relabel({0:4, 1:3, 2:2, 3:1, 4:0}).as_folding().scaling_facto
Finite family {0: 2, 1: 1, 2: 1, 3: 1, 4: 1}
```

5.1.233 Root system data for Cartan types with marked nodes

class sage.combinat.root_system.type_marked.**AmbientSpace**(*root_system*, *base_ring*)
Bases: sage.combinat.root_system.ambient_space.AmbientSpace

Ambient space for a marked finite Cartan type.

It is constructed in the canonical way from the ambient space of the original Cartan type.

EXAMPLES:

```
sage: L = CartanType(["F",4]).marked_nodes([1,3]).root_system().ambient_space(); L
Ambient space of the Root system of type ['F', 4] with nodes (1, 3) marked
sage: TestSuite(L).run()
```

dimension()
Return the dimension of this ambient space.

See also:

sage.combinat.root_system.ambient_space.AmbientSpace.dimension()

EXAMPLES:

```
sage: L = CartanType(["F",4]).marked_nodes([1,3]).root_system().ambient_space()
sage: L.dimension()
4
```

fundamental_weight(*i*)
Return the `i`-th fundamental weight.

It is constructed by looking up the corresponding simple coroot in the ambient space for the original Cartan type.

EXAMPLES:

```
sage: L = CartanType(["F",4]).marked_nodes([1,3]).root_system().ambient_space()
sage: L.fundamental_weight(1)
(1, 1, 0, 0)
sage: L.fundamental_weights()
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0),
               3: (3/2, 1/2, 1/2, 1/2), 4: (1, 0, 0, 0)}
```

simple_root(*i*)

Return the `i`-th simple root.

It is constructed by looking up the corresponding simple coroot in the ambient space for the original Cartan type.

EXAMPLES:

```
sage: L = CartanType(["F",4]).marked_nodes([1,3]).root_system().ambient_space()
sage: L.simple_root(1)
(0, 1, -1, 0)
sage: L.simple_roots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1),
               3: (0, 0, 0, 1), 4: (1/2, -1/2, -1/2, -1/2)}
sage: L.simple_coroots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1),
               3: (0, 0, 0, 2), 4: (1, -1, -1, -1)}
```

class sage.combinat.root_system.type_marked.**CartanType**(*ct*, *marked_nodes*)

Bases: sage.combinat.root_system.cartan_type.CartanType_decorator

A class for Cartan types with marked nodes.

INPUT:

- `ct` – a Cartan type
- `marked_nodes` – a list of marked nodes

EXAMPLES:

We take the Cartan type B_4:

```
sage: T = CartanType(['B',4])
sage: T.dynkin_diagram()
O---O---O=>=O
1   2   3   4
B4
```

And mark some of its nodes:

```
sage: T = T.marked_nodes([2,3])
sage: T.dynkin_diagram()
O---X---X=>=O
1   2   3   4
B4 with nodes (2, 3) marked
```

Markings are not additive:

```
sage: T.marked_nodes([1,4]).dynkin_diagram()
X---O---O=>=X
```

```
1   2   3   4
B4 with nodes (1, 4) marked
```

And trivial relabelling are honoured nicely:

```
sage: T = T.marked_nodes([])
sage: T.dynkin_diagram()
O---O---O=>=O
1   2   3   4
B4
```

ascii_art(*label=<function <lambda>>*, *node=None*)

Return an ascii art representation of this Cartan type.

EXAMPLES:

```
sage: print CartanType(["G", 2]).marked_nodes([2]).ascii_art()
  3
O=<=X
1   2
sage: print CartanType(["B", 3, 1]).marked_nodes([0, 3]).ascii_art()
    X 0
    |
    |
O---O=>=X
1   2   3
sage: print CartanType(["F", 4, 1]).marked_nodes([0, 2]).ascii_art()
X---O---X=>=O---O
0   1   2   3   4
```

dual()

Implements sage.combinat.root_system.cartan_type.CartanType_abstract.dual(), using that taking the dual and marking nodes are commuting operations.

EXAMPLES:

```
sage: T = CartanType(["BC",3, 2])
sage: T.marked_nodes([1,3]).dual().dynkin_diagram()
O=>=X---O=>=X
0   1   2   3
BC3~* with nodes (1, 3) marked
sage: T.dual().marked_nodes([1,3]).dynkin_diagram()
O=>=X---O=>=X
0   1   2   3
BC3~* with nodes (1, 3) marked
```

dynkin_diagram()

Returns the Dynkin diagram for this Cartan type.

EXAMPLES:

```
sage: CartanType(["G", 2]).marked_nodes([2]).dynkin_diagram()
  3
O=<=X
1   2
G2 with node 2 marked
```

TESTS:

To be compared with the examples in ascii_art():

```
sage: sorted(CartanType(["G", 2]).relabel({1:2,2:1}).dynkin_diagram().edges())
[(1, 2, 3), (2, 1, 1)]
sage: sorted(CartanType(["B", 3, 1]).relabel([1,3,2,0]).dynkin_diagram().edges())
[(0, 2, 1), (1, 2, 1), (2, 0, 2), (2, 1, 1), (2, 3, 1), (3, 2, 1)]
sage: sorted(CartanType(["F", 4, 1]).relabel(lambda n: 4-n).dynkin_diagram().edges())
[(0, 1, 1), (1, 0, 1), (1, 2, 1), (2, 1, 2), (2, 3, 1), (3, 2, 1), (3, 4, 1), (4, 3, 1)]
```

marked_nodes (*marked_nodes*)

Return `self` with nodes `marked_nodes` marked.

EXAMPLES:

```
sage: ct = CartanType(['A',12])
sage: m = ct.marked_nodes([1,4,6,7,8,12]); m
['A', 12] with nodes (1, 4, 6, 7, 8, 12) marked
sage: m.marked_nodes([2])
['A', 12] with node 2 marked
sage: m.marked_nodes([]) is ct
True
```

relabel (*relabelling*)

Return the relabelling of `self`.

EXAMPLES:

```
sage: T = CartanType(["BC",3, 2])
sage: T.marked_nodes([1,3]).relabel(lambda x: x+2).dynkin_diagram()
O=<=X---O=<=X
2   3   4   5
BC3~ relabelled by {0: 2, 1: 3, 2: 4, 3: 5} with nodes (3, 5) marked
sage: T.relabel(lambda x: x+2).marked_nodes([3,5]).dynkin_diagram()
O=<=X---O=<=X
2   3   4   5
BC3~ relabelled by {0: 2, 1: 3, 2: 4, 3: 5} with nodes (3, 5) marked
```

type()

Return the type of `self` or `None` if unknown.

EXAMPLES:

```
sage: ct = CartanType(['F', 4]).marked_nodes([1,3])
sage: ct.type()
'F'
```

class sage.combinat.root_system.type_marked.**CartanType_affine**(*ct*, *marked_nodes*)

Bases: sage.combinat.root_system.type_marked.CartanType, sage.combinat.root_system.cartan_type.CartanType_affine

TESTS:

```
sage: ct = CartanType(['B',3,1]).marked_nodes([1,3])
sage: ct
['B', 3, 1] with nodes (1, 3) marked

sage: L = ct.root_system().ambient_space(); L
Ambient space of the Root system of type ['B', 3, 1] with nodes (1, 3) marked
sage: L.classical()
Ambient space of the Root system of type ['B', 3] with nodes (1, 3) marked
sage: TestSuite(L).run()
```

basic_untwisted()
Return the basic untwisted Cartan type associated with this affine Cartan type.

Given an affine type $X_n^{(r)}$, the basic untwisted type is X_n. In other words, it is the classical Cartan type that is twisted to obtain `self`.

EXAMPLES:

```
sage: CartanType(['A', 7, 2]).marked_nodes([1,3]).basic_untwisted()
['A', 7] with nodes (1, 3) marked
sage: CartanType(['D', 4, 3]).marked_nodes([0,2]).basic_untwisted()
['D', 4] with node 2 marked
```

classical()
Return the classical Cartan type associated with `self`.

EXAMPLES:

```
sage: T = CartanType(['A',4,1]).marked_nodes([0,2,4])
sage: T.dynkin_diagram()
0
X-----------+
|           |
|           |
O---X---O---X
1   2   3   4
A4~ with nodes (0, 2, 4) marked

sage: T0 = T.classical()
sage: T0
['A', 4] with nodes (2, 4) marked
sage: T0.dynkin_diagram()
O---X---O---X
1   2   3   4
A4 with nodes (2, 4) marked
```

is_untwisted_affine()
Implements :meth:'CartanType_affine.is_untwisted_affine'.

A marked Cartan type is untwisted affine if the original is.

EXAMPLES:

```
sage: CartanType(['B', 3, 1]).marked_nodes([1,3]).is_untwisted_affine()
True
```

special_node()
Return the special node of the Cartan type.

See also:

`special_node()`

It is the special node of the non-marked Cartan type..

EXAMPLES:

```
sage: CartanType(['B', 3, 1]).marked_nodes([1,3]).special_node()
0
```

class sage.combinat.root_system.type_marked.**CartanType_finite**(*ct*, *marked_nodes*)
Bases: sage.combinat.root_system.type_marked.CartanType, sage.combinat.root_system.cartan_type.CartanType_finite

Return an isomorphic Cartan type obtained by marking the nodes of the Dynkin diagram.

TESTS:

Test that the produced Cartan type is in the appropriate abstract classes:

```
sage: ct = CartanType(['B',4]).marked_nodes([1,2])
sage: TestSuite(ct).run()
sage: from sage.combinat.root_system import cartan_type
sage: isinstance(ct, cartan_type.CartanType_finite)
True
sage: isinstance(ct, cartan_type.CartanType_simple)
True
sage: isinstance(ct, cartan_type.CartanType_affine)
False
sage: isinstance(ct, cartan_type.CartanType_crystallographic)
True
sage: isinstance(ct, cartan_type.CartanType_simply_laced)
False

sage: ct = CartanType(['A',3,1]).marked_nodes([1,2])
sage: TestSuite(ct).run()
sage: isinstance(ct, cartan_type.CartanType_simple)
True
sage: isinstance(ct, cartan_type.CartanType_finite)
False
sage: isinstance(ct, cartan_type.CartanType_affine)
True
sage: isinstance(ct, cartan_type.CartanType_crystallographic)
True
sage: isinstance(ct, cartan_type.CartanType_simply_laced)
True
```

AmbientSpace
alias of AmbientSpace

affine()
Return the affine Cartan type associated with self.

EXAMPLES:

```
sage: B4 = CartanType(['B',4]).marked_nodes([1,3])
sage: B4.dynkin_diagram()
X---O---X=>=O
1   2   3   4
B4 with nodes (1, 3) marked
sage: B4.affine().dynkin_diagram()
    O 0
    |
    |
X---O---X=>=O
1   2   3   4
B4~ with nodes (1, 3) marked
```

TESTS:

Check that we don't inadvertently change the internal marking of ct:

```
sage: ct = CartanType(['F', 4]).marked_nodes([1,3])
sage: ct.affine()
['F', 4, 1] with nodes (1, 3) marked
sage: ct
```

```
['F', 4] with nodes (1, 3) marked
```

5.1.234 Root system data for reducible Cartan types

class sage.combinat.root_system.type_reducible.**AmbientSpace**(*root_system*, *base_ring*)

Bases: sage.combinat.root_system.ambient_space.AmbientSpace

EXAMPLES:

```
sage: RootSystem("A2xB2").ambient_space()
Ambient space of the Root system of type A2xB2
```

ambient_spaces()

Returns a list of the irreducible Cartan types of which the given reducible Cartan type is a product.

EXAMPLES:

```
sage: RootSystem("A2xB2").ambient_space().ambient_spaces()
[Ambient space of the Root system of type ['A', 2],
 Ambient space of the Root system of type ['B', 2]]
```

cartan_type()

EXAMPLES:

```
sage: RootSystem("A2xB2").ambient_space().cartan_type()
A2xB2
```

component_types()

EXAMPLES:

```
sage: RootSystem("A2xB2").ambient_space().component_types()
[['A', 2], ['B', 2]]
```

dimension()

EXAMPLES:

```
sage: RootSystem("A2xB2").ambient_space().dimension()
5
```

fundamental_weights()

EXAMPLES:

```
sage: RootSystem("A2xB2").ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0, 0), 2: (1, 1, 0, 0, 0), 3: (0, 0, 0, 1, 0), 4: (0, 0, 0, 1/2,
```

inject_weights(*i*, *v*)

Produces the corresponding element of the lattice.

INPUT:

- i - an integer in range(self.components)
- v - a vector in the i-th component weight lattice

EXAMPLES:

```
sage: V = RootSystem("A2xB2").ambient_space()
sage: [V.inject_weights(i,V.ambient_spaces()[i].fundamental_weights()[1]) for i in range(2)]
[(1, 0, 0, 0, 0), (0, 0, 0, 1, 0)]
```

```
sage: [V.inject_weights(i,V.ambient_spaces()[i].fundamental_weights()[2]) for i in range(2)]
[(1, 1, 0, 0, 0), (0, 0, 0, 1/2, 1/2)]
```

negative_roots()

EXAMPLES:

```
sage: RootSystem("A1xA2").ambient_space().negative_roots()
[(-1, 1, 0, 0, 0), (0, 0, -1, 1, 0), (0, 0, -1, 0, 1), (0, 0, 0, -1, 1)]
```

positive_roots()

EXAMPLES:

```
sage: RootSystem("A1xA2").ambient_space().positive_roots()
[(1, -1, 0, 0, 0), (0, 0, 1, -1, 0), (0, 0, 1, 0, -1), (0, 0, 0, 1, -1)]
```

simple_coroot(*i*)

EXAMPLES:

```
sage: A = RootSystem("A1xB2").ambient_space()
sage: A.simple_coroot(2)
(0, 0, 1, -1)
sage: A.simple_coroots()
Finite family {1: (1, -1, 0, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 2)}
```

simple_root(*i*)

EXAMPLES:

```
sage: A = RootSystem("A1xB2").ambient_space()
sage: A.simple_root(2)
(0, 0, 1, -1)
sage: A.simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 1)}
```

class sage.combinat.root_system.type_reducible.**CartanType**(*types*)

Bases: sage.structure.sage_object.SageObject, sage.combinat.root_system.cartan_type.Cartar

A class for reducible Cartan types.

Reducible root systems are ones that can be factored as direct products. Strictly speaking type D_2 (corresponding to orthogonal groups of degree 4) is reducible since it is isomorphic to $A_1 \times A_1$. However type D_2 is not built using this class for our purposes.

INPUT:

- types - a list of simple Cartan types

EXAMPLES:

```
sage: [t1,t2]=[CartanType(x) for x in ['A',1],['B',2]]
sage: CartanType([t1,t2])
A1xB2
sage: t = CartanType("A2xB2")
```

A reducible Cartan type is finite (resp. crystallographic, simply laced) if all its components are:

```
sage: t.is_finite()
True
sage: t.is_crystallographic()
True
sage: t.is_simply_laced()
False
```

This is implemented by inserting the appropriate abstract super classes (see `_add_abstract_superclass()`):

```
sage: t.__class__.mro()
[<class 'sage.combinat.root_system.type_reducible.CartanType_with_superclass'>, <class 'sage.com
```

The index set of the reducible Cartan type is obtained by relabelling successively the nodes of the Dynkin diagrams of the components by 1,2,...:

```
sage: t = CartanType(["A",4], ["BC",5,2], ["C",3])
sage: t.index_set()
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)

sage: t.dynkin_diagram()
O---O---O---O
1   2   3   4
O=<=O---O---O---O=<=O
5   6   7   8   9   10
O---O=<=O
11   12   13
A4xBC5~xC3
```

AmbientSpace
alias of `AmbientSpace`

ascii_art(*label=<function <lambda>>*, *node=None*)
Return an ascii art representation of this reducible Cartan type.

EXAMPLES:

```
sage: print CartanType("F4xA2").ascii_art(label = lambda x: x+2)
O---O=>=O---O
3   4   5   6
O---O
7   8

sage: print CartanType(["BC",5,2], ["A",4]).ascii_art()
O=<=O---O---O---O=<=O
1   2   3   4   5   6
O---O---O---O
7   8   9   10

sage: print CartanType(["A",4], ["BC",5,2], ["C",3]).ascii_art()
O---O---O---O
1   2   3   4
O=<=O---O---O---O=<=O
5   6   7   8   9   10
O---O=<=O
11   12   13
```

cartan_matrix(*subdivide=True*)
Return the Cartan matrix associated with `self`. By default the Cartan matrix is a subdivided block matrix showing the reducibility but the subdivision can be suppressed with the option `subdivide = False`.

Todo

Currently `subdivide` is currently ignored.

EXAMPLES:

```
sage: ct = CartanType("A2","B2")
sage: ct.cartan_matrix()
[ 2 -1| 0  0]
[-1  2| 0  0]
[-----+-----]
[ 0  0| 2 -1]
[ 0  0|-2  2]
sage: ct.cartan_matrix(subdivide=False)
[ 2 -1  0  0]
[-1  2  0  0]
[ 0  0  2 -1]
[ 0  0 -2  2]
```

component_types()

A list of Cartan types making up the reducible type.

EXAMPLES:

```
sage: CartanType(['A',2],['B',2]).component_types()
[['A', 2], ['B', 2]]
```

dual()

EXAMPLES:

```
sage: CartanType("A2xB2").dual()
A2xC2
```

dynkin_diagram()

Returns a Dynkin diagram for type reducible.

EXAMPLES:

```
sage: dd = CartanType("A2xB2xF4").dynkin_diagram()
sage: dd
O---O
1   2
O=>=O
3   4
O---O=>=O---O
5   6   7   8
A2xB2xF4
sage: dd.edges()
[(1, 2, 1), (2, 1, 1), (3, 4, 2), (4, 3, 1), (5, 6, 1), (6, 5, 1), (6, 7, 2), (7, 6, 1), (7,

sage: CartanType("F4xA2").dynkin_diagram()
O---O=>=O---O
1   2   3   4
O---O
5   6
F4xA2
```

index_set()

Implements `CartanType_abstract.index_set()`.

For the moment, the index set is always of the form $\{1, \ldots, n\}$.

EXAMPLES:

```
sage: CartanType("A2","A1").index_set()
(1, 2, 3)
```

is_affine()
Report that this reducible Cartan type is not affine

EXAMPLES:

```
sage: CartanType(['A',2],['B',2]).is_affine()
False
```

is_finite()
EXAMPLES:

```
sage: ct1 = CartanType(['A',2],['B',2])
sage: ct1.is_finite()
True
sage: ct2 = CartanType(['A',2],['B',2,1])
sage: ct2.is_finite()
False
```

TESTS:

```
sage: isinstance(ct1, sage.combinat.root_system.cartan_type.CartanType_finite)
True
sage: isinstance(ct2, sage.combinat.root_system.cartan_type.CartanType_finite)
False
```

is_irreducible()
Report that this Cartan type is not irreducible.

EXAMPLES:

```
sage: ct = CartanType(['A',2],['B',2])
sage: ct.is_irreducible()
False
```

rank()
Returns the rank of self.

EXAMPLES:

```
sage: CartanType("A2","A1").rank()
3
```

type()
Returns "reducible" since the type is reducible.

EXAMPLES:

```
sage: CartanType(['A',2],['B',2]).type()
'reducible'
```

5.1.235 Root system data for relabelled Cartan types

class sage.combinat.root_system.type_relabel.**AmbientSpace**(*root_system*, *base_ring*)
Bases: sage.combinat.root_system.ambient_space.AmbientSpace

Ambient space for a relabelled finite Cartan type.

It is constructed in the canonical way from the ambient space of the original Cartan type, by relabelling the simple roots, fundamental weights, etc.

EXAMPLES:

```
sage: cycle = {1:2, 2:3, 3:4, 4:1}
sage: L = CartanType(["F",4]).relabel(cycle).root_system().ambient_space(); L
Ambient space of the Root system of type ['F', 4] relabelled by {1: 2, 2: 3, 3: 4, 4: 1}
sage: TestSuite(L).run()
```

dimension()

Return the dimension of this ambient space.

See also:

sage.combinat.root_system.ambient_space.AmbientSpace.dimension()

EXAMPLES:

```
sage: cycle = {1:2, 2:3, 3:4, 4:1}
sage: L = CartanType(["F",4]).relabel(cycle).root_system().ambient_space()
sage: L.dimension()
4
```

fundamental_weight(*i*)

Return the `i`-th fundamental weight.

It is constructed by looking up the corresponding simple coroot in the ambient space for the original Cartan type.

EXAMPLES:

```
sage: cycle = {1:2, 2:3, 3:4, 4:1}
sage: L = CartanType(["F",4]).relabel(cycle).root_system().ambient_space()
sage: K = CartanType(["F",4]).root_system().ambient_space()
sage: K.fundamental_weights()
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0), 3: (3/2, 1/2, 1/2, 1/2), 4: (1, 0, 0, 0)}
sage: L.fundamental_weight(1)
(1, 0, 0, 0)
sage: L.fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (2, 1, 1, 0), 4: (3/2, 1/2, 1/2, 1/2)}
```

simple_root(*i*)

Return the `i`-th simple root.

It is constructed by looking up the corresponding simple coroot in the ambient space for the original Cartan type.

EXAMPLES:

```
sage: cycle = {1:2, 2:3, 3:4, 4:1}
sage: L = CartanType(["F",4]).relabel(cycle).root_system().ambient_space()
sage: K = CartanType(["F",4]).root_system().ambient_space()
sage: K.simple_roots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 1), 4: (1/2, -1/2, -1/2, -1/
sage: K.simple_coroots()
Finite family {1: (0, 1, -1, 0), 2: (0, 0, 1, -1), 3: (0, 0, 0, 2), 4: (1, -1, -1, -1)}
sage: L.simple_root(1)
(1/2, -1/2, -1/2, -1/2)

sage: L.simple_roots()
Finite family {1: (1/2, -1/2, -1/2, -1/2), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1), 4: (0, 0, 0,

sage: L.simple_coroots()
Finite family {1: (1, -1, -1, -1), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1), 4: (0, 0, 0, 2)}
```

class sage.combinat.root_system.type_relabel.**CartanType**(*type*, *relabelling*)
Bases: sage.combinat.root_system.cartan_type.CartanType_decorator

A class for relabelled Cartan types.

ascii_art(*label=<function <lambda>>*, *node=None*)
Return an ascii art representation of this Cartan type.

EXAMPLES:

```
sage: print CartanType(["G", 2]).relabel({1:2,2:1}).ascii_art()
  3
O=<=O
2   1
sage: print CartanType(["B", 3, 1]).relabel([1,3,2,0]).ascii_art()
    O 1
    |
    |
O---O=>=O
3   2   0
sage: print CartanType(["F", 4, 1]).relabel(lambda n: 4-n).ascii_art()
O---O---O=>=O---O
4   3   2   1   0
```

dual()
Implements sage.combinat.root_system.cartan_type.CartanType_abstract.dual(), using that taking the dual and relabelling are commuting operations.

EXAMPLES:

```
sage: T = CartanType(["BC",3, 2])
sage: cycle = {1:2, 2:3, 3:0, 0:1}
sage: T.relabel(cycle).dual().dynkin_diagram()
O=>=O---O=>=O
1   2   3   0
BC3~* relabelled by {0: 1, 1: 2, 2: 3, 3: 0}
sage: T.dual().relabel(cycle).dynkin_diagram()
O=>=O---O=>=O
1   2   3   0
BC3~* relabelled by {0: 1, 1: 2, 2: 3, 3: 0}
```

dynkin_diagram()
Returns the Dynkin diagram for this Cartan type.

EXAMPLES:

```
sage: CartanType(["G", 2]).relabel({1:2,2:1}).dynkin_diagram()
  3
O=<=O
2   1
G2 relabelled by {1: 2, 2: 1}
```

TESTS:

To be compared with the examples in ascii_art():

```
sage: sorted(CartanType(["G", 2]).relabel({1:2,2:1}).dynkin_diagram().edges())
[(1, 2, 3), (2, 1, 1)]
sage: sorted(CartanType(["B", 3, 1]).relabel([1,3,2,0]).dynkin_diagram().edges())
[(0, 2, 1), (1, 2, 1), (2, 0, 2), (2, 1, 1), (2, 3, 1), (3, 2, 1)]
sage: sorted(CartanType(["F", 4, 1]).relabel(lambda n: 4-n).dynkin_diagram().edges())
[(0, 1, 1), (1, 0, 1), (1, 2, 1), (2, 1, 2), (2, 3, 1), (3, 2, 1), (3, 4, 1), (4, 3, 1)]
```

index_set()

EXAMPLES:

```
sage: ct = CartanType(['G', 2]).relabel({1:2,2:1})
sage: ct.index_set()
(1, 2)
```

type()

Return the type of `self` or `None` if unknown.

EXAMPLES:

```
sage: ct = CartanType(['G', 2]).relabel({1:2,2:1})
sage: ct.type()
'G'
```

class sage.combinat.root_system.type_relabel.**CartanType_affine**(*type*, *relabelling*)

Bases: sage.combinat.root_system.type_relabel.CartanType, sage.combinat.root_system.cartan_type.CartanType_affine

TESTS:

```
sage: ct = CartanType(['D',4,3]); ct
['G', 2, 1]^* relabelled by {0: 0, 1: 2, 2: 1}

sage: L = ct.root_system().ambient_space(); L
Ambient space of the Root system of type ['G', 2, 1]^* relabelled by {0: 0, 1: 2, 2: 1}
sage: L.classical()
Ambient space of the Root system of type ['G', 2]
sage: TestSuite(L).run()
```

basic_untwisted()

Return the basic untwisted Cartan type associated with this affine Cartan type.

Given an affine type $X_n^{(r)}$, the basic untwisted type is X_n. In other words, it is the classical Cartan type that is twisted to obtain `self`.

EXAMPLES:

```
sage: ct = CartanType(['A', 5, 2]).relabel({0:1, 1:0, 2:2, 3:3})
sage: ct.basic_untwisted()
['A', 5]
```

classical()

Return the classical Cartan type associated with `self`.

EXAMPLES:

```
sage: A41 = CartanType(['A',4,1])
sage: A41.dynkin_diagram()
0
O-----------+
|           |
|           |
O---O---O---O
1   2   3   4
A4~

sage: T = A41.relabel({0:1, 1:2, 2:3, 3:4, 4:0})
sage: T
['A', 4, 1] relabelled by {0: 1, 1: 2, 2: 3, 3: 4, 4: 0}
sage: T.dynkin_diagram()
```

```
1
O-----------+
|           |
|           |
O---O---O---O
2   3   4   0
A4~ relabelled by {0: 1, 1: 2, 2: 3, 3: 4, 4: 0}

sage: T0 = T.classical()
sage: T0
['A', 4] relabelled by {1: 2, 2: 3, 3: 4, 4: 0}
sage: T0.dynkin_diagram()
O---O---O---O
2   3   4   0
A4 relabelled by {1: 2, 2: 3, 3: 4, 4: 0}
```

is_untwisted_affine()

Implements :meth:'CartanType_affine.is_untwisted_affine'

A relabelled Cartan type is untwisted affine if the original is.

EXAMPLES:

```
sage: CartanType(['B', 3, 1]).relabel({1:2, 2:3, 3:0, 0:1}).is_untwisted_affine()
True
```

special_node()

Returns a special node of the Dynkin diagram

See also:

`special_node()`

It is obtained by relabelling of the special node of the non relabelled Dynkin diagram.

EXAMPLES:

```
sage: CartanType(['B', 3, 1]).special_node()
0
sage: CartanType(['B', 3, 1]).relabel({1:2, 2:3, 3:0, 0:1}).special_node()
1
```

class sage.combinat.root_system.type_relabel.**CartanType_finite**(*type*, *relabelling*)

Bases: sage.combinat.root_system.type_relabel.CartanType, sage.combinat.root_system.cartan_type.CartanType_finite

INPUT:

- `type` – a Cartan type
- `relabelling` – a function (or a list, or a dictionary)

Returns an isomorphic Cartan type obtained by relabelling the nodes of the Dynkin diagram. Namely the node with label `i` is relabelled `f(i)` (or, by `f[i]` if `f` is a list or dictionary).

EXAMPLES:

We take the Cartan type B_4:

```
sage: T = CartanType(['B',4])
sage: T.dynkin_diagram()
O---O---O=>=O
```

```
1   2   3   4
B4
```

And relabel its nodes:

```
sage: cycle = {1:2, 2:3, 3:4, 4:1}

sage: T = T.relabel(cycle)
sage: T.dynkin_diagram()
O---O---O=>=O
2   3   4   1
B4 relabelled by {1: 2, 2: 3, 3: 4, 4: 1}
sage: sorted(T.dynkin_diagram().edges())
[(1, 4, 1), (2, 3, 1), (3, 2, 1), (3, 4, 1), (4, 1, 2), (4, 3, 1)]
```

Multiple relabelling are recomposed into a single one:

```
sage: T = T.relabel(cycle)
sage: T.dynkin_diagram()
O---O---O=>=O
3   4   1   2
B4 relabelled by {1: 3, 2: 4, 3: 1, 4: 2}

sage: T = T.relabel(cycle)
sage: T.dynkin_diagram()
O---O---O=>=O
4   1   2   3
B4 relabelled by {1: 4, 2: 1, 3: 2, 4: 3}
```

And trivial relabelling are honoured nicely:

```
sage: T = T.relabel(cycle)
sage: T.dynkin_diagram()
O---O---O=>=O
1   2   3   4
B4
```

Test that the produced Cartan type is in the appropriate abstract classes (see trac ticket #13724):

```
sage: ct = CartanType(['B',4]).relabel(cycle)
sage: TestSuite(ct).run()
sage: from sage.combinat.root_system import cartan_type
sage: isinstance(ct, cartan_type.CartanType_finite)
True
sage: isinstance(ct, cartan_type.CartanType_simple)
True
sage: isinstance(ct, cartan_type.CartanType_affine)
False
sage: isinstance(ct, cartan_type.CartanType_crystallographic)
True
sage: isinstance(ct, cartan_type.CartanType_simply_laced)
False

sage: ct = CartanType(['A',3,1]).relabel({0:3,1:2, 2:1,3:0})
sage: TestSuite(ct).run()
sage: isinstance(ct, cartan_type.CartanType_simple)
True
sage: isinstance(ct, cartan_type.CartanType_finite)
False
sage: isinstance(ct, cartan_type.CartanType_affine)
```

```
True
sage: isinstance(ct, cartan_type.CartanType_crystallographic)
True
sage: isinstance(ct, cartan_type.CartanType_simply_laced)
True
```

Check for the original issues of trac ticket #13724:

```
sage: A3 = CartanType("A3")
sage: A3.cartan_matrix()
[ 2 -1  0]
[-1  2 -1]
[ 0 -1  2]
sage: A3r = A3.relabel({1:2,2:3,3:1})
sage: A3r.cartan_matrix()
[ 2  0 -1]
[ 0  2 -1]
[-1 -1  2]

sage: ct = CartanType(["D",4,3]).classical(); ct
['G', 2]
sage: ct.symmetrizer()
Finite family {1: 1, 2: 3}
```

AmbientSpace
alias of AmbientSpace

affine()
Return the affine Cartan type associated with self.

EXAMPLES:

```
sage: B4 = CartanType(['B',4])
sage: B4.dynkin_diagram()
O---O---O=>=O
1   2   3   4
B4
sage: B4.affine().dynkin_diagram()
    O 0
    |
    |
O---O---O=>=O
1   2   3   4
B4~
```

If possible, this reuses the original label for the special node:

```
sage: T = B4.relabel({1:2, 2:3, 3:4, 4:1}); T.dynkin_diagram()
O---O---O=>=O
2   3   4   1
B4 relabelled by {1: 2, 2: 3, 3: 4, 4: 1}
sage: T.affine().dynkin_diagram()
    O 0
    |
    |
O---O---O=>=O
2   3   4   1
B4~ relabelled by {0: 0, 1: 2, 2: 3, 3: 4, 4: 1}
```

Otherwise, it chooses a label for the special_node in $0, 1, ...$:

```
sage: T = B4.relabel({1:0, 2:1, 3:2, 4:3}); T.dynkin_diagram()
O---O---O=>=O
0   1   2   3
B4 relabelled by {1: 0, 2: 1, 3: 2, 4: 3}
sage: T.affine().dynkin_diagram()
    O 4
    |
    |
O---O---O=>=O
0   1   2   3
B4~ relabelled by {0: 4, 1: 0, 2: 1, 3: 2, 4: 3}
```

This failed before trac ticket #13724:

```
sage: ct = CartanType(["G",2]).dual(); ct
['G', 2] relabelled by {1: 2, 2: 1}
sage: ct.affine()
['G', 2, 1] relabelled by {0: 0, 1: 2, 2: 1}

sage: ct = CartanType(["F",4]).dual(); ct
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
sage: ct.affine()
['F', 4, 1] relabelled by {0: 0, 1: 4, 2: 3, 3: 2, 4: 1}
```

Check that we don't inadvertently change the internal relabelling of `ct`:

```
sage: ct
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

5.1.236 Weight lattice realizations

class sage.combinat.root_system.weight_lattice_realizations.**WeightLatticeRealizations**(*base*, *name=No*

Bases: sage.categories.category_types.Category_over_base_ring

The category of weight lattice realizations over a given base ring

A *weight lattice realization* L over a base ring R is a free module (or vector space if R is a field) endowed with an embedding of the root lattice of some root system. By restriction, this embedding defines an embedding of the root lattice of this root system, which makes L a root lattice realization.

Typical weight lattice realizations over $\mathbf{Z}$ include the weight lattice, and ambient lattice. Typical weight lattice realizations over $\mathbf{Q}$ include the weight space, and ambient space.

To describe the embedding, a weight lattice realization must implement a method `fundamental_weight`(i) returning for each `i() in the index set the image of the fundamental weight Λ_i under the embedding.

In order to be a proper root lattice realization, a weight lattice realization should also implement the scalar product with the coroot lattice; on the other hand, the embedding of the simple roots is given for free.

See also:

- RootSystem
- RootLatticeRealizations
- WeightSpace
- AmbientSpace

EXAMPLES:

Here, we consider the root system of type A_7, and embed the weight lattice element $x = \Lambda_1 + 2\Lambda_3$ in several root lattice realizations:

```
sage: R = RootSystem(["A",7])
sage: Lambda = R.weight_lattice().fundamental_weights()
sage: x = Lambda[2] + 2 * Lambda[5]

sage: L = R.weight_space()
sage: L(x)
Lambda[2] + 2*Lambda[5]

sage: L = R.ambient_lattice()
sage: L(x)
(3, 3, 2, 2, 2, 0, 0, 0)
```

We embed the weight space element $x = \Lambda_1 + 1/2\Lambda_3$ in the ambient space:

```
sage: Lambda = R.weight_space().fundamental_weights()
sage: x = Lambda[2] + 1/2 * Lambda[5]

sage: L = R.ambient_space()
sage: L(x)
(3/2, 3/2, 1/2, 1/2, 1/2, 0, 0, 0)
```

Of course, one can't embed the weight space in the ambient lattice:

```
sage: L = R.ambient_lattice()
sage: L(x)
Traceback (most recent call last):
...
TypeError: do not know how to make x (= Lambda[2] + 1/2*Lambda[5]) an element of self (=Ambient
```

If K_1 is a subring of K_2, then one could in theory have an embedding from the weight space over K_1 to any weight lattice realization over K_2; this is not implemented:

```
sage: K1 = QQ
sage: K2 = QQ['q']
sage: L = R.ambient_space(K2)

sage: Lambda = R.weight_space(K2).fundamental_weights()
sage: L(Lambda[1])
(1, 0, 0, 0, 0, 0, 0, 0)

sage: Lambda = R.weight_space(K1).fundamental_weights()
sage: L(Lambda[1])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= Lambda[1]) an element of self (=Ambient space of the Roo
```

class ElementMethods

symmetric_form(*la*)

Return the symmetric form of `self` with `la`.

Return the pairing $(|)$ on the weight lattice. See Chapter 6 in Kac, Infinite Dimensional Lie Algebras for more details.

Warning: For affine root systems, if you are not working in the extended weight lattice/space, this may return incorrect results.

EXAMPLES:

```
sage: P = RootSystem(['C',2]).weight_lattice()
sage: al = P.simple_roots()
sage: al[1].symmetric_form(al[1])
2
sage: al[1].symmetric_form(al[2])
-2
sage: al[2].symmetric_form(al[1])
-2
sage: Q = RootSystem(['C',2]).root_lattice()
sage: alQ = Q.simple_roots()
sage: all(al[i].symmetric_form(al[j]) == alQ[i].symmetric_form(alQ[j])
....:     for i in P.index_set() for j in P.index_set())
True

sage: P = RootSystem(['C',2,1]).weight_lattice(extended=True)
sage: al = P.simple_roots()
sage: al[1].symmetric_form(al[1])
2
sage: al[1].symmetric_form(al[2])
-2
sage: al[1].symmetric_form(al[0])
-2
sage: al[0].symmetric_form(al[1])
-2
sage: Q = RootSystem(['C',2,1]).root_lattice()
sage: alQ = Q.simple_roots()
sage: all(al[i].symmetric_form(al[j]) == alQ[i].symmetric_form(alQ[j])
....:     for i in P.index_set() for j in P.index_set())
True
sage: La = P.basis()
sage: [La['delta'].symmetric_form(al) for al in P.simple_roots()]
[0, 0, 0]
sage: [La[0].symmetric_form(al) for al in P.simple_roots()]
[1, 0, 0]

sage: P = RootSystem(['C',2,1]).weight_lattice()
sage: Q = RootSystem(['C',2,1]).root_lattice()
sage: al = P.simple_roots()
sage: alQ = Q.simple_roots()
sage: all(al[i].symmetric_form(al[j]) == alQ[i].symmetric_form(alQ[j])
....:     for i in P.index_set() for j in P.index_set())
True
```

The result of $(\Lambda_0|\alpha_0)$ should be 1, however we get 0 because we are not working in the extended weight lattice:

```
sage: La = P.basis()
sage: [La[0].symmetric_form(al) for al in P.simple_roots()]
[0, 0, 0]
```

TESTS:

We check that $A_{2n}^{(2)}$ has 3 different root lengths:

```
sage: P = RootSystem(['A',4,2]).weight_lattice()
sage: al = P.simple_roots()
sage: [al[i].symmetric_form(al[i]) for i in P.index_set()]
[2, 4, 8]
```

to_weight_space(*base_ring=None*)

Map `self` to the weight space.

Warning: Implemented for finite Cartan type.

EXAMPLES:

```
sage: b = CartanType(['B',2]).root_system().ambient_space().from_vector(vector([1,-2]));
(1, -2)
sage: b.to_weight_space()
3*Lambda[1] - 4*Lambda[2]
sage: b = CartanType(['B',2]).root_system().ambient_space().from_vector(vector([1/2,0]))
(1/2, 0)
sage: b.to_weight_space()
1/2*Lambda[1]
sage: b.to_weight_space(ZZ)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
sage: b = CartanType(['G',2]).root_system().ambient_space().from_vector(vector([4,-5,1]))
(4, -5, 1)
sage: b.to_weight_space()
-6*Lambda[1] + 5*Lambda[2]
```

class WeightLatticeRealizations.**ParentMethods**

dynkin_diagram_automorphism_of_alcove_morphism(*f*)

Returns the Dynkin diagram automorphism induced by an alcove morphism

INPUT:

- `f` - a linear map from `self` to `self` which preserves alcoves

This method returns the Dynkin diagram automorphism for the decomposition $f = dw$ (see `reduced_word_of_alcove_morphism()`), as a dictionary mapping elements of the index set to itself.

EXAMPLES:

```
sage: R = RootSystem(["A",2,1]).weight_lattice()
sage: alpha = R.simple_roots()
sage: Lambda = R.fundamental_weights()
```

Translations by elements of the root lattice induce a trivial Dynkin diagram automorphism:

```
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(alpha[0].translation)
{0: 0, 1: 1, 2: 2}
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(alpha[1].translation)
{0: 0, 1: 1, 2: 2}
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(alpha[2].translation)
{0: 0, 1: 1, 2: 2}
```

This is no more the case for translations by general elements of the (classical) weight lattice at level 0:

```
sage: omega1 = Lambda[1] - Lambda[0]
sage: omega2 = Lambda[2] - Lambda[0]
```

```
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(omega1.translation)
{0: 1, 1: 2, 2: 0}
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(omega2.translation)
{0: 2, 1: 0, 2: 1}

sage: R = RootSystem(['C',2,1]).weight_lattice()
sage: alpha = R.simple_roots()
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(alpha[1].translation)
{0: 2, 1: 1, 2: 0}

sage: R = RootSystem(['D',5,1]).weight_lattice()
sage: Lambda = R.fundamental_weights()
sage: omega1 = Lambda[1] - Lambda[0]
sage: omega2 = Lambda[2] - 2*Lambda[0]
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(omega1.translation)
{0: 1, 1: 0, 2: 2, 3: 3, 4: 5, 5: 4}
sage: R.dynkin_diagram_automorphism_of_alcove_morphism(omega2.translation)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5}
```

Algorithm: computes w of the decomposition, and see how $f \circ w^{-1}$ permutes the simple roots.

embed_at_level (*x*, *level=1*)

Embed the classical weight x in the level `level` hyperplane

This is achieved by translating the straightforward embedding of x by $c\Lambda_0$ for c some appropriate scalar.

INPUT:

- `x` – an element of the corresponding classical weight/ambient lattice
- `level` – an integer or element of the base ring (default: 1)

EXAMPLES:

```
sage: L = RootSystem(["B",3,1]).weight_space()
sage: L0 = L.classical()
sage: alpha = L0.simple_roots()
sage: omega = L0.fundamental_weights()
sage: L.embed_at_level(omega[1], 1)
Lambda[1]
sage: L.embed_at_level(omega[2], 1)
-Lambda[0] + Lambda[2]
sage: L.embed_at_level(omega[3], 1)
Lambda[3]
sage: L.embed_at_level(alpha[1], 1)
Lambda[0] + 2*Lambda[1] - Lambda[2]
```

fundamental_weight (*i*)

Returns the i^{th} fundamental weight

INPUT:

- `i` – an element of the index set

By a slight notational abuse, for an affine type this method should also accept `"delta"` as input, and return the image of δ of the extended weight lattice in this realization.

This should be overridden by any subclass, and typically be implemented as a cached method for efficiency.

EXAMPLES:

```
sage: L = RootSystem(["A",3]).ambient_lattice()
sage: L.fundamental_weight(1)
(1, 0, 0, 0)
```

```
sage: L = RootSystem(["A",3,1]).weight_lattice(extended=True)
sage: L.fundamental_weight(1)
Lambda[1]
sage: L.fundamental_weight("delta")
delta
```

TESTS:

```
sage: super(sage.combinat.root_system.weight_space.WeightSpace, L).fundamental_weight(1)
Traceback (most recent call last):
...
NotImplementedError: <abstract method fundamental_weight at ...>
```

fundamental_weights()

Returns the family $(\Lambda_i)_{i \in I}$ of the fundamental weights.

EXAMPLES:

```
sage: e = RootSystem(['A',3]).ambient_lattice()
sage: f = e.fundamental_weights()
sage: [f[i] for i in [1,2,3]]
[(1, 0, 0, 0), (1, 1, 0, 0), (1, 1, 1, 0)]
```

is_extended()

Returns whether this is a realization of the extended weight lattice

See also:

sage.combinat.root_system.weight_space.WeightSpace

EXAMPLES:

```
sage: RootSystem(["A",3,1]).weight_lattice().is_extended()
False
sage: RootSystem(["A",3,1]).weight_lattice(extended=True).is_extended()
True
```

This method is irrelevant for finite root systems, since the weight lattice need not be extended to ensure that the root lattice embeds faithfully:

```
sage: RootSystem(["A",3]).weight_lattice().is_extended()
False
```

reduced_word_of_alcove_morphism(f)

INPUT:

- f – a linear map from `self` to `self` which preserves alcoves.

Let A be the fundamental alcove. This returns a reduced word $i_1, ..., i_k$ such that the affine Weyl group element $w = s_{i_1} \circ \cdots \circ s_{i_k}$ maps the alcove $f(A)$ back to A. In other words, the alcove walk $i_1, ..., i_k$ brings the fundamental alcove to the corresponding translated alcove.

Let us throw in a bit of context to explain the main use case. It is customary to realize the alcove picture in the coroot or coweight lattice $R^\vee$. The extended affine Weyl group is then the group of linear maps on $R^\vee$ which preserve the alcoves. By [Kac "Infinite-dimensional Lie algebra", Proposition 6.5] the affine Weyl group is the semidirect product of the associated finite Weyl group and the group of translations in the coroot lattice (the extended affine Weyl group uses the coweight lattice instead). In other words, an element of the extended affine Weyl group admits a unique decomposition of the form:

$$f = dw,$$

where w is in the Weyl group, and d is a function which maps the fundamental alcove to itself. As d permutes the walls of the fundamental alcove, it permutes accordingly the corresponding simple roots, which induces an automorphism of the Dynkin diagram.

This method returns a reduced word for w, whereas the method dynkin_diagram_automorphism_of_alcove_morphism() returns d as a permutation of the nodes of the Dynkin diagram.

Nota bene: recall that the coroot (resp. coweight) lattice is implemented as the root (resp weight) lattice of the dual root system. Hence, this method is implemented for weight lattice realizations, but in practice is most of the time used on the dual side.

EXAMPLES:

We start with type A which is simply laced; hence we do not have to worry about the distinction between the weight and coweight lattice:

```
sage: R = RootSystem(["A",2,1]).weight_lattice()
sage: alpha = R.simple_roots()
sage: Lambda = R.fundamental_weights()
```

We consider first translations by elements of the root lattice:

```
sage: R.reduced_word_of_alcove_morphism(alpha[0].translation)
[1, 2, 1, 0]
sage: R.reduced_word_of_alcove_morphism(alpha[1].translation)
[0, 2, 0, 1]
sage: R.reduced_word_of_alcove_morphism(alpha[2].translation)
[0, 1, 0, 2]
```

We continue with translations by elements of the classical weight lattice, embedded at level 0:
sage: omega1 = Lambda[1] - Lambda[0] sage: omega2 = Lambda[2] - Lambda[0]

sage: R.reduced_word_of_alcove_morphism(omega1.translation) [0, 2] sage: R.reduced_word_of_alcove_morphism(omega2.translation) [0, 1]

The following tests ensure that the code agrees with the tables in Kashiwara's private notes on affine quantum algebras (2008).

TESTS:

```
sage: R = RootSystem(['A',5,1]).weight_lattice()
sage: alpha = R.simple_roots()
sage: Lambda = R.fundamental_weights()
sage: omega1 = Lambda[1] - Lambda[0]
sage: R.reduced_word_of_alcove_morphism(omega1.translation)
[0, 5, 4, 3, 2]
sage: R.reduced_word_of_alcove_morphism(alpha[0].translation)
[1, 2, 3, 4, 5, 4, 3, 2, 1, 0]

sage: R = RootSystem(['C',3,1]).weight_lattice()
sage: alpha = R.simple_roots()
sage: Lambda = R.fundamental_weights()
sage: omega1 = 2*(Lambda[1] - Lambda[0])
sage: omega2 = 2*(Lambda[2] - Lambda[0])
sage: omega3 = Lambda[3] - Lambda[0]
sage: R.reduced_word_of_alcove_morphism(omega1.translation)
[0, 1, 2, 3, 2, 1]
sage: R.reduced_word_of_alcove_morphism(omega2.translation)
[0, 1, 0, 2, 1, 3, 2, 1, 3, 2]
sage: R.reduced_word_of_alcove_morphism(omega3.translation)
[0, 1, 0, 2, 1, 0]
sage: W = WeylGroup(['C',3,1])
sage: s = W.simple_reflections()
sage: w = s[0]*s[1]*s[2]*s[3]*s[2]
sage: W.from_reduced_word(R.reduced_word_of_alcove_morphism(omega2.translation)) == w*w
True
```

```
sage: w = s[0]*s[1]*s[2]*s[0]*s[1]*s[0]
sage: W.from_reduced_word(R.reduced_word_of_alcove_morphism(omega3.translation)) == w
True

sage: R = RootSystem(['D',4,1]).weight_lattice()
sage: Lambda = R.fundamental_weights()
sage: omega1 = Lambda[1] - Lambda[0]
sage: omega2 = Lambda[2] - 2*Lambda[0]
sage: omega3 = Lambda[3] - Lambda[0]
sage: omega4 = Lambda[4] - Lambda[0]
sage: R.reduced_word_of_alcove_morphism(omega1.translation)
[0, 2, 3, 4, 2, 0]
sage: R.reduced_word_of_alcove_morphism(omega2.translation)
[0, 2, 1, 3, 2, 4, 2, 1, 3, 2]
sage: R.reduced_word_of_alcove_morphism(omega3.translation)
[0, 2, 1, 4, 2, 0]
sage: R.reduced_word_of_alcove_morphism(omega4.translation)
[0, 2, 1, 3, 2, 0]
sage: W = WeylGroup(['D',4,1])
sage: s = W.simple_reflections()
sage: w = s[0]*s[2]*s[3]*s[4]*s[2]
sage: w1= s[1]*s[2]*s[3]*s[4]*s[2]
sage: W.from_reduced_word(R.reduced_word_of_alcove_morphism(omega2.translation)) == w*w1
True

sage: R = RootSystem(['D',5,1]).weight_lattice()
sage: Lambda = R.fundamental_weights()
sage: omega1 = Lambda[1] - Lambda[0]
sage: omega2 = Lambda[2] - 2*Lambda[0]
sage: R.reduced_word_of_alcove_morphism(omega1.translation)
[0, 2, 3, 4, 5, 3, 2, 0]
sage: W = WeylGroup(['D',5,1])
sage: s = W.simple_reflections()
sage: w = s[0]*s[2]*s[3]*s[4]*s[5]*s[3]*s[2]
sage: w1= s[1]*s[2]*s[3]*s[4]*s[5]*s[3]*s[2]
sage: W.from_reduced_word(R.reduced_word_of_alcove_morphism(omega2.translation)) == w*w1
True
```

reduced_word_of_translation(*t*)

Given an element of the root lattice, this returns a reduced word $i_1, ..., i_k$ such that the Weyl group element $s_{i_1} \circ \cdots \circ s_{i_k}$ implements the "translation" where x maps to $x + level(x) * t$. In other words, the alcove walk $i_1, ..., i_k$ brings the fundamental alcove to the corresponding translated alcove.

Note: there are some technical conditions for t to actually be a translation; those are not tested (TODO: detail).

EXAMPLES:

```
sage: R = RootSystem(["A",2,1]).weight_lattice()
sage: alpha = R.simple_roots()
sage: R.reduced_word_of_translation(alpha[1])
[0, 2, 0, 1]
sage: R.reduced_word_of_translation(alpha[2])
[0, 1, 0, 2]
sage: R.reduced_word_of_translation(alpha[0])
[1, 2, 1, 0]

sage: R = RootSystem(['D',5,1]).weight_lattice()
sage: Lambda = R.fundamental_weights()
```

```
sage: omega1 = Lambda[1] - Lambda[0]
sage: omega2 = Lambda[2] - 2*Lambda[0]
sage: R.reduced_word_of_translation(omega1)
[0, 2, 3, 4, 5, 3, 2, 0]
sage: R.reduced_word_of_translation(omega2)
[0, 2, 1, 3, 2, 4, 3, 5, 3, 2, 1, 4, 3, 2]
```

A non simply laced case:

```
sage: R = RootSystem(["C",2,1]).weight_lattice()
sage: Lambda = R.fundamental_weights()
sage: c = R.cartan_type().translation_factors()
sage: c
Finite family {0: 1, 1: 2, 2: 1}
sage: R.reduced_word_of_translation((Lambda[1]-Lambda[0]) * c[1])
[0, 1, 2, 1]
sage: R.reduced_word_of_translation((Lambda[2]-Lambda[0]) * c[2])
[0, 1, 0]
```

See also `_test_reduced_word_of_translation()`.

TODO:

- Add a picture in the doc
- Add a method which, given an element of the classical weight lattice, constructs the appropriate value for t

rho()

EXAMPLES:

```
sage: RootSystem(['A',3]).ambient_lattice().rho()
(3, 2, 1, 0)
```

rho_classical()

Return the embedding at level 0 of ρ of the classical lattice.

EXAMPLES:

```
sage: RootSystem(['C',4,1]).weight_lattice().rho_classical()
-4*Lambda[0] + Lambda[1] + Lambda[2] + Lambda[3] + Lambda[4]
sage: L = RootSystem(['D',4,1]).weight_lattice()
sage: L.rho_classical().scalar(L.null_coroot())
0
```

Warning: In affine type BC dual, this does not live in the weight lattice:

```
sage: L = CartanType(["BC",2,2]).dual().root_system().weight_space()
sage: L.rho_classical()
-3/2*Lambda[0] + Lambda[1] + Lambda[2]
sage: L = CartanType(["BC",2,2]).dual().root_system().weight_lattice()
sage: L.rho_classical()
Traceback (most recent call last):
...
ValueError: 5 is not divisible by 2
```

signs_of_alcovewalk(*walk*)

Let walk = $[i_1, \ldots, i_n]$ denote an alcove walk starting from the fundamental alcove y_0, crossing at step 1 the wall i_1, and so on.

For each k, set $w_k = s_{i_1} \circ s_{i_k}$, and denote by $y_k = w_k(y_0)$ the alcove reached after k steps. Then, y_k

is obtained recursively from y_{k-1} by applying the following reflection:

$$y_k = s_{w_{k-1}\alpha_{i_k}} y_{k-1}$$

The step is said positive if $w_{k-1}\alpha_{i_k}$ is a negative root (considering w_{k-1} as element of the classical Weyl group and α_{i_k} as a classical root) and negative otherwise. The algorithm implemented here use the equivalent property:

```
.. MATH:: \langle w_{k-1}^{-1} \rho_0, \alpha^\vee_{i_k}\rangle > 0
```

Where ρ_0 is the sum of the classical fundamental weights embedded at level 0 in this space (see `rho_classical()`), and $\alpha^\vee_{i_k}$ is the simple coroot associated to α_{i_k}.

This function returns a list of the form $[+1, +1, -1, ...]$, where the k^{th} entry denotes whether the k^{th} step was positive or negative.

See equation 3.4, of Ram: Alcove walks ..., arxiv:math/0601343v1 [math.RT]

EXAMPLES:

```
sage: L = RootSystem(['C',2,1]).weight_lattice()
sage: L.signs_of_alcovewalk([1,2,0,1,2,1,2,0,1,2])
[-1, -1, 1, -1, 1, 1, 1, 1, 1, 1]

sage: L = RootSystem(['A',2,1]).weight_lattice()
sage: L.signs_of_alcovewalk([0,1,2,1,2,0,1,2,0,1,2,0])
[1, 1, 1, 1, -1, 1, -1, 1, -1, 1, -1, 1]

sage: L = RootSystem(['B',2,1]).coweight_lattice()
sage: L.signs_of_alcovewalk([0,1,2,0,1,2])
[1, -1, 1, -1, 1, 1]
```

Warning: This method currently does not work in the weight lattice for type BC dual because ρ_0 does not live in this lattice (but an integral multiple of it would do the job as well).

simple_root(i)

Returns the i-th simple root

This default implementation takes the i-th simple root in the weight lattice and embeds it in `self`.

EXAMPLES:

Since all the weight lattice realizations in Sage currently implement a simple_root method, we have to call this one by hand:

```
sage: from sage.combinat.root_system.weight_lattice_realizations import WeightLatticeRea
sage: simple_root = WeightLatticeRealizations(QQ).parent_class.simple_root.f
sage: L = RootSystem("A3").ambient_space()
sage: simple_root(L, 1)
(1, -1, 0, 0)
sage: simple_root(L, 2)
(0, 1, -1, 0)
sage: simple_root(L, 3)
(1, 1, 2, 0)
```

Note that this last root differs from the one implemented in `L` by a multiple of the vector `(1,1,1,1)`:

```
sage: L.simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1)}
```

This is a harmless artefact of the SL versus GL interpretation of type A; see the thematic tutorial on Lie Methods and Related Combinatorics in Sage for details.

weyl_dimension(*highest_weight*)
Return the dimension of the highest weight representation of highest weight `highest_weight`.

EXAMPLES:

```
sage: RootSystem(['A',3]).ambient_lattice().weyl_dimension([2,1,0,0])
20
sage: P = RootSystem(['C',2]).weight_lattice()
sage: La = P.basis()
sage: P.weyl_dimension(La[1]+La[2])
16

sage: type(RootSystem(['A',3]).ambient_lattice().weyl_dimension([2,1,0,0]))
<type 'sage.rings.integer.Integer'>
```

`WeightLatticeRealizations.`**super_categories**()
EXAMPLES:

```
sage: from sage.combinat.root_system.weight_lattice_realizations import WeightLatticeRealiza
sage: WeightLatticeRealizations(QQ).super_categories()
[Category of root lattice realizations over Rational Field]
```

5.1.237 Weight lattices and weight spaces

class `sage.combinat.root_system.weight_space.`**WeightSpace**(*root_system*, *base_ring*, *extended*)
Bases: `sage.combinat.free_module.CombinatorialFreeModule`

INPUT:

- `root_system` – a root system
- `base_ring` – a ring R
- `extended` – a boolean (default: False)

The weight space (or lattice if `base_ring` is $\mathbf{Z}$) of a root system is the formal free module $\bigoplus_i R\Lambda_i$ generated by the fundamental weights $(\Lambda_i)_{i\in I}$ of the root system.

This class is also used for coweight spaces (or lattices).

See also:

- `RootSystem()`
- `RootSystem.weight_lattice()` and `RootSystem.weight_space()`
- `WeightLatticeRealizations()`

EXAMPLES:

```
sage: Q = RootSystem(['A', 3]).weight_lattice(); Q
Weight lattice of the Root system of type ['A', 3]
sage: Q.simple_roots()
Finite family {1: 2*Lambda[1] - Lambda[2], 2: -Lambda[1] + 2*Lambda[2] - Lambda[3], 3: -Lambda[2

sage: Q = RootSystem(['A', 3, 1]).weight_lattice(); Q
Weight lattice of the Root system of type ['A', 3, 1]
sage: Q.simple_roots()
Finite family {0: 2*Lambda[0] -   Lambda[1]               -   Lambda[3],
               1:  -Lambda[0] + 2*Lambda[1] -   Lambda[2],
```

```
                2:                 -Lambda[1] + 2*Lambda[2] -   Lambda[3],
                3:  -Lambda[0]                -   Lambda[2] + 2*Lambda[3]}
```

For infinite types, the Cartan matrix is singular, and therefore the embedding of the root lattice is not faithful:

```
sage: sum(Q.simple_roots())
0
```

In particular, the null root is zero:

```
sage: Q.null_root()
0
```

This can be compensated by extending the basis of the weight space and slightly deforming the simple roots to make them linearly independent, without affecting the scalar product with the coroots. This feature is currently only implemented for affine types. In that case, if `extended` is set, then the basis of the weight space is extended by an element δ:

```
sage: Q = RootSystem(['A', 3, 1]).weight_lattice(extended = True); Q
Extended weight lattice of the Root system of type ['A', 3, 1]
sage: Q.basis().keys()
{0, 1, 2, 3, 'delta'}
```

And the simple root α_0 associated to the special node is deformed as follows:

```
sage: Q.simple_roots()
Finite family {0: 2*Lambda[0] -   Lambda[1]                -   Lambda[3] + delta,
               1:  -Lambda[0] + 2*Lambda[1] -   Lambda[2],
               2:                 -Lambda[1] + 2*Lambda[2] -   Lambda[3],
               3:  -Lambda[0]                -   Lambda[2] + 2*Lambda[3]}
```

Now, the null root is nonzero:

```
sage: Q.null_root()
delta
```

Warning: By a slight notational abuse, the extra basis element used to extend the fundamental weights is called `\delta` in the current implementation. However, in the literature, `\delta` usually denotes instead the null root. Most of the time, those two objects coincide, but not for type BC (aka. $A_{2n}^{(2)}$). Therefore we currently have:

```
sage: Q = RootSystem(["A",4,2]).weight_lattice(extended=True)
sage: Q.simple_root(0)
2*Lambda[0] - Lambda[1] + delta
sage: Q.null_root()
2*delta
```

whereas, with the standard notations from the literature, one would expect to get respectively $2\Lambda_0-\Lambda_1+1/2\delta$ and δ.
Other than this notational glitch, the implementation remains correct for type BC.
The notations may get improved in a subsequent version, which might require changing the index of the extra basis element. To guarantee backward compatibility in code not included in Sage, it is recommended to use the following idiom to get that index:

```
sage: F = Q.basis_extension(); F
Finite family {'delta': delta}
sage: index = F.keys()[0]; index
'delta'
```

Then, for example, the coefficient of an element of the extended weight lattice on that basis element can be recovered with:

```
sage: Q.null_root()[index]
2
```

TESTS:

```
sage: for ct in CartanType.samples(crystallographic=True)+[CartanType(["A",2],["C",5,1])]:
...       TestSuite(ct.root_system().weight_lattice()).run()
...       TestSuite(ct.root_system().weight_space()).run()
sage: for ct in CartanType.samples(affine=True):
...       if ct.is_implemented():
...           P = ct.root_system().weight_space(extended=True)
...           TestSuite(P).run()
```

Element
alias of WeightSpaceElement

basis_extension()
Return the basis elements used to extend the fundamental weights

EXAMPLES:

```
sage: Q = RootSystem(["A",3,1]).weight_lattice()
sage: Q.basis_extension()
Family ()

sage: Q = RootSystem(["A",3,1]).weight_lattice(extended=True)
sage: Q.basis_extension()
Finite family {'delta': delta}
```

This method is irrelevant for finite types:

```
sage: Q = RootSystem(["A",3]).weight_lattice()
sage: Q.basis_extension()
Family ()
```

fundamental_weight(i)

Returns the i-th fundamental weight

INPUT:

- i – an element of the index set or "delta"

By a slight notational abuse, for an affine type this method also accepts "delta" as input, and returns the image of δ of the extended weight lattice in this realization.

EXAMPLES:

```
sage: Q = RootSystem(["A",3]).weight_lattice()
sage: Q.fundamental_weight(1)
Lambda[1]

sage: Q = RootSystem(["A",3,1]).weight_lattice(extended=True)
sage: Q.fundamental_weight(1)
Lambda[1]
sage: Q.fundamental_weight("delta")
delta
```

is_extended()

Returns whether this is an extended weight lattice

EXAMPLES:

```
sage: RootSystem(["A",3,1]).weight_lattice().is_extended()
False
sage: RootSystem(["A",3,1]).weight_lattice(extended=True).is_extended()
True
```

simple_root(j)

Returns the j^{th} simple root

EXAMPLES:

```
sage: L = RootSystem(["C",4]).weight_lattice()
sage: L.simple_root(3)
-Lambda[2] + 2*Lambda[3] - Lambda[4]
```

Its coefficients are given by the corresponding column of the Cartan matrix:

```
sage: L.cartan_type().cartan_matrix()[:,2]
[ 0]
[-1]
[ 2]
[-1]
```

Here are all simple roots:

```
sage: L.simple_roots()
Finite family {1:  2*Lambda[1]   - Lambda[2],
               2:   -Lambda[1] + 2*Lambda[2]   - Lambda[3],
               3:   -Lambda[2] + 2*Lambda[3]   - Lambda[4],
               4:               -2*Lambda[3] + 2*Lambda[4]}
```

For the extended weight lattice of an affine type, the simple root associated to the special node is deformed by adding δ, where δ is the null root:

```
sage: L = RootSystem(["C",4,1]).weight_lattice(extended=True)
sage: L.simple_root(0)
2*Lambda[0] - 2*Lambda[1] + delta
```

In fact δ is really $1/a_0$ times the null root (see the discussion in WeightSpace) but this only makes a difference in type BC:

```
sage: L = RootSystem(CartanType(["BC",4,2])).weight_lattice(extended=True)
sage: L.simple_root(0)
2*Lambda[0] - Lambda[1] + delta
sage: L.null_root()
2*delta
```

See also:

- simple_root()
- CartanType.col_annihilator()

to_ambient_space_morphism()

The morphism from self to its associated ambient space.

EXAMPLES:

```
sage: CartanType(['A',2]).root_system().weight_lattice().to_ambient_space_morphism()
Generic morphism:
From: Weight lattice of the Root system of type ['A', 2]
To:   Ambient space of the Root system of type ['A', 2]
```

Warning: Implemented only for finite Cartan type.

class sage.combinat.root_system.weight_space.**WeightSpaceElement**(M, x)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

is_dominant()

Checks whether an element in the weight space lies in the positive cone spanned by the basis elements (fundamental weights).

EXAMPLES:

```
sage: W = RootSystem(['A',3]).weight_space()
sage: Lambda = W.basis()
sage: w = Lambda[1]+Lambda[3]
sage: w.is_dominant()
True
sage: w = Lambda[1]-Lambda[2]
```

```
sage: w.is_dominant()
False
```

In the extended affine weight lattice, 'delta' is orthogonal to the positive coroots, so adding or subtracting it should not effect dominance

```
sage: P = RootSystem(['A',2,1]).weight_lattice(extended=true)
sage: Lambda = P.fundamental_weights()
sage: delta = P.null_root()
sage: w = Lambda[1]-delta
sage: w.is_dominant()
True
```

scalar(*lambdacheck*)

The canonical scalar product between the weight lattice and the coroot lattice.

Todo

- merge with_apply_multi_module_morphism
- allow for any root space / lattice
- define properly the return type (depends on the base rings of the two spaces)
- make this robust for extended weight lattices (i might be "delta")

EXAMPLES:

```
sage: L = RootSystem(["C",4,1]).weight_lattice()
sage: Lambda     = L.fundamental_weights()
sage: alphacheck = L.simple_coroots()
sage: Lambda[1].scalar(alphacheck[1])
1
sage: Lambda[1].scalar(alphacheck[2])
0
```

The fundamental weights and the simple coroots are dual bases:

```
sage: matrix([ [ Lambda[i].scalar(alphacheck[j])
...              for i in L.index_set() ]
...            for j in L.index_set() ])
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

Note that the scalar product is not yet implemented between the weight space and the coweight space; in any cases, that won't be the job of this method:

```
sage: R = RootSystem(["A",3])
sage: alpha = R.weight_space().roots()
sage: alphacheck = R.coweight_space().roots()
sage: alpha[1].scalar(alphacheck[1])
Traceback (most recent call last):
...
ValueError: -Lambdacheck[1] + 2*Lambdacheck[2] - Lambdacheck[3] is not in the coroot space
```

to_ambient()

Maps `self` to the ambient space.

EXAMPLES:

```
sage: mu = CartanType(['B',2]).root_system().weight_lattice().an_element(); mu
2*Lambda[1] + 2*Lambda[2]
sage: mu.to_ambient()
(3, 1)
```

..warning:

```
Only implemented in finite Cartan type.
Does not work for coweight lattices because there is no implemented map
from the coweight lattice to the ambient space.
```

to_weight_space()

Map `self` to the weight space.

Since $self.parent()$ is the weight space, this map just returns `self`. This overrides the generic method in $WeightSpaceRealizations$.

EXAMPLES:

```
sage: mu = CartanType(['A',2]).root_system().weight_lattice().an_element(); mu
2*Lambda[1] + 2*Lambda[2]
sage: mu.to_weight_space()
2*Lambda[1] + 2*Lambda[2]
```

5.1.238 Weyl Character Rings

class sage.combinat.root_system.weyl_characters.**WeightRing**(*parent*, *prefix*)

Bases: sage.combinat.free_module.CombinatorialFreeModule

The weight ring, which is the group algebra over a weight lattice.

A Weyl character may be regarded as an element of the weight ring. In fact, an element of the weight ring is an element of the `Weyl character ring` if and only if it is invariant under the action of the Weyl group.

The advantage of the weight ring over the Weyl character ring is that one may conduct calculations in the weight ring that involve sums of weights that are not Weyl group invariant.

EXAMPLES:

```
sage: A2 = WeylCharacterRing(['A',2])
sage: a2 = WeightRing(A2)
sage: wd = prod(a2(x/2)-a2(-x/2) for x in a2.space().positive_roots()); wd
a2(-1,1,0) - a2(-1,0,1) - a2(1,-1,0) + a2(1,0,-1) + a2(0,-1,1) - a2(0,1,-1)
sage: chi = A2([5,3,0]); chi
A2(5,3,0)
sage: a2(chi)
a2(1,2,5) + 2*a2(1,3,4) + 2*a2(1,4,3) + a2(1,5,2) + a2(2,1,5)
+ 2*a2(2,2,4) + 3*a2(2,3,3) + 2*a2(2,4,2) + a2(2,5,1) + 2*a2(3,1,4)
+ 3*a2(3,2,3) + 3*a2(3,3,2) + 2*a2(3,4,1) + a2(3,5,0) + a2(3,0,5)
+ 2*a2(4,1,3) + 2*a2(4,2,2) + 2*a2(4,3,1) + a2(4,4,0) + a2(4,0,4)
+ a2(5,1,2) + a2(5,2,1) + a2(5,3,0) + a2(5,0,3) + a2(0,3,5)
+ a2(0,4,4) + a2(0,5,3)
sage: a2(chi)*wd
-a2(-1,3,6) + a2(-1,6,3) + a2(3,-1,6) - a2(3,6,-1) - a2(6,-1,3) + a2(6,3,-1)
sage: sum((-1)^w.length()*a2([6,3,-1]).weyl_group_action(w) for w in a2.space().weyl_group())
-a2(-1,3,6) + a2(-1,6,3) + a2(3,-1,6) - a2(3,6,-1) - a2(6,-1,3) + a2(6,3,-1)
sage: a2(chi)*wd == sum((-1)^w.length()*a2([6,3,-1]).weyl_group_action(w) for w in a2.space().we
True
```

class Element(*M*, *x*)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

A class for weight ring elements.

cartan_type()

Return the Cartan type.

EXAMPLES:

```
sage: A2=WeylCharacterRing("A2")
sage: a2 = WeightRing(A2)
sage: a2([0,1,0]).cartan_type()
['A', 2]
```

character()

Assuming that `self` is invariant under the Weyl group, this will express it as a linear combination of characters. If `self` is not Weyl group invariant, this method will not terminate.

EXAMPLES:

```
sage: A2 = WeylCharacterRing(['A',2])
sage: a2 = WeightRing(A2)
sage: W = a2.space().weyl_group()
sage: mu = a2(2,1,0)
sage: nu = sum(mu.weyl_group_action(w) for w in W) ; nu
a2(1,2,0) + a2(1,0,2) + a2(2,1,0) + a2(2,0,1) + a2(0,1,2) + a2(0,2,1)
sage: nu.character()
-2*A2(1,1,1) + A2(2,1,0)
```

demazure(*w*, *debug=False*)

Return the result of applying the Demazure operator ∂_w to `self`.

INPUT:

- `w` – a Weyl group element, or its reduced word

If $w = s_i$ is a simple reflection, the operation ∂_w sends the weight λ to

$$\frac{\lambda - s_i \cdot \lambda + \alpha_i}{1 + \alpha_i}$$

where the numerator is divisible the denominator in the weight ring. This is extended by multiplicativity to all w in the Weyl group.

EXAMPLES:

```
sage: B2 = WeylCharacterRing("B2",style="coroots")
sage: b2=WeightRing(B2)
sage: b2(1,0).demazure([1])
b2(1,0) + b2(-1,2)
sage: b2(1,0).demazure([2])
b2(1,0)
sage: r=b2(1,0).demazure([1,2]); r
b2(1,0) + b2(-1,2)
sage: r.demazure([1])
b2(1,0) + b2(-1,2)
sage: r.demazure([2])
b2(0,0) + b2(1,0) + b2(1,-2) + b2(-1,2)
```

demazure_lusztig(*i*, *v*)

Return the result of applying the Demazure-Lusztig operator T_i to `self`.

INPUT:

- `i` – an element of the index set (or a reduced word or Weyl group element)

•v – an element of the base ring

If R is the parent WeightRing, the Demazure-Lusztig operator T_i is the linear map $R \to R$ that sends (for a weight λ) $R(\lambda)$ to

$$(R(\alpha_i) - 1)^{-1}\big(R(\lambda) - R(s_i\lambda) - v(R(\lambda) - R(\alpha_i + s_i\lambda))\big)$$

where the numerator is divisible by the denominator in R. The Demazure-Lusztig operators give a representation of the Iwahori–Hecke algebra associated to the Weyl group. See

•Lusztig, Equivariant K-theory and representations of Hecke algebras, Proc. Amer. Math. Soc. 94 (1985), no. 2, 337-342.

•Cherednik, *Nonsymmetric Macdonald polynomials*. IMRN 10, 483-515 (1995).

In the examples, we confirm the braid and quadratic relations for type B_2.

EXAMPLES:

```
sage: P.<v> = PolynomialRing(QQ)
sage: B2 = WeylCharacterRing("B2",style="coroots",base_ring=P); b2 = B2.ambient()
sage: def T1(f) : return f.demazure_lusztig(1,v)
sage: def T2(f) : return f.demazure_lusztig(2,v)
sage: T1(T2(T1(T2(b2(1,-1)))))
(v^2-v)*b2(0,-1) + v^2*b2(-1,1)
sage: [T1(T1(f))==(v-1)*T1(f)+v*f for f in [b2(0,0), b2(1,0), b2(2,3)]]
[True, True, True]
sage: [T1(T2(T1(T2(b2(i,j))))) == T2(T1(T2(T1(b2(i,j))))) for i in [-2..2] for j in [-1,
[True, True, True, True, True, True, True, True, True, True]
```

Instead of an index i one may use a reduced word or Weyl group element:

```
sage: b2(1,0).demazure_lusztig([2,1],v)==T2(T1(b2(1,0)))
True
sage: W = B2.space().weyl_group(prefix="s")
sage: [s1,s2]=W.simple_reflections()
sage: b2(1,0).demazure_lusztig(s2*s1,v)==T2(T1(b2(1,0)))
True
```

scale(*k*)

Multiplies a weight by k. The operation is extended by linearity to the weight ring.

INPUT:

•k – a nonzero integer

EXAMPLES:

```
sage: g2 = WeylCharacterRing("G2",style="coroots").ambient()
sage: g2(2,3).scale(2)
g2(4,6)
```

shift(*mu*)

Add μ to any weight. Extended by linearity to the weight ring.

INPUT:

•mu – a weight

EXAMPLES:

```
sage: g2 = WeylCharacterRing("G2",style="coroots").ambient()
sage: [g2(1,2).shift(fw) for fw in g2.fundamental_weights()]
[g2(2,2), g2(1,3)]
```

weyl_group_action(*w*)

Return the action of the Weyl group element `w` on `self`.

EXAMPLES:

```
sage: G2 = WeylCharacterRing(['G',2])
sage: g2 = WeightRing(G2)
sage: L = g2.space()
sage: [fw1, fw2] = L.fundamental_weights()
sage: sum(g2(fw2).weyl_group_action(w) for w in L.weyl_group())
2*g2(-2,1,1) + 2*g2(-1,-1,2) + 2*g2(-1,2,-1) + 2*g2(1,-2,1) + 2*g2(1,1,-2) + 2*g2(2,-1,-
```

WeightRing.**cartan_type**()

Return the Cartan type.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: WeightRing(A2).cartan_type()
['A', 2]
```

WeightRing.**fundamental_weights**()

Return the fundamental weights.

EXAMPLES:

```
sage: WeightRing(WeylCharacterRing("G2")).fundamental_weights()
Finite family {1: (1, 0, -1), 2: (2, -1, -1)}
```

WeightRing.**one_basis**()

Return the index of 1.

EXAMPLES:

```
sage: A3=WeylCharacterRing("A3")
sage: WeightRing(A3).one_basis()
(0, 0, 0, 0)
sage: WeightRing(A3).one()
a3(0,0,0,0)
```

WeightRing.**parent**()

Return the parent Weyl character ring.

EXAMPLES:

```
sage: A2=WeylCharacterRing("A2")
sage: a2=WeightRing(A2)
sage: a2.parent()
The Weyl Character Ring of Type A2 with Integer Ring coefficients
sage: a2.parent() == A2
True
```

WeightRing.**positive_roots**()

Return the positive roots.

EXAMPLES:

```
sage: WeightRing(WeylCharacterRing("G2")).positive_roots()
[(0, 1, -1), (1, -2, 1), (1, -1, 0), (1, 0, -1), (1, 1, -2), (2, -1, -1)]
```

WeightRing.**product_on_basis**(a, b)

Return the product of basis elements indexed by `a` and `b`.

EXAMPLES:

```
sage: A2=WeylCharacterRing("A2")
sage: a2=WeightRing(A2)
```

```
sage: a2(1,0,0) * a2(0,1,0) # indirect doctest
a2(1,1,0)
```

WeightRing.**simple_roots**()
Return the simple roots.

EXAMPLES:

```
sage: WeightRing(WeylCharacterRing("G2")).simple_roots()
Finite family {1: (0, 1, -1), 2: (1, -2, 1)}
```

WeightRing.**some_elements**()
Return some elements of `self`.

EXAMPLES:

```
sage: A3=WeylCharacterRing("A3")
sage: a3=WeightRing(A3)
sage: a3.some_elements()
[a3(1,0,0,0), a3(1,1,0,0), a3(1,1,1,0)]
```

WeightRing.**space**()
Return the weight space realization associated to `self`.

EXAMPLES:

```
sage: E8 = WeylCharacterRing(['E',8])
sage: e8 = WeightRing(E8)
sage: e8.space()
Ambient space of the Root system of type ['E', 8]
```

WeightRing.**weyl_character_ring**()
Return the parent Weyl Character Ring. A synonym for `self.parent()`.

EXAMPLES:

```
sage: A2=WeylCharacterRing("A2")
sage: a2=WeightRing(A2)
sage: a2.weyl_character_ring()
The Weyl Character Ring of Type A2 with Integer Ring coefficients
```

WeightRing.**wt_repr**(*wt*)
Return a string representing the irreducible character with highest weight vector `wt`. Uses coroot notation if the associated Weyl character ring is defined with `style="coroots"`.

EXAMPLES:

```
sage: G2 = WeylCharacterRing("G2")
sage: [G2.ambient().wt_repr(x) for x in G2.fundamental_weights()]
['g2(1,0,-1)', 'g2(2,-1,-1)']
sage: G2 = WeylCharacterRing("G2",style="coroots")
sage: [G2.ambient().wt_repr(x) for x in G2.fundamental_weights()]
['g2(1,0)', 'g2(0,1)']
```

class sage.combinat.root_system.weyl_characters.**WeylCharacterRing**(*ct*, *base_ring=Integer Ring*, *prefix=None*, *style='lattice'*)

Bases: sage.combinat.free_module.CombinatorialFreeModule

A class for rings of Weyl characters.

Let K be a compact Lie group, which we assume is semisimple and simply-connected. Its complexified Lie algebra L is the Lie algebra of a complex analytic Lie group G. The following three categories are equivalent: finite-dimensional representations of K; finite-dimensional representations of L; and finite-dimensional analytic representations of G. In every case, there is a parametrization of the irreducible representations by their highest weight vectors. For this theory of Weyl, see (for example):

- Adams, *Lectures on Lie groups*
- Broecker and Tom Dieck, *Representations of Compact Lie groups*
- Bump, *Lie Groups*
- Fulton and Harris, *Representation Theory*
- Goodman and Wallach, *Representations and Invariants of the Classical Groups*
- Hall, *Lie Groups, Lie Algebras and Representations*
- Humphreys, *Introduction to Lie Algebras and their representations*
- Procesi, *Lie Groups*
- Samelson, *Notes on Lie Algebras*
- Varadarajan, *Lie groups, Lie algebras, and their representations*
- Zhelobenko, *Compact Lie Groups and their Representations*.

Computations that you can do with these include computing their weight multiplicities, products (thus decomposing the tensor product of a representation into irreducibles) and branching rules (restriction to a smaller group).

There is associated with K, L or G as above a lattice, the weight lattice, whose elements (called weights) are characters of a Cartan subgroup or subalgebra. There is an action of the Weyl group W on the lattice, and elements of a fixed fundamental domain for W, the positive Weyl chamber, are called dominant. There is for each representation a unique highest dominant weight that occurs with nonzero multiplicity with respect to a certain partial order, and it is called the highest weight vector.

EXAMPLES:

```
sage: L = RootSystem("A2").ambient_space()
sage: [fw1,fw2] = L.fundamental_weights()
sage: R = WeylCharacterRing(['A',2], prefix="R")
sage: [R(1),R(fw1),R(fw2)]
[R(0,0,0), R(1,0,0), R(1,1,0)]
```

Here `R(1)`, `R(fw1)`, and `R(fw2)` are irreducible representations with highest weight vectors 0, Λ_1, and Λ_2 respecitively (the first two fundamental weights).

For type A (also G_2, F_4, E_6 and E_7) we will take as the weight lattice not the weight lattice of the semisimple group, but for a larger one. For type A, this means we are concerned with the representation theory of $K = U(n)$ or $G = GL(n, \mathbf{C})$ rather than $SU(n)$ or $SU(n, \mathbf{C})$. This is useful since the representation theory of $GL(n)$ is ubiquitous, and also since we may then represent the fundamental weights (in `sage.combinat.root_system.root_system`) by vectors with integer entries. If you are only interested in $SL(3)$, say, use `WeylCharacterRing(['A',2])` as above but be aware that `R([a,b,c])` and `R([a+1,b+1,c+1])` represent the same character of $SL(3)$ since `R([1,1,1])` is the determinant.

For more information, see the thematic tutorial *Lie Methods and Related Combinatorics in Sage*, available at:

http://www.sagemath.org/doc/thematic_tutorials/lie.html

class Element(*M*, *x*)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

A class for Weyl characters.

adams_operation(*r*)

Return the r-th Adams operation of self.

INPUT:

- r – a positive integer

This is a virtual character, whose weights are the weights of self, each multiplied by r.

EXAMPLES:

```
sage: A2=WeylCharacterRing("A2")
sage: A2(1,1,0).adams_operation(3)
A2(2,2,2) - A2(3,2,1) + A2(3,3,0)
```

branch(*S*, *rule='default'*)

Return the restriction of the character to the subalgebra. If no rule is specified, we will try to specify one.

INPUT:

- S – a Weyl character ring for a Lie subgroup or subalgebra
- rule – a branching rule

See branch_weyl_character() for more information about branching rules.

EXAMPLES:

```
sage: B3 = WeylCharacterRing(['B',3])
sage: A2 = WeylCharacterRing(['A',2])
sage: [B3(w).branch(A2,rule="levi") for w in B3.fundamental_weights()]
[A2(0,0,0) + A2(1,0,0) + A2(0,0,-1),
A2(0,0,0) + A2(1,0,0) + A2(1,1,0) + A2(1,0,-1) + A2(0,-1,-1) + A2(0,0,-1),
A2(-1/2,-1/2,-1/2) + A2(1/2,-1/2,-1/2) + A2(1/2,1/2,-1/2) + A2(1/2,1/2,1/2)]
```

cartan_type()

Return the Cartan type of self.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: A2([1,0,0]).cartan_type()
['A', 2]
```

degree()

The degree of self, that is, the dimension of module.

EXAMPLES:

```
sage: B3 = WeylCharacterRing(['B',3])
sage: [B3(x).degree() for x in B3.fundamental_weights()]
[7, 21, 8]
```

exterior_power(*k*)

Return the k-th exterior power of self.

INPUT:

- k – a nonnegative integer

The algorithm is based on the identity $ke_k = \sum_{r=1}^{k}(-1)^{k-1}p_k e_{k-r}$ relating the power-sum and elementary symmetric polynomials. Applying this to the eigenvalues of an element of the parent Lie group in the representation self, the e_k become exterior powers and the p_k become Adams operations, giving an efficient recursive implementation.

EXAMPLES:

```
sage: B3=WeylCharacterRing("B3",style="coroots")
sage: spin=B3(0,0,1)
sage: spin.exterior_power(6)
B3(1,0,0) + B3(0,1,0)
```

exterior_square()

Return the exterior square of the character.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2",style="coroots")
sage: A2(1,0).exterior_square()
A2(0,1)
```

frobenius_schur_indicator()

Return:

- 1 if the representation is real (orthogonal)
- -1 if the representation is quaternionic (symplectic)
- 0 if the representation is complex (not self dual)

The Frobenius-Schur indicator of a character χ of a compact group G is the Haar integral over the group of $\chi(g^2)$. Its value is 1, -1 or 0. This method computes it for irreducible characters of compact Lie groups by checking whether the symmetric and exterior square characters contain the trivial character.

Todo

Try to compute this directly without actually calculating the full symmetric and exterior squares.

EXAMPLES:

```
sage: B2 = WeylCharacterRing("B2",style="coroots")
sage: B2(1,0).frobenius_schur_indicator()
 1
sage: B2(0,1).frobenius_schur_indicator()
 -1
```

inner_product(*other*)

Compute the inner product with another character.

The irreducible characters are an orthonormal basis with respect to the usual inner product of characters, interpreted as functions on a compact Lie group, by Schur orthogonality.

INPUT:

- `other` – another character

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: [f1,f2]=A2.fundamental_weights()
sage: r1 = A2(f1)*A2(f2); r1
A2(1,1,1) + A2(2,1,0)
sage: r2 = A2(f1)^3; r2
A2(1,1,1) + 2*A2(2,1,0) + A2(3,0,0)
sage: r1.inner_product(r2)
3
```

invariant_degree()

Return the multiplicity of the trivial representation in `self`.

Multiplicities of other irreducibles may be obtained using `multiplicity()`.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2",style="coroots")
sage: rep = A2(1,0)^2*A2(0,1)^2; rep
2*A2(0,0) + A2(0,3) + 4*A2(1,1) + A2(3,0) + A2(2,2)
sage: rep.invariant_degree()
2
```

is_irreducible()

Return whether `self` is an irreducible character.

EXAMPLES:

```
sage: B3 = WeylCharacterRing(['B',3])
sage: [B3(x).is_irreducible() for x in B3.fundamental_weights()]
[True, True, True]
sage: sum(B3(x) for x in B3.fundamental_weights()).is_irreducible()
False
```

multiplicity(*other*)

Return the multiplicity of the irreducible `other` in `self`.

INPUT:

- `other` – an irreducible character

EXAMPLES:

```
sage: B2 = WeylCharacterRing("B2",style="coroots")
sage: rep = B2(1,1)^2; rep
B2(0,0) + B2(1,0) + 2*B2(0,2) + B2(2,0) + 2*B2(1,2) + B2(0,4) + B2(3,0) + B2(2,2)
sage: rep.multiplicity(B2(0,2))
2
```

symmetric_power(k)

Return the k-th symmetric power of `self`.

INPUT:

- k – a nonnegative integer

The algorithm is based on the identity $kh_k = \sum_{r=1}^{k} p_k h_{k-r}$ relating the power-sum and complete symmetric polynomials. Applying this to the eigenvalues of an element of the parent Lie group in the representation `self`, the h_k become symmetric powers and the p_k become Adams operations, giving an efficient recursive implementation.

EXAMPLES:

```
sage: B3=WeylCharacterRing("B3",style="coroots")
sage: spin=B3(0,0,1)
sage: spin.symmetric_power(6)
B3(0,0,0) + B3(0,0,2) + B3(0,0,4) + B3(0,0,6)
```

symmetric_square()

Return the symmetric square of the character.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2",style="coroots")
sage: A2(1,0).symmetric_square()
A2(2,0)
```

weight_multiplicities()

Produce the dictionary of weight multiplicities for the Weyl character `self`. The character does not have to be irreducible.

EXAMPLES:

```
sage: from pprint import pprint
sage: B2=WeylCharacterRing("B2",style="coroots")
```

```
sage: pprint(B2(0,1).weight_multiplicities())
{(-1/2, -1/2): 1, (-1/2, 1/2): 1, (1/2, -1/2): 1, (1/2, 1/2): 1}
```

WeylCharacterRing.**adjoint_representation**()

Returns the adjoint representation as an element of the WeylCharacterRing".

EXAMPLES:

```
sage: G2=WeylCharacterRing("G2",style="coroots")
sage: G2.adjoint_representation()
G2(0,1)
```

WeylCharacterRing.**ambient**()

Returns the weight ring of `self`.

EXAMPLES:

```
sage: WeylCharacterRing("A2").ambient()
The Weight ring attached to The Weyl Character Ring of Type A2 with Integer Ring coefficient
```

WeylCharacterRing.**base_ring**()

Return the base ring of `self`.

EXAMPLES:

```
sage: R = WeylCharacterRing(['A',3], base_ring = CC); R.base_ring()
Complex Field with 53 bits of precision
```

WeylCharacterRing.**cartan_type**()

Return the Cartan type of `self`.

EXAMPLES:

```
sage: WeylCharacterRing("A2").cartan_type()
['A', 2]
```

WeylCharacterRing.**char_from_weights**(*mdict*)

Construct a Weyl character from an invariant linear combination of weights.

INPUT:

- `mdict` – a dictionary mapping weights to coefficients, and representing a linear combination of weights which shall be invariant under the action of the Weyl group

OUTPUT: the corresponding Weyl character

EXAMPLES:

```
sage: from pprint import pprint
sage: A2 = WeylCharacterRing("A2")
sage: v = A2._space([3,1,0]); v
(3, 1, 0)
sage: d = dict([(x,1) for x in v.orbit()]); pprint(d)
{(1, 3, 0): 1,
 (1, 0, 3): 1,
 (3, 1, 0): 1,
 (3, 0, 1): 1,
 (0, 1, 3): 1,
 (0, 3, 1): 1}
sage: A2.char_from_weights(d)
-A2(2,1,1) - A2(2,2,0) + A2(3,1,0)
```

WeylCharacterRing.**demazure_character**(*hwv*, *word*, *debug=False*)

Compute the Demazure character.

INPUT:

- hwv – a (usually dominant) weight
- word – a Weyl group word

Produces the Demazure character with highest weight hwv and word as an element of the weight ring. Only available if style="coroots". The Demazure operators are also available as methods of WeightRing elements, and as methods of crystals. Given a CrystalOfTableaux with given highest weight vector, the Demazure method on the crystal will give the equivalent of this method, except that the Demazure character of the crystal is given as a sum of monomials instead of an element of the WeightRing.

See WeightRing.Element.demazure() and sage.categories.classical_crystals.ClassicalCrys

EXAMPLES:

```
sage: A2=WeylCharacterRing("A2",style="coroots")
sage: h=sum(A2.fundamental_weights()); h
(2, 1, 0)
sage: A2.demazure_character(h,word=[1,2])
a2(0,0) + a2(-2,1) + a2(2,-1) + a2(1,1) + a2(-1,2)
sage: A2.demazure_character((1,1),word=[1,2])
a2(0,0) + a2(-2,1) + a2(2,-1) + a2(1,1) + a2(-1,2)
```

WeylCharacterRing.**dot_reduce**(*a*)

Auxiliary function for product_on_basis().

Return a pair $[\epsilon, b]$ where b is a dominant weight and ϵ is 0, 1 or -1. To describe b, let w be an element of the Weyl group such that $w(a+\rho)$ is dominant. If $w(a+\rho)-\rho$ is dominant, then ϵ is the sign of w and b is $w(a+\rho)-\rho$. Otherwise, ϵ is zero.

INPUT:

- a – a weight

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: weights = sorted(A2(2,1,0).weight_multiplicities().keys(), key=str); weights
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 1, 1), (1, 2, 0), (2, 0, 1), (2, 1, 0)]
sage: [A2.dot_reduce(x) for x in weights]
[[0, (0, 0, 0)], [-1, (1, 1, 1)], [-1, (1, 1, 1)], [1, (1, 1, 1)], [0, (0, 0, 0)], [0, (0, 0
```

WeylCharacterRing.**dynkin_diagram**()

Return the Dynkin diagram of self.

EXAMPLES:

```
sage: WeylCharacterRing("E7").dynkin_diagram()
        O 2
        |
        |
O---O---O---O---O---O
1   3   4   5   6   7
E7
```

WeylCharacterRing.**extended_dynkin_diagram**()

Return the extended Dynkin diagram, which is the Dynkin diagram of the corresponding untwisted affine type.

EXAMPLES:

```
sage: WeylCharacterRing("E7").extended_dynkin_diagram()
            O 2
            |
            |
O---O---O---O---O---O---O
0   1   3   4   5   6   7
E7~
```

WeylCharacterRing.**fundamental_weights**()

Return the fundamental weights.

EXAMPLES:

```
sage: WeylCharacterRing("G2").fundamental_weights()
Finite family {1: (1, 0, -1), 2: (2, -1, -1)}
```

WeylCharacterRing.**highest_root**()

Return the highest_root.

EXAMPLES:

```
sage: WeylCharacterRing("G2").highest_root()
 (2, -1, -1)
```

WeylCharacterRing.**irr_repr**(*hwv*)

Return a string representing the irreducible character with highest weight vector `hwv`.

EXAMPLES:

```
sage: B3 = WeylCharacterRing("B3")
sage: [B3.irr_repr(v) for v in B3.fundamental_weights()]
['B3(1,0,0)', 'B3(1,1,0)', 'B3(1/2,1/2,1/2)']
sage: B3 = WeylCharacterRing("B3", style="coroots")
sage: [B3.irr_repr(v) for v in B3.fundamental_weights()]
['B3(1,0,0)', 'B3(0,1,0)', 'B3(0,0,1)']
```

WeylCharacterRing.**lift**()

The embedding of `self` into its weight ring.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: A2.lift
Generic morphism:
  From: The Weyl Character Ring of Type A2 with Integer Ring coefficients
  To:   The Weight ring attached to The Weyl Character Ring of Type A2 with Integer Ring coe

sage: x = -A2(2,1,1) - A2(2,2,0) + A2(3,1,0)
sage: A2.lift(x)
a2(1,3,0) + a2(1,0,3) + a2(3,1,0) + a2(3,0,1) + a2(0,1,3) + a2(0,3,1)
```

As a shortcut, you may also do:

```
sage: x.lift()
a2(1,3,0) + a2(1,0,3) + a2(3,1,0) + a2(3,0,1) + a2(0,1,3) + a2(0,3,1)
```

Or even:

```
sage: a2 = WeightRing(A2)
sage: a2(x)
a2(1,3,0) + a2(1,0,3) + a2(3,1,0) + a2(3,0,1) + a2(0,1,3) + a2(0,3,1)
```

WeylCharacterRing.**lift_on_basis**(*irr*)

Expand the basis element indexed by the weight `irr` into the weight ring of `self`.

INPUT:

- `irr` – a dominant weight

This is used to implement `lift()`.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: v = A2._space([2,1,0]); v
(2, 1, 0)
sage: A2.lift_on_basis(v)
2*a2(1,1,1) + a2(1,2,0) + a2(1,0,2) + a2(2,1,0) + a2(2,0,1) + a2(0,1,2) + a2(0,2,1)
```

This is consistent with the analoguous calculation with symmetric Schur functions:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s[2,1].expand(3)
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
```

WeylCharacterRing.**maximal_subgroup**(*ct*)

INPUT:

- `ct` – the Cartan type of a maximal subgroup of self.

Returns a branching rule. In rare cases where there is more than one maximal subgroup (up to outer automorphisms) with the given Cartan type, the function returns a list of branching rules.

EXAMPLES:

```
sage: WeylCharacterRing("E7").maximal_subgroup("A2")
miscellaneous branching rule E7 => A2
sage: WeylCharacterRing("E7").maximal_subgroup("A1")
[iii branching rule E7 => A1, iv branching rule E7 => A1]
```

For more information, see the related method `maximal_subgroups()`.

WeylCharacterRing.**maximal_subgroups**()

This method is only available if the Cartan type of self is irreducible and of rank no greater than 8. This method produces a list of the maximal subgroups of self, up to (possibly outer) automorphisms. Each line in the output gives the Cartan type of a maximal subgroup followed by a command that creates the branching rule.

EXAMPLES:

```
sage: WeylCharacterRing("E6").maximal_subgroups()
D5:branching_rule("E6","D5","levi")
C4:branching_rule("E6","C4","symmetric")
F4:branching_rule("E6","F4","symmetric")
A2:branching_rule("E6","A2","miscellaneous")
G2:branching_rule("E6","G2","miscellaneous")
A2xG2:branching_rule("E6","A2xG2","miscellaneous")
A1xA5:branching_rule("E6","A1xA5","extended")
A2xA2xA2:branching_rule("E6","A2xA2xA2","extended")
```

Note that there are other embeddings of (for example A_2 into E_6 as nonmaximal subgroups. These embeddings may be constructed by composing branching rules through various subgroups.

Once you know which maximal subgroup you are interested in, to create the branching rule, you may either paste the command to the right of the colon from the above output onto the command line, or alternatively invoke the related method `maximal_subgroup()`:

```
sage: branching_rule("E6","G2","miscellaneous")
miscellaneous branching rule E6 => G2
sage: WeylCharacterRing("E6").maximal_subgroup("G2")
miscellaneous branching rule E6 => G2
```

It is believed that the list of maximal subgroups is complete, except that some subgroups may be not be invariant under outer automorphisms. It is reasonable to want a list of maximal subgroups that is complete up to conjugation, but to obtain such a list you may have to apply outer automorphisms. The group of outer automorphisms modulo inner automorphisms is isomorphic to the group of symmetries of the Dynkin diagram, and these are available as branching rules. The following example shows that while a branching rule from D_4 to $A_1 imes C_2$ is supplied, another different one may be obtained by composing it with the triality automorphism of D_4:

```
sage: [D4,A1xC2]=[WeylCharacterRing(x,style="coroots") for x in ["D4","A1xC2"]]
sage: fw = D4.fundamental_weights()
sage: b = D4.maximal_subgroup("A1xC2")
sage: [D4(fw).branch(A1xC2,rule=b) for fw in D4.fundamental_weights()]
[A1xC2(1,1,0),
A1xC2(2,0,0) + A1xC2(2,0,1) + A1xC2(0,2,0),
A1xC2(1,1,0),
A1xC2(2,0,0) + A1xC2(0,0,1)]
sage: b1 = branching_rule("D4","D4","triality")*b
sage: [D4(fw).branch(A1xC2,rule=b1) for fw in D4.fundamental_weights()]
[A1xC2(1,1,0),
A1xC2(2,0,0) + A1xC2(2,0,1) + A1xC2(0,2,0),
A1xC2(2,0,0) + A1xC2(0,0,1),
A1xC2(1,1,0)]
```

WeylCharacterRing.**one_basis**()

Return the index of 1 in `self`.

EXAMPLES:

```
sage: WeylCharacterRing("A3").one_basis()
(0, 0, 0, 0)
sage: WeylCharacterRing("A3").one()
A3(0,0,0,0)
```

WeylCharacterRing.**positive_roots**()

Return the positive roots.

EXAMPLES:

```
sage: WeylCharacterRing("G2").positive_roots()
[(0, 1, -1), (1, -2, 1), (1, -1, 0), (1, 0, -1), (1, 1, -2), (2, -1, -1)]
```

WeylCharacterRing.**product_on_basis**(a, b)

Compute the tensor product of two irreducible representations `a` and `b`.

EXAMPLES:

```
sage: D4 = WeylCharacterRing(['D',4])
sage: spin_plus = D4(1/2,1/2,1/2,1/2)
sage: spin_minus = D4(1/2,1/2,1/2,-1/2)
sage: spin_plus * spin_minus # indirect doctest
D4(1,0,0,0) + D4(1,1,1,0)
sage: spin_minus * spin_plus
```

```
D4(1,0,0,0) + D4(1,1,1,0)
```

Uses the Brauer-Klimyk method.

WeylCharacterRing.**rank**()
Return the rank.

EXAMPLES:

```
sage: WeylCharacterRing("G2").rank()
2
```

WeylCharacterRing.**retract**()
The partial inverse map from the weight ring into `self`.

EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: a2 = WeightRing(A2)
sage: A2.retract
Generic morphism:
  From: The Weight ring attached to The Weyl Character Ring of Type A2 with Integer Ring coe
  To:   The Weyl Character Ring of Type A2 with Integer Ring coefficients

sage: v = A2._space([3,1,0]); v
(3, 1, 0)
sage: chi = a2.sum_of_monomials(v.orbit()); chi
a2(1,3,0) + a2(1,0,3) + a2(3,1,0) + a2(3,0,1) + a2(0,1,3) + a2(0,3,1)
sage: A2.retract(chi)
-A2(2,1,1) - A2(2,2,0) + A2(3,1,0)
```

The input should be invariant:

```
sage: A2.retract(a2.monomial(v))
Traceback (most recent call last):
...
ValueError: multiplicity dictionary may not be Weyl group invariant
```

As a shortcut, you may use conversion:

```
sage: A2(chi)
-A2(2,1,1) - A2(2,2,0) + A2(3,1,0)
sage: A2(a2.monomial(v))
Traceback (most recent call last):
...
ValueError: multiplicity dictionary may not be Weyl group invariant
```

WeylCharacterRing.**simple_coroots**()
Return the simple coroots.

EXAMPLES:

```
sage: WeylCharacterRing("G2").simple_coroots()
Finite family {1: (0, 1, -1), 2: (1/3, -2/3, 1/3)}
```

WeylCharacterRing.**simple_roots**()
Return the simple roots.

EXAMPLES:

```
sage: WeylCharacterRing("G2").simple_roots()
Finite family {1: (0, 1, -1), 2: (1, -2, 1)}
```

WeylCharacterRing.**some_elements**()

Return some elements of `self`.

EXAMPLES:

```
sage: WeylCharacterRing("A3").some_elements()
[A3(1,0,0,0), A3(1,1,0,0), A3(1,1,1,0)]
```

WeylCharacterRing.**space**()

Return the weight space associated to `self`.

EXAMPLES:

```
sage: WeylCharacterRing(['E',8]).space()
Ambient space of the Root system of type ['E', 8]
```

sage.combinat.root_system.weyl_characters.**irreducible_character_freudenthal**(*hwv*, *debug=False*)

Return the dictionary of multiplicities for the irreducible character with highest weight λ.

The weight multiplicities are computed by the Freudenthal multiplicity formula. The algorithm is based on recursion relation that is stated, for example, in Humphrey's book on Lie Algebras. The multiplicities are invariant under the Weyl group, so to compute them it would be sufficient to compute them for the weights in the positive Weyl chamber. However after some testing it was found to be faster to compute every weight using the recursion, since the use of the Weyl group is expensive in its current implementation.

INPUT:

- `hwv` – a dominant weight in a weight lattice.
- `L` – the ambient space

EXAMPLES:

```
sage: from pprint import pprint
sage: pprint(WeylCharacterRing("A2")(2,1,0).weight_multiplicities()) # indirect doctest
{(1, 1, 1): 2, (1, 2, 0): 1, (1, 0, 2): 1, (2, 1, 0): 1,
 (2, 0, 1): 1, (0, 1, 2): 1, (0, 2, 1): 1}
```

5.1.239 Weyl Groups

AUTHORS:

- Daniel Bump (2008): initial version
- Mike Hansen (2008): initial version
- Anne Schilling (2008): initial version
- Nicolas Thiery (2008): initial version
- Volker Braun (2013): LibGAP-based matrix groups

EXAMPLES:

More examples on Weyl Groups should be added here...

The Cayley graph of the Weyl Group of type ['A', 3]:

```
sage: w = WeylGroup(['A',3])
sage: d = w.cayley_graph(); d
Digraph on 24 vertices
sage: d.show3d(color_by_label=True, edge_size=0.01, vertex_size=0.03)
```

The Cayley graph of the Weyl Group of type ['D', 4]:

```
sage: w = WeylGroup(['D',4])
sage: d = w.cayley_graph(); d
Digraph on 192 vertices
sage: d.show3d(color_by_label=True, edge_size=0.01, vertex_size=0.03) #long time (less than one minut
```

class sage.combinat.root_system.weyl_group.**ClassicalWeylSubgroup**(*domain*, *prefix*)

Bases: sage.combinat.root_system.weyl_group.WeylGroup_gens

A class for Classical Weyl Subgroup of an affine Weyl Group

EXAMPLES:

```
sage: G = WeylGroup(["A",3,1]).classical()
sage: G
Parabolic Subgroup of the Weyl Group of type ['A', 3, 1] (as a matrix group acting on the root s
sage: G.category()
Category of finite weyl groups
sage: G.cardinality()
24
sage: G.index_set()
(1, 2, 3)
sage: TestSuite(G).run()
```

TESTS:

```
sage: from sage.combinat.root_system.weyl_group import ClassicalWeylSubgroup
sage: H = ClassicalWeylSubgroup(RootSystem(["A", 3, 1]).root_space(), prefix=None)
sage: H is G
True
```

Caveat: the interface is likely to change. The current main application is for plots.

Todo

implement:

- Parabolic subrootsystems
- Parabolic subgroups with a set of nodes as argument

---

cartan_type()

EXAMPLES:

```
sage: WeylGroup(['A',3,1]).classical().cartan_type()
['A', 3]
sage: WeylGroup(['A',3,1]).classical().index_set()
(1, 2, 3)
```

Note: won't be needed, once the lattice will be a parabolic sub root system

simple_reflections()

EXAMPLES:

```
sage: WeylGroup(['A',2,1]).classical().simple_reflections()
Finite family {1: [ 1  0  0]
                  [ 1 -1  1]
                  [ 0  0  1],
               2: [ 1  0  0]
                  [ 0  1  0]
                  [ 1  1 -1]}
```

Note: won't be needed, once the lattice will be a parabolic sub root system

weyl_group (*prefix='hereditary'*)

Return the Weyl group associated to the parabolic subgroup.

EXAMPLES:

```
sage: WeylGroup(['A',4,1]).classical().weyl_group()
Weyl Group of type ['A', 4, 1] (as a matrix group acting on the root space)
sage: WeylGroup(['C',4,1]).classical().weyl_group()
Weyl Group of type ['C', 4, 1] (as a matrix group acting on the root space)
sage: WeylGroup(['E',8,1]).classical().weyl_group()
Weyl Group of type ['E', 8, 1] (as a matrix group acting on the root space)
```

sage.combinat.root_system.weyl_group.**WeylGroup** (*x*, *prefix=None*)

Returns the Weyl group of the root system defined by the Cartan type (or matrix) `ct`.

INPUT:

- `x` - a root system or a Cartan type (or matrix)

OPTIONAL:

- `prefix` – changes the representation of elements from matrices to products of simple reflections

EXAMPLES:

The following constructions yield the same result, namely a weight lattice and its corresponding Weyl group:

```
sage: G = WeylGroup(['F',4])
sage: L = G.domain()
```

or alternatively and equivalently:

```
sage: L = RootSystem(['F',4]).ambient_space()
sage: G = L.weyl_group()
sage: W = WeylGroup(L)
```

Either produces a weight lattice, with access to its roots and weights.

```
sage: G = WeylGroup(['F',4])
sage: G.order()
1152
sage: [s1,s2,s3,s4] = G.simple_reflections()
sage: w = s1*s2*s3*s4; w
[ 1/2  1/2  1/2  1/2]
[-1/2  1/2  1/2 -1/2]
[ 1/2  1/2 -1/2 -1/2]
[ 1/2 -1/2  1/2 -1/2]
sage: type(w) == G.element_class
True
sage: w.order()
12
```

```
sage: w.length() # length function on Weyl group
4
```

The default representation of Weyl group elements is as matrices. If you prefer, you may specify a prefix, in which case the elements are represented as products of simple reflections.

```
sage: W=WeylGroup("C3",prefix="s")
sage: [s1,s2,s3]=W.simple_reflections() # lets Sage parse its own output
sage: s2*s1*s2*s3
s1*s2*s3*s1
sage: s2*s1*s2*s3 == s1*s2*s3*s1
True
sage: (s2*s3)^2==(s3*s2)^2
True
sage: (s1*s2*s3*s1).matrix()
[ 0  0 -1]
[ 0  1  0]
[ 1  0  0]

sage: L = G.domain()
sage: fw = L.fundamental_weights(); fw
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0), 3: (3/2, 1/2, 1/2, 1/2), 4: (1, 0, 0, 0)}
sage: rho = sum(fw); rho
(11/2, 5/2, 3/2, 1/2)
sage: w.action(rho) # action of G on weight lattice
(5, -1, 3, 2)
```

We can also do the same for arbitrary Cartan matrices:

```
sage: cm = CartanMatrix([[2,-5,0],[-2,2,-1],[0,-1,2]])
sage: W = WeylGroup(cm)
sage: W.gens()
(
[-1  5  0]  [ 1  0  0]  [ 1  0  0]
[ 0  1  0]  [ 2 -1  1]  [ 0  1  0]
[ 0  0  1], [ 0  0  1], [ 0  1 -1]
)
sage: s0,s1,s2 = W.gens()
sage: s1*s2*s1
[ 1  0  0]
[ 2  0 -1]
[ 2 -1  0]
sage: s2*s1*s2
[ 1  0  0]
[ 2  0 -1]
[ 2 -1  0]
sage: s0*s1*s0*s2*s0
[ 9  0 -5]
[ 2  0 -1]
[ 0  1 -1]
```

Same Cartan matrix, but with a prefix to display using simple reflections:

```
sage: W = WeylGroup(cm, prefix='s')
sage: s0,s1,s2 = W.gens()
sage: s0*s2*s1
s2*s0*s1
sage: (s1*s2)^3
1
```

```
sage: (s0*s1)^5
s0*s1*s0*s1*s0*s1*s0*s1*s0*s1
sage: s0*s1*s2*s1*s2
s2*s0*s1
sage: s0*s1*s2*s0*s2
s0*s1*s0
```

TESTS:

```
sage: TestSuite(WeylGroup(["A",3])).run()
sage: TestSuite(WeylGroup(["A",3,1])).run()  # long time

sage: W = WeylGroup(['A',3,1])
sage: s = W.simple_reflections()
sage: w = s[0]*s[1]*s[2]
sage: w.reduced_word()
[0, 1, 2]
sage: w = s[0]*s[2]
sage: w.reduced_word()
[2, 0]
sage: W = groups.misc.WeylGroup(['A',3,1])
```

class sage.combinat.root_system.weyl_group.**WeylGroupElement**(*parent*, *g*, *check=False*)

Bases: sage.groups.matrix_gps.group_element.MatrixGroupElement_gap

Class for a Weyl Group elements

action(*v*)

Returns the action of self on the vector v.

EXAMPLES:

```
sage: W = WeylGroup(['A',2])
sage: s = W.simple_reflections()
sage: v = W.domain()([1,0,0])
sage: s[1].action(v)
(0, 1, 0)

sage: W = WeylGroup(RootSystem(['A',2]).root_lattice())
sage: s = W.simple_reflections()
sage: alpha = W.domain().simple_roots()
sage: s[1].action(alpha[1])
-alpha[1]

sage: W=WeylGroup(['A',2,1])
sage: alpha = W.domain().simple_roots()
sage: s = W.simple_reflections()
sage: s[1].action(alpha[1])
-alpha[1]
sage: s[1].action(alpha[0])
alpha[0] + alpha[1]
```

apply_simple_reflection(*i*, *side='right'*)

domain()

Returns the ambient lattice associated with self.

EXAMPLES:

```
sage: W = WeylGroup(['A',2])
sage: s1 = W.simple_reflection(1)
```

```
sage: s1.domain()
Ambient space of the Root system of type ['A', 2]
```

has_descent(*i*, *positive=False*, *side='right'*)

Tests if self has a descent at position i, that is if self is on the strict negative side of the i^{th} simple reflection hyperplane.

If positive is True, tests if it is on the strict positive side instead.

EXAMPLES:

```
sage: W = WeylGroup(['A',3])
sage: s = W.simple_reflections()
sage: [W.one().has_descent(i) for i in W.domain().index_set()]
[False, False, False]
sage: [s[1].has_descent(i) for i in W.domain().index_set()]
[True, False, False]
sage: [s[2].has_descent(i) for i in W.domain().index_set()]
[False, True, False]
sage: [s[3].has_descent(i) for i in W.domain().index_set()]
[False, False, True]
sage: [s[3].has_descent(i, True) for i in W.domain().index_set()]
[True, True, False]
sage: W = WeylGroup(['A',3,1])
sage: s = W.simple_reflections()
sage: [W.one().has_descent(i) for i in W.domain().index_set()]
[False, False, False, False]
sage: [s[0].has_descent(i) for i in W.domain().index_set()]
[True, False, False, False]
sage: w = s[0] * s[1]
sage: [w.has_descent(i) for i in W.domain().index_set()]
[False, True, False, False]
sage: [w.has_descent(i, side = "left") for i in W.domain().index_set()]
[True, False, False, False]
sage: w = s[0] * s[2]
sage: [w.has_descent(i) for i in W.domain().index_set()]
[True, False, True, False]
sage: [w.has_descent(i, side = "left") for i in W.domain().index_set()]
[True, False, True, False]

sage: W = WeylGroup(['A',3])
sage: W.one().has_descent(0)
True
sage: W.w0.has_descent(0)
False
```

has_left_descent(*i*)

Tests if self has a left descent at position i.

EXAMPLES:

```
sage: W = WeylGroup(['A',3])
sage: s = W.simple_reflections()
sage: [W.one().has_descent(i) for i in W.domain().index_set()]
[False, False, False]
sage: [s[1].has_descent(i) for i in W.domain().index_set()]
[True, False, False]
sage: [s[2].has_descent(i) for i in W.domain().index_set()]
[False, True, False]
sage: [s[3].has_descent(i) for i in W.domain().index_set()]
```

```
[False, False, True]
sage: [s[3].has_descent(i, True) for i in W.domain().index_set()]
[True, True, False]
```

to_permutation()

A first approximation of to_permutation ...

This assumes types A,B,C,D on the ambient lattice

This further assume that the basis is indexed by 0,1,... and returns a permutation of (5,4,2,3,1) (beuargl), as a tuple

to_permutation_string()

EXAMPLES:

```
sage: W = WeylGroup(["A",3])
sage: s = W.simple_reflections()
sage: (s[1]*s[2]*s[3]).to_permutation_string()
'2341'
```

class sage.combinat.root_system.weyl_group.**WeylGroup_gens**(*domain*, *prefix*)

Bases: sage.misc.cachefunc.ClearCacheOnPickle, sage.structure.unique_representation.Unique
sage.groups.matrix_gps.finitely_generated.FinitelyGeneratedMatrixGroup_gap

EXAMPLES:

```
sage: G = WeylGroup(['B',3])
sage: TestSuite(G).run()
sage: cm = CartanMatrix([[2,-5,0],[-2,2,-1],[0,-1,2]])
sage: W = WeylGroup(cm)
sage: TestSuite(W).run() # long time
```

Element

alias of WeylGroupElement

bruhat_graph(*x*, *y*)

Return the Bruhat graph as a directed graph, with an edge $u \to v$ if and only if $u < v$ in the Bruhat order, and $u = r \cdot v$.

The Bruhat graph $\Gamma(x, y)$, defined if $x \leq y$ in the Bruhat order, has as its vertices the Bruhat interval $\{t | x \leq t \leq y\}$, and as its edges are the pairs (u, v) such that $u = r \cdot v$ where r is a reflection, that is, a conjugate of a simple reflection.

REFERENCES:

Carrell, The Bruhat graph of a Coxeter group, a conjecture of Deodhar, and rational smoothness of Schubert varieties. Algebraic groups and their generalizations: classical methods (University Park, PA, 1991), 53–61, Proc. Sympos. Pure Math., 56, Part 1, Amer. Math. Soc., Providence, RI, 1994.

EXAMPLES:

```
sage: W = WeylGroup("A3", prefix="s")
sage: s1, s2, s3 = W.simple_reflections()
sage: G = W.bruhat_graph(s1*s3, s1*s2*s3*s2*s1); G
Digraph on 10 vertices
```

Check that the graph has the correct number of edges (see trac ticket #17744):

```
sage: len(G.edges())
16
```

cartan_type()
Returns the CartanType associated to self.

EXAMPLES:

```
sage: G = WeylGroup(['F',4])
sage: G.cartan_type()
['F', 4]
```

character_table()
Returns the character table as a matrix

Each row is an irreducible character. For larger tables you may preface this with a command such as gap.eval("SizeScreen([120,40])") in order to widen the screen.

EXAMPLES:

```
sage: WeylGroup(['A',3]).character_table()
CT1

     2  3  2  2  .  3
     3  1  .  .  1  .

       1a 4a 2a 3a 2b

X.1     1 -1 -1  1  1
X.2     3  1 -1  . -1
X.3     2  .  . -1  2
X.4     3 -1  1  . -1
X.5     1  1  1  1  1
```

classical()
If `self` is a Weyl group from an affine Cartan Type, this give the classical parabolic subgroup of `self`.

Caveat: we assume that 0 is a special node of the Dynkin diagram

TODO: extract parabolic subgroup method

EXAMPLES:

```
sage: G = WeylGroup(['A',3,1])
sage: G.classical()
Parabolic Subgroup of the Weyl Group of type ['A', 3, 1]
 (as a matrix group acting on the root space)
sage: WeylGroup(['A',3]).classical()
Traceback (most recent call last):
...
ValueError: classical subgroup only defined for affine types
```

coxeter_matrix()
Return the Coxeter matrix associated to `self`.

EXAMPLES:

```
sage: G = WeylGroup(['A',3])
sage: G.coxeter_matrix()
[1 3 2]
[3 1 3]
[2 3 1]
```

domain()
Returns the domain of the element of `self`, that is the root lattice realization on which they act.

EXAMPLES:

```
sage: G = WeylGroup(['F',4])
sage: G.domain()
Ambient space of the Root system of type ['F', 4]
sage: G = WeylGroup(['A',3,1])
sage: G.domain()
Root space over the Rational Field of the Root system of type ['A', 3, 1]
```

from_morphism(*f*)

index_set()

Returns the index set of self.

EXAMPLES:

```
sage: G = WeylGroup(['F',4])
sage: G.index_set()
(1, 2, 3, 4)
sage: G = WeylGroup(['A',3,1])
sage: G.index_set()
(0, 1, 2, 3)
```

long_element_hardcoded()

Returns the long Weyl group element (hardcoded data)

Do we really want to keep it? There is a generic implementation which works in all cases. The hardcoded should have a better complexity (for large classical types), but there is a cache, so does this really matter?

EXAMPLES:

```
sage: types = [ ['A',5],['B',3],['C',3],['D',4],['G',2],['F',4],['E',6] ]
sage: [WeylGroup(t).long_element().length() for t in types]
[15, 9, 9, 12, 6, 24, 36]
sage: all( WeylGroup(t).long_element() == WeylGroup(t).long_element_hardcoded() for t in typ
True
```

morphism_matrix(*f*)

one()

Returns the unit element of the Weyl group

EXAMPLES:

```
sage: W = WeylGroup(['A',3])
sage: e = W.one(); e
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: type(e) == W.element_class
True
```

reflections()

Return the reflections of `self`.

The reflections of a Coxeter group W are the conjugates of the simple reflections. They are in bijection with the positive roots, for given a positive root, we may have the reflection in the hyperplane orthogonal to it. This method returns a family indexed by the positive roots taking values in the reflections. This requires `self` to be a finite Weyl group.

Note: Prior to trac ticket #20027, the reflections were the keys of the family and the values were the

positive roots.

EXAMPLES:

```
sage: W = WeylGroup("B2", prefix="s")
sage: refdict = W.reflections(); refdict
Finite family {(1, -1): s1, (1, 1): s2*s1*s2, (1, 0): s1*s2*s1, (0, 1): s2}
sage: [r+refdict[r].action(r) for r in refdict.keys()]
[(0, 0), (0, 0), (0, 0), (0, 0)]
```

simple_reflection(*i*)

Returns the i^{th} simple reflection.

EXAMPLES:

```
sage: G = WeylGroup(['F',4])
sage: G.simple_reflection(1)
[1 0 0 0]
[0 0 1 0]
[0 1 0 0]
[0 0 0 1]
sage: W=WeylGroup(['A',2,1])
sage: W.simple_reflection(1)
[ 1  0  0]
[ 1 -1  1]
[ 0  0  1]
```

simple_reflections()

Returns the simple reflections of self, as a family.

EXAMPLES:

There are the simple reflections for the symmetric group:

```
sage: W=WeylGroup(['A',2])
sage: s = W.simple_reflections(); s
Finite family {1: [0 1 0]
[1 0 0]
[0 0 1], 2: [1 0 0]
[0 0 1]
[0 1 0]}
```

As a special feature, for finite irreducible root systems, s[0] gives the reflection along the highest root:

```
sage: s[0]
[0 0 1]
[0 1 0]
[1 0 0]
```

We now look at some further examples:

```
sage: W=WeylGroup(['A',2,1])
sage: W.simple_reflections()
Finite family {0: [-1  1  1]
[ 0  1  0]
[ 0  0  1], 1: [ 1  0  0]
[ 1 -1  1]
[ 0  0  1], 2: [ 1  0  0]
[ 0  1  0]
[ 1  1 -1]}
sage: W = WeylGroup(['F',4])
```

```
sage: [s1,s2,s3,s4] = W.simple_reflections()
sage: w = s1*s2*s3*s4; w
[ 1/2  1/2  1/2  1/2]
[-1/2  1/2  1/2 -1/2]
[ 1/2  1/2 -1/2 -1/2]
[ 1/2 -1/2  1/2 -1/2]
sage: s4^2 == W.one()
True
sage: type(w) == W.element_class
True
```

unit()

Returns the unit element of the Weyl group

EXAMPLES:

```
sage: W = WeylGroup(['A',3])
sage: e = W.one(); e
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: type(e) == W.element_class
True
```

5.1.240 Rooted (Unordered) Trees

AUTHORS:

- Florent Hivert (2011): initial version

class sage.combinat.rooted_tree.**LabelledRootedTree**(*parent*, *children*, *label=None*, *check=True*)

Bases: sage.combinat.abstract_tree.AbstractLabelledClonableTree, sage.combinat.rooted_tree.RootedTree

Labelled rooted trees.

A labelled rooted tree is a rooted tree with a label attached at each node.

More formally: The *labelled rooted trees* are an inductive datatype defined as follows: A labelled rooted tree is a multiset of labelled rooted trees, endowed with a label (which can be any object, including `None`). The trees that belong to this multiset are said to be the *children* of the tree. (Notice that the labels of these children may and may not be of the same type as the label of the tree). A labelled rooted tree which has no children (so the only information it carries is its label) is said to be a *leaf*.

Every labelled rooted tree gives rise to an unlabelled rooted tree (`RootedTree`) by forgetting the labels. (This is implemented as a conversion.)

INPUT:

- `children` – a list or tuple or more generally any iterable of trees or objects convertible to trees
- `label` – any hashable Sage object (default is `None`)

EXAMPLES:

```
sage: x = LabelledRootedTree([], label = 3); x
3[]
sage: LabelledRootedTree([x, x, x], label = 2)
2[3[], 3[], 3[]]
```

```
sage: LabelledRootedTree((x, x, x), label = 2)
2[3[], 3[], 3[]]
sage: LabelledRootedTree([[],[[], []]], label = 3)
3[None[], None[None[], None[]]]
```

Children are reordered using the value of the `sort_key()` method:

```
sage: y = LabelledRootedTree([], label = 5); y
5[]
sage: xyy2 = LabelledRootedTree((x, y, y), label = 2); xyy2
2[3[], 5[], 5[]]
sage: yxy2 = LabelledRootedTree((y, x, y), label = 2); yxy2
2[3[], 5[], 5[]]
sage: xyy2 == yxy2
True
```

Converting labelled into unlabelled rooted trees by forgetting the labels, and back (the labels are initialized as `None`):

```
sage: yxy2crude = RootedTree(yxy2); yxy2crude
[[], [], []]
sage: LabelledRootedTree(yxy2crude)
None[None[], None[], None[]]
```

TESTS:

```
sage: xyy2._get_list() == yxy2._get_list()
True
```

sort_key()

Return a tuple of nonnegative integers encoding the labelled rooted tree `self`.

The first entry of the tuple is a pair consisting of the number of children of the root and the label of the root. Then the rest of the tuple is obtained as follows: List the tuples corresponding to all children (we are regarding the children themselves as trees). Order this list (not the tuples!) in lexicographically increasing order, and flatten it into a single tuple.

This tuple characterizes the labelled rooted tree uniquely, and can be used to sort the labelled rooted trees provided that the labels belong to a type which is totally ordered.

Note: The tree `self` must be normalized before calling this method (see `normalize()`). This doesn't matter unless you are inside the `clone()` context manager, because outside of it every rooted tree is already normalized.

Note: This method overrides `RootedTree.sort_key()` and returns a result different from what the latter would return, as it wants to encode the whole labelled tree including its labelling rather than just the unlabelled tree. Therefore, be careful with using this method on subclasses of `RootedOrderedTree`; under some circumstances they could inherit it from another superclass instead of from `RootedTree`, which would cause the method to forget the labelling. See the docstrings of `RootedTree.sort_key()` and `sage.combinat.ordered_tree.OrderedTree.sort_key()`.

EXAMPLES:

```
sage: LRT = LabelledRootedTrees(); LRT
Labelled rooted trees
sage: x = LRT([], label = 3); x
3[]
```

```
sage: x.sort_key()
((0, 3),)
sage: y = LRT([x, x, x], label = 2); y
2[3[], 3[], 3[]]
sage: y.sort_key()
((3, 2), (0, 3), (0, 3), (0, 3))
sage: LRT.an_element().sort_key()
((3, 'alpha'), (0, 3), (1, 5), (0, None), (2, 42), (0, 3), (0, 3))
sage: lb = RootedTrees()([[],[[], []]]).canonical_labelling()
sage: lb.sort_key()
((2, 1), (0, 2), (2, 3), (0, 4), (0, 5))
```

class sage.combinat.rooted_tree.**LabelledRootedTrees**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

This is a parent stub to serve as a factory class for labelled rooted trees.

EXAMPLES:

```
sage: LRT = LabelledRootedTrees(); LRT
Labelled rooted trees
sage: x = LRT([], label = 3); x
3[]
sage: x.parent() is LRT
True
sage: y = LRT([x, x, x], label = 2); y
2[3[], 3[], 3[]]
sage: y.parent() is LRT
True
```

Todo

Add the possibility to restrict the labels to a fixed set.

---

class sage.combinat.rooted_tree.**LabelledRootedTrees_all**(*category=None*)

Bases: sage.combinat.rooted_tree.LabelledRootedTrees

Class of all (unordered) labelled rooted trees.

See LabelledRootedTree for a definition.

Element

alias of LabelledRootedTree

labelled_trees()

Return the set of labelled trees associated to self.

EXAMPLES:

```
sage: LabelledRootedTrees().labelled_trees()
Labelled rooted trees
```

unlabelled_trees()

Return the set of unlabelled trees associated to self.

EXAMPLES:

```
sage: LabelledRootedTrees().unlabelled_trees()
Rooted trees
```

class sage.combinat.rooted_tree.**RootedTree**(*parent=None*, *children=[ ]*, *check=True*)

Bases: sage.combinat.abstract_tree.AbstractClonableTree, sage.structure.list_clone.NormalizedClonableList

The class for unordered rooted trees.

The *unordered rooted trees* are an inductive datatype defined as follows: An unordered rooted tree is a multiset of unordered rooted trees. The trees that belong to this multiset are said to be the *children* of the tree. The tree that has no children is called a *leaf*.

The *labelled rooted trees* (LabelledRootedTree) form a subclass of this class; they carry additional data.

One can create a tree from any list (or more generally iterable) of trees or objects convertible to a tree.

EXAMPLES:

```
sage: RootedTree([])
[]
sage: RootedTree([[], [[]]])
[[], [[]]]
sage: RootedTree([[[]], []])
[[], [[]]]
sage: O = OrderedTree([[[]], []]); O
[[[]], []]
sage: RootedTree(O)  # this is O with the ordering forgotten
[[], [[]]]
```

One can also enter any small rooted tree ("small" meaning that no vertex has more than 15 children) by using a simple numerical encoding of rooted trees, namely, the from_hexacode() function. (This function actually parametrizes ordered trees, and here we make it parametrize unordered trees by forgetting the ordering.)

```
sage: from sage.combinat.abstract_tree import from_hexacode
sage: RT = RootedTrees()
sage: from_hexacode('32001010', RT)
[[[]], [[]], [[], []]]
```

Note: Unlike an ordered tree, an (unordered) rooted tree is a multiset (rather than a list) of children. That is, two ordered trees which differ from each other by switching the order of children are equal to each other as (unordered) rooted trees. Internally, rooted trees are encoded as sage.structure.list_clone.NormalizedClonableList instances, and instead of storing their children as an actual multiset, they store their children as a list which is sorted according to their sort_key() value. This is as good as storing them as multisets, since the sort_key() values are sortable and distinguish different (unordered) trees. However, if you wish to define a subclass of RootedTree which implements rooted trees with extra structure (say, a class of edge-colored rooted trees, or a class of rooted trees with a cyclic order on the list of children), then the inherited sort_key() method will no longer distinguish different trees (and, as a consequence, equal trees will be regarded as distinct). Thus, you will have to override the method by one that does distinguish different trees.

---

graft_list(*other*)

Return the list of trees obtained by grafting other on self.

Here grafting means that one takes the disjoint union of self and other, chooses a node of self, and adds the root of other to the list of children of this node. The root of the resulting tree is the root of self. (This can be done for each node of self; this method returns the list of all results.)

This is useful for free pre-Lie algebras.

EXAMPLES:

```
sage: RT = RootedTree
sage: x = RT([])
sage: y = RT([x, x])
sage: x.graft_list(x)
[[[]]]
sage: l = y.graft_list(x); l
[[[], [], []], [[], [[]]], [[], [[]]]]
sage: [parent(i) for i in l]
[Rooted trees, Rooted trees, Rooted trees]
```

TESTS:

```
sage: x = RootedTrees(1)([])
sage: y = RootedTrees(3)([x, x])
sage: l = y.graft_list(x); l
[[[], [], []], [[], [[]]], [[], [[]]]]
sage: [parent(i) for i in l]
[Rooted trees, Rooted trees, Rooted trees]

sage: x = RootedTree([[[], []], []])
sage: y = RootedTree([[], []])
sage: len(uniq(x.graft_list(y)))
4
```

graft_on_root(*other*)

Return the tree obtained by grafting `other` on the root of `self`.

Here grafting means that one takes the disjoint union of `self` and `other`, and adds the root of `other` to the list of children of `self`. The root of the resulting tree is the root of `self`.

This is useful for free Nap algebras.

EXAMPLES:

```
sage: RT = RootedTree
sage: x = RT([])
sage: y = RT([x, x])
sage: x.graft_on_root(x)
[[]]
sage: y.graft_on_root(x)
[[], [], []]
sage: x.graft_on_root(y)
[[[], []]]
```

is_empty()

Return if `self` is the empty tree.

For rooted trees, this always returns `False`.

Note: This is not the same as `bool(t)`, which returns whether `t` has some child or not.

EXAMPLES:

```
sage: t = RootedTrees(4)([[],[[]]])
sage: t.is_empty()
False
sage: bool(t)
True
sage: t = RootedTrees(1)([])
sage: t.is_empty()
```

```
False
sage: bool(t)
False
```

normalize()

Normalize `self`.

This function is at the core of the implementation of rooted (unordered) trees. The underlying structure is provided by ordered rooted trees. Every rooted tree is represented by a normalized element in the set of its planar embeddings.

There should be no need to call `normalize` directly as it is called automatically upon creation and cloning or modification (by `NormalizedClonableList`).

The normalization has a recursive definition. It means first that every sub-tree is itself normalized, and also that sub-trees are sorted. Here the sort is performed according to the values of the `sort_key()` method.

EXAMPLES:

```
sage: RT = RootedTree
sage: RT([[],[[]]]) == RT([[[]],[]])  # indirect doctest
True
sage: rt1 = RT([[],[[]]])
sage: rt2 = RT([[[]],[]])
sage: rt1 is rt2
False
sage: rt1 == rt2
True
sage: rt1._get_list() == rt2._get_list()
True
```

sort_key()

Return a tuple of nonnegative integers encoding the rooted tree `self`.

The first entry of the tuple is the number of children of the root. Then the rest of the tuple is obtained as follows: List the tuples corresponding to all children (we are regarding the children themselves as trees). Order this list (not the tuples!) in lexicographically increasing order, and flatten it into a single tuple.

This tuple characterizes the rooted tree uniquely, and can be used to sort the rooted trees.

Note: The tree `self` must be normalized before calling this method (see `normalize()`). This doesn't matter unless you are inside the `clone()` context manager, because outside of it every rooted tree is already normalized.

Note: By default, this method does not encode any extra structure that `self` might have. If you have a subclass inheriting from `RootedTree` which allows for some extra structure, you need to override `sort_key()` in order to preserve this structure (for example, the `LabelledRootedTree` class does this in `LabelledRootedTree.sort_key()`). See the note in the docstring of `sage.combinat.ordered_tree.OrderedTree.sort_key()` for a pitfall.

EXAMPLES:

```
sage: RT = RootedTree
sage: RT([[],[[]]]).sort_key()
(2, 0, 1, 0)
sage: RT([[[]],[]]).sort_key()
(2, 0, 1, 0)
```

class sage.combinat.rooted_tree.**RootedTrees**
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Factory class for rooted trees.

INPUT:

- size – (optional) an integer

OUTPUT:

the set of all rooted trees (of the given size size if specified)

EXAMPLES:

```
sage: RootedTrees()
Rooted trees

sage: RootedTrees(2)
Rooted trees with 2 nodes
```

class sage.combinat.rooted_tree.**RootedTrees_all**
Bases: sage.sets.disjoint_union_enumerated_sets.DisjointUnionEnumeratedSets, sage.combinat.rooted_tree.RootedTrees

Class of all (unordered, unlabelled) rooted trees.

See RootedTree for a definition.

Element
alias of RootedTree

labelled_trees()
Return the set of labelled trees associated to self.

EXAMPLES:

```
sage: RootedTrees().labelled_trees()
Labelled rooted trees
```

As a consequence:

```
sage: lb = RootedTrees()([[],[[], []]]).canonical_labelling()
sage: lb
1[2[], 3[4[], 5[]]]
sage: lb.__class__
<class 'sage.combinat.rooted_tree.LabelledRootedTrees_all_with_category.element_class'>
sage: lb.parent()
Labelled rooted trees
```

leaf()
Return a leaf tree with self as parent.

EXAMPLES:

```
sage: RootedTrees().leaf()
[]
```

unlabelled_trees()
Return the set of unlabelled trees associated to self.

EXAMPLES:

```
sage: RootedTrees().unlabelled_trees()
Rooted trees
```

class sage.combinat.rooted_tree.**RootedTrees_size**(*n*)

Bases: sage.combinat.rooted_tree.RootedTrees

The enumerated set of rooted trees with a given number of nodes.

The number of nodes of a rooted tree is defined recursively: The number of nodes of a rooted tree with a children is a plus the sum of the number of nodes of each of these children.

TESTS:

```
sage: from sage.combinat.rooted_tree import RootedTrees_size
sage: for i in range(1, 6): TestSuite(RootedTrees_size(i)).run()
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: RootedTrees(1).cardinality()
1
sage: RootedTrees(3).cardinality()
2
```

check_element(*el*, *check=True*)

Check that a given tree actually belongs to `self`.

This just checks the number of vertices.

EXAMPLES:

```
sage: RT3 = RootedTrees(3)
sage: RT3([[],[]])      # indirect doctest
[[], []]
sage: RT3([[],[],[]])  # indirect doctest
Traceback (most recent call last):
...
ValueError: wrong number of nodes
```

element_class()

TESTS:

```
sage: S = RootedTrees(3)
sage: S.element_class
<class 'sage.combinat.rooted_tree.RootedTrees_all_with_category.element_class'>
sage: S.first().__class__ == RootedTrees().first().__class__
True
```

sage.combinat.rooted_tree.**number_of_rooted_trees**(*n*)

Return the number of rooted trees with n nodes.

Compute the number $a(n)$ of rooted trees with n nodes using the recursive formula ([SL000081]):

$$a(n+1) = \frac{1}{n}\sum_{k=1}^{n}\left(\sum_{d|k} d a(d)\right) a(n-k+1)$$

EXAMPLES:

```
sage: from sage.combinat.rooted_tree import number_of_rooted_trees
sage: [number_of_rooted_trees(i) for i in range(10)]
[0, 1, 1, 2, 4, 9, 20, 48, 115, 286]
```

REFERENCES:

5.1.241 Robinson-Schensted-Knuth correspondence

AUTHORS:

- Travis Scrimshaw (2012-12-07): Initial version

EXAMPLES:

We can perform RSK and the inverse on a variety of objects:

```
sage: p = Tableau([[1,2,2],[2]]); q = Tableau([[1,3,3],[2]])
sage: gp = RSK_inverse(p, q); gp
[[1, 2, 3, 3], [2, 1, 2, 2]]
sage: RSK(*gp)
[[[1, 2, 2], [2]], [[1, 3, 3], [2]]]
sage: m = RSK_inverse(p, q, 'matrix'); m
[0 1]
[1 0]
[0 2]
sage: RSK(m)
[[[1, 2, 2], [2]], [[1, 3, 3], [2]]]
```

TESTS:

Check that it is a correspondence between all types of input and the input is preserved:

```
sage: t1 = Tableau([[1, 2, 5], [3], [4]])
sage: t2 = Tableau([[1, 2, 3], [4], [5]])
sage: gp = RSK_inverse(t1, t2); gp
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
sage: w = RSK_inverse(t1, t2, 'word'); w
word: 14532
sage: m = RSK_inverse(t1, t2, 'matrix'); m
[1 0 0 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
[0 0 1 0 0]
[0 1 0 0 0]
sage: p = RSK_inverse(t1, t2, 'permutation'); p
[1, 4, 5, 3, 2]
sage: t1
[[1, 2, 5], [3], [4]]
sage: t2
[[1, 2, 3], [4], [5]]
sage: RSK(*gp) == [t1, t2]
True
sage: RSK(w) == [t1, t2]
True
sage: RSK(m) == [t1, t2]
True
sage: RSK(p) == [t1, t2]
True
```

```
sage: gp
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
sage: w
word: 14532
sage: m
[1 0 0 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
[0 0 1 0 0]
[0 1 0 0 0]
sage: p
[1, 4, 5, 3, 2]
```

REFERENCES:

sage.combinat.rsk.**RSK**(*obj1=None*, *obj2=None*, *insertion='RSK'*, *check_standard=False*, ***options*)

Perform the Robinson-Schensted-Knuth (RSK) correspondence.

The Robinson-Schensted-Knuth (RSK) correspondence (also known as the RSK algorithm) is most naturally stated as a bijection between generalized permutations (also known as two-line arrays, biwords, ...) and pairs of semi-standard Young tableaux (P, Q) of identical shape. The tableau P is known as the insertion tableau, and Q is known as the recording tableau.

The basic operation is known as row insertion $P \leftarrow k$ (where P is a given semi-standard Young tableau, and k is an integer). Row insertion is a recursive algorithm which starts by setting $k_0 = k$, and in its i-th step inserts the number k_i into the i-th row of P (we start counting the rows at 0) by replacing the first integer greater than k_i in the row by k_i and defines k_{i+1} as the integer that has been replaced. If no integer greater than k_i exists in the i-th row, then k_i is simply appended to the row and the algorithm terminates at this point.

Now the RSK algorithm, applied to a generalized permutation $p = ((j_0, k_0), (j_1, k_1), \ldots, (j_{\ell-1}, k_{\ell-1}))$ (encoded as a lexicographically sorted list of pairs) starts by initializing two semi-standard tableaux P_0 and Q_0 as empty tableaux. For each nonnegative integer t starting at 0, take the pair (j_t, k_t) from p and set $P_{t+1} = P_t \leftarrow k_t$, and define Q_{t+1} by adding a new box filled with j_t to the tableau Q_t at the same location the row insertion on P_t ended (that is to say, adding a new box with entry j_t such that P_{t+1} and Q_{t+1} have the same shape). The iterative process stops when t reaches the size of p, and the pair (P_t, Q_t) at this point is the image of p under the Robinson-Schensted-Knuth correspondence.

This correspondence has been introduced in [Knu1970], where it has been referred to as "Construction A".

For more information, see Chapter 7 in [Sta-EC2].

We also note that integer matrices are in bijection with generalized permutations. Furthermore, we can convert any word w (and, in particular, any permutation) to a generalized permutation by considering the top line to be $(1, 2, \ldots, n)$ where n is the length of w.

The optional argument `insertion` allows to specify an alternative insertion procedure to be used instead of the standard Robinson-Schensted-Knuth insertion. If the input is a reduced word of a permutation (i.e., an element of a type-A Coxeter group), one can set `insertion` to `'EG'`, which gives Edelman-Greene insertion, an algorithm defined in [EG1987] Definition 6.20 (where it is referred to as Coxeter-Knuth insertion). The Edelman-Greene insertion is similar to the standard row insertion except that if k_i and $k_i + 1$ both exist in row i, we *only* set $k_{i+1} = k_i + 1$ and continue.

One can also perform a "Hecke RSK algorithm", defined using the Hecke insertion studied in [BKSTY06] (but using rows instead of columns). The algorithm proceeds similarly to the classical RSK algorithm. However, it is not clear in what generality it works; thus, following [BKSTY06], we shall assume that our biword p has top line $(1, 2, \ldots, n)$ (or, at least, has its top line strictly increasing). The Hecke RSK algorithm returns a pair of an increasing tableau and a set-valued standard tableau. If $p = ((j_0, k_0), (j_1, k_1), \ldots, (j_{\ell-1}, k_{\ell-1}))$, then the algorithm recursively constructs pairs $(P_0, Q_0), (P_1, Q_1), \ldots, (P_\ell, Q_\ell)$ of tableaux. The construction of P_{t+1} and Q_{t+1} from P_t, Q_t, j_t and k_t proceeds as follows: Set $i = j_t$, $x = k_t$, $P = P_t$ and $Q = Q_t$. We are going

to insert x into the increasing tableau P and update the set-valued "recording tableau" Q accordingly. As in the classical RSK algorithm, we first insert x into row 1 of P, then into row 2 of the resulting tableau, and so on, until the construction terminates. The details are different: Suppose we are inserting x into row R of P. If (Case 1) there exists an entry y in row R such that $x < y$, then let y be the minimal such entry. We replace this entry y with x if the result is still an increasing tableau; in either subcase, we then continue recursively, inserting y into the next row of P. If, on the other hand, (Case 2) no such y exists, then we append x to the end of R if the result is an increasing tableau (Subcase 2.1), and otherwise (Subcase 2.2) do nothing. Furthermore, in Subcase 2.1, we add the box that we have just filled with x in P to the shape of Q, and fill it with the one-element set $\{i\}$. In Subcase 2.2, we find the bottommost box of the column containing the rightmost box of row R, and add i to the entry of Q in this box (this entry is a set, since Q is a set-valued). In either subcase, we terminate the recursion, and set $P_{t+1} = P$ and $Q_{t+1} = Q$.

Notice that set-valued tableaux are encoded as tableaux whose entries are tuples of positive integers; each such tuple is strictly increasing and encodes a set (namely, the set of its entries).

INPUT:

- `obj1, obj2` – Can be one of the following:
  - A word in an ordered alphabet
  - An integer matrix
  - Two lists of equal length representing a generalized permutation
  - Any object which has a method `_rsk_iter()` which returns an iterator over the object represented as generalized permutation or a pair of lists.
- `insertion` – (Default: `'RSK'`) The following types of insertion are currently supported:
  - `'RSK'` – Robinson-Schensted-Knuth
  - `'EG'` – Edelman-Greene (only for reduced words of permutations/elements of a type-A Coxeter group)
  - `'hecke'` – Hecke insertion (only guaranteed for generalized permutations whose top row is strictly increasing)
- `check_standard` – (Default: `False`) Check if either of the resulting tableaux is a standard tableau, and if so, typecast it as such

EXAMPLES:

If we only give one line, we treat the top line as being $(1, 2, \ldots, n)$:

```
sage: RSK([3,3,2,4,1])
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
sage: RSK(Word([3,3,2,4,1]))
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
sage: RSK(Word([2,3,3,2,1,3,2,3]))
[[[1, 2, 2, 3, 3], [2, 3], [3]], [[1, 2, 3, 6, 8], [4, 7], [5]]]
```

With a generalized permutation:

```
sage: RSK([1, 2, 2, 2], [2, 1, 1, 2])
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
sage: RSK(Word([1,1,3,4,4]), [1,4,2,1,3])
[[[1, 1, 3], [2], [4]], [[1, 1, 4], [3], [4]]]
sage: RSK([1,3,3,4,4], Word([6,2,2,1,7]))
[[[1, 2, 7], [2], [6]], [[1, 3, 4], [3], [4]]]
```

If we give it a matrix:

```
sage: RSK(matrix([[0,1],[2,1]]))
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
```

We can also give it something looking like a matrix:

```
sage: RSK([[0,1],[2,1]])
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
```

There are also variations of the insertion algorithm in RSK. Here we consider Edelman-Greene insertion:

```
sage: RSK([2,1,2,3,2], insertion='EG')
[[[1, 2, 3], [2, 3]], [[1, 3, 4], [2, 5]]]
```

We reproduce figure 6.4 in [EG1987]:

```
sage: RSK([2,3,2,1,2,3], insertion='EG')
[[[1, 2, 3], [2, 3], [3]], [[1, 2, 6], [3, 5], [4]]]
```

Hecke insertion is also supported. We construct Example 2.1 in Arxiv 0801.1319v2:

```
sage: w = [5, 4, 1, 3, 4, 2, 5, 1, 2, 1, 4, 2, 4]
sage: RSK(w, insertion='hecke')
[[[1, 2, 4, 5], [2, 4, 5], [3, 5], [4], [5]],
 [[(1,), (4,), (5,), (7,)],
  [(2,), (9,), (11, 13)],
  [(3,), (12,)],
  [(6,)],
  [(8, 10)]]]
```

There is also `RSK_inverse()` which performs the inverse of the bijection on a pair of semistandard tableaux. We note that the inverse function takes 2 separate tableaux as inputs, so to compose with `RSK()`, we need to use the python $\star$ on the output:

```
sage: RSK_inverse(*RSK([1, 2, 2, 2], [2, 1, 1, 2]))
[[1, 2, 2, 2], [2, 1, 1, 2]]
sage: P,Q = RSK([1, 2, 2, 2], [2, 1, 1, 2])
sage: RSK_inverse(P, Q)
[[1, 2, 2, 2], [2, 1, 1, 2]]
```

TESTS:

Empty objects:

```
sage: RSK(Permutation([]))
[[], []]
sage: RSK(Word([]))
[[], []]
sage: RSK(matrix([[]]))
[[], []]
sage: RSK([], [])
[[], []]
sage: RSK([[]])
[[], []]
sage: RSK(Word([]), insertion='EG')
[[], []]
sage: RSK(Word([]), insertion='hecke')
[[], []]
```

sage.combinat.rsk.**RSK_inverse**(*p*, *q*, *output='array'*, *insertion='RSK'*)

Return the generalized permutation corresponding to the pair of tableaux (p, q) under the inverse of the

Robinson-Schensted-Knuth algorithm.

For more information on the bijection, see `RSK()`.

INPUT:

- `p`, `q` – Two semi-standard tableaux of the same shape, or (in the case when Hecke insertion is used) an increasing tableau and a set-valued tableau of the same shape (see the note below for the format of the set-valued tableau)
- `output` – (Default: `'array'`) if `q` is semi-standard:
  - `'array'` – as a two-line array (i.e. generalized permutation or biword)
  - `'matrix'` – as an integer matrix

  and if `q` is standard, we can have the output:
  - `'word'` – as a word

  and additionally if `p` is standard, we can also have the output:
  - `'permutation'` – as a permutation
- `insertion` – (Default: `RSK`) The insertion algorithm used in the bijection. Currently the following are supported:
  - `'RSK'` – Robinson-Schensted-Knuth insertion
  - `'EG'` – Edelman-Greene insertion
  - `'hecke'` – Hecke insertion

Note: In the case of Hecke insertion, the input variable `q` should be a set-valued tableau, encoded as a tableau whose entries are strictly increasing tuples of positive integers. Each such tuple encodes the set of its entries.

EXAMPLES:

If both `p` and `q` are standard:

```
sage: t1 = Tableau([[1, 2, 5], [3], [4]])
sage: t2 = Tableau([[1, 2, 3], [4], [5]])
sage: RSK_inverse(t1, t2)
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
sage: RSK_inverse(t1, t2, 'word')
word: 14532
sage: RSK_inverse(t1, t2, 'matrix')
[1 0 0 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
[0 0 1 0 0]
[0 1 0 0 0]
sage: RSK_inverse(t1, t2, 'permutation')
[1, 4, 5, 3, 2]
sage: RSK_inverse(t1, t1, 'permutation')
[1, 4, 3, 2, 5]
sage: RSK_inverse(t2, t2, 'permutation')
[1, 2, 5, 4, 3]
sage: RSK_inverse(t2, t1, 'permutation')
[1, 5, 4, 2, 3]
```

If the first tableau is semistandard:

```
sage: p = Tableau([[1,2,2],[3]]); q = Tableau([[1,2,4],[3]])
sage: ret = RSK_inverse(p, q); ret
[[1, 2, 3, 4], [1, 3, 2, 2]]
sage: RSK_inverse(p, q, 'word')
word: 1322
```

In general:

```
sage: p = Tableau([[1,2,2],[2]]); q = Tableau([[1,3,3],[2]])
sage: RSK_inverse(p, q)
[[1, 2, 3, 3], [2, 1, 2, 2]]
sage: RSK_inverse(p, q, 'matrix')
[0 1]
[1 0]
[0 2]
```

Using Edelman-Greene insertion:

```
sage: pq = RSK([2,1,2,3,2], insertion='EG'); pq
[[[1, 2, 3], [2, 3]], [[1, 3, 4], [2, 5]]]
sage: RSK_inverse(*pq, insertion='EG')
[2, 1, 2, 3, 2]
```

Using Hecke insertion:

```
sage: w = [5, 4, 1, 3, 4, 2, 5, 1, 2, 1, 4, 2, 4]
sage: pq = RSK(w, insertion='hecke')
sage: RSK_inverse(*pq, insertion='hecke', output='list')
[5, 4, 1, 3, 4, 2, 5, 1, 2, 1, 4, 2, 4]
```

Note: The constructor of `Tableau` accepts not only semistandard tableaux, but also arbitrary lists that are fillings of a partition diagram. (And such lists are used, e.g., for the set-valued tableau `q` that is passed to `RSK_inverse(p, q, insertion='hecke')`.) The user is responsible for ensuring that the tableaux passed to `RSK_inverse` are of the right types (semistandard, standard, increasing, set-valued as needed).

TESTS:

From empty tableaux:

```
sage: RSK_inverse(Tableau([]), Tableau([]))
[[], []]
```

Check that `RSK_inverse()` is the inverse of `RSK()` on the different types of inputs/outputs:

```
sage: f = lambda p: RSK_inverse(*RSK(p), output='permutation')
sage: all(p == f(p) for n in range(7) for p in Permutations(n))
True
sage: all(RSK_inverse(*RSK(w), output='word') == w for n in range(4) for w in Words(5, n))
True
sage: from sage.combinat.integer_matrices import IntegerMatrices
sage: M = IntegerMatrices([1,2,2,1], [3,1,1,1])
sage: all(RSK_inverse(*RSK(m), output='matrix') == m for m in M)
True

sage: n = ZZ.random_element(200)
sage: p = Permutations(n).random_element()
sage: is_fine = True if p == f(p) else p ; is_fine
True
```

Same for Edelman-Greene (but we are checking only the reduced words that can be obtained using the `reduced_word()` method from permutations):

```
sage: g = lambda w: RSK_inverse(*RSK(w, insertion='EG'), insertion='EG', output='permutation')
sage: all(p.reduced_word() == g(p.reduced_word()) for n in range(7) for p in Permutations(n))
True

sage: n = ZZ.random_element(200)
sage: p = Permutations(n).random_element()
sage: is_fine = True if p == f(p) else p ; is_fine
True
```

Both tableaux must be of the same shape:

```
sage: RSK_inverse(Tableau([[1,2,3]]), Tableau([[1,2]]))
Traceback (most recent call last):
...
ValueError: p(=[[1, 2, 3]]) and q(=[[1, 2]]) must have the same shape
```

sage.combinat.rsk.**hecke_insertion**(*obj1*, *obj2=None*)

Return the Hecke insertion of the pair `[obj1, obj2]`.

See also:

`RSK()`

EXAMPLES:

```
sage: w = [5, 4, 1, 3, 4, 2, 5, 1, 2, 1, 4, 2, 4]
sage: RSK(w, insertion='hecke')
[[[1, 2, 4, 5], [2, 4, 5], [3, 5], [4], [5]],
 [[(1,), (4,), (5,), (7,)],
  [(2,), (9,), (11, 13)],
  [(3,), (12,)],
  [(6,)],
  [(8, 10)]]]
```

sage.combinat.rsk.**hecke_insertion_reverse**(*p*, *q*, *output='array'*)

Return the reverse Hecke insertion of `(p, q)`.

See also:

`RSK_inverse()`

EXAMPLES:

```
sage: w = [5, 4, 1, 3, 4, 2, 5, 1, 2, 1, 4, 2, 4]
sage: P,Q = RSK(w, insertion='hecke')
sage: wp = RSK_inverse(P, Q, insertion='hecke', output='list'); wp
[5, 4, 1, 3, 4, 2, 5, 1, 2, 1, 4, 2, 4]
sage: wp == w
True
```

sage.combinat.rsk.**robinson_schensted_knuth**(*obj1=None*, *obj2=None*, *insertion='RSK'*, *check_standard=False*, ***options*)

Perform the Robinson-Schensted-Knuth (RSK) correspondence.

The Robinson-Schensted-Knuth (RSK) correspondence (also known as the RSK algorithm) is most naturally stated as a bijection between generalized permutations (also known as two-line arrays, biwords, ...) and pairs of semi-standard Young tableaux (P, Q) of identical shape. The tableau P is known as the insertion tableau, and Q is known as the recording tableau.

The basic operation is known as row insertion $P \leftarrow k$ (where P is a given semi-standard Young tableau, and k is an integer). Row insertion is a recursive algorithm which starts by setting $k_0 = k$, and in its i-th step inserts the number k_i into the i-th row of P (we start counting the rows at 0) by replacing the first integer greater than k_i in the row by k_i and defines k_{i+1} as the integer that has been replaced. If no integer greater than k_i exists in the i-th row, then k_i is simply appended to the row and the algorithm terminates at this point.

Now the RSK algorithm, applied to a generalized permutation $p = ((j_0, k_0), (j_1, k_1), \ldots, (j_{\ell-1}, k_{\ell-1}))$ (encoded as a lexicographically sorted list of pairs) starts by initializing two semi-standard tableaux P_0 and Q_0 as empty tableaux. For each nonnegative integer t starting at 0, take the pair (j_t, k_t) from p and set $P_{t+1} = P_t \leftarrow k_t$, and define Q_{t+1} by adding a new box filled with j_t to the tableau Q_t at the same location the row insertion on P_t ended (that is to say, adding a new box with entry j_t such that P_{t+1} and Q_{t+1} have the same shape). The iterative process stops when t reaches the size of p, and the pair (P_t, Q_t) at this point is the image of p under the Robinson-Schensted-Knuth correspondence.

This correspondence has been introduced in [Knu1970], where it has been referred to as "Construction A".

For more information, see Chapter 7 in [Sta-EC2].

We also note that integer matrices are in bijection with generalized permutations. Furthermore, we can convert any word w (and, in particular, any permutation) to a generalized permutation by considering the top line to be $(1, 2, \ldots, n)$ where n is the length of w.

The optional argument `insertion` allows to specify an alternative insertion procedure to be used instead of the standard Robinson-Schensted-Knuth insertion. If the input is a reduced word of a permutation (i.e., an element of a type-A Coxeter group), one can set `insertion` to `'EG'`, which gives Edelman-Greene insertion, an algorithm defined in [EG1987] Definition 6.20 (where it is referred to as Coxeter-Knuth insertion). The Edelman-Greene insertion is similar to the standard row insertion except that if k_i and $k_i + 1$ both exist in row i, we *only* set $k_{i+1} = k_i + 1$ and continue.

One can also perform a "Hecke RSK algorithm", defined using the Hecke insertion studied in [BKSTY06] (but using rows instead of columns). The algorithm proceeds similarly to the classical RSK algorithm. However, it is not clear in what generality it works; thus, following [BKSTY06], we shall assume that our biword p has top line $(1, 2, \ldots, n)$ (or, at least, has its top line strictly increasing). The Hecke RSK algorithm returns a pair of an increasing tableau and a set-valued standard tableau. If $p = ((j_0, k_0), (j_1, k_1), \ldots, (j_{\ell-1}, k_{\ell-1}))$, then the algorithm recursively constructs pairs $(P_0, Q_0), (P_1, Q_1), \ldots, (P_\ell, Q_\ell)$ of tableaux. The construction of P_{t+1} and Q_{t+1} from P_t, Q_t, j_t and k_t proceeds as follows: Set $i = j_t$, $x = k_t$, $P = P_t$ and $Q = Q_t$. We are going to insert x into the increasing tableau P and update the set-valued "recording tableau" Q accordingly. As in the classical RSK algorithm, we first insert x into row 1 of P, then into row 2 of the resulting tableau, and so on, until the construction terminates. The details are different: Suppose we are inserting x into row R of P. If (Case 1) there exists an entry y in row R such that $x < y$, then let y be the minimal such entry. We replace this entry y with x if the result is still an increasing tableau; in either subcase, we then continue recursively, inserting y into the next row of P. If, on the other hand, (Case 2) no such y exists, then we append x to the end of R if the result is an increasing tableau (Subcase 2.1), and otherwise (Subcase 2.2) do nothing. Furthermore, in Subcase 2.1, we add the box that we have just filled with x in P to the shape of Q, and fill it with the one-element set $\{i\}$. In Subcase 2.2, we find the bottommost box of the column containing the rightmost box of row R, and add i to the entry of Q in this box (this entry is a set, since Q is a set-valued). In either subcase, we terminate the recursion, and set $P_{t+1} = P$ and $Q_{t+1} = Q$.

Notice that set-valued tableaux are encoded as tableaux whose entries are tuples of positive integers; each such tuple is strictly increasing and encodes a set (namely, the set of its entries).

INPUT:

- `obj1, obj2` – Can be one of the following:
  - –A word in an ordered alphabet
  - –An integer matrix
  - –Two lists of equal length representing a generalized permutation

– Any object which has a method `_rsk_iter()` which returns an iterator over the object represented as generalized permutation or a pair of lists.

- `insertion` – (Default: `'RSK'`) The following types of insertion are currently supported:
  - `'RSK'` – Robinson-Schensted-Knuth
  - `'EG'` – Edelman-Greene (only for reduced words of permutations/elements of a type-A Coxeter group)
  - `'hecke'` – Hecke insertion (only guaranteed for generalized permutations whose top row is strictly increasing)
- `check_standard` – (Default: `False`) Check if either of the resulting tableaux is a standard tableau, and if so, typecast it as such

EXAMPLES:

If we only give one line, we treat the top line as being $(1, 2, \ldots, n)$:

```
sage: RSK([3,3,2,4,1])
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
sage: RSK(Word([3,3,2,4,1]))
[[[1, 3, 4], [2], [3]], [[1, 2, 4], [3], [5]]]
sage: RSK(Word([2,3,3,2,1,3,2,3]))
[[[1, 2, 2, 3, 3], [2, 3], [3]], [[1, 2, 3, 6, 8], [4, 7], [5]]]
```

With a generalized permutation:

```
sage: RSK([1, 2, 2, 2], [2, 1, 1, 2])
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
sage: RSK(Word([1,1,3,4,4]), [1,4,2,1,3])
[[[1, 1, 3], [2], [4]], [[1, 1, 4], [3], [4]]]
sage: RSK([1,3,3,4,4], Word([6,2,2,1,7]))
[[[1, 2, 7], [2], [6]], [[1, 3, 4], [3], [4]]]
```

If we give it a matrix:

```
sage: RSK(matrix([[0,1],[2,1]]))
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
```

We can also give it something looking like a matrix:

```
sage: RSK([[0,1],[2,1]])
[[[1, 1, 2], [2]], [[1, 2, 2], [2]]]
```

There are also variations of the insertion algorithm in RSK. Here we consider Edelman-Greene insertion:

```
sage: RSK([2,1,2,3,2], insertion='EG')
[[[1, 2, 3], [2, 3]], [[1, 3, 4], [2, 5]]]
```

We reproduce figure 6.4 in [EG1987]:

```
sage: RSK([2,3,2,1,2,3], insertion='EG')
[[[1, 2, 3], [2, 3], [3]], [[1, 2, 6], [3, 5], [4]]]
```

Hecke insertion is also supported. We construct Example 2.1 in Arxiv 0801.1319v2:

```
sage: w = [5, 4, 1, 3, 4, 2, 5, 1, 2, 1, 4, 2, 4]
sage: RSK(w, insertion='hecke')
[[[1, 2, 4, 5], [2, 4, 5], [3, 5], [4], [5]],
 [[(1,), (4,), (5,), (7,)],
  [(2,), (9,), (11, 13)],
  [(3,), (12,)],
```

```
[(6,)],
[(8, 10)]]]
```

There is also `RSK_inverse()` which performs the inverse of the bijection on a pair of semistandard tableaux. We note that the inverse function takes 2 separate tableaux as inputs, so to compose with `RSK()`, we need to use the python $\star$ on the output:

```
sage: RSK_inverse(*RSK([1, 2, 2, 2], [2, 1, 1, 2]))
[[1, 2, 2, 2], [2, 1, 1, 2]]
sage: P,Q = RSK([1, 2, 2, 2], [2, 1, 1, 2])
sage: RSK_inverse(P, Q)
[[1, 2, 2, 2], [2, 1, 1, 2]]
```

TESTS:

Empty objects:

```
sage: RSK(Permutation([]))
[[], []]
sage: RSK(Word([]))
[[], []]
sage: RSK(matrix([[]]))
[[], []]
sage: RSK([], [])
[[], []]
sage: RSK([[]])
[[], []]
sage: RSK(Word([]), insertion='EG')
[[], []]
sage: RSK(Word([]), insertion='hecke')
[[], []]
```

sage.combinat.rsk.**robinson_schensted_knuth_inverse**(*p*, *q*, *output='array'*, *insertion='RSK'*)

Return the generalized permutation corresponding to the pair of tableaux (p, q) under the inverse of the Robinson-Schensted-Knuth algorithm.

For more information on the bijection, see `RSK()`.

INPUT:

- `p`, `q` – Two semi-standard tableaux of the same shape, or (in the case when Hecke insertion is used) an increasing tableau and a set-valued tableau of the same shape (see the note below for the format of the set-valued tableau)
- `output` – (Default: `'array'`) if `q` is semi-standard:
  - `'array'` – as a two-line array (i.e. generalized permutation or biword)
  - `'matrix'` – as an integer matrix

  and if `q` is standard, we can have the output:
  - `'word'` – as a word

  and additionally if `p` is standard, we can also have the output:
  - `'permutation'` – as a permutation
- `insertion` – (Default: `RSK`) The insertion algorithm used in the bijection. Currently the following are supported:
  - `'RSK'` – Robinson-Schensted-Knuth insertion

– 'EG' – Edelman-Greene insertion

– 'hecke' – Hecke insertion

Note: In the case of Hecke insertion, the input variable q should be a set-valued tableau, encoded as a tableau whose entries are strictly increasing tuples of positive integers. Each such tuple encodes the set of its entries.

EXAMPLES:

If both p and q are standard:

```
sage: t1 = Tableau([[1, 2, 5], [3], [4]])
sage: t2 = Tableau([[1, 2, 3], [4], [5]])
sage: RSK_inverse(t1, t2)
[[1, 2, 3, 4, 5], [1, 4, 5, 3, 2]]
sage: RSK_inverse(t1, t2, 'word')
word: 14532
sage: RSK_inverse(t1, t2, 'matrix')
[1 0 0 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
[0 0 1 0 0]
[0 1 0 0 0]
sage: RSK_inverse(t1, t2, 'permutation')
[1, 4, 5, 3, 2]
sage: RSK_inverse(t1, t1, 'permutation')
[1, 4, 3, 2, 5]
sage: RSK_inverse(t2, t2, 'permutation')
[1, 2, 5, 4, 3]
sage: RSK_inverse(t2, t1, 'permutation')
[1, 5, 4, 2, 3]
```

If the first tableau is semistandard:

```
sage: p = Tableau([[1,2,2],[3]]); q = Tableau([[1,2,4],[3]])
sage: ret = RSK_inverse(p, q); ret
[[1, 2, 3, 4], [1, 3, 2, 2]]
sage: RSK_inverse(p, q, 'word')
word: 1322
```

In general:

```
sage: p = Tableau([[1,2,2],[2]]); q = Tableau([[1,3,3],[2]])
sage: RSK_inverse(p, q)
[[1, 2, 3, 3], [2, 1, 2, 2]]
sage: RSK_inverse(p, q, 'matrix')
[0 1]
[1 0]
[0 2]
```

Using Edelman-Greene insertion:

```
sage: pq = RSK([2,1,2,3,2], insertion='EG'); pq
[[[1, 2, 3], [2, 3]], [[1, 3, 4], [2, 5]]]
sage: RSK_inverse(*pq, insertion='EG')
[2, 1, 2, 3, 2]
```

Using Hecke insertion:

```
sage: w = [5, 4, 1, 3, 4, 2, 5, 1, 2, 1, 4, 2, 4]
sage: pq = RSK(w, insertion='hecke')
```

```
sage: RSK_inverse(*pq, insertion='hecke', output='list')
[5, 4, 1, 3, 4, 2, 5, 1, 2, 1, 4, 2, 4]
```

Note: The constructor of `Tableau` accepts not only semistandard tableaux, but also arbitrary lists that are fillings of a partition diagram. (And such lists are used, e.g., for the set-valued tableau `q` that is passed to `RSK_inverse(p, q, insertion='hecke')`.) The user is responsible for ensuring that the tableaux passed to `RSK_inverse` are of the right types (semistandard, standard, increasing, set-valued as needed).

TESTS:

From empty tableaux:

```
sage: RSK_inverse(Tableau([]), Tableau([]))
[[], []]
```

Check that `RSK_inverse()` is the inverse of `RSK()` on the different types of inputs/outputs:

```
sage: f = lambda p: RSK_inverse(*RSK(p), output='permutation')
sage: all(p == f(p) for n in range(7) for p in Permutations(n))
True
sage: all(RSK_inverse(*RSK(w), output='word') == w for n in range(4) for w in Words(5, n))
True
sage: from sage.combinat.integer_matrices import IntegerMatrices
sage: M = IntegerMatrices([1,2,2,1], [3,1,1,1])
sage: all(RSK_inverse(*RSK(m), output='matrix') == m for m in M)
True

sage: n = ZZ.random_element(200)
sage: p = Permutations(n).random_element()
sage: is_fine = True if p == f(p) else p ; is_fine
True
```

Same for Edelman-Greene (but we are checking only the reduced words that can be obtained using the `reduced_word()` method from permutations):

```
sage: g = lambda w: RSK_inverse(*RSK(w, insertion='EG'), insertion='EG', output='permutation')
sage: all(p.reduced_word() == g(p.reduced_word()) for n in range(7) for p in Permutations(n))
True

sage: n = ZZ.random_element(200)
sage: p = Permutations(n).random_element()
sage: is_fine = True if p == f(p) else p ; is_fine
True
```

Both tableaux must be of the same shape:

```
sage: RSK_inverse(Tableau([[1,2,3]]), Tableau([[1,2]]))
Traceback (most recent call last):
...
ValueError: p(=[[1, 2, 3]]) and q(=[[1, 2]]) must have the same shape
```

`sage.combinat.rsk.`**`to_matrix`**`(t, b)`

Return the integer matrix corresponding to a two-line array.

INPUT:

- `t` – The top line of the array
- `b` – The bottom line of the array

OUTPUT:

An $m \times n$-matrix (where m and n are the maximum entries in t and b respectively) whose (i, j)-th entry, for any i and j, is the number of all positions k satisfying $t_k = i$ and $b_k = j$.

EXAMPLES:

```
sage: from sage.combinat.rsk import to_matrix
sage: to_matrix([1, 1, 3, 3, 4], [2, 3, 1, 1, 3])
[0 1 1]
[0 0 0]
[2 0 0]
[0 0 1]
```

5.1.242 Schubert Polynomials

sage.combinat.schubert_polynomial.**SchubertPolynomialRing**(*R*)

Returns the Schubert polynomial ring over R on the X basis.

EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ); X
Schubert polynomial ring with X basis over Integer Ring
sage: X(1)
X[1]
sage: X([1,2,3])*X([2,1,3])
X[2, 1]
sage: X([2,1,3])*X([2,1,3])
X[3, 1, 2]
sage: X([2,1,3])+X([3,1,2,4])
X[2, 1] + X[3, 1, 2]
sage: a = X([2,1,3])+X([3,1,2,4])
sage: a^2
X[3, 1, 2] + 2*X[4, 1, 2, 3] + X[5, 1, 2, 3, 4]
```

class sage.combinat.schubert_polynomial.**SchubertPolynomialRing_xbasis**(*R*)

Bases: sage.combinat.combinatorial_algebra.CombinatorialAlgebra

EXAMPLES:

```
sage: X = SchubertPolynomialRing(QQ)
sage: X == loads(dumps(X))
True
```

Element

alias of SchubertPolynomial_class

class sage.combinat.schubert_polynomial.**SchubertPolynomial_class**(*M*, *x*)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

divided_difference(*i*)

EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ)
sage: a = X([3,2,1])
sage: a.divided_difference(1)
X[2, 3, 1]
sage: a.divided_difference([3,2,1])
X[1]
```

expand()

EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ)
sage: X([2,1,3]).expand()
x0
sage: map(lambda x: x.expand(), [X(p) for p in Permutations(3)])
[1, x0 + x1, x0, x0*x1, x0^2, x0^2*x1]
```

TESTS:

Calling .expand() should always return an element of an MPolynomialRing

```
sage: X = SchubertPolynomialRing(ZZ)
sage: f = X([1]); f
X[1]
sage: type(f.expand())
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular'>
sage: f.expand()
1
sage: f = X([1,2])
sage: type(f.expand())
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular'>
sage: f = X([1,3,2,4])
sage: type(f.expand())
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular'>
```

multiply_variable(*i*)

Returns the Schubert polynomial obtained by multiplying self by the variable x_i.

EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ)
sage: a = X([3,2,4,1])
sage: a.multiply_variable(0)
X[4, 2, 3, 1]
sage: a.multiply_variable(1)
X[3, 4, 2, 1]
sage: a.multiply_variable(2)
X[3, 2, 5, 1, 4] - X[3, 4, 2, 1] - X[4, 2, 3, 1]
sage: a.multiply_variable(3)
X[3, 2, 4, 5, 1]
```

scalar_product(*x*)

Returns the standard scalar product of self and x.

EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ)
sage: a = X([3,2,4,1])
sage: a.scalar_product(a)
0
sage: b = X([4,3,2,1])
sage: b.scalar_product(a)
X[1, 3, 4, 6, 2, 5]
sage: Permutation([1, 3, 4, 6, 2, 5, 7]).to_lehmer_code()
[0, 1, 1, 2, 0, 0, 0]
sage: s = SymmetricFunctions(ZZ).schur()
sage: c = s([2,1,1])
sage: b.scalar_product(a).expand()
x0^2*x1*x2 + x0*x1^2*x2 + x0*x1*x2^2 + x0^2*x1*x3 + x0*x1^2*x3 + x0^2*x2*x3 + 3*x0*x1*x2*x3
sage: c.expand(4)
x0^2*x1*x2 + x0*x1^2*x2 + x0*x1*x2^2 + x0^2*x1*x3 + x0*x1^2*x3 + x0^2*x2*x3 + 3*x0*x1*x2*x3
```

sage.combinat.schubert_polynomial.**is_SchubertPolynomial**(*x*)
Returns True if x is a Schubert polynomial and False otherwise.

EXAMPLES:

```
sage: from sage.combinat.schubert_polynomial import is_SchubertPolynomial
sage: X = SchubertPolynomialRing(ZZ)
sage: a = 1
sage: is_SchubertPolynomial(a)
False
sage: b = X(1)
sage: is_SchubertPolynomial(b)
True
sage: c = X([2,1,3])
sage: is_SchubertPolynomial(c)
True
```

5.1.243 Set Partitions

AUTHORS:

- Mike Hansen
- MuPAD-Combinat developers (for algorithms and design inspiration).
- Travis Scrimshaw (2013-02-28): Removed `CombinatorialClass` and added entry point through `SetPartition`.

class sage.combinat.set_partition.**SetPartition**(*parent*, *s*)
Bases: sage.structure.list_clone.ClonableArray

A partition of a set.

A set partition p of a set S is a partition of S into subsets called parts and represented as a set of sets. By extension, a set partition of a nonnegative integer n is the set partition of the integers from 1 to n. The number of set partitions of n is called the n-th Bell number.

There is a natural integer partition associated with a set partition, namely the nonincreasing sequence of sizes of all its parts.

There is a classical lattice associated with all set partitions of n. The infimum of two set partitions is the set partition obtained by intersecting all the parts of both set partitions. The supremum is obtained by transitive closure of the relation i related to j if and only if they are in the same part in at least one of the set partitions.

We will use terminology from partitions, in particular the *length* of a set partition $A = \{A_1, \ldots, A_k\}$ is the number of parts of A and is denoted by $|A| := k$. The *size* of A is the cardinality of S. We will also sometimes use the notation $[n] := \{1, 2, \ldots, n\}$.

EXAMPLES:

There are 5 set partitions of the set $\{1, 2, 3\}$:

```
sage: SetPartitions(3).cardinality()
5
```

Here is the list of them:

```
sage: SetPartitions(3).list()
[{{1, 2, 3}},
 {{1}, {2, 3}},
 {{1, 3}, {2}},
 {{1, 2}, {3}},
 {{1}, {2}, {3}}]
```

There are 6 set partitions of $\{1, 2, 3, 4\}$ whose underlying partition is $[2, 1, 1]$:

```
sage: SetPartitions(4, [2,1,1]).list()
[{{1}, {2}, {3, 4}},
 {{1}, {2, 4}, {3}},
 {{1}, {2, 3}, {4}},
 {{1, 4}, {2}, {3}},
 {{1, 3}, {2}, {4}},
 {{1, 2}, {3}, {4}}]
```

Since trac ticket #14140, we can create a set partition directly by `SetPartition`, which creates the base set by taking the union of the parts passed in:

```
sage: s = SetPartition([[1,3],[2,4]]); s
{{1, 3}, {2, 4}}
sage: s.parent()
Set partitions
```

apply_permutation(*p*)

Apply `p` to the underlying set of `self`.

INPUT:

- `p` – A permutation

EXAMPLES:

```
sage: x = SetPartition([[1,2], [3,5,4]])
sage: p = Permutation([2,1,4,5,3])
sage: x.apply_permutation(p)
{{1, 2}, {3, 4, 5}}
sage: q = Permutation([3,2,1,5,4])
sage: x.apply_permutation(q)
{{1, 4, 5}, {2, 3}}
```

arcs()

Return `self` as a list of arcs.

Consider a set partition $X = \{X_1, X_2, \ldots, X_n\}$, then the arcs are the pairs $i - j$ such that $i, j \in X_k$ for some k.

EXAMPLES:

```
sage: A = SetPartition([[1],[2,3],[4]])
sage: A.arcs()
[(2, 3)]
sage: B = SetPartition([[1,3,6,7],[2,5],[4]])
sage: B.arcs()
[(1, 3), (3, 6), (6, 7), (2, 5)]
```

base_set()

Return the base set of `self`, which is the union of all parts of `self`.

EXAMPLES:

```
sage: SetPartition([[1], [2,3], [4]]).base_set()
{1, 2, 3, 4}
sage: SetPartition([[1,2,3,4]]).base_set()
{1, 2, 3, 4}
sage: SetPartition([]).base_set()
{}
```

base_set_cardinality()

Return the cardinality of the base set of `self`, which is the sum of the sizes of the parts of `self`.

This is also known as the *size* (sometimes the *weight*) of a set partition.

EXAMPLES:

```
sage: SetPartition([[1], [2,3], [4]]).base_set_cardinality()
4
sage: SetPartition([[1,2,3,4]]).base_set_cardinality()
4
```

cardinality()

Returns the len of `self`

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: len(IncreasingArrays()([1,2,3]))
3
```

check()

Check that we are a valid ordered set partition.

EXAMPLES:

```
sage: OS = OrderedSetPartitions(4)
sage: s = OS([[1, 3], [2, 4]])
sage: s.check()
```

coarsenings()

Return a list of coarsenings of `self`.

EXAMPLES:

```
sage: SetPartition([[1,3],[2,4]]).coarsenings()
[{{1, 2, 3, 4}}, {{1, 3}, {2, 4}}]
sage: SetPartition([[1],[2,4],[3]]).coarsenings()
[{{1, 2, 3, 4}},
 {{1}, {2, 3, 4}},
 {{1, 3}, {2, 4}},
 {{1, 2, 4}, {3}},
 {{1}, {2, 4}, {3}}]
```

```
sage: SetPartition([]).coarsenings()
[{}]
```

inf(*other*)

The product of the set partitions `self` and `other`.

The product of two set partitions B and C is defined as the set partition whose parts are the nonempty intersections between each part of B and each part of C. This product is also the infimum of B and C in the classical set partition lattice (that is, the coarsest set partition which is finer than each of B and C). Consequently, `inf` acts as an alias for this method.

See also:

`sup()`

EXAMPLES:

```
sage: x = SetPartition([ [1,2], [3,5,4] ])
sage: y = SetPartition(( (3,1,2), (5,4) ))
sage: x * y
{{1, 2}, {3}, {4, 5}}

sage: S = SetPartitions(4)
sage: sp1 = S([[2,3,4], [1]])
sage: sp2 = S([[1,3], [2,4]])
sage: s = S([[2,4], [3], [1]])
sage: sp1.inf(sp2) == s
True
```

TESTS:

Here is a different implementation of the `__mul__` method (one that was formerly used for the `inf` method, before it was realized that the methods do the same thing):

```
sage: def mul2(s, t):
....:     temp = [ss.intersection(ts) for ss in s for ts in t]
....:     temp = filter(lambda x: x != Set([]), temp)
....:     return s.__class__(s.parent(), temp)
```

Let us check that this gives the same as `__mul__` on set partitions of $\{1, 2, 3, 4\}$:

```
sage: all( all( mul2(s, t) == s * t for s in SetPartitions(4) )
....:      for t in SetPartitions(4) )
True
```

is_atomic()

Return if `self` is an atomic set partition.

A (standard) set partition A can be split if there exist $j < i$ such that $\max(A_j) < \min(A_i)$ where A is ordered by minimal elements. This means we can write $A = B|C$ for some nonempty set partitions B and C. We call a set partition *atomic* if it cannot be split and is nonempty. Here, the pipe symbol $|$ is as defined in method `pipe()`.

EXAMPLES:

```
sage: SetPartition([[1,3], [2]]).is_atomic()
True
sage: SetPartition([[1,3], [2], [4]]).is_atomic()
False
sage: SetPartition([[1], [2,4], [3]]).is_atomic()
False
```

```
sage: SetPartition([[1,2,3,4]]).is_atomic()
True
sage: SetPartition([[1, 4], [2], [3]]).is_atomic()
True
sage: SetPartition([]).is_atomic()
False
```

is_noncrossing()

Check if `self` is noncrossing.

EXAMPLES:

```
sage: x = SetPartition([[1,2],[3,4]])
sage: x.is_noncrossing()
True
sage: x = SetPartition([[1,3],[2,4]])
sage: x.is_noncrossing()
False
```

AUTHOR: Florent Hivert

ordered_set_partition_action(s)

Return the action of an ordered set partition `s` on `self`.

Let $A = \{A_1, A_2, \ldots, A_k\}$ be a set partition of some set S and s be an ordered set partition (i.e., set composition) of a subset of $[k]$. Let $A^{\downarrow}$ denote the standardization of A, and $A_{\{i_1,i_2,\ldots,i_m\}}$ denote the sub-partition $\{A_{i_1}, A_{i_2}, \ldots, A_{i_m}\}$ for any subset $\{i_1, \ldots, i_m\}$ of $\{1, \ldots, k\}$. We define the set partition $s(A)$ by

$$s(A) = A^{\downarrow}_{s_1} | A^{\downarrow}_{s_2} | \cdots | A^{\downarrow}_{s_q}.$$

where $s = (s_1, s_2, \ldots, s_q)$. Here, the pipe symbol $|$ is as defined in method `pipe()`.

This is $s[A]$ in section 2.3 in [LM2011].

INPUT:

- `s` – an ordered set partition with base set a subset of $\{1, \ldots, k\}$

EXAMPLES:

```
sage: A = SetPartition([[1], [2,4], [3]])
sage: s = OrderedSetPartition([[1,3], [2]])
sage: A.ordered_set_partition_action(s)
{{1}, {2}, {3, 4}}
sage: s = OrderedSetPartition([[2,3], [1]])
sage: A.ordered_set_partition_action(s)
{{1, 3}, {2}, {4}}
```

We create Figure 1 in [LM2011] (we note that there is a typo in the lower-left corner of the table in the published version of the paper, whereas the arXiv version gives the correct partition):

```
sage: A = SetPartition([[1,3], [2,9], [4,5,8], [7]])
sage: B = SetPartition([[1,3], [2,8], [4,5,6], [7]])
sage: C = SetPartition([[1,5], [2,8], [3,4,6], [7]])
sage: s = OrderedSetPartition([[1,3], [2]])
sage: t = OrderedSetPartition([[2], [3,4]])
sage: u = OrderedSetPartition([[1], [2,3,4]])
sage: A.ordered_set_partition_action(s)
{{1, 2}, {3, 4, 5}, {6, 7}}
sage: A.ordered_set_partition_action(t)
{{1, 2}, {3, 4, 6}, {5}}
```

```
sage: A.ordered_set_partition_action(u)
{{1, 2}, {3, 8}, {4, 5, 7}, {6}}
sage: B.ordered_set_partition_action(s)
{{1, 2}, {3, 4, 5}, {6, 7}}
sage: B.ordered_set_partition_action(t)
{{1, 2}, {3, 4, 5}, {6}}
sage: B.ordered_set_partition_action(u)
{{1, 2}, {3, 8}, {4, 5, 6}, {7}}
sage: C.ordered_set_partition_action(s)
{{1, 4}, {2, 3, 5}, {6, 7}}
sage: C.ordered_set_partition_action(t)
{{1, 2}, {3, 4, 5}, {6}}
sage: C.ordered_set_partition_action(u)
{{1, 2}, {3, 8}, {4, 5, 6}, {7}}
```

REFERENCES:

pipe(*other*)

Return the pipe of the set partitions `self` and `other`.

The pipe of two set partitions is defined as follows:

For any integer k and any subset I of $\mathbf{Z}$, let $I + k$ denote the subset of $\mathbf{Z}$ obtained by adding k to every element of k.

If B and C are set partitions of $[n]$ and $[m]$, respectively, then the pipe of B and C is defined as the set partition

$$\{B_1, B_2, \ldots, B_b, C_1 + n, C_2 + n, \ldots, C_c + n\}$$

of $[n + m]$, where $B = \{B_1, B_2, \ldots, B_b\}$ and $C = \{C_1, C_2, \ldots, C_c\}$. This pipe is denoted by $B|C$.

EXAMPLES:

```
sage: SetPartition([[1,3],[2,4]]).pipe(SetPartition([[1,3],[2]]))
{{1, 3}, {2, 4}, {5, 7}, {6}}
sage: SetPartition([]).pipe(SetPartition([[1,2],[3,5],[4]]))
{{1, 2}, {3, 5}, {4}}
sage: SetPartition([[1,2],[3,5],[4]]).pipe(SetPartition([]))
{{1, 2}, {3, 5}, {4}}
sage: SetPartition([[1,2],[3]]).pipe(SetPartition([[1]]))
{{1, 2}, {3}, {4}}
```

refinements()

Return a list of refinements of `self`.

EXAMPLES:

```
sage: SetPartition([[1,3],[2,4]]).refinements()
[{{1, 3}, {2, 4}},
 {{1, 3}, {2}, {4}},
 {{1}, {2, 4}, {3}},
 {{1}, {2}, {3}, {4}}]
sage: SetPartition([[1],[2,4],[3]]).refinements()
[{{1}, {2, 4}, {3}}, {{1}, {2}, {3}, {4}}]
sage: SetPartition([]).refinements()
[{}]
```

restriction(*I*)

Return the restriction of `self` to a subset `I` (which is given as a set or list or any other iterable).

EXAMPLES:

```
sage: A = SetPartition([[1], [2,3]])
sage: A.restriction([1,2])
{{1}, {2}}
sage: A.restriction([2,3])
{{2, 3}}
sage: A.restriction([])
{}
sage: A.restriction([4])
{}
```

shape()

Return the integer partition whose parts are the sizes of the sets in `self`.

EXAMPLES:

```
sage: S = SetPartitions(5)
sage: x = S([[1,2], [3,5,4]])
sage: x.shape()
[3, 2]
sage: y = S([[2], [3,1], [5,4]])
sage: y.shape()
[2, 2, 1]
```

shape_partition()

Return the integer partition whose parts are the sizes of the sets in `self`.

EXAMPLES:

```
sage: S = SetPartitions(5)
sage: x = S([[1,2], [3,5,4]])
sage: x.shape()
[3, 2]
sage: y = S([[2], [3,1], [5,4]])
sage: y.shape()
[2, 2, 1]
```

size()

Return the cardinality of the base set of `self`, which is the sum of the sizes of the parts of `self`.

This is also known as the *size* (sometimes the *weight*) of a set partition.

EXAMPLES:

```
sage: SetPartition([[1], [2,3], [4]]).base_set_cardinality()
4
sage: SetPartition([[1,2,3,4]]).base_set_cardinality()
4
```

standard_form()

Return `self` as a list of lists.

When the ground set is totally ordered, the elements of each block are listed in increasing order.

This is not related to standard set partitions (which simply means set partitions of $[n] = \{1, 2, \ldots, n\}$ for some integer n) or standardization (`standardization()`).

EXAMPLES:

```
sage: [x.standard_form() for x in SetPartitions(4, [2,2])]
[[[1, 2], [3, 4]], [[1, 3], [2, 4]], [[1, 4], [2, 3]]]
```

TESTS:

```
sage: SetPartition([(1, 9, 8), (2, 3, 4, 5, 6, 7)]).standard_form()
[[1, 8, 9], [2, 3, 4, 5, 6, 7]]
```

standardization()

Return the standardization of `self`.

Given a set partition $A = \{A_1, \ldots, A_n\}$ of an ordered set S, the standardization of A is the set partition of $\{1, 2, \ldots, |S|\}$ obtained by replacing the elements of the parts of A by the integers $1, 2, \ldots, |S|$ in such a way that their relative order is preserved (i. e., the smallest element in the whole set partition is replaced by 1, the next-smallest by 2, and so on).

EXAMPLES:

```
sage: SetPartition([[4], [1, 3]]).standardization()
{{1, 2}, {3}}
sage: SetPartition([[4], [6, 3]]).standardization()
{{1, 3}, {2}}
sage: SetPartition([]).standardization()
{}
```

strict_coarsenings()

Return all strict coarsenings of `self`.

Strict coarsening is the binary relation on set partitions defined as the transitive-and-reflexive closure of the relation $\prec$ defined as follows: For two set partitions A and B, we have $A \prec B$ if there exist parts A_i, A_j of A such that $\max(A_i) < \min(A_j)$ and $B = A \setminus \{A_i, A_j\} \cup \{A_i \cup A_j\}$.

EXAMPLES:

```
sage: A = SetPartition([[1],[2,3],[4]])
sage: A.strict_coarsenings()
[{{1}, {2, 3}, {4}}, {{1, 2, 3}, {4}}, {{1, 4}, {2, 3}},
 {{1}, {2, 3, 4}}, {{1, 2, 3, 4}}]
sage: SetPartition([[1],[2,4],[3]]).strict_coarsenings()
[{{1}, {2, 4}, {3}}, {{1, 2, 4}, {3}}, {{1, 3}, {2, 4}}]
sage: SetPartition([]).strict_coarsenings()
[{}]
```

sup(*t*)

Return the supremum of `self` and `t` in the classical set partition lattice.

The supremum of two set partitions B and C is obtained as the transitive closure of the relation which relates i to j if and only if i and j are in the same part in at least one of the set partitions B and C.

See also:

`__mul__()`

EXAMPLES:

```
sage: S = SetPartitions(4)
sage: sp1 = S([[2,3,4], [1]])
sage: sp2 = S([[1,3], [2,4]])
sage: s = S([[1,2,3,4]])
sage: sp1.sup(sp2) == s
True
```

to_partition()

Return the integer partition whose parts are the sizes of the sets in `self`.

EXAMPLES:

```
sage: S = SetPartitions(5)
sage: x = S([[1,2], [3,5,4]])
sage: x.shape()
[3, 2]
sage: y = S([[2], [3,1], [5,4]])
sage: y.shape()
[2, 2, 1]
```

to_permutation()

Convert a set partition of $\{1, ..., n\}$ to a permutation by considering the blocks of the partition as cycles.

The cycles are such that the number of excedences is maximised, that is, each cycle is of the form $(a_1, a_2, ..., a_k)$ with $a_1 < a_2 < ... < a_k$.

EXAMPLES:

```
sage: s = SetPartition([[1,3],[2,4]])
sage: s.to_permutation()
[3, 4, 1, 2]
```

class sage.combinat.set_partition.**SetPartitions**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

An (unordered) partition of a set S is a set of pairwise disjoint nonempty subsets with union S, and is represented by a sorted list of such subsets.

SetPartitions(s) returns the class of all set partitions of the set s, which can be given as a set or a string; if a string, each character is considered an element.

SetPartitions(n), where n is an integer, returns the class of all set partitions of the set $\{1, 2, \ldots, n\}$.

You may specify a second argument k. If k is an integer, SetPartitions returns the class of set partitions into k parts; if it is an integer partition, SetPartitions returns the class of set partitions whose block sizes correspond to that integer partition.

The Bell number B_n, named in honor of Eric Temple Bell, is the number of different partitions of a set with n elements.

EXAMPLES:

```
sage: S = [1,2,3,4]
sage: SetPartitions(S,2)
Set partitions of {1, 2, 3, 4} with 2 parts
sage: SetPartitions([1,2,3,4], [3,1]).list()
[{{1}, {2, 3, 4}}, {{1, 3, 4}, {2}}, {{1, 2, 4}, {3}}, {{1, 2, 3}, {4}}]
sage: SetPartitions(7, [3,3,1]).cardinality()
70
```

In strings, repeated letters are not considered distinct as of trac ticket #14140:

```
sage: SetPartitions('abcde').cardinality()
52
sage: SetPartitions('aabcd').cardinality()
15
```

REFERENCES:

- Wikipedia article Partition_of_a_set

Element

alias of SetPartition

is_less_than(*s*, *t*)

Check if $s < t$ in the refinement ordering on set partitions.

This means that s is a refinement of t and satisfies $s \neq t$.

A set partition s is said to be a refinement of a set partition t of the same set if and only if each part of s is a subset of a part of t.

EXAMPLES:

```
sage: S = SetPartitions(4)
sage: s = S([[1,3],[2,4]])
sage: t = S([[1],[2],[3],[4]])
sage: S.is_less_than(t, s)
True
sage: S.is_less_than(s, t)
False
sage: S.is_less_than(s, s)
False
```

is_strict_refinement(*s*, *t*)

Return `True` if `s` is a strict refinement of `t` and satisfies $s \neq t$.

A set partition s is said to be a strict refinement of a set partition t of the same set if and only if one can obtain t from s by repeatedly combining pairs of parts whose convex hulls don't intersect (i. e., whenever we are combining two parts, the maximum of each of them should be smaller than the minimum of the other).

EXAMPLES:

```
sage: S = SetPartitions(4)
sage: s = S([[1],[2],[3],[4]])
sage: t = S([[1,3],[2,4]])
sage: u = S([[1,2,3,4]])
sage: S.is_strict_refinement(s, t)
True
sage: S.is_strict_refinement(t, u)
False
sage: A = SetPartition([[1,3],[2,4]])
sage: B = SetPartition([[1,2,3,4]])
sage: S.is_strict_refinement(s, A)
True
sage: S.is_strict_refinement(t, B)
False
```

lt(*s*, *t*)

Check if $s < t$ in the refinement ordering on set partitions.

This means that s is a refinement of t and satisfies $s \neq t$.

A set partition s is said to be a refinement of a set partition t of the same set if and only if each part of s is a subset of a part of t.

EXAMPLES:

```
sage: S = SetPartitions(4)
sage: s = S([[1,3],[2,4]])
sage: t = S([[1],[2],[3],[4]])
sage: S.is_less_than(t, s)
True
sage: S.is_less_than(s, t)
False
```

```
sage: S.is_less_than(s, s)
False
```

class sage.combinat.set_partition.**SetPartitions_all**
Bases: sage.combinat.set_partition.SetPartitions

All set partitions.

class sage.combinat.set_partition.**SetPartitions_set**(*s*)
Bases: sage.combinat.set_partition.SetPartitions

Set partitions of a fixed set S.

base_set()
Return the base set of `self`.

EXAMPLES:
```
sage: SetPartitions(3).base_set()
{1, 2, 3}
```

base_set_cardinality()
Return the cardinality of the base set of `self`.

EXAMPLES:
```
sage: SetPartitions(3).base_set_cardinality()
3
```

cardinality()
Return the number of set partitions of the set S.

The cardinality is given by the n-th Bell number where n is the number of elements in the set S.

EXAMPLES:
```
sage: SetPartitions([1,2,3,4]).cardinality()
15
sage: SetPartitions(3).cardinality()
5
sage: SetPartitions(3,2).cardinality()
3
sage: SetPartitions([]).cardinality()
1
```

class sage.combinat.set_partition.**SetPartitions_setn**(*s*, *n*)
Bases: sage.combinat.set_partition.SetPartitions_set

TESTS:
```
sage: S = SetPartitions(5, 3)
sage: TestSuite(S).run()
```

cardinality()
The Stirling number of the second kind is the number of partitions of a set of size n into k blocks.

EXAMPLES:
```
sage: SetPartitions(5, 3).cardinality()
25
sage: stirling_number2(5,3)
25
```

class sage.combinat.set_partition.**SetPartitions_setparts**(*s*, *parts*)
Bases: sage.combinat.set_partition.SetPartitions_set

Class of all set partitions with fixed partition sizes corresponding to an integer partition λ.

cardinality()
Return the cardinality of self.

This algorithm counts for each block of the partition the number of ways to fill it using values from the set. Then, for each distinct value v of block size, we divide the result by the number of ways to arrange the blocks of size v in the set partition.

For example, if we want to count the number of set partitions of size 13 having [3,3,3,2,2] as underlying partition we compute the number of ways to fill each block of the partition, which is $\binom{13}{3}\binom{10}{3}\binom{7}{3}\binom{4}{2}\binom{2}{2}$ and as we have three blocks of size 3 and two blocks of size 2, we divide the result by 3!2! which gives us 600600.

EXAMPLES:

```
sage: SetPartitions(3, [2,1]).cardinality()
3
sage: SetPartitions(13, Partition([3,3,3,2,2])).cardinality()
600600
```

TESTS:

```
sage: all((len(SetPartitions(size, part)) == SetPartitions(size, part).cardinality() for siz
True
sage: sum((SetPartitions(13, p).cardinality() for p in Partitions(13))) == SetPartitions(13)
True
```

sage.combinat.set_partition.**cyclic_permutations_of_set_partition**(*set_part*)
Returns all combinations of cyclic permutations of each cell of the set partition.

AUTHORS:

- Robert L. Miller

EXAMPLES:

```
sage: from sage.combinat.set_partition import cyclic_permutations_of_set_partition
sage: cyclic_permutations_of_set_partition([[1,2,3,4],[5,6,7]])
[[[1, 2, 3, 4], [5, 6, 7]],
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]
```

sage.combinat.set_partition.**cyclic_permutations_of_set_partition_iterator**(*set_part*)
Iterates over all combinations of cyclic permutations of each cell of the set partition.

AUTHORS:

- Robert L. Miller

EXAMPLES:

```
sage: from sage.combinat.set_partition import cyclic_permutations_of_set_partition_iterator
sage: list(cyclic_permutations_of_set_partition_iterator([[1,2,3,4],[5,6,7]]))
[[[1, 2, 3, 4], [5, 6, 7]],
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]
```

5.1.244 Ordered Set Partitions

AUTHORS:

- Mike Hansen
- MuPAD-Combinat developers (for algorithms and design inspiration)
- Travis Scrimshaw (2013-02-28): Removed `CombinatorialClass` and added entry point through `OrderedSetPartition`.

class sage.combinat.set_partition_ordered.**OrderedSetPartition**(*parent*, *s*)
Bases: sage.structure.list_clone.ClonableArray

An ordered partition of a set.

An ordered set partition p of a set s is a list of pairwise disjoint nonempty subsets of s such that the union of these subsets is s. These subsets are called the parts of the partition. We represent an ordered set partition as a list of sets. By extension, an ordered set partition of a nonnegative integer n is the set partition of the integers from 1 to n. The number of ordered set partitions of n is called the n-th ordered Bell number.

There is a natural integer composition associated with an ordered set partition, that is the sequence of sizes of all its parts in order.

The number T_n of ordered set partitions of $\{1, 2, ..., n\}$ is the so-called n-th *Fubini number* (also known as the n-th ordered Bell number; see Wikipedia article Ordered Bell number). Its exponential generating function is

$$\sum_n \frac{T_n}{n!} x^n = \frac{1}{2 - e^x}.$$

(See sequence A000670 in OEIS.)

EXAMPLES:

There are 13 ordered set partitions of $\{1, 2, 3\}$:

```
sage: OrderedSetPartitions(3).cardinality()
13
```

Here is the list of them:

```
sage: OrderedSetPartitions(3).list()
[[{1}, {2}, {3}],
 [{1}, {3}, {2}],
 [{2}, {1}, {3}],
```

```
 [{3}, {1}, {2}],
 [{2}, {3}, {1}],
 [{3}, {2}, {1}],
 [{1}, {2, 3}],
 [{2}, {1, 3}],
 [{3}, {1, 2}],
 [{1, 2}, {3}],
 [{1, 3}, {2}],
 [{2, 3}, {1}],
 [{1, 2, 3}]]
```

There are 12 ordered set partitions of $\{1, 2, 3, 4\}$ whose underlying composition is $[1, 2, 1]$:

```
sage: OrderedSetPartitions(4,[1,2,1]).list()
[[{1}, {2, 3}, {4}],
 [{1}, {2, 4}, {3}],
 [{1}, {3, 4}, {2}],
 [{2}, {1, 3}, {4}],
 [{2}, {1, 4}, {3}],
 [{3}, {1, 2}, {4}],
 [{4}, {1, 2}, {3}],
 [{3}, {1, 4}, {2}],
 [{4}, {1, 3}, {2}],
 [{2}, {3, 4}, {1}],
 [{3}, {2, 4}, {1}],
 [{4}, {2, 3}, {1}]]
```

Since trac ticket #14140, we can create an ordered set partition directly by `OrderedSetPartition` which creates the parent object by taking the union of the partitions passed in. However it is recommended and (marginally) faster to create the parent first and then create the ordered set partition from that.

```
sage: s = OrderedSetPartition([[1,3],[2,4]]); s
[{1, 3}, {2, 4}]
sage: s.parent()
Ordered set partitions of {1, 2, 3, 4}
```

REFERENCES:

Wikipedia article Ordered_partition_of_a_set

check()

Check that we are a valid ordered set partition.

EXAMPLES:

```
sage: OS = OrderedSetPartitions(4)
sage: s = OS([[1, 3], [2, 4]])
sage: s.check()
```

to_composition()

Return the integer composition whose parts are the sizes of the sets in `self`.

EXAMPLES:

```
sage: S = OrderedSetPartitions(5)
sage: x = S([[3,5,4], [1, 2]])
sage: x.to_composition()
[3, 2]
sage: y = S([[3,1], [2], [5,4]])
sage: y.to_composition()
[2, 1, 2]
```

class sage.combinat.set_partition_ordered.**OrderedSetPartitions**(*s*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Return the combinatorial class of ordered set partitions of `s`.

EXAMPLES:

```
sage: OS = OrderedSetPartitions([1,2,3,4]); OS
Ordered set partitions of {1, 2, 3, 4}
sage: OS.cardinality()
75
sage: OS.first()
[{1}, {2}, {3}, {4}]
sage: OS.last()
[{1, 2, 3, 4}]
sage: OS.random_element()
[{3}, {1}, {2}, {4}]

sage: OS = OrderedSetPartitions([1,2,3,4], [2,2]); OS
Ordered set partitions of {1, 2, 3, 4} into parts of size [2, 2]
sage: OS.cardinality()
6
sage: OS.first()
[{1, 2}, {3, 4}]
sage: OS.last()
[{3, 4}, {1, 2}]
sage: OS.list()
[[{1, 2}, {3, 4}],
 [{1, 3}, {2, 4}],
 [{1, 4}, {2, 3}],
 [{2, 3}, {1, 4}],
 [{2, 4}, {1, 3}],
 [{3, 4}, {1, 2}]]

sage: OS = OrderedSetPartitions("cat"); OS
Ordered set partitions of {'a', 'c', 't'}
sage: OS.list()
[[{'a'}, {'c'}, {'t'}],
 [{'a'}, {'t'}, {'c'}],
 [{'c'}, {'a'}, {'t'}],
 [{'t'}, {'a'}, {'c'}],
 [{'c'}, {'t'}, {'a'}],
 [{'t'}, {'c'}, {'a'}],
 [{'a'}, {'c', 't'}],
 [{'c'}, {'a', 't'}],
 [{'t'}, {'a', 'c'}],
 [{'a', 'c'}, {'t'}],
 [{'a', 't'}, {'c'}],
 [{'c', 't'}, {'a'}],
 [{'a', 'c', 't'}]]
```

Element

alias of OrderedSetPartition

class sage.combinat.set_partition_ordered.**OrderedSetPartitions_s**(*s*)

Bases: sage.combinat.set_partition_ordered.OrderedSetPartitions

Class of ordered partitions of a set S.

cardinality()

EXAMPLES:

```
sage: OrderedSetPartitions(0).cardinality()
1
sage: OrderedSetPartitions(1).cardinality()
1
sage: OrderedSetPartitions(2).cardinality()
3
sage: OrderedSetPartitions(3).cardinality()
13
sage: OrderedSetPartitions([1,2,3]).cardinality()
13
sage: OrderedSetPartitions(4).cardinality()
75
sage: OrderedSetPartitions(5).cardinality()
541
```

class sage.combinat.set_partition_ordered.**OrderedSetPartitions_scomp**(s, *comp*)

Bases: sage.combinat.set_partition_ordered.OrderedSetPartitions

TESTS:

```
sage: OS = OrderedSetPartitions([1,2,3,4], [2,1,1])
sage: OS == loads(dumps(OS))
True
```

cardinality()

Return the cardinality of `self`.

The number of ordered set partitions of a set of length k with composition shape μ is equal to

$$\frac{k!}{\prod_{\mu_i \neq 0} \mu_i!}.$$

EXAMPLES:

```
sage: OrderedSetPartitions(5,[2,3]).cardinality()
10
sage: OrderedSetPartitions(0, []).cardinality()
1
sage: OrderedSetPartitions(0, [0]).cardinality()
1
sage: OrderedSetPartitions(0, [0,0]).cardinality()
1
sage: OrderedSetPartitions(5, [2,0,3]).cardinality()
10
```

class sage.combinat.set_partition_ordered.**OrderedSetPartitions_sn**(s, n)

Bases: sage.combinat.set_partition_ordered.OrderedSetPartitions

TESTS:

```
sage: OS = OrderedSetPartitions([1,2,3,4], 2)
sage: OS == loads(dumps(OS))
True
```

cardinality()

Return the cardinality of `self`.

The number of ordered partitions of a set of size n into k parts is equal to $k!S(n,k)$ where $S(n,k)$ denotes the Stirling number of the second kind.

EXAMPLES:

```
sage: OrderedSetPartitions(4,2).cardinality()
14
sage: OrderedSetPartitions(4,1).cardinality()
1
```

class sage.combinat.set_partition_ordered.**SplitNK**(*s*, *comp*)

Bases: sage.combinat.set_partition_ordered.OrderedSetPartitions_scomp

TESTS:

```
sage: OS = OrderedSetPartitions([1,2,3,4], [2,1,1])
sage: OS == loads(dumps(OS))
True
```

5.1.245 Symmetric Functions

- Introduction to Symmetric Functions
- *Symmetric Functions*
- *Symmetric functions, with their multiple realizations*
- *Classical symmetric functions.*
- *Schur symmetric functions*
- *Monomial symmetric functions*
- *Multiplicative symmetric functions*
- *Elementary symmetric functions*
- *Homogeneous symmetric functions*
- *Power sum symmetric functions*
- *Characters of the symmetric group as bases of the symmetric functions*
- *Orthogonal Symmetric Functions*
- *Symplectic Symmetric Functions*
- *Generic dual bases symmetric functions*
- *Symmetric functions defined by orthogonality and triangularity.*
- *Kostka-Foulkes Polynomials*
- *Hall-Littlewood Polynomials*
- *Jack Symmetric Functions*
- *k-Schur Functions*
- *Quotient of symmetric function space by ideal generated by Hall-Littlewood symmetric functions*
- *LLT symmetric functions*
- *Macdonald Polynomials*

- *Non-symmetric Macdonald Polynomials*
- *Witt symmetric functions*

5.1.246 Symmetric function features that are imported by default in the interpreter namespace

5.1.247 Characters of the symmetric group as bases of the symmetric functions

Just as the Schur functions are the irreducible characters of Gl_n and form a basis of the symmetric functions, the irreducible symmetric group character basis are the irreducible characters of of S_n when the group is realized as the permutation matrices.

REFERENCES:

class sage.combinat.sf.character.**character_basis**(*Sym*, *other_basis*, *bname*, *pfix*)

Bases: sage.combinat.sf.character.generic_character

General code for a character basis (irreducible and induced trivial).

This is a basis of the symmetric functions that has the property that `self(la).character_to_frobenius_image(n)` is equal to `other([n-sum(la)]+la)`.

It should also have the property that the (outer) structure constants are the analogue of the stable Kronecker coefficients on the `other` basis (where `other` is either the Schur or homogeneous bases).

These bases are introduced in [OZ2015].

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: h = Sym.h()
sage: ht = SymmetricFunctions(QQ).ht()
sage: st = SymmetricFunctions(QQ).st()
sage: ht(s[2,1])
ht[1, 1] + ht[2, 1] - ht[3]
sage: s(ht[2,1])
s[1] - 2*s[1, 1] - 2*s[2] + s[2, 1] + s[3]
sage: ht(h[2,1])
ht[1] + 2*ht[1, 1] + ht[2, 1]
sage: h(ht[2,1])
h[1] - 2*h[1, 1] + h[2, 1]
sage: st(ht[2,1])
st[] + 2*st[1] + st[1, 1] + 2*st[2] + st[2, 1] + st[3]
sage: ht(st[2,1])
ht[1] - ht[1, 1] + ht[2, 1] - ht[3]
sage: ht[2]*ht[1,1]
ht[1, 1] + 2*ht[1, 1, 1] + ht[2, 1, 1]
sage: h[4,2].kronecker_product(h[4,1,1])
h[2, 2, 1, 1] + 2*h[3, 1, 1, 1] + h[4, 1, 1]
sage: s(st[2,1])
3*s[1] - 2*s[1, 1] - 2*s[2] + s[2, 1]
sage: st(s[2,1])
st[] + 3*st[1] + 2*st[1, 1] + 2*st[2] + st[2, 1]
sage: st[2]*st[1]
st[1] + st[1, 1] + st[2] + st[2, 1] + st[3]
sage: s[4,2].kronecker_product(s[5,1])
s[3, 2, 1] + s[3, 3] + s[4, 1, 1] + s[4, 2] + s[5, 1]
```

class sage.combinat.sf.character.**generic_character**(*Sym*, *basis_name=None*, *prefix=None*, *graded=True*)

Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic

Initializes the symmetric function algebra.

INPUT:

- Sym – the ring of symmetric functions
- basis_name – name of basis (default: None)
- prefix – prefix used to display basis
- graded – (default: True) if True, then the basis is considered to be graded, otherwise the basis is filtered

TESTS:

```
sage: from sage.combinat.sf.classical import SymmetricFunctionAlgebra_classical
sage: s = SymmetricFunctions(QQ).s()
sage: isinstance(s, SymmetricFunctionAlgebra_classical)
True
sage: TestSuite(s).run()
```

class sage.combinat.sf.character.**irreducible_character_basis**(*Sym*, *pfix*)

Bases: sage.combinat.sf.character.generic_character

The irreducible symmetric group character basis of the symmetric functions.

This is a basis of the symmetric functions that has the property that self(la).character_to_frobenius_image(n) is equal to s([n-sum(la)]+la).

It should also have the property that the (outer) structure constants are the analogue of the stable kronecker coefficients on the Schur basis (where other is either the Schur or homogeneous bases).

This basis is introduced in [OZ2015].

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: h = Sym.h()
sage: ht = SymmetricFunctions(QQ).ht()
sage: st = SymmetricFunctions(QQ).st()
sage: st(ht[2,1])
st[] + 2*st[1] + st[1, 1] + 2*st[2] + st[2, 1] + st[3]
sage: ht(st[2,1])
ht[1] - ht[1, 1] + ht[2, 1] - ht[3]
sage: s(st[2,1])
3*s[1] - 2*s[1, 1] - 2*s[2] + s[2, 1]
sage: st(s[2,1])
st[] + 3*st[1] + 2*st[1, 1] + 2*st[2] + st[2, 1]
sage: st[2]*st[1]
st[1] + st[1, 1] + st[2] + st[2, 1] + st[3]
sage: s[4,2].kronecker_product(s[5,1])
s[3, 2, 1] + s[3, 3] + s[4, 1, 1] + s[4, 2] + s[5, 1]
sage: st[1,1,1].counit()
-1
sage: all(sum(c*st(la)*st(mu).antipode() for
....:     ((la,mu),c) in st(ga).coproduct())==st(st(ga).counit())
....:     for ga in Partitions(3))
True
```

TESTS:

```
sage: TestSuite(st).run()
```

5.1.248 Classical symmetric functions.

class sage.combinat.sf.classical.**SymmetricFunctionAlgebra_classical**(*Sym*, *basis_name=None*, *prefix=None*, *graded=True*)

Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic

The class of classical symmetric functions.

Todo

delete this class once all coercions will be handled by Sage's coercion model

---

TESTS:

```
sage: TestSuite(SymmetricFunctions(QQ).s()).run()
sage: TestSuite(SymmetricFunctions(QQ).h()).run()
sage: TestSuite(SymmetricFunctions(QQ).m()).run()
sage: TestSuite(SymmetricFunctions(QQ).e()).run()
sage: TestSuite(SymmetricFunctions(QQ).p()).run()
```

class Element(*M*, *x*)

Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic_Element

A symmetric function.

sage.combinat.sf.classical.**init**()

Set up the conversion functions between the classical bases.

EXAMPLES:

```
sage: from sage.combinat.sf.classical import init
sage: sage.combinat.sf.classical.conversion_functions = {}
sage: init()
sage: sage.combinat.sf.classical.conversion_functions[('Schur', 'powersum')]
<built-in function t_SCHUR_POWSYM_symmetrica>
```

The following checks if the bug described in trac ticket #15312 is fixed.:

```
sage: change = sage.combinat.sf.classical.conversion_functions[('powersum', 'Schur')]
sage: hideme = change({Partition([1]*47):ZZ(1)}) # long time
sage: change({Partition([2,2]):QQ(1)})
s[1, 1, 1, 1] - s[2, 1, 1] + 2*s[2, 2] - s[3, 1] + s[4]
```

5.1.249 Generic dual bases symmetric functions

class sage.combinat.sf.dual.**SymmetricFunctionAlgebra_dual**(*dual_basis*, *scalar*, *scalar_name=''*, *basis_name=None*, *prefix=None*)

Bases: sage.combinat.sf.classical.SymmetricFunctionAlgebra_classical

Generic dual basis of a basis of symmetric functions.

INPUT:

- `dual_basis` – a basis of the ring of symmetric functions
- `scalar` – A function z on partitions which determines the scalar product on the power sum basis by $\langle p_\mu, p_\mu \rangle = z(\mu)$. (Independently on the function chosen, the power sum basis will always be orthogonal; the function `scalar` only determines the norms of the basis elements.) This defaults to the function `zee` defined in `sage.combinat.sf.sfa`, that is, the function is defined by:

$$\lambda \mapsto \prod_{i=1}^{\infty} m_i(\lambda)! i^{m_i(\lambda)},$$

  where $m_i(\lambda)$ means the number of times i appears in λ. This default function gives the standard Hall scalar product on the ring of symmetric functions.
- `scalar_name` – (default: the empty string) a string giving a description of the scalar product specified by the parameter `scalar`
- `basis_name` – (optional) a string to serve as name for the basis to be generated (such as "forgotten" in "the forgotten basis"); don't set it to any of the already existing basis names (such as `homogeneous`, `monomial`, `forgotten`, etc.).
- `prefix` – (default: `'d'` and the prefix for `dual_basis`) a string to use as the symbol for the basis

OUTPUT:

The basis of the ring of symmetric functions dual to the basis `dual_basis` with respect to the scalar product determined by `scalar`.

EXAMPLES:

```
sage: e = SymmetricFunctions(QQ).e()
sage: f = e.dual_basis(prefix = "m", basis_name="Forgotten symmetric functions"); f
Symmetric Functions over Rational Field in the Forgotten symmetric functions basis
sage: TestSuite(f).run(elements = [f[1,1]+2*f[2], f[1]+3*f[1,1]])
sage: TestSuite(f).run() # long time (11s on sage.math, 2011)
```

This class defines canonical coercions between `self` and `self^*`, as follow:

Lookup for the canonical isomorphism from `self` to P (=powersum), and build the adjoint isomorphism from P^* to `self^*`. Since P is self-adjoint for this scalar product, derive an isomorphism from P to `self^*`, and by composition with the above get an isomorphism from `self` to `self^*` (and similarly for the isomorphism `self^*` to `self`).

This should be striped down to just (auto?) defining canonical isomorphism by adjunction (as in MuPAD-Combinat), and let the coercion handle the rest.

Inversions may not be possible if the base ring is not a field:

```
sage: m = SymmetricFunctions(ZZ).m()
sage: h = m.dual_basis(lambda x: 1)
sage: h[2,1]
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

By transitivity, this defines indirect coercions to and from all other bases:

```
sage: s = SymmetricFunctions(QQ['t'].fraction_field()).s()
sage: t = QQ['t'].fraction_field().gen()
sage: zee_hl = lambda x: x.centralizer_size(t=t)
```

```
sage: S = s.dual_basis(zee_hl)
sage: S(s([2,1]))
(-t/(t^5-2*t^4+t^3-t^2+2*t-1))*d_s[1, 1, 1] + ((-t^2-1)/(t^5-2*t^4+t^3-t^2+2*t-1))*d_s[2, 1] + (
```

TESTS:

Regression test for trac ticket #12489. This ticket improving equality test revealed that the conversion back from the dual basis did not strip cancelled terms from the dictionary:

```
sage: y = e[1, 1, 1, 1] - 2*e[2, 1, 1] + e[2, 2]
sage: sorted(f.element_class(f, dual = y))
[([1, 1, 1, 1], 6), ([2, 1, 1], 2), ([2, 2], 1)]
```

class **Element** (*A*, *dictionary=None*, *dual=None*)

Bases: sage.combinat.sf.classical.SymmetricFunctionAlgebra_classical.Element

An element in the dual basis.

INPUT:

At least one of the following must be specified. The one (if any) which is not provided will be computed.

- `dictionary` – an internal dictionary for the monomials and coefficients of `self`
- `dual` – self as an element of the dual basis.

dual ()

Return `self` in the dual basis.

OUTPUT:

- the element `self` expanded in the dual basis to `self.parent()`

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).monomial()
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(scalar=zee)
sage: a = h([2,1])
sage: a.parent()
Dual basis to Symmetric Functions over Rational Field in the monomial basis
sage: a.dual()
3*m[1, 1, 1] + 2*m[2, 1] + m[3]
```

expand (*n*, *alphabet='x'*)

Expand the symmetric function `self` as a symmetric polynomial in `n` variables.

INPUT:

- `n` – a nonnegative integer
- `alphabet` – (default: `'x'`) a variable for the expansion

OUTPUT:

A monomial expansion of `self` in the n variables labelled by `alphabet`.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).monomial()
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(zee)
sage: a = h([2,1])+h([3])
sage: a.expand(2)
2*x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + 2*x1^3
sage: a.dual().expand(2)
2*x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + 2*x1^3
sage: a.expand(2,alphabet='y')
```

```
2*y0^3 + 3*y0^2*y1 + 3*y0*y1^2 + 2*y1^3
sage: a.expand(2,alphabet='x,y')
2*x^3 + 3*x^2*y + 3*x*y^2 + 2*y^3
sage: h([1]).expand(0)
0
sage: (3*h([])).expand(0)
3
```

omega()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1}p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda|-\ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the :meth‘omega()‘ method.

OUTPUT:

- the result of applying omega to `self`

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).monomial()
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(zee)
sage: hh = SymmetricFunctions(QQ).homogeneous()
sage: hh([2,1]).omega()
h[1, 1, 1] - h[2, 1]
sage: h([2,1]).omega()
d_m[1, 1, 1] - d_m[2, 1]
```

omega_involution()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1}p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda|-\ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the :meth‘omega()‘ method.

OUTPUT:

•the result of applying omega to `self`

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).monomial()
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(zee)
sage: hh = SymmetricFunctions(QQ).homogeneous()
sage: hh([2,1]).omega()
h[1, 1, 1] - h[2, 1]
sage: h([2,1]).omega()
d_m[1, 1, 1] - d_m[2, 1]
```

scalar(*x*)

Return the standard scalar product of `self` and `x`.

INPUT:

•`x` – element of the symmetric functions

OUTPUT:

•the scalar product between `x` and `self`

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).monomial()
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(scalar=zee)
sage: a = h([2,1])
sage: a.scalar(a)
2
```

scalar_hl(*x*)

Return the Hall-Littlewood scalar product of `self` and `x`.

INPUT:

•`x` – element of the same dual basis as `self`

OUTPUT:

•the Hall-Littlewood scalar product between `x` and `self`

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).monomial()
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(scalar=zee)
sage: a = h([2,1])
sage: a.scalar_hl(a)
(t + 2)/(-t^4 + 2*t^3 - 2*t + 1)
```

SymmetricFunctionAlgebra_dual.**transition_matrix**(*basis*, *n*)

Returns the transition matrix between the n^{th} homogeneous components of `self` and `basis`.

INPUT:

•`basis` – a target basis of the ring of symmetric functions

•`n` – nonnegative integer

OUTPUT:

•A transition matrix from `self` to `basis` for the elements of degree `n`. The indexing order of the rows and columns is the order of `Partitions(n)`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: e = Sym.elementary()
sage: f = e.dual_basis()
```

```
sage: f.transition_matrix(s, 5)
[ 1 -1  0  1  0 -1  1]
[-2  1  1 -1 -1  1  0]
[-2  2 -1 -1  1  0  0]
[ 3 -1 -1  1  0  0  0]
[ 3 -2  1  0  0  0  0]
[-4  1  0  0  0  0  0]
[ 1  0  0  0  0  0  0]
sage: Partitions(5).list()
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]
sage: s(f[2,2,1])
s[3, 2] - 2*s[4, 1] + 3*s[5]
sage: e.transition_matrix(s, 5).inverse().transpose()
[ 1 -1  0  1  0 -1  1]
[-2  1  1 -1 -1  1  0]
[-2  2 -1 -1  1  0  0]
[ 3 -1 -1  1  0  0  0]
[ 3 -2  1  0  0  0  0]
[-4  1  0  0  0  0  0]
[ 1  0  0  0  0  0  0]
```

5.1.250 Elementary symmetric functions

class sage.combinat.sf.elementary.**SymmetricFunctionAlgebra_elementary**(*Sym*)

Bases: sage.combinat.sf.multiplicative.SymmetricFunctionAlgebra_multiplicative

A class for methods for the elementary basis of the symmetric functions.

INPUT:

- self – an elementary basis of the symmetric functions
- Sym – an instance of the ring of symmetric functions

TESTS:

```
sage: e = SymmetricFunctions(QQ).e()
sage: e == loads(dumps(e))
True
sage: TestSuite(e).run(skip=['_test_associativity', '_test_distributivity', '_test_prod'])
sage: TestSuite(e).run(elements = [e[1,1]+e[2], e[1]+2*e[1,1]])
```

class Element(*M*, *x*)

Bases: sage.combinat.sf.classical.SymmetricFunctionAlgebra_classical.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

expand(*n*, *alphabet='x'*)

Expand the symmetric function self as a symmetric polynomial in n variables.

INPUT:

- n – a nonnegative integer
- alphabet – (default: 'x') a variable for the expansion

OUTPUT:

A monomial expansion of self in the n variables labelled by alphabet.

EXAMPLES:

```
sage: e = SymmetricFunctions(QQ).e()
sage: e([2,1]).expand(3)
x0^2*x1 + x0*x1^2 + x0^2*x2 + 3*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
sage: e([1,1,1]).expand(2)
x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + x1^3
sage: e([3]).expand(2)
0
sage: e([2]).expand(3)
x0*x1 + x0*x2 + x1*x2
sage: e([3]).expand(4,alphabet='x,y,z,t')
x*y*z + x*y*t + x*z*t + y*z*t
sage: e([3]).expand(4,alphabet='y')
y0*y1*y2 + y0*y1*y3 + y0*y2*y3 + y1*y2*y3
sage: e([]).expand(2)
1
sage: e([]).expand(0)
1
sage: (3*e([])).expand(0)
3
```

omega()

Return the image of self under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1}p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda|-\ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (length()) of the partition λ, where λ' denotes the conjugate partition (conjugate()) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

omega_involution() is a synonym for the :meth‘omega()‘ method.

EXAMPLES:

```
sage: e = SymmetricFunctions(QQ).e()
sage: a = e([2,1]); a
e[2, 1]
sage: a.omega()
e[1, 1, 1] - e[2, 1]

sage: h = SymmetricFunctions(QQ).h()
sage: h(e([2,1]).omega())
h[2, 1]
```

omega_involution()

Return the image of self under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1}p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda|-\ell(\lambda)}p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the :meth‘omega()‘ method.

EXAMPLES:

```
sage: e = SymmetricFunctions(QQ).e()
sage: a = e([2,1]); a
e[2, 1]
sage: a.omega()
e[1, 1, 1] - e[2, 1]

sage: h = SymmetricFunctions(QQ).h()
sage: h(e([2,1]).omega())
h[2, 1]
```

verschiebung(*n*)

Return the image of the symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the unique algebra endomorphism V of the ring of symmetric functions that satisfies $V(h_r) = h_{r/n}$ for every positive integer r divisible by n, and satisfies $V(h_r) = 0$ for every positive integer r not divisible by n. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every nonnegative integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(h_r) = h_{r/n}, \quad \mathbf{V}_n(p_r) = np_{r/n}, \quad \mathbf{V}_n(e_r) = (-1)^{r-r/n}e_{r/n}$$

(where h is the complete homogeneous basis, p is the powersum basis, and e is the elementary basis). For every nonnegative integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(h_r) = \mathbf{V}_n(p_r) = \mathbf{V}_n(e_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism. Its name derives from the Verschiebung (German for “shift”) endomorphism of the Witt vectors.

The n-th Verschiebung operator is adjoint to the n-th Frobenius operator (see `frobenius()` for its definition) with respect to the Hall scalar product (`scalar()`).

The action of the n-th Verschiebung operator on the Schur basis can also be computed explicitly. The following (probably clumsier than necessary) description can be obtained by solving exercise 7.61 in Stanley [STA].

Let λ be a partition. Let n be a positive integer. If the n-core of λ is nonempty, then $\mathbf{V}_n(s_\lambda) = 0$. Otherwise, the following method computes $\mathbf{V}_n(s_\lambda)$: Write the partition λ in the form $(\lambda_1, \lambda_2, ..., \lambda_{ns})$ for some nonnegative integer s. (If n does not divide the length of λ, then this is achieved by adding trailing zeroes to λ.) Set $\beta_i = \lambda_i + ns - i$ for every $s \in \{1, 2, \ldots, ns\}$. Then, $(\beta_1, \beta_2, ..., \beta_{ns})$ is a strictly decreasing sequence of nonnegative integers. Stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $-1 - \beta_i$ modulo n. Let ξ be the sign of the permutation that is used for this sorting. Let ψ be the sign of the permutation that is used to stably sort the list $(1, 2, \ldots, ns)$

in order of (weakly) increasing remainder of $i-1$ modulo n. (Notice that $\psi = (-1)^{n(n-1)s(s-1)/4}$.) Then, $\mathbf{V}_n(s_\lambda) = \xi\psi \prod_{i=0}^{n-1} s_{\lambda^{(i)}}$, where $(\lambda^{(0)}, \lambda^{(1)}, \ldots, \lambda^{(n-1)})$ is the n-quotient of λ.

INPUT:

- n – a positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of symmetric functions) to self.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: e = Sym.e()
sage: e[3].verschiebung(2)
0
sage: e[4].verschiebung(4)
-e[1]
```

The Verschiebung endomorphisms are multiplicative:

```
sage: all( all( e(lam).verschiebung(2) * e(mu).verschiebung(2)
....:           == (e(lam) * e(mu)).verschiebung(2)
....:           for mu in Partitions(4) )
....:       for lam in Partitions(4) )
True
```

TESTS:

Let us check that this method on the elementary basis gives the same result as the implementation in :module‘sage.combinat.sf.sfa‘ on the complete homogeneous basis:

```
sage: Sym = SymmetricFunctions(QQ)
sage: e = Sym.e(); h = Sym.h()
sage: all( h(e(lam)).verschiebung(3) == h(e(lam).verschiebung(3))
....:       for lam in Partitions(6) )
True
sage: all( e(h(lam)).verschiebung(2) == e(h(lam).verschiebung(2))
....:       for lam in Partitions(4) )
True
```

SymmetricFunctionAlgebra_elementary.**coproduct_on_generators**(i)

Returns the coproduct on self[i].

INPUT:

- self – an elementary basis of the symmetric functions
- i – a nonnegative integer

OUTPUT:

- returns the coproduct on the elementary generator $e(i)$

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: e = Sym.elementary()
sage: e.coproduct_on_generators(2)
e[] # e[2] + e[1] # e[1] + e[2] # e[]
sage: e.coproduct_on_generators(0)
e[] # e[]
```

5.1.251 Hall-Littlewood Polynomials

Notation used in the definitions follows mainly [Mac1995].

REFERENCES:

class sage.combinat.sf.hall_littlewood.**HallLittlewood**(*Sym*, *t='t'*)

Bases: sage.structure.unique_representation.UniqueRepresentation

The family of Hall-Littlewood symmetric function bases.

The Hall-Littlewood symmetric functions are a family of symmetric functions that depend on a parameter t.

INPUT:

By default the parameter for these functions is t, and whatever the parameter is, it must be in the base ring.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).hall_littlewood(1)
Hall-Littlewood polynomials with t=1 over Rational Field
sage: SymmetricFunctions(QQ['t'].fraction_field()).hall_littlewood()
Hall-Littlewood polynomials over Fraction Field of Univariate Polynomial Ring in t over Rational
```

P()

Return the algebra of symmetric functions in the Hall-Littlewood P basis. This is the same as the HL basis in John Stembridge's SF examples file.

INPUT:

- self – a class of Hall-Littlewood symmetric function bases

OUTPUT:

The class of the Hall-Littlewood P basis.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP = Sym.hall_littlewood().P(); HLP
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: SP = Sym.hall_littlewood(t=-1).P(); SP
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: s = Sym.schur()
sage: s(HLP([2,1]))
(-t^2-t)*s[1, 1, 1] + s[2, 1]
```

The Hall-Littlewood polynomials in the P basis at $t = 0$ are the Schur functions:

```
sage: Sym = SymmetricFunctions(QQ)
sage: HLP = Sym.hall_littlewood(t=0).P()
sage: s = Sym.schur()
sage: s(HLP([2,1])) == s([2,1])
True
```

The Hall-Littlewood polynomials in the P basis at $t = 1$ are the monomial symmetric functions:

```
sage: Sym = SymmetricFunctions(QQ)
sage: HLP = Sym.hall_littlewood(t=1).P()
sage: m = Sym.monomial()
sage: m(HLP([2,2,1])) == m([2,2,1])
True
```

We end with some examples of coercions between:

1.Hall-Littlewood P basis.

2.Hall-Littlewood polynomials in the Q basis

3.Hall-Littlewood polynomials in the Q' basis (via the Schurs)

4.Classical symmetric functions

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP  = Sym.hall_littlewood().P()
sage: HLQ  = Sym.hall_littlewood().Q()
sage: HLQp = Sym.hall_littlewood().Qp()
sage: s = Sym.schur()
sage: p = Sym.power()
sage: HLP(HLQ([2])) # indirect doctest
(-t+1)*HLP[2]
sage: HLP(HLQp([2]))
t*HLP[1, 1] + HLP[2]
sage: HLP(s([2]))
t*HLP[1, 1] + HLP[2]
sage: HLP(p([2]))
(t-1)*HLP[1, 1] + HLP[2]
sage: s = HLQp.symmetric_function_ring().s()
sage: HLQp.transition_matrix(s,3)
[      1       0       0]
[      t       1       0]
[    t^3 t^2 + t       1]
sage: s.transition_matrix(HLP,3)
[      1       t     t^3]
[      0       1 t^2 + t]
[      0       0       1]
```

The method sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic_Element.hl_creation_ is a creation operator for the Q basis:

```
sage: HLQp[1].hl_creation_operator([3]).hl_creation_operator([3])
HLQp[3, 3, 1]
```

Transitions between bases with the parameter t specialized:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['y','z']))
sage: (y,z) = Sym.base_ring().gens()
sage: HLy = Sym.hall_littlewood(t=y)
sage: HLz = Sym.hall_littlewood(t=z)
sage: Qpy = HLy.Qp()
sage: Qpz = HLz.Qp()
sage: s = Sym.schur()
sage: s( Qpy[3,1] + z*Qpy[2,2] )
z*s[2, 2] + (y*z+1)*s[3, 1] + (y^2*z+y)*s[4]
sage: s( Qpy[3,1] + y*Qpz[2,2] )
y*s[2, 2] + (y*z+1)*s[3, 1] + (y*z^2+y)*s[4]
sage: s( Qpy[3,1] + y*Qpy[2,2] )
y*s[2, 2] + (y^2+1)*s[3, 1] + (y^3+y)*s[4]

sage: Qy = HLy.Q()
sage: Qz = HLz.Q()
sage: Py = HLy.P()
sage: Pz = HLz.P()
sage: Pz(Qpy[2,1])
(y*z^3+z^2+z)*HLP[1, 1, 1] + (y*z+1)*HLP[2, 1] + y*HLP[3]
sage: Pz(Qz[2,1])
```

```
(z^2-2*z+1)*HLP[2, 1]
sage: Qz(Py[2])
((-y+z)/(z^3-z^2-z+1))*HLQ[1, 1] + (1/(-z+1))*HLQ[2]
sage: Qy(Pz[2])
((y-z)/(y^3-y^2-y+1))*HLQ[1, 1] + (1/(-y+1))*HLQ[2]
sage: Qy.hall_littlewood_family() == HLy
True
sage: Qy.hall_littlewood_family() == HLz
False
sage: Qz.symmetric_function_ring() == Qy.symmetric_function_ring()
True

sage: Sym = SymmetricFunctions(FractionField(QQ['q']))
sage: q = Sym.base_ring().gen()
sage: HL = Sym.hall_littlewood(t=q)
sage: HLQp = HL.Qp()
sage: HLQ = HL.Q()
sage: HLP = HL.P()
sage: s = Sym.schur()
sage: s(HLQp[3,2].plethysm((1-q)*s[1]))/(1-q)^2
(-q^5-q^4)*s[1, 1, 1, 1, 1] + (q^3+q^2)*s[2, 1, 1, 1] - q*s[2, 2, 1] - q*s[3, 1, 1] + s[3, 2
sage: s(HLP[3,2])
(-q^5-q^4)*s[1, 1, 1, 1, 1] + (q^3+q^2)*s[2, 1, 1, 1] - q*s[2, 2, 1] - q*s[3, 1, 1] + s[3, 2
```

The P and Q-Schur at $t = -1$ indexed by strict partitions are a basis for the space algebraically generated by the odd power sum symmetric functions:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q']))
sage: SP = Sym.hall_littlewood(t=-1).P()
sage: SQ = Sym.hall_littlewood(t=-1).Q()
sage: p = Sym.power()
sage: SP(SQ[3,2,1])
8*HLP[3, 2, 1]
sage: SP(SQ[2,2,1])
0
sage: p(SP[3,2,1])
1/45*p[1, 1, 1, 1, 1, 1] - 1/9*p[3, 1, 1, 1] - 1/9*p[3, 3] + 1/5*p[5, 1]
sage: SP(p[3,3])
-4*HLP[3, 2, 1] + 2*HLP[4, 2] - 2*HLP[5, 1] + HLP[6]
sage: SQ( SQ[1]*SQ[3] -2*(1-q)*SQ[4] )
HLQ[3, 1] + 2*q*HLQ[4]
```

TESTS:

```
sage: HLP(s[[]])
HLP[]
sage: HLQ(s[[]])
HLQ[]
sage: HLQp(s[[]])
HLQp[]
```

Q()

Returns the algebra of symmetric functions in Hall-Littlewood Q basis. This is the same as the Q basis in John Stembridge's SF examples file.

More extensive examples can be found in the documentation for the Hall-Littlewood P basis.

INPUT:

- self – a class of Hall-Littlewood symmetric function bases

OUTPUT:

- returns the class of the Hall-Littlewood Q basis

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLQ = Sym.hall_littlewood().Q(); HLQ
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: SQ = SymmetricFunctions(QQ).hall_littlewood(t=-1).Q(); SQ
Symmetric Functions over Rational Field in the Hall-Littlewood Q with t=-1 basis
```

Qp()

Returns the algebra of symmetric functions in Hall-Littlewood Q' (Qp) basis. This is dual to the Hall-Littlewood P basis with respect to the standard scalar product.

More extensive examples can be found in the documentation for the Hall-Littlewood P basis.

INPUT:

- self – a class of Hall-Littlewood symmetric function bases

OUTPUT:

- returns the class of the Hall-Littlewood Qp-basis

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLQp = Sym.hall_littlewood().Qp(); HLQp
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
```

base_ring()

Returns the base ring of the symmetric functions where the Hall-Littlewood symmetric functions live

INPUT:

- self – a class of Hall-Littlewood symmetric function bases

OUTPUT:

The base ring of the symmetric functions.

EXAMPLES

```
sage: HL = SymmetricFunctions(QQ['t'].fraction_field()).hall_littlewood(t=1)
sage: HL.base_ring()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

symmetric_function_ring()

The ring of symmetric functions associated to the class of Hall-Littlewood symmetric functions

INPUT:

- self – a class of Hall-Littlewood symmetric function bases

OUTPUT:

- returns the ring of symmetric functions

EXAMPLES

```
sage: HL = SymmetricFunctions(FractionField(QQ['t'])).hall_littlewood()
sage: HL.symmetric_function_ring()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
```

class sage.combinat.sf.hall_littlewood.**HallLittlewood_generic**(*hall_littlewood*)
Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic

A class with methods for working with Hall-Littlewood symmetric functions which are common to all bases.

INPUT:

- self – a Hall-Littlewood symmetric function basis
- hall_littlewood – a class of Hall-Littlewood bases

TESTS:

```
sage: SymmetricFunctions(QQ['t'].fraction_field()).hall_littlewood().P()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Field i
sage: SymmetricFunctions(QQ).hall_littlewood(t=2).P()
Symmetric Functions over Rational Field in the Hall-Littlewood P with t=2 basis
```

class **Element**(*M*, *x*)
Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic_Element

Methods for elements of a Hall-Littlewood basis that are common to all bases.

expand(*n*, *alphabet='x'*)
Expands the symmetric function as a symmetric polynomial in n variables.

INPUT:
- self – an element of a Hall-Littlewood basis
- n – a positive integer
- alphabet – a string representing a variable name (default: 'x')

OUTPUT:
- returns a symmetric polynomial of self in n variables

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP = Sym.hall_littlewood().P()
sage: HLQ = Sym.hall_littlewood().Q()
sage: HLQp = Sym.hall_littlewood().Qp()
sage: HLP([2]).expand(2)
x0^2 + (-t + 1)*x0*x1 + x1^2
sage: HLQ([2]).expand(2)
(-t + 1)*x0^2 + (t^2 - 2*t + 1)*x0*x1 + (-t + 1)*x1^2
sage: HLQp([2]).expand(2)
x0^2 + x0*x1 + x1^2
sage: HLQp([2]).expand(2, 'y')
y0^2 + y0*y1 + y1^2
sage: HLQp([2]).expand(1)
x^2
```

scalar(*x*, *zee=None*)
Returns standard scalar product between self and x.

This is the default implementation that converts both self and x into Schur functions and performs the scalar product that basis.

The Hall-Littlewood P basis is dual to the Qp basis with respect to this scalar product.

INPUT:
- self – an element of a Hall-Littlewood basis
- x – another symmetric element of the symmetric functions

OUTPUT:

•returns the scalar product between `self` and `x`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP = Sym.hall_littlewood().P()
sage: HLQ = Sym.hall_littlewood().Q()
sage: HLQp = Sym.hall_littlewood().Qp()
sage: HLP([2]).scalar(HLQp([2]))
1
sage: HLP([2]).scalar(HLQp([1,1]))
0
sage: HLP([2]).scalar(HLQ([2]), lambda mu: mu.centralizer_size(t = HLP.t))
1
sage: HLP([2]).scalar(HLQ([1,1]), lambda mu: mu.centralizer_size(t = HLP.t))
0
```

scalar_hl(*x*, *t=None*)

Returns the Hall-Littlewood (with parameter `t`) scalar product of `self` and `x`.

The Hall-Littlewood scalar product is defined in Macdonald's book [Mac1995]. The power sum basis is orthogonal and $\langle p_\mu, p_\mu \rangle = z_\mu \prod_i 1/(1-t^{\mu_i})$

The Hall-Littlewood P basis is dual to the Q basis with respect to this scalar product.

INPUT:

•`self` – an element of a Hall-Littlewood basis

•`x` – another symmetric element of the symmetric functions

•`t` – an optional parameter, if this parameter is not specified then the value of the `t` from the basis is used in the calculation

OUTPUT:

•returns the Hall-Littlewood scalar product between `self` and `x`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP = Sym.hall_littlewood().P()
sage: HLQ = Sym.hall_littlewood().Q()
sage: HLP([2]).scalar_hl(HLQ([2]))
1
sage: HLP([2]).scalar_hl(HLQ([1,1]))
0
sage: HLQ([2]).scalar_hl(HLQ([2]))
-t + 1
sage: HLQ([2]).scalar_hl(HLQ([1,1]))
0
sage: HLP([2]).scalar_hl(HLP([2]))
1/(-t + 1)
```

HallLittlewood_generic.**hall_littlewood_family**()

The family of Hall-Littlewood bases associated to `self`

INPUT:

•`self` – a Hall-Littlewood symmetric function basis

OUTPUT:

•returns the class of Hall-Littlewood bases

EXAMPLES

```
sage: HLP = SymmetricFunctions(FractionField(QQ['t'])).hall_littlewood(1).P()
sage: HLP.hall_littlewood_family()
Hall-Littlewood polynomials with t=1 over Fraction Field of Univariate Polynomial Ring in t
```

HallLittlewood_generic.**transition_matrix**(*basis*, *n*)

Returns the transitions matrix between `self` and `basis` for the homogeneous component of degree `n`.

INPUT:

- `self` – a Hall-Littlewood symmetric function basis
- `basis` – another symmetric function basis
- `n` – a non-negative integer representing the degree

OUTPUT:

- Returns a $r \times r$ matrix of elements of the base ring of `self` where r is the number of partitions of `n`. The entry corresponding to row μ, column ν is the coefficient of `basis` (ν) in `self` (μ)

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP = Sym.hall_littlewood().P()
sage: s   = Sym.schur()
sage: HLP.transition_matrix(s, 4)
[             1             -t              0             t^2           -t^3]
[             0              1             -t              -t      t^3 + t^2]
[             0              0              1              -t             t^3]
[             0              0              0               1 -t^3 - t^2 - t]
[             0              0              0               0              1]
sage: HLQ = Sym.hall_littlewood().Q()
sage: HLQ.transition_matrix(s,3)
[                         -t + 1                        t^2 - t                      -t^3 + t^
[                              0                  t^2 - 2*t + 1           -t^4 + t^3 + t^2 -
[                              0                              0 -t^6 + t^5 + t^4 - t^2 - t +
sage: HLQp = Sym.hall_littlewood().Qp()
sage: HLQp.transition_matrix(s,3)
[      1       0       0]
[      t       1       0]
[    t^3 t^2 + t       1]
```

class sage.combinat.sf.hall_littlewood.**HallLittlewood_p**(*hall_littlewood*)

Bases: sage.combinat.sf.hall_littlewood.HallLittlewood_generic

A class representing the Hall-Littlewood P basis of symmetric functions

class **Element**(*M*, *x*)

Bases: sage.combinat.sf.hall_littlewood.HallLittlewood_generic.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.sf.hall_littlewood.**HallLittlewood_q**(*hall_littlewood*)

Bases: sage.combinat.sf.hall_littlewood.HallLittlewood_generic

The Q basis is defined as a normalization of the P basis.

INPUT:

- `self` – an instance of the Hall-Littlewood P basis
- `hall_littlewood` – a class for the family of Hall-Littlewood bases

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: Q = Sym.hall_littlewood().Q()
sage: TestSuite(Q).run(skip=['_test_associativity', '_test_distributivity', '_test_prod']) # prc
sage: TestSuite(Q).run(elements = [Q.t*Q[1,1]+Q[2], Q[1]+(1+Q.t)*Q[1,1]])  # long time (depends

sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP = Sym.hall_littlewood().P()
sage: HLQ = Sym.hall_littlewood().Q()
sage: HLQp = Sym.hall_littlewood().Qp()
sage: s = Sym.schur(); p = Sym.power()
sage: HLQ( HLP([2,1]) + HLP([3]) )
(1/(t^2-2*t+1))*HLQ[2, 1] + (1/(-t+1))*HLQ[3]
sage: HLQ(HLQp([2])) # indirect doctest
(t/(t^3-t^2-t+1))*HLQ[1, 1] + (1/(-t+1))*HLQ[2]
sage: HLQ(s([2]))
(t/(t^3-t^2-t+1))*HLQ[1, 1] + (1/(-t+1))*HLQ[2]
sage: HLQ(p([2]))
(1/(t^2-1))*HLQ[1, 1] + (1/(-t+1))*HLQ[2]
```

class **Element** (M, x)

Bases: sage.combinat.sf.hall_littlewood.HallLittlewood_generic.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.sf.hall_littlewood.**HallLittlewood_qp** (*hall_littlewood*)

Bases: sage.combinat.sf.hall_littlewood.HallLittlewood_generic

The Hall-Littlewood Qp basis is calculated through the symmetrica library (see the function `HallLittlewood_qp._to_s()`).

INPUT:

- `self` – an instance of the Hall-Littlewood P basis
- `hall_littlewood` – a class for the family of Hall-Littlewood bases

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: Qp = Sym.hall_littlewood().Q()
sage: TestSuite(Qp).run(skip=['_test_passociativity', '_test_distributivity', '_test_prod']) # p
sage: TestSuite(Qp).run(elements = [Qp.t*Qp[1,1]+Qp[2], Qp[1]+(1+Qp.t)*Qp[1,1]])  # long time (c

sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLP = Sym.hall_littlewood().P()
sage: HLQ = Sym.hall_littlewood().Q()
sage: HLQp = Sym.hall_littlewood().Qp()
```

```
sage: s = Sym.schur(); p = Sym.power()
sage: HLQp(HLP([2])) # indirect doctest
-t*HLQp[1, 1] + (t^2+1)*HLQp[2]
sage: HLQp(s(HLQ([2]))) # work around bug reported in ticket #12969
(t^2-t)*HLQp[1, 1] + (-t^3+t^2-t+1)*HLQp[2]
sage: HLQp(s([2]))
HLQp[2]
sage: HLQp(p([2]))
-HLQp[1, 1] + (t+1)*HLQp[2]
sage: s = HLQp.symmetric_function_ring().s()
sage: HLQp.transition_matrix(s,3)
[      1       0       0]
[      t       1       0]
[    t^3 t^2 + t       1]
sage: s.transition_matrix(HLP,3)
[      1       t     t^3]
[      0       1 t^2 + t]
[      0       0       1]
```

class **Element** (M, x)

Bases: sage.combinat.sf.hall_littlewood.HallLittlewood_generic.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

5.1.252 Homogeneous symmetric functions

By this we mean the basis formed of the complete homogeneous symmetric functions h_λ, not an arbitrary graded basis.

class sage.combinat.sf.homogeneous.**SymmetricFunctionAlgebra_homogeneous** (Sym)

Bases: sage.combinat.sf.multiplicative.SymmetricFunctionAlgebra_multiplicative

A class of methods specific to the homogeneous basis of symmetric functions.

INPUT:

- self – a homogeneous basis of symmetric functions
- Sym – an instance of the ring of symmetric functions

TESTS:

```
sage: h = SymmetricFunctions(QQ).e()
sage: h == loads(dumps(h))
True
sage: TestSuite(h).run(skip=['_test_associativity', '_test_distributivity', '_test_prod'])
sage: TestSuite(h).run(elements = [h[1,1]+h[2], h[1]+2*h[1,1]])
```

class **Element** (M, x)

Bases: sage.combinat.sf.classical.SymmetricFunctionAlgebra_classical.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

expand(*n*, *alphabet='x'*)

Expand the symmetric function `self` as a symmetric polynomial in `n` variables.

INPUT:

- `n` – a nonnegative integer
- `alphabet` – (default: `'x'`) a variable for the expansion

OUTPUT:

A monomial expansion of `self` in the n variables labelled by `alphabet`.

EXAMPLES:

```
sage: h = SymmetricFunctions(QQ).h()
sage: h([3]).expand(2)
x0^3 + x0^2*x1 + x0*x1^2 + x1^3
sage: h([1,1,1]).expand(2)
x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + x1^3
sage: h([2,1]).expand(3)
x0^3 + 2*x0^2*x1 + 2*x0*x1^2 + x1^3 + 2*x0^2*x2 + 3*x0*x1*x2 + 2*x1^2*x2 + 2*x0*x2^2 + 2
sage: h([3]).expand(2,alphabet='y')
y0^3 + y0^2*y1 + y0*y1^2 + y1^3
sage: h([3]).expand(2,alphabet='x,y')
x^3 + x^2*y + x*y^2 + y^3
sage: h([3]).expand(3,alphabet='x,y,z')
x^3 + x^2*y + x*y^2 + y^3 + x^2*z + x*y*z + y^2*z + x*z^2 + y*z^2 + z^3
sage: (h([]) + 2*h([1])).expand(3)
2*x0 + 2*x1 + 2*x2 + 1
sage: h([1]).expand(0)
0
sage: (3*h([])).expand(0)
3
```

omega()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1} p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda| - \ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

omega_involution() is a synonym for the omega() method.

OUTPUT:

•the image of self under the omega automorphism

EXAMPLES:

```
sage: h = SymmetricFunctions(QQ).h()
sage: a = h([2,1]); a
h[2, 1]
sage: a.omega()
h[1, 1, 1] - h[2, 1]
sage: e = SymmetricFunctions(QQ).e()
sage: e(h([2,1]).omega())
e[2, 1]
```

omega_involution()

Return the image of self under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1}p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda|-\ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (length()) of the partition λ, where λ' denotes the conjugate partition (conjugate()) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

omega_involution() is a synonym for the omega() method.

OUTPUT:

•the image of self under the omega automorphism

EXAMPLES:

```
sage: h = SymmetricFunctions(QQ).h()
sage: a = h([2,1]); a
h[2, 1]
sage: a.omega()
h[1, 1, 1] - h[2, 1]
sage: e = SymmetricFunctions(QQ).e()
sage: e(h([2,1]).omega())
e[2, 1]
```

SymmetricFunctionAlgebra_homogeneous.**coproduct_on_generators**(i)

Returns the coproduct on h_i.

INPUT:

•self – a homogeneous basis of symmetric functions

•i – a nonnegative integer

OUTPUT:

•the sum $\sum_{r=0}^{i} h_r \otimes h_{i-r}$

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: h = Sym.homogeneous()
sage: h.coproduct_on_generators(2)
h[] # h[2] + h[1] # h[1] + h[2] # h[]
sage: h.coproduct_on_generators(0)
h[] # h[]
```

5.1.253 Jack Symmetric Functions

Jack's symmetric functions appear in [Ma1995] Chapter VI, section 10. Zonal polynomials are the subject of [Ma1995] Chapter VII. The parameter α in that reference is the parameter t in this implementation in sage.

REFERENCES:

class sage.combinat.sf.jack.**Jack**(*Sym*, *t='t'*)

Bases: sage.structure.unique_representation.UniqueRepresentation

The family of Jack symmetric functions including the P, Q, J, Qp bases. The default parameter is t.

INPUT:

- self – the family of Jack symmetric function bases
- Sym – a ring of symmetric functions
- t – an optional parameter (default : 't')

EXAMPLES:

```
sage: SymmetricFunctions(FractionField(QQ['t'])).jack()
Jack polynomials over Fraction Field of Univariate Polynomial Ring in t over Rational Field
sage: SymmetricFunctions(QQ).jack(1)
Jack polynomials with t=1 over Rational Field
```

J()

Returns the algebra of Jack polynomials in the J basis.

INPUT:

- self – the family of Jack symmetric function bases

OUTPUT: the J basis of the Jack symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JJ = Sym.jack().J(); JJ
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: Sym = SymmetricFunctions(QQ)
sage: Sym.jack(t=-1).J()
Symmetric Functions over Rational Field in the Jack J with t=-1 basis
```

At $t = 1$, the Jack polynomials in the J basis are scalar multiples of the Schur functions with the scalar given by a Partition's hook_product() method at 1:

```
sage: Sym = SymmetricFunctions(QQ)
sage: JJ = Sym.jack(t=1).J()
sage: s = Sym.schur()
sage: p = Partition([3,2,1,1])
sage: s(JJ(p)) == p.hook_product(1)*s(p)  # long time (4s on sage.math, 2012)
True
```

At $t = 2$, the Jack polynomials in the J basis are scalar multiples of the zonal polynomials with the scalar given by a Partition's `hook_product()` method at 2.

```
sage: Sym = SymmetricFunctions(QQ)
sage: JJ = Sym.jack(t=2).J()
sage: Z = Sym.zonal()
sage: p = Partition([2,2,1])
sage: Z(JJ(p)) == p.hook_product(2)*Z(p)
True

sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JJ = Sym.jack().J()
sage: JP = Sym.jack().P()
sage: JJ(sum(JP(p) for p in Partitions(3)))
1/6*JackJ[1, 1, 1] + (1/(t+2))*JackJ[2, 1] + (1/(2*t^2+3*t+1))*JackJ[3]

sage: s = Sym.schur()
sage: JJ(s([3])) # indirect doctest
((t^2-3*t+2)/(6*t^2+18*t+12))*JackJ[1, 1, 1] + ((2*t-2)/(2*t^2+5*t+2))*JackJ[2, 1] + (1/(2*t
sage: JJ(s([2,1]))
((t-1)/(3*t+6))*JackJ[1, 1, 1] + (1/(t+2))*JackJ[2, 1]
sage: JJ(s([1,1,1]))
1/6*JackJ[1, 1, 1]
```

P()

Returns the algebra of Jack polynomials in the P basis.

INPUT:

- `self` – the family of Jack symmetric function bases

OUTPUT:

- the P basis of the Jack symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JP = Sym.jack().P(); JP
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: Sym.jack(t=-1).P()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
```

At $t = 1$, the Jack polynomials in the P basis are the Schur symmetric functions.

```
sage: Sym = SymmetricFunctions(QQ)
sage: JP = Sym.jack(t=1).P()
sage: s = Sym.schur()
sage: s(JP([2,2,1]))
s[2, 2, 1]
sage: JP(s([2,2,1]))
JackP[2, 2, 1]
sage: JP([2,1])^2
JackP[2, 2, 1, 1] + JackP[2, 2, 2] + JackP[3, 1, 1, 1] + 2*JackP[3, 2, 1] + JackP[3, 3] + Ja
```

At $t = 2$, the Jack polynomials in the P basis are the zonal polynomials.

```
sage: Sym = SymmetricFunctions(QQ)
sage: JP = Sym.jack(t=2).P()
sage: Z = Sym.zonal()
sage: Z(JP([2,2,1]))
Z[2, 2, 1]
```

```
sage: JP(Z[2, 2, 1])
JackP[2, 2, 1]
sage: JP([2])^2
64/45*JackP[2, 2] + 16/21*JackP[3, 1] + JackP[4]
sage: Z([2])^2
64/45*Z[2, 2] + 16/21*Z[3, 1] + Z[4]

sage: Sym = SymmetricFunctions(QQ['a','b'].fraction_field())
sage: (a,b) = Sym.base_ring().gens()
sage: Jacka = Sym.jack(t=a)
sage: Jackb = Sym.jack(t=b)
sage: m = Sym.monomial()
sage: JPa = Jacka.P()
sage: JPb = Jackb.P()
sage: m(JPa[2,1])
(6/(a+2))*m[1, 1, 1] + m[2, 1]
sage: m(JPb[2,1])
(6/(b+2))*m[1, 1, 1] + m[2, 1]
sage: m(a*JPb([2,1]) + b*JPa([2,1]))
((6*a^2+6*b^2+12*a+12*b)/(a*b+2*a+2*b+4))*m[1, 1, 1] + (a+b)*m[2, 1]
sage: JPa(JPb([2,1]))
((6*a-6*b)/(a*b+2*a+2*b+4))*JackP[1, 1, 1] + JackP[2, 1]

sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JQ = Sym.jack().Q()
sage: JP = Sym.jack().P()
sage: JJ = Sym.jack().J()

sage: JP(JQ([2,1]))
((t+2)/(2*t^3+t^2))*JackP[2, 1]
sage: JP(JQ([3]))
((2*t^2+3*t+1)/(6*t^3))*JackP[3]
sage: JP(JQ([1,1,1]))
(6/(t^3+3*t^2+2*t))*JackP[1, 1, 1]

sage: JP(JJ([3]))
(2*t^2+3*t+1)*JackP[3]
sage: JP(JJ([2,1]))
(t+2)*JackP[2, 1]
sage: JP(JJ([1,1,1]))
6*JackP[1, 1, 1]

sage: s = Sym.schur()
sage: JP(s([2,1]))
((2*t-2)/(t+2))*JackP[1, 1, 1] + JackP[2, 1]
sage: s(_)
s[2, 1]
```

Q()

Returns the algebra of Jack polynomials in the Q basis.

INPUT:

- `self` – the family of Jack symmetric function bases

OUTPUT:

- the Q basis of the Jack symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JQ = Sym.jack().Q(); JQ
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: Sym = SymmetricFunctions(QQ)
sage: Sym.jack(t=-1).Q()
Symmetric Functions over Rational Field in the Jack Q with t=-1 basis

sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JQ = Sym.jack().Q()
sage: JP = Sym.jack().P()
sage: JQ(sum(JP(p) for p in Partitions(3)))
(1/6*t^3+1/2*t^2+1/3*t)*JackQ[1, 1, 1] + ((2*t^3+t^2)/(t+2))*JackQ[2, 1] + (6*t^3/(2*t^2+3*t

sage: s = Sym.schur()
sage: JQ(s([3])) # indirect doctest
(1/6*t^3-1/2*t^2+1/3*t)*JackQ[1, 1, 1] + ((2*t^3-2*t^2)/(t+2))*JackQ[2, 1] + (6*t^3/(2*t^2+3
sage: JQ(s([2,1]))
(1/3*t^3-1/3*t)*JackQ[1, 1, 1] + ((2*t^3+t^2)/(t+2))*JackQ[2, 1]
sage: JQ(s([1,1,1]))
(1/6*t^3+1/2*t^2+1/3*t)*JackQ[1, 1, 1]
```

Qp()

Returns the algebra of Jack polynomials in the Qp, which is dual to the P basis with respect to the standard scalar product.

INPUT:

- `self` – the family of Jack symmetric function bases

OUTPUT:

- the Q' basis of the Jack symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JP = Sym.jack().P()
sage: JQp = Sym.jack().Qp(); JQp
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: a = JQp([2])
sage: a.scalar(JP([2]))
1
sage: a.scalar(JP([1,1]))
0
sage: JP(JQp([2]))                          # todo: missing auto normalization
((t-1)/(t+1))*JackP[1, 1] + JackP[2]
sage: JP._normalize(JP(JQp([2])))
((t-1)/(t+1))*JackP[1, 1] + JackP[2]
```

base_ring()

Returns the base ring of the symmetric functions in which the Jack symmetric functions live

INPUT:

- `self` – the family of Jack symmetric function bases

OUTPUT:

- the base ring of the symmetric functions ring of `self`

EXAMPLES:

```
sage: J2 = SymmetricFunctions(QQ).jack(t=2)
sage: J2.base_ring()
Rational Field
```

symmetric_function_ring()

Returns the base ring of the symmetric functions of the Jack symmetric function bases

INPUT:

- self – the family of Jack symmetric function bases

OUTPUT:

- the symmetric functions ring of self

EXAMPLES:

```
sage: Jacks = SymmetricFunctions(FractionField(QQ['t'])).jack()
sage: Jacks.symmetric_function_ring()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
```

class sage.combinat.sf.jack.**JackPolynomials_generic**(*jack*)

Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic

A class of methods which are common to all Jack bases of the symmetric functions

INPUT:

- self – a Jack basis of the symmetric functions
- jack – a family of Jack symmetric function bases

EXAMPLES

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JP = Sym.jack().P(); JP.base_ring()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
sage: Sym = SymmetricFunctions(QQ)
sage: JP = Sym.jack(t=2).P(); JP.base_ring()
Rational Field
```

class Element(*M*, *x*)

Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic_Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

scalar_jack(*x*, *t=None*)

A scalar product where the power sums are orthogonal and $\langle p_\mu, p_\mu \rangle = z_\mu t^{length(\mu)}$

INPUT:

- self – an element of a Jack basis of the symmetric functions
- x – an element of the symmetric functions
- **t – an optional parameter (default** [None uses the parameter from] the basis)

OUTPUT:

- returns the Jack scalar product between `x` and `self`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JP = Sym.jack().P()
sage: JQ = Sym.jack().Q()
sage: p = Partitions(3).list()
sage: matrix([[JP(a).scalar_jack(JQ(b)) for a in p] for b in p])
[1 0 0]
[0 1 0]
[0 0 1]
```

`JackPolynomials_generic.`**`c1`**(*part*)

Returns the t-Jack scalar product between `J(part)` and `P(part)`.

INPUT:

- `self` – a Jack basis of the symmetric functions
- `part` – a partition
- `t` – an optional parameter (default: uses the parameter t from the Jack basis)

OUTPUT:

- a polynomial in the parameter `t` which is equal to the scalar product of `J(part)` and `P(part)`

EXAMPLES

```
sage: JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
sage: JP.c1(Partition([2,1]))
t + 2
```

`JackPolynomials_generic.`**`c2`**(*part*)

Returns the t-Jack scalar product between `J(part)` and `Q(part)`.

INPUT:

- `self` – a Jack basis of the symmetric functions
- `part` – a partition
- **`t` – an optional parameter (default: uses the parameter t from the** Jack basis)

OUTPUT:

- a polynomial in the parameter `t` which is equal to the scalar product of `J(part)` and `Q(part)`

EXAMPLES:

```
sage: JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
sage: JP.c2(Partition([2,1]))
2*t^3 + t^2
```

`JackPolynomials_generic.`**`coproduct_by_coercion`**(*elt*)

Returns the coproduct of the element `elt` by coercion to the Schur basis.

INPUT:

- `self` – a Jack symmetric function basis
- `elt` – an instance of this basis

OUTPUT:

•The coproduct acting on `elt`, the result is an element of the tensor squared of the Jack symmetric function basis

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'].fraction_field())
sage: Sym.jack().P()[2,2].coproduct() #indirect doctest
JackP[] # JackP[2, 2] + (2/(t+1))*JackP[1] # JackP[2, 1] + ((8*t+4)/(t^3+4*t^2+5*t+2))*JackP
```

`JackPolynomials_generic.`**`jack_family`**`()`

Returns the family of Jack bases associated to the basis `self`

INPUT:

•`self` – a Jack basis of the symmetric functions

OUTPUT:

•the family of Jack symmetric functions associated to `self`

EXAMPLES:

```
sage: JackP = SymmetricFunctions(QQ).jack(t=2).P()
sage: JackP.jack_family()
Jack polynomials with t=2 over Rational Field
```

class `sage.combinat.sf.jack.`**`JackPolynomials_j`**(*jack*)

Bases: `sage.combinat.sf.jack.JackPolynomials_generic`

The J basis is a defined as a normalized form of the P basis

INPUT:

•`self` – an instance of the Jack P basis of the symmetric functions

•`jack` – a family of Jack symmetric function bases

EXAMPLES:

```
sage: J = SymmetricFunctions(FractionField(QQ['t'])).jack().J()
sage: TestSuite(J).run(skip=['_test_associativity', '_test_distributivity', '_test_prod']) # prc
sage: TestSuite(J).run(elements = [J.t*J[1,1]+J[2], J[1]+(1+J.t)*J[1,1]])  # long time (3s on sa
```

class **`Element`**(*M*, *x*)

Bases: `sage.combinat.sf.jack.JackPolynomials_generic.Element`

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class `sage.combinat.sf.jack.`**`JackPolynomials_p`**(*jack*)

Bases: `sage.combinat.sf.jack.JackPolynomials_generic`

The P basis is uni-triangularly related to the monomial basis and orthogonal with respect to the Jack scalar product.

INPUT:

- self – an instance of the Jack P basis of the symmetric functions
- jack – a family of Jack symmetric function bases

EXAMPLES:

```
sage: P = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
sage: TestSuite(P).run(skip=['_test_associativity', '_test_distributivity', '_test_prod']) # pro
sage: TestSuite(P).run(elements = [P.t*P[1,1]+P[2], P[1]+(1+P.t)*P[1,1]])
```

class Element(*M*, *x*)

Bases: sage.combinat.sf.jack.JackPolynomials_generic.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

scalar_jack(*x*, *t=None*)

The scalar product on the symmetric functions where the power sums are orthogonal and $\langle p_\mu, p_\mu \rangle = z_\mu t^{length(mu)}$ where the t parameter from the Jack symmetric function family.

INPUT:

- self – an element of the Jack P basis
- x – an element of the P basis

EXAMPLES

```
sage: JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
sage: l = [JP(p) for p in Partitions(3)]
sage: matrix([[a.scalar_jack(b) for a in l] for b in l])
[  6*t^3/(2*t^2 + 3*t + 1)                         0                           0]
[                       0     (2*t^3 + t^2)/(t + 2)                           0]
[                       0                         0 1/6*t^3 + 1/2*t^2 + 1/3*t]
```

JackPolynomials_p.**scalar_jack_basis**(*part1*, *part2=None*)

Returns the scalar product of $P(part1)$ and $P(part2)$.

This is equation (10.16) of [Mc1995] on page 380.

INPUT:

- self – an instance of the Jack P basis of the symmetric functions
- part1 – a partition
- part2 – an optional partition (default : None)

OUTPUT:

- the scalar product between $P(part1)$ and $P(part2)$ (or itself if $part2$ is None)

REFRENCES:

EXAMPLES:

```
sage: JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
sage: JJ = SymmetricFunctions(FractionField(QQ['t'])).jack().J()
sage: JP.scalar_jack_basis(Partition([2,1]), Partition([1,1,1]))
0
sage: JP._normalize_coefficients(JP.scalar_jack_basis(Partition([3,2,1]), Partition([3,2,1])
(12*t^6 + 20*t^5 + 11*t^4 + 2*t^3)/(2*t^3 + 11*t^2 + 20*t + 12)
sage: JJ(JP[3,2,1]).scalar_jack(JP[3,2,1])
(12*t^6 + 20*t^5 + 11*t^4 + 2*t^3)/(2*t^3 + 11*t^2 + 20*t + 12)
```

With a single argument, takes $part2 = part1$:

```
sage: JP.scalar_jack_basis(Partition([2,1]), Partition([2,1]))
(2*t^3 + t^2)/(t + 2)
sage: JJ(JP[2,1]).scalar_jack(JP[2,1])
(2*t^3 + t^2)/(t + 2)
```

class sage.combinat.sf.jack.**JackPolynomials_q**(*jack*)

Bases: sage.combinat.sf.jack.JackPolynomials_generic

The Q basis is defined as a normalized form of the P basis

INPUT:

- `self` – an instance of the Jack Q basis of the symmetric functions
- `jack` – a family of Jack symmetric function bases

EXAMPLES:

```
sage: Q = SymmetricFunctions(FractionField(QQ['t'])).jack().Q()
sage: TestSuite(Q).run(skip=['_test_associativity', '_test_distributivity', '_test_prod']) # pro
sage: TestSuite(Q).run(elements = [Q.t*Q[1,1]+Q[2], Q[1]+(1+Q.t)*Q[1,1]])  # long time (3s on sa
```

class **Element**(*M*, *x*)

Bases: sage.combinat.sf.jack.JackPolynomials_generic.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.sf.jack.**JackPolynomials_qp**(*jack*)

Bases: sage.combinat.sf.jack.JackPolynomials_generic

The Qp basis is the dual basis to the P basis with respect to the standard scalar product

INPUT:

- `self` – an instance of the Jack Qp basis of the symmetric functions
- `jack` – a family of Jack symmetric function bases

EXAMPLES:

```
sage: Qp = SymmetricFunctions(FractionField(QQ['t'])).jack().Qp()
sage: TestSuite(Qp).run(skip=['_test_associativity', '_test_distributivity', '_test_prod']) # pr
sage: TestSuite(Qp).run(elements = [Qp.t*Qp[1,1]+Qp[2], Qp[1]+(1+Qp.t)*Qp[1,1]])  # long time (3
```

class Element(*M*, *x*)

Bases: sage.combinat.sf.jack.JackPolynomials_generic.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

JackPolynomials_qp.**coproduct_by_coercion**(*elt*)

Returns the coproduct of the element elt by coercion to the Schur basis.

INPUT:

- elt – an instance of the Qp basis

OUTPUT:

- The coproduct acting on elt, the result is an element of the tensor squared of the Qp symmetric function basis

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'].fraction_field())
sage: JQp = Sym.jack().Qp()
sage: JQp[2,2].coproduct()   #indirect doctest
JackQp[] # JackQp[2, 2] + (2*t/(t+1))*JackQp[1] # JackQp[2, 1] + JackQp[1, 1] # JackQp[1, 1]
```

class sage.combinat.sf.jack.**SymmetricFunctionAlgebra_zonal**(*Sym*)

Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic

Returns the algebra of zonal polynomials.

INPUT:

- self – a zonal basis of the symmetric functions
- Sym – a ring of the symmetric functions

EXAMPLES

```
sage: Z = SymmetricFunctions(QQ).zonal()
sage: Z([2])^2
64/45*Z[2, 2] + 16/21*Z[3, 1] + Z[4]
sage: Z = SymmetricFunctions(QQ).zonal()
sage: TestSuite(Z).run(skip=['_test_associativity', '_test_distributivity', '_test_prod']) # prc
sage: TestSuite(Z).run(elements = [Z[1,1]+Z[2], Z[1]+2*Z[1,1]])
```

class Element(*M*, *x*)

Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic_Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

scalar_zonal(*x*)

The zonal scalar product has the power sum basis and the zonal symmetric functions are orthogonal. In particular, $\langle p_\mu, p_\mu \rangle = z_\mu 2^{length(\mu)}$.

INPUT:

- `self` – an element of the zonal basis
- `x` – an element of the symmetric function

OUTPUT:

- the scalar product between `self` and `x`

EXAMPLES

```
sage: Sym = SymmetricFunctions(QQ)
sage: Z = Sym.zonal()
sage: parts = Partitions(3).list()
sage: matrix([[Z(a).scalar_zonal(Z(b)) for a in parts] for b in parts])
[16/5    0    0]
[   0    5    0]
[   0    0    4]
sage: p = Z.symmetric_function_ring().power()
sage: matrix([[Z(p(a)).scalar_zonal(p(b)) for a in parts] for b in parts])
[ 6  0  0]
[ 0  8  0]
[ 0  0 48]
```

sage.combinat.sf.jack.**c1**(*part*, *t*)

Returns the t-Jack scalar product between `J(part)` and `P(part)`.

INPUT:

- `part` – a partition
- `t` – an optional parameter (default: uses the parameter t from the Jack basis)

OUTPUT:

- a polynomial in the parameter `t` which is equal to the scalar product of `J(part)` and `P(part)`

EXAMPLES:

```
sage: from sage.combinat.sf.jack import c1
sage: t = QQ['t'].gen()
sage: [c1(p,t) for p in Partitions(3)]
[2*t^2 + 3*t + 1, t + 2, 6]
```

sage.combinat.sf.jack.**c2**(*part*, *t*)

Returns the t-Jack scalar product between `J(part)` and `Q(part)`.

INPUT:

- `self` – a Jack basis of the symmetric functions
- `part` – a partition
- `t` – an optional parameter (default: uses the parameter t from the Jack basis)

OUTPUT:

- a polynomial in the parameter `t` which is equal to the scalar product of `J(part)` and `Q(part)`

EXAMPLES:

```
sage: from sage.combinat.sf.jack import c2
sage: t = QQ['t'].gen()
sage: [c2(p,t) for p in Partitions(3)]
[6*t^3, 2*t^3 + t^2, t^3 + 3*t^2 + 2*t]
```

sage.combinat.sf.jack.**normalize_coefficients**(*self*, *c*)

If our coefficient ring is the field of fractions over a univariate polynomial ring over the rationals, then we should clear both the numerator and denominator of the denominators of their coefficients.

INPUT:

- `self` – a Jack basis of the symmetric functions
- `c` – a coefficient in the base ring of `self`

OUTPUT:

- divide numerator and denominator by the greatest common divisor

EXAMPLES:

```
sage: JP = SymmetricFunctions(FractionField(QQ['t'])).jack().P()
sage: t = JP.base_ring().gen()
sage: a = 2/(1/2*t+1/2)
sage: JP._normalize_coefficients(a)
4/(t + 1)
sage: a = 1/(1/3+1/6*t)
sage: JP._normalize_coefficients(a)
6/(t + 2)
sage: a = 24/(4*t^2 + 12*t + 8)
sage: JP._normalize_coefficients(a)
6/(t^2 + 3*t + 2)
```

sage.combinat.sf.jack.**part_scalar_jack**(*part1*, *part2*, *t*)

Returns the Jack scalar product between `p(part1)` and `p(part2)` where p is the power-sum basis.

INPUT:

- `part1`, `part2` – two partitions
- `t` – a parameter

OUTPUT:

- returns the scalar product between the power sum indexed by `part1` and `part2`

EXAMPLES:

```
sage: Q.<t> = QQ[]
sage: from sage.combinat.sf.jack import part_scalar_jack
sage: matrix([[part_scalar_jack(p1,p2,t) for p1 in Partitions(4)] for p2 in Partitions(4)])
[   4*t      0      0      0      0]
[     0  3*t^2      0      0      0]
[     0      0  8*t^2      0      0]
[     0      0      0  4*t^3      0]
[     0      0      0      0 24*t^4]
```

5.1.254 Quotient of symmetric function space by ideal generated by Hall-Littlewood symmetric functions

The quotient of symmetric functions by the ideal generated by the Hall-Littlewood P symmetric functions indexed by partitions with first part greater than k. When $t = 1$ this space is the quotient of the symmetric functions by the ideal generated by the monomial symmetric functions indexed by partitions with first part greater than k.

AUTHORS:

- Chris Berg (2012-12-01)
- Mike Zabrocki - k-bounded Hall Littlewood P and dual k-Schur functions (2012-12-02)

class sage.combinat.sf.k_dual.**AffineSchurFunctions**(*kBoundedRing*)
Bases: sage.combinat.sf.k_dual.KBoundedQuotientBasis

This basis is dual to the k-Schur functions at $t = 1$. This realization follows the monomial expansion given by Lam [Lam2006].

REFERENCES:

class sage.combinat.sf.k_dual.**DualkSchurFunctions**(*kBoundedRing*)
Bases: sage.combinat.sf.k_dual.KBoundedQuotientBasis

This basis is dual to the k-Schur functions. The expansion is given in Section 4.12 of [LLMSSZ]. When $t = 1$ this basis is equal to the AffineSchurFunctions and that basis is more efficient in this case.

REFERENCES:

class sage.combinat.sf.k_dual.**KBoundedQuotient**(*Sym*, *k*, *t='t'*)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Initialization of the ring of Symmetric functions modulo the ideal of monomial symmetric functions which are indexed by partitions whose first part is greater than k.

INPUT:

- Sym – an element of class sage.combinat.sf.sf.SymmetricFunctions
- k – a positive integer
- R – a ring

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Q = Sym.kBoundedQuotient(3,t=1)
sage: Q
3-Bounded Quotient of Symmetric Functions over Rational Field with t=1
sage: km = Q.km()
sage: km
3-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the 3-bounded monomial
sage: F = Q.affineSchur()
sage: F(km(F[3,1,1])) == F[3,1,1]
True
sage: km(F(km([3,2]))) == km[3,2]
True
sage: F[3,2].lift()
m[1, 1, 1, 1, 1] + m[2, 1, 1, 1] + m[2, 2, 1] + m[3, 1, 1] + m[3, 2]
sage: F[2,1]*F[2,1]
2*F3[1, 1, 1, 1, 1, 1] + 4*F3[2, 1, 1, 1, 1] + 4*F3[2, 2, 1, 1] + 4*F3[2, 2, 2] + 2*F3[3, 1, 1,
sage: F[1,2]
Traceback (most recent call last):
```

```
...
ValueError: [1, 2] is not a valid partition
sage: F[4,2]
Traceback (most recent call last):
...
ValueError: Partition is not 3-bounded
sage: km[2,1]*km[2,1]
4*m3[2, 2, 1, 1] + 6*m3[2, 2, 2] + 2*m3[3, 2, 1] + 2*m3[3, 3]
sage: HLPk = Q.kHallLittlewoodP()
sage: HLPk[2,1]*HLPk[2,1]
4*HLP3[2, 2, 1, 1] + 6*HLP3[2, 2, 2] + 2*HLP3[3, 2, 1] + 2*HLP3[3, 3]
sage: dks = Q.dual_k_Schur()
sage: dks[2,1]*dks[2,1]
2*dks3[1, 1, 1, 1, 1, 1] + 4*dks3[2, 1, 1, 1, 1] + 4*dks3[2, 2, 1, 1] + 4*dks3[2, 2, 2] + 2*dks3

sage: Q = Sym.kBoundedQuotient(3)
Traceback (most recent call last):
...
TypeError: unable to convert t to a rational
sage: Sym = SymmetricFunctions(QQ['t'].fraction_field())
sage: Q = Sym.kBoundedQuotient(3)
sage: km = Q.km()
sage: F = Q.affineSchur()
sage: F(km(F[3,1,1])) == F[3,1,1]
True
sage: km(F(km([3,2]))) == km[3,2]
True
sage: dks = Q.dual_k_Schur()
sage: HLPk = Q.kHallLittlewoodP()
sage: dks(HLPk(dks[3,1,1])) == dks[3,1,1]
True
sage: km(dks(km([3,2]))) == km[3,2]
True
sage: dks[2,1]*dks[2,1]
(t^3+t^2)*dks3[1, 1, 1, 1, 1, 1] + (2*t^2+2*t)*dks3[2, 1, 1, 1, 1] + (t^2+2*t+1)*dks3[2, 2, 1, 1
```

TESTS:

```
sage: TestSuite(Q).run()
```

AffineGrothendieckPolynomial(*la*, *m*)

Returns the affine Grothendieck polynomial indexed by the partition `la`. Because this belongs to the completion of the algebra, and hence is an infinite sum, it computes only up to those symmetric functions of degree at most `m`. See `_AffineGrothendieckPolynomial()` for the code.

INPUT:

- `la` – A k-bounded partition
- `m` – An integer

EXAMPLES:

```
sage: Q = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1)
sage: Q.AffineGrothendieckPolynomial([2,1],4)
2*m3[1, 1, 1] - 8*m3[1, 1, 1, 1] + m3[2, 1] - 3*m3[2, 1, 1] - m3[2, 2]
```

F()

The affine Schur basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions. This is also equal to the affine Stanley symmetric functions (see

WeylGroups.ElementMethods.stanley_symmetric_function()) indexed by an affine Grassmannian permutation.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).kBoundedQuotient(2,t=1).affineSchur()
2-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the 2-bounded affi
```

a_realization()

Returns a particular realization of self (the basis of k-bounded monomials if $t = 1$ and the basis of k-bounded Hall-Littlewood functions otherwise).

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Q = Sym.kBoundedQuotient(3,t=1)
sage: Q.a_realization()
3-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the 3-bounded monc
sage: Q = Sym.kBoundedQuotient(3,t=2)
sage: Q.a_realization()
3-Bounded Quotient of Symmetric Functions over Rational Field with t=2 in the 3-bounded Hall
```

affineSchur()

The affine Schur basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions. This is also equal to the affine Stanley symmetric functions (see WeylGroups.ElementMethods.stanley_symmetric_function()) indexed by an affine Grassmannian permutation.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).kBoundedQuotient(2,t=1).affineSchur()
2-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the 2-bounded affi
```

ambient()

Returns the Symmetric Functions over the same ring as self. This is needed to realize our ring as a quotient.

TESTS:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Q = Sym.kBoundedQuotient(3,t=1)
sage: Q.ambient()
Symmetric Functions over Rational Field
```

an_element()

Returns an element of the quotient ring of k-bounded symmetric functions. This method is here to make the TestSuite run properly.

EXAMPLES:

```
sage: Q = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1)
sage: Q.an_element()
2*m3[] + 2*m3[1] + 3*m3[2]
```

dks()

The dual k-Schur basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions. At $t = 1$ this is also equal to the affine Schur basis and calculations will be faster using elements in the affineSchur() basis.

EXAMPLES:

```
sage: SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(2).dual_k_Schur()
2-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate Polynomial Ring
```

dual_k_Schur()

The dual k-Schur basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions. At $t = 1$ this is also equal to the affine Schur basis and calculations will be faster using elements in the affineSchur() basis.

EXAMPLES:

```
sage: SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(2).dual_k_Schur()
2-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate Polynomial Ring
```

kHLP()

The Hall-Littlewood P basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions. At $t = 1$ this basis is equal to the k-bounded monomial basis and calculations will be faster using elements in the k-bounded monomial basis (see kmonomial()).

EXAMPLES:

```
sage: SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(2).kHallLittlewoodP()
2-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate Polynomial Ring
```

kHallLittlewoodP()

The Hall-Littlewood P basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions. At $t = 1$ this basis is equal to the k-bounded monomial basis and calculations will be faster using elements in the k-bounded monomial basis (see kmonomial()).

EXAMPLES:

```
sage: SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(2).kHallLittlewoodP()
2-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate Polynomial Ring
```

km()

The monomial basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).kBoundedQuotient(2,t=1).kmonomial()
2-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the 2-bounded monc
```

kmonomial()

The monomial basis of the k-bounded quotient of symmetric functions, indexed by k-bounded partitions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).kBoundedQuotient(2,t=1).kmonomial()
2-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the 2-bounded monc
```

lift(*la*)

Gives the lift map from the quotient ring of k-bounded symmetric functions to the symmetric functions. This method is here to make the TestSuite run properly.

INPUT:

- la – A k-bounded partition

OUTPUT:

- **The monomial element or a Hall-Littlewood P element of the symmetric functions** indexed by the partition la.

EXAMPLES:

```
sage: Q = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1)
sage: Q.lift([2,1])
m[2, 1]
sage: Q = SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(3)
sage: Q.lift([2,1])
HLP[2, 1]
```

one()

Returns the unit of the quotient ring of k-bounded symmetric functions. This method is here to make the TestSuite run properly.

EXAMPLES:

```
sage: Q = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1)
sage: Q.one()
m3[]
```

realizations()

A list of realizations of the k-bounded quotient.

EXAMPLES:

```
sage: kQ = SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(3)
sage: kQ.realizations()
[3-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate Polynomial Ring
sage: HLP = kQ.ambient().hall_littlewood().P()
sage: all( rzn(HLP[3,2,1]).lift() == HLP[3,2,1] for rzn in kQ.realizations())
True
sage: kQ = SymmetricFunctions(QQ).kBoundedQuotient(3,1)
sage: kQ.realizations()
[3-Bounded Quotient of Symmetric Functions over Rational Field with t=1 in the 3-bounded mon
sage: m = kQ.ambient().m()
sage: all( rzn(m[3,2,1]).lift() == m[3,2,1] for rzn in kQ.realizations())
True
```

retract(*la*)

Gives the retract map from the symmetric functions to the quotient ring of k-bounded symmetric functions. This method is here to make the TestSuite run properly.

INPUT:

- `la` – A partition

OUTPUT:

- The monomial element of the k-bounded quotient indexed by `la`.

EXAMPLES:

```
sage: Q = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1)
sage: Q.retract([2,1])
m3[2, 1]
```

class sage.combinat.sf.k_dual.**KBoundedQuotientBases**(*base*)

Bases: sage.categories.realizations.Category_realization_of_parent

The category of bases for the k-bounded subspace of symmetric functions.

class ElementMethods

class KBoundedQuotientBases.**ParentMethods**

ambient()

Returns the symmetric functions.

EXAMPLES:

```
sage: km = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).km()
sage: km.ambient()
Symmetric Functions over Rational Field
```

antipode(*element*)

Return the antipode of `element` via lifting to the symmetric functions and then retracting into the k-bounded quotient basis.

INPUT:

- `element` – an element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: dks3 = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).dual_k_Schur()
sage: dks3[3,2].antipode()
-dks3[1, 1, 1, 1, 1]
sage: km = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).km()
sage: km[3,2].antipode()
m[3, 2]
sage: km.antipode(km[3,2])
m[3, 2]
sage: m = SymmetricFunctions(QQ).m()
sage: m[3,2].antipode()
m[3, 2] + 2*m[5]
```

```
sage: km = SymmetricFunctions(FractionField(QQ['t'])).kBoundedQuotient(3).km()
sage: km[1,1,1,1].antipode()
(t^3-3*t^2+3*t)*m[1, 1, 1, 1] + (-t^2+2*t)*m[2, 1, 1] + t*m[2, 2] + t*m[3, 1]
sage: kHP = SymmetricFunctions(FractionField(QQ['t'])).kBoundedQuotient(3).kHLP()
sage: kHP[2,2].antipode()
(t^9-t^6-t^5+t^2)*HLP3[1, 1, 1, 1] + (t^6-t^3-t^2+t)*HLP3[2, 1, 1] + (t^5-t^2+1)*HLP3[2,
sage: dks = SymmetricFunctions(FractionField(QQ['t'])).kBoundedQuotient(3).dks()
sage: dks[2,2].antipode()
dks3[2, 2]
sage: dks[3,2].antipode()
-t^2*dks3[1, 1, 1, 1, 1] + (t^2-1)*dks3[2, 2, 1] + (-t^5+t)*dks3[3, 2]
```

coproduct(*element*)

Return the coproduct of `element` via lifting to the symmetric functions and then returning to the k-bounded quotient basis. This method is implemented for all t but is (weakly) conjectured to not be the correct operation for arbitrary t because the coproduct on dual-k-Schur functions does not have a positive expansion.

INPUT:

- `element` – an element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Q3 = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1)
sage: km = Q3.km()
sage: km[3,2].coproduct()
m[] # m[3, 2] + m[2] # m[3] + m[3] # m[2] + m[3, 2] # m[]
sage: dks3 = Q3.dual_k_Schur()
sage: dks3[2,2].coproduct()
dks3[] # dks3[2, 2] + dks3[1] # dks3[2, 1] + dks3[1, 1] # dks3[1, 1] + dks3[2] # dks3[2]
```

```
sage: Q3t = SymmetricFunctions(FractionField(QQ['t'])).kBoundedQuotient(3)
sage: km = Q3t.km()
sage: km[3,2].coproduct()
m3[] # m3[3, 2] + m3[2] # m3[3] + m3[3] # m3[2] + m3[3, 2] # m3[]
sage: dks = Q3t.dks()
sage: dks[2,1,1].coproduct()
dks3[] # dks3[2, 1, 1] + (-t+1)*dks3[1] # dks3[1, 1, 1] + dks3[1] # dks3[2, 1] + (-t+1)*
sage: kHLP = Q3t.kHLP()
sage: kHLP[2,1].coproduct()
HLP3[] # HLP3[2, 1] + (-t^2+1)*HLP3[1] # HLP3[1, 1] + HLP3[1] # HLP3[2] + (-t^2+1)*HLP3[
sage: km.coproduct(km[3,2])
m3[] # m3[3, 2] + m3[2] # m3[3] + m3[3] # m3[2] + m3[3, 2] # m3[]
```

counit(*element*)

Return the counit of `element`.

The counit is the constant term of `element`.

INPUT:

- `element` – an element in a basis

EXAMPLES:

```
sage: km = SymmetricFunctions(FractionField(QQ['t'])).kBoundedQuotient(3).km()
sage: f = 2*km[2,1] - 3*km([])
sage: f.counit()
-3
sage: km.counit(f)
-3
```

degree_on_basis(*b*)

Return the degree of the basis element indexed by `b`.

INPUT:

- `b` – a partition

EXAMPLES:

```
sage: F = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).affineSchur()
sage: F.degree_on_basis(Partition([3,2]))
5
```

indices()

The set of k-bounded partitions of all non-negative integers.

EXAMPLES:

```
sage: km = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).km()
sage: km.indices()
3-Bounded Partitions
```

lift(*la*)

Implements the lift map from the basis `self` to the monomial basis of symmetric functions.

INPUT:

- `la` – A k-bounded partition.

OUTPUT:

- A symmetric function in the monomial basis.

EXAMPLES:

```
sage: F = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).affineSchur()
sage: F.lift([3,1])
m[1, 1, 1, 1] + m[2, 1, 1] + m[2, 2] + m[3, 1]
sage: Sym = SymmetricFunctions(QQ['t'].fraction_field())
sage: dks = Sym.kBoundedQuotient(3).dual_k_Schur()
```

```
sage: dks.lift([3,1])
t^5*HLP[1, 1, 1, 1] + t^2*HLP[2, 1, 1] + t*HLP[2, 2] + HLP[3, 1]
sage: dks = Sym.kBoundedQuotient(3,t=1).dual_k_Schur()
sage: dks.lift([3,1])
m[1, 1, 1, 1] + m[2, 1, 1] + m[2, 2] + m[3, 1]
```

one_basis()

Return the basis element indexing 1.

EXAMPLES:

```
sage: F = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).affineSchur()
sage: F.one()  # indirect doctest
F3[]
```

product(*x*, *y*)

Returns the product of two elements x and y.

INPUT:

•x, y – Elements of the k-bounded quotient of symmetric functions.

OUTPUT:

•A k-bounded symmetric function in the dual k-Schur function basis

EXAMPLES:

```
sage: dks3 = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).dual_k_Schur()
sage: dks3.product(dks3[2,1],dks3[1,1])
2*dks3[1, 1, 1, 1, 1] + 2*dks3[2, 1, 1, 1] + 2*dks3[2, 2, 1] + dks3[3, 1, 1] + dks3[3, 2
sage: dks3.product(dks3[2,1]+dks3[1], dks3[1,1])
dks3[1, 1, 1] + 2*dks3[1, 1, 1, 1, 1] + dks3[2, 1] + 2*dks3[2, 1, 1, 1] + 2*dks3[2, 2, 1
sage: dks3.product(dks3[2,1]+dks3[1], dks3([]))
dks3[1] + dks3[2, 1]
sage: dks3.product(dks3([]), dks3([]))
dks3[]
sage: dks3.product(dks3([]), dks3([4,1]))
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 1]) an element of self (=3-Bounded Quotient
```

```
sage: dks3 = SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(3).dual_k_Schu
sage: dks3.product(dks3[2,1],dks3[1,1])
(t^2+t)*dks3[1, 1, 1, 1, 1] + (t+1)*dks3[2, 1, 1, 1] + (t+1)*dks3[2, 2, 1] + dks3[3, 1,
sage: dks3.product(dks3[2,1]+dks3[1], dks3[1,1])
dks3[1, 1, 1] + (t^2+t)*dks3[1, 1, 1, 1, 1] + dks3[2, 1] + (t+1)*dks3[2, 1, 1, 1] + (t+1
sage: dks3.product(dks3[2,1]+dks3[1], dks3([]))
dks3[1] + dks3[2, 1]
sage: dks3.product(dks3([]), dks3([]))
dks3[]
```

```
sage: F = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).affineSchur()
sage: F.product(F[2,1],F[1,1])
2*F3[1, 1, 1, 1, 1] + 2*F3[2, 1, 1, 1] + 2*F3[2, 2, 1] + F3[3, 1, 1] + F3[3, 2]
sage: F.product(F[2,1]+F[1], F[1,1])
F3[1, 1, 1] + 2*F3[1, 1, 1, 1, 1] + F3[2, 1] + 2*F3[2, 1, 1, 1] + 2*F3[2, 2, 1] + F3[3,
sage: F.product(F[2,1]+F[1], F([]))
F3[1] + F3[2, 1]
sage: F.product(F([]), F([]))
F3[]
sage: F.product(F([]), F([4,1]))
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 1]) an element of self (=3-Bounded Quotient
```

```
sage: F = SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(3).affineSchur()
sage: F.product(F[2,1],F[1,1])
2*F3[1, 1, 1, 1, 1] + 2*F3[2, 1, 1, 1] + 2*F3[2, 2, 1] + F3[3, 1, 1] + F3[3, 2]
sage: F.product(F[2,1],F[2])
(t^4+t^3-2*t^2+1)*F3[1, 1, 1, 1, 1] + (-t^2+t+1)*F3[2, 1, 1, 1] + (-t^2+t+2)*F3[2, 2, 1]
sage: F.product(F[2,1]+F[1], F[1,1])
F3[1, 1, 1] + 2*F3[1, 1, 1, 1, 1] + F3[2, 1] + 2*F3[2, 1, 1, 1] + 2*F3[2, 2, 1] + F3[3,
sage: F.product(F[2,1]+F[1], F([]))
F3[1] + F3[2, 1]
sage: F.product(F([]), F([]))
F3[]

sage: km = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).km()
sage: km.product(km[2,1],km[2,1])
4*m3[2, 2, 1, 1] + 6*m3[2, 2, 2] + 2*m3[3, 2, 1] + 2*m3[3, 3]
sage: Q3 = SymmetricFunctions(FractionField(QQ['t'])).kBoundedQuotient(3)
sage: km = Q3.km()
sage: km.product(km[2,1],km[2,1])
(t^5+7*t^4-8*t^3-28*t^2+47*t-19)*m3[1, 1, 1, 1, 1, 1] + (t^4-3*t^3-9*t^2+23*t-12)*m3[2,
sage: dks = Q3.dual_k_Schur()
sage: km.product(dks[2,1],dks[1,1])
20*m3[1, 1, 1, 1, 1] + 9*m3[2, 1, 1, 1] + 4*m3[2, 2, 1] + 2*m3[3, 1, 1] + m3[3, 2]
```

retract(*la*)

Gives the retract map from the symmetric functions to the quotient ring of k-bounded symmetric functions. This method is here to make the TestSuite run properly.

INPUT:

- `la` – A partition

OUTPUT:

- The monomial element of the k-bounded quotient indexed by `la`.

EXAMPLES:

```
sage: Q = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1)
sage: Q.retract([2,1])
m3[2, 1]
```

KBoundedQuotientBases.**super_categories**()

The super categories of `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'])
sage: from sage.combinat.sf.k_dual import KBoundedQuotientBases
sage: Q = Sym.kBoundedQuotient(3,t=1)
sage: KQB = KBoundedQuotientBases(Q)
sage: KQB.super_categories()
[Category of realizations of 3-Bounded Quotient of Symmetric Functions over Univariate Polyn
 Join of Category of graded hopf algebras with basis over Univariate Polynomial Ring in t ov
         Category of quotients of algebras over Univariate Polynomial Ring in t over Rationa
```

class sage.combinat.sf.k_dual.**KBoundedQuotientBasis**(*kBoundedRing*, *prefix*)

Bases: sage.combinat.free_module.CombinatorialFreeModule

Abstract base class for the bases of the k-bounded quotient.

class sage.combinat.sf.k_dual.**kMonomial**(*kBoundedRing*)

Bases: sage.combinat.sf.k_dual.KBoundedQuotientBasis

The basis of monomial symmetric functions indexed by partitions with first part less than or equal to k.

lift(*la*)

Implements the lift function on the monomial basis. Given a k-bounded partition `la`, the lift will return the corresponding monomial basis element.

INPUT:

- `la` – A k-bounded partition

OUTPUT:

- A monomial symmetric function.

EXAMPLES:

```
sage: km = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).km()
sage: km.lift(Partition([3,1]))
m[3, 1]
sage: km.lift([])
m[]
sage: km.lift(Partition([4,1]))
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 1]) an element of self (=3-Bounded Quotient of S
```

retract(*la*)

Implements the retract function on the monomial basis. Given a partition `la`, the retract will return the corresponding k-bounded monomial basis element if `la` is k-bounded; zero otherwise.

INPUT:

- `la` – A partition

OUTPUT:

- **A k-bounded monomial symmetric function in the k-quotient of symmetric** functions.

EXAMPLES:

```
sage: km = SymmetricFunctions(QQ).kBoundedQuotient(3,t=1).km()
sage: km.retract(Partition([3,1]))
m3[3, 1]
sage: km.retract(Partition([4,1]))
0
sage: km.retract([])
m3[]
sage: m = SymmetricFunctions(QQ).m()
sage: km(m[3, 1])
m3[3, 1]
sage: km(m[4, 1])
0

sage: km = SymmetricFunctions(FractionField(QQ['t'])).kBoundedQuotient(3).km()
sage: km.retract(Partition([3,1]))
m3[3, 1]
sage: km.retract(Partition([4,1]))
(t^4+t^3-9*t^2+11*t-4)*m3[1, 1, 1, 1, 1] + (-3*t^2+6*t-3)*m3[2, 1, 1, 1] + (-t^2+3*t-2)*m3[2
sage: m = SymmetricFunctions(FractionField(QQ['t'])).m()
sage: km(m[3, 1])
m3[3, 1]
sage: km(m[4, 1])
(t^4+t^3-9*t^2+11*t-4)*m3[1, 1, 1, 1, 1] + (-3*t^2+6*t-3)*m3[2, 1, 1, 1] + (-t^2+3*t-2)*m3[2
```

class sage.combinat.sf.k_dual.**kbounded_HallLittlewoodP**(*kBoundedRing*)

Bases: sage.combinat.sf.k_dual.KBoundedQuotientBasis

The basis of P Hall-Littlewood symmetric functions indexed by partitions with first part less than or equal to k.

lift(*la*)

Implements the lift function on the Hall-Littlewood P basis. Given a k-bounded partition `la`, the lift will return the corresponding Hall-Littlewood P basis element.

INPUT:

- `la` – A k-bounded partition

OUTPUT:

- A Hall-Littlewood symmetric function.

EXAMPLES:

```
sage: kHLP = SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(3).kHallLittlewoo
sage: kHLP.lift(Partition([3,1]))
HLP[3, 1]
sage: kHLP.lift([])
HLP[]
sage: kHLP.lift(Partition([4,1]))
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 1]) an element of self (=3-Bounded Quotient of S
```

retract(*la*)

Implements the retract function on the Hall-Littlewood P basis. Given a partition `la`, the retract will return the corresponding k-bounded Hall-Littlewood P basis element if `la` is k-bounded; zero otherwise.

INPUT:

- `la` – A partition

OUTPUT:

- **A k-bounded Hall-Littlewood P symmetric function in the k-quotient of** symmetric functions.

EXAMPLES:

```
sage: kHLP = SymmetricFunctions(QQ['t'].fraction_field()).kBoundedQuotient(3).kHallLittlewoo
sage: kHLP.retract(Partition([3,1]))
HLP3[3, 1]
sage: kHLP.retract(Partition([4,1]))
0
sage: kHLP.retract([])
HLP3[]
sage: m = kHLP.realization_of().ambient().m()
sage: kHLP(m[2,2])
(t^4-t^3-t+1)*HLP3[1, 1, 1, 1] + (t-1)*HLP3[2, 1, 1] + HLP3[2, 2]
```

5.1.255 Kostka-Foulkes Polynomials

Based on the algorithms in John Stembridge's SF package for Maple which can be found at http://www.math.lsa.umich.edu/~jrs/maple.html .

`sage.combinat.sf.kfpoly.`**KostkaFoulkesPolynomial**(*mu*, *nu*, *t=None*)

Returns the Kostka-Foulkes polynomial $K_{\mu,\nu}(t)$.

INPUT:

- `mu`, `nu` – partitions

- t – an optional parameter (default: None)

OUTPUT:

- the Koskta-Foulkes polynomial indexed by partitions mu and nu and evaluated at the parameter t. If t is None the resulting polynomial is in the polynomial ring $ZZ['t']$.

EXAMPLES:

```
sage: KostkaFoulkesPolynomial([2,2],[2,2])
1
sage: KostkaFoulkesPolynomial([2,2],[4])
0
sage: KostkaFoulkesPolynomial([2,2],[1,1,1,1])
t^4 + t^2
sage: KostkaFoulkesPolynomial([2,2],[2,1,1])
t
sage: q = PolynomialRing(QQ,'q').gen()
sage: KostkaFoulkesPolynomial([2,2],[2,1,1],q)
q
```

sage.combinat.sf.kfpoly.**compat**(*n*, *mu*, *nu*)

Generate all possible partitions of n that can precede μ, ν in a rigging sequence.

INPUT:

- n – a positive integer
- mu, nu – partitions

OUTPUT:

- a list of partitions

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import *
sage: compat(4, [1], [2,1])
[[1, 1, 1, 1], [2, 1, 1], [2, 2], [3, 1], [4]]
sage: compat(3, [1], [2,1])
[[1, 1, 1], [2, 1], [3]]
sage: compat(2, [1], [])
[[2]]
sage: compat(3, [1], [])
[[2, 1], [3]]
sage: compat(3, [2], [1])
[[3]]
sage: compat(4, [1,1], [])
[[2, 2], [3, 1], [4]]
sage: compat(4, [2], [])
[[4]]
```

sage.combinat.sf.kfpoly.**dom**(*mu*, *snu*)

Returns True if sum(mu[:i+1]) >= snu[i] for all 0 <= i < len(snu); otherwise, it returns False.

INPUT:

- mu – a partition
- snu – a sequence of positive integers

OUTPUT:

- a boolean value

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import *
sage: dom([3,2,1],[2,4,5])
True
sage: dom([3,2,1],[2,4,7])
False
sage: dom([3,2,1],[2,6,5])
False
sage: dom([3,2,1],[4,4,4])
False
```

sage.combinat.sf.kfpoly.**kfpoly**(*mu*, *nu*, *t=None*)

Returns the Kostka-Foulkes polynomial $K_{\mu,\nu}(t)$ by generating all rigging sequences for the shape μ, and then selecting those of content ν.

INPUT:

- mu, nu – partitions
- t – an optional parameter (default: None)

OUTPUT:

- the Koskta-Foulkes polynomial indexed by partitions mu and nu and evaluated at the parameter t. If t is None the resulting polynomial is in the polynomial ring $\mathbb{Z}['t']$.

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import kfpoly
sage: kfpoly([2,2], [2,1,1])
t
sage: kfpoly([4], [2,1,1])
t^3
sage: kfpoly([4], [2,2])
t^2
sage: kfpoly([1,1,1,1], [2,2])
0
```

sage.combinat.sf.kfpoly.**riggings**(*part*)

Generate all possible rigging sequences for a fixed sage.combinat.partition.

INPUT:

- part – a partition

OUTPUT:

- returns a list of riggings associated to the partition part

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import *
sage: riggings([3])
[[[1, 1, 1]], [[2, 1]], [[3]]]
sage: riggings([2,1])
[[[2, 1], [1]], [[3], [1]]]
sage: riggings([1,1,1])
[[[3], [2], [1]]]
sage: riggings([2,2])
[[[2, 2], [1, 1]], [[3, 1], [1, 1]], [[4], [1, 1]], [[4], [2]]]
sage: riggings([2,2,2])
[[[3, 3], [2, 2], [1, 1]],
 [[4, 2], [2, 2], [1, 1]],
```

```
 [[5, 1], [2, 2], [1, 1]],
 [[6], [2, 2], [1, 1]],
 [[5, 1], [3, 1], [1, 1]],
 [[6], [3, 1], [1, 1]],
 [[6], [4], [2]]]
```

sage.combinat.sf.kfpoly.**schur_to_hl**(*mu*, *t=None*)

Returns a dictionary corresponding to s_μ in Hall-Littlewood P basis.

INPUT:

- mu – a partition
- t – an optional parameter (default : the generator from $\mathbb{Z}['t']$)

OUTPUT:

- a dictionary with the coefficients $K_{\mu\nu}(t)$ for ν smaller in dominance order than μ

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import *
sage: schur_to_hl([1,1,1])
{[1, 1, 1]: 1}
sage: a = schur_to_hl([2,1])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1] t^2 + t
[2, 1] 1
sage: a = schur_to_hl([3])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1] t^3
[2, 1] t
[3] 1
sage: a = schur_to_hl([4])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1, 1] t^6
[2, 1, 1] t^3
[2, 2] t^2
[3, 1] t
[4] 1
sage: a = schur_to_hl([3,1])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1, 1] t^5 + t^4 + t^3
[2, 1, 1] t^2 + t
[2, 2] t
[3, 1] 1
sage: a = schur_to_hl([2,2])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1, 1] t^4 + t^2
[2, 1, 1] t
[2, 2] 1
sage: a = schur_to_hl([2,1,1])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1, 1] t^3 + t^2 + t
[2, 1, 1] 1
sage: a = schur_to_hl([1,1,1,1])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1, 1] 1
sage: a = schur_to_hl([2,2,2])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1, 1, 1, 1] t^9 + t^7 + t^6 + t^5 + t^3
```

```
[2, 1, 1, 1, 1] t^4 + t^2
[2, 2, 1, 1] t
[2, 2, 2] 1
```

sage.combinat.sf.kfpoly.**weight**(*rg*, *t=None*)

Returns the weight of a rigging.

INPUT:

- `rg` – a rigging, a list of partitions
- `t` – an optional parameter, (default : uses the generator from $ZZ['t']$)

OUTPUT:

- a polynomial in the parameter `t`

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import weight
sage: weight([[2,1], [1]])
1
sage: weight([[3], [1]])
t^2 + t
sage: weight([[2,1], [3]])
t^4
sage: weight([[2, 2], [1, 1]])
1
sage: weight([[3, 1], [1, 1]])
t
sage: weight([[4], [1, 1]], 2)
16
sage: weight([[4], [2]], t=2)
4
```

5.1.256 LLT symmetric functions

REFERENCES:

class sage.combinat.sf.llt.**LLT_class**(*Sym*, *k*, *t='t'*)

Bases: sage.structure.unique_representation.UniqueRepresentation

A class for working with LLT symmetric functions.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: L3 = Sym.llt(3); L3
level 3 LLT polynomials over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: L3.cospin([3,2,1])
(t+1)*m[1, 1] + m[2]
sage: HC3 = L3.hcospin(); HC3
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Field i
sage: m = Sym.monomial()
sage: m( HC3[1,1] )
(t+1)*m[1, 1] + m[2]
```

We require that the parameter t must be in the base ring:

```
sage: Symxt = SymmetricFunctions(QQ['x','t'].fraction_field())
sage: (x,t) = Symxt.base_ring().gens()
sage: LLT3x = Symxt.llt(3,t=x)
sage: LLT3 = Symxt.llt(3)
sage: HS3x = LLT3x.hspin()
sage: HS3t = LLT3.hspin()
sage: s = Symxt.schur()
sage: s(HS3x[2,1])
s[2, 1] + x*s[3]
sage: s(HS3t[2,1])
s[2, 1] + t*s[3]
sage: HS3x(HS3t[2,1])
HSp3[2, 1] + (-x+t)*HSp3[3]
sage: s(HS3x(HS3t[2,1]))
s[2, 1] + t*s[3]
sage: LLT3t2 = Symxt.llt(3,t=2)
sage: HC3t2 = LLT3t2.hcospin()
sage: HS3x(HC3t2[3,1])
2*HSp3[3, 1] + (-2*x+1)*HSp3[4]
```

base_ring()

Returns the base ring of `self`.

INPUT:

- `self` – a family of LLT symmetric functions bases

OUTPUT:

- returns the base ring of the symmetric function ring associated to `self`

EXAMPLES:

```
sage: SymmetricFunctions(FractionField(QQ['t'])).llt(3).base_ring()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

cospin(*skp*)

Calculates a single instance of the cospin symmetric functions. These are the functions defined in [LLT1997] equation (26).

INPUT:

- `self` – a family of LLT symmetric functions bases
- `skp` – a partition or a list of partitions or a list of skew partitions

OUTPUT:

- returns the monomial expansion of the LLT symmetric function cospin functions indexed by `skp`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: L3 = Sym.llt(3)
sage: L3.cospin([2,1])
m[1]
sage: L3.cospin([3,2,1])
(t+1)*m[1, 1] + m[2]
sage: s = Sym.schur()
sage: s(L3.cospin([[2],[1],[2]]))
t^4*s[2, 2, 1] + t^3*s[3, 1, 1] + (t^3+t^2)*s[3, 2] + (t^2+t)*s[4, 1] + s[5]
```

hcospin()

Returns the HCospin basis. This basis is defined [LLT1997] equation (27).

INPUT:

- `self` – a family of LLT symmetric functions bases

OUTPUT:

- returns the h-cospin basis of the LLT symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HCosp3 = Sym.llt(3).hcospin(); HCosp3
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: HCosp3([1])^2
1/t*HCosp3[1, 1] + ((t-1)/t)*HCosp3[2]

sage: s = Sym.schur()
sage: HCosp3(s([2]))
HCosp3[2]
sage: HCosp3(s([1,1]))
1/t*HCosp3[1, 1] - 1/t*HCosp3[2]
sage: s(HCosp3([2,1]))
t*s[2, 1] + s[3]
```

hspin()

Returns the HSpin basis. This basis is defined [LLT1997] equation (28).

INPUT:

- `self` – a family of LLT symmetric functions bases

OUTPUT:

- returns the h-spin basis of the LLT symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HSp3 = Sym.llt(3).hspin(); HSp3
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: HSp3([1])^2
HSp3[1, 1] + (-t+1)*HSp3[2]

sage: s = Sym.schur()
sage: HSp3(s([2]))
HSp3[2]
sage: HSp3(s([1,1]))
HSp3[1, 1] - t*HSp3[2]
sage: s(HSp3([2,1]))
s[2, 1] + t*s[3]
```

level()

Returns the level of `self`.

INPUT:

- `self` – a family of LLT symmetric functions bases

OUTPUT:

- the level is the parameter of k in the basis

EXAMPLES:

```
sage: SymmetricFunctions(FractionField(QQ['t'])).llt(3).level()
3
```

spin_square(*skp*)

Calculates a single instance of a spin squared LLT symmetric function associated with a partition, list of partitions, or a list of skew partitions. This family of symmetric functions is defined in [LT2000] equation (43).

INPUT:

- `self` – a family of LLT symmetric functions bases
- `skp` – a partition of a list of partitions or a list of skew partitions

OUTPUT:

- returns the monomial expansion of the LLT symmetric function spin-square functions indexed by `skp`

EXAMPLES:

```
sage: L3 = SymmetricFunctions(FractionField(QQ['t'])).llt(3)
sage: L3.spin_square([2,1])
t*m[1]
sage: L3.spin_square([3,2,1])
(t^3+t)*m[1, 1] + t^3*m[2]
sage: L3.spin_square([[1],[1],[1]])
(t^6+2*t^4+2*t^2+1)*m[1, 1, 1] + (t^6+t^4+t^2)*m[2, 1] + t^6*m[3]
sage: L3.spin_square([[[2,2],[1]],[[2,1],[]]])
(2*t^4+3*t^2+1)*m[1, 1, 1, 1] + (t^4+t^2)*m[2, 1, 1] + t^4*m[2, 2]
```

symmetric_function_ring()

The symmetric function algebra associated to the family of LLT symmetric function bases

INPUT:

- `self` – a family of LLT symmetric functions bases

OUTPUT:

- returns the symmetric function ring associated to `self`.

EXAMPLES

```
sage: L3 = SymmetricFunctions(FractionField(QQ['t'])).llt(3)
sage: L3.symmetric_function_ring()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
```

class sage.combinat.sf.llt.**LLT_cospin**(*llt*)

Bases: sage.combinat.sf.llt.LLT_generic

A class of methods for the h-cospin LLT basis of the symmetric functions.

INPUT:

- `self` – an instance of the LLT hcospin basis
- `llt` – a family of LLT symmetric function bases

TESTS:

```
sage: HCosp3 = SymmetricFunctions(FractionField(QQ['t'])).llt(3).hcospin()
sage: TestSuite(HCosp3).run(skip = ["_test_associativity", "_test_distributivity", "_test_prod"]
sage: TestSuite(HCosp3).run(elements = [HCosp3.t*HCosp3[1,1]+HCosp3.t*HCosp3[2], HCosp3[1]+(1+HC
```

```
sage: HC3t2 = SymmetricFunctions(QQ).llt(3,t=2).hcospin()
sage: TestSuite(HC3t2).run() # products are too expensive, long time (6s on sage.math, 2012)

sage: HC3x = SymmetricFunctions(FractionField(QQ['x'])).llt(3,t=x).hcospin()
sage: TestSuite(HC3x).run(skip = ["_test_associativity", "_test_distributivity", "_test_prod"])
sage: TestSuite(HC3x).run(elements = [HC3x.t*HC3x[1,1]+HC3x.t*HC3x[2], HC3x[1]+(1+HC3x.t)*HC3x[1
```

class **Element** (*M*, *x*)

Bases: sage.combinat.sf.llt.LLT_generic.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.sf.llt.**LLT_generic** (*llt*, *prefix*)

Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic

A class of methods which are common to both the hspin and hcospin of the LLT symmetric functions.

INPUT:

- self – an instance of the LLT hspin or hcospin basis
- llt – a family of LLT symmetric functions

EXAMPLES:

```
sage: SymmetricFunctions(FractionField(QQ['t'])).llt(3).hspin()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Field i
sage: SymmetricFunctions(QQ).llt(3,t=2).hspin()
Symmetric Functions over Rational Field in the level 3 LLT spin with t=2 basis
sage: QQz = FractionField(QQ['z']); z = QQz.gen()
sage: SymmetricFunctions(QQz).llt(3,t=z).hspin()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in z over Rational Field i
```

class **Element** (*M*, *x*)

Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic_Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

LLT_generic.**level**()

Returns the level of self.

INPUT:

- self – an instance of the LLT hspin or hcospin basis

OUTPUT:

- returns the level associated to the basis self.

EXAMPLES:

```
sage: HSp3 = SymmetricFunctions(FractionField(QQ['t'])).llt(3).hspin()
sage: HSp3.level()
3
```

LLT_generic.**llt_family**()

The family of the llt bases of the symmetric functions.

INPUT:

- self – an instance of the LLT hspin or hcospin basis

OUTPUT:

- returns an instance of the family of LLT bases associated to self.

EXAMPLES:

```
sage: HSp3 = SymmetricFunctions(FractionField(QQ['t'])).llt(3).hspin()
sage: HSp3.llt_family()
level 3 LLT polynomials over Fraction Field of Univariate Polynomial Ring in t over Rational
```

class sage.combinat.sf.llt.**LLT_spin**(*llt*)

Bases: sage.combinat.sf.llt.LLT_generic

A class of methods for the h-spin LLT basis of the symmetric functions.

INPUT:

- self – an instance of the LLT hcospin basis
- llt – a family of LLT symmetric function bases

TESTS:

```
sage: HSp3 = SymmetricFunctions(FractionField(QQ['t'])).llt(3).hspin()
sage: TestSuite(HSp3).run(skip = ["_test_associativity", "_test_distributivity", "_test_prod"])
sage: TestSuite(HSp3).run(elements = [HSp3.t*HSp3[1,1]+HSp3.t*HSp3[2], HSp3[1]+(1+HSp3.t)*HSp3[1

sage: HS3t2 = SymmetricFunctions(QQ).llt(3,t=2).hspin()
sage: TestSuite(HS3t2).run() # products are too expensive, long time (7s on sage.math, 2012)

sage: HS3x = SymmetricFunctions(FractionField(QQ['x'])).llt(3,t=x).hspin()
sage: TestSuite(HS3x).run(skip = ["_test_associativity", "_test_distributivity", "_test_prod"])
sage: TestSuite(HS3x).run(elements = [HS3x.t*HS3x[1,1]+HS3x.t*HS3x[2], HS3x[1]+(1+HS3x.t)*HS3x[1
```

class Element(*M*, *x*)

Bases: sage.combinat.sf.llt.LLT_generic.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
```

```
sage: f == loads(dumps(f))
True
```

5.1.257 Macdonald Polynomials

Notation used in the definitions follows mainly [Macdonald1995].

The integral forms of the bases H and Ht do not appear in Macdonald's book. They correspond to the two bases $H_\mu[X;q,t] = \sum_\nu K_{\nu\mu}(q,t)s_\mu[X]$ and $\tilde{H}_\mu[X;q,t] = t^{n(\mu)}\sum_\nu K_{\nu\mu}(q,1/t)s_\nu[X]$ where $K_{\mu\nu}(q,t)$ are the Macdonald q,t-Koskta coefficients.

The Ht in this case is short for $\tilde{H}$ and is the basis which is the graded Frobenius image of the Garsia-Haiman modules [GH1993].

REFERENCES:

class sage.combinat.sf.macdonald.**Macdonald**(*Sym*, *q='q'*, *t='t'*)

Bases: sage.structure.unique_representation.UniqueRepresentation

Macdonald Symmetric functions including P, Q, J, H, Ht bases also including the S basis which is the plethystic transformation of the Schur basis (that which is dual to the Schur basis with respect to the Macdonald q,t-scalar product)

INPUT:

- self – a family of Macdonald symmetric function bases

EXAMPLES

```
sage: t = QQ['t'].gen(); SymmetricFunctions(QQ['t'].fraction_field()).macdonald(q=t,t=1)
Macdonald polynomials with q=t and t=1 over Fraction Field of Univariate Polynomial Ring in t ov
sage: Sym = SymmetricFunctions(FractionField(QQ['t'])).macdonald()
Traceback (most recent call last):
...
TypeError: unable to evaluate 'q' in Fraction Field of Univariate Polynomial Ring in t over Rati
```

H()

Returns the Macdonald polynomials on the H basis. When the H basis is expanded on the Schur basis, the coefficients are the qt-Kostka numbers.

INPUT:

- self – a family of Macdonald symmetric function bases

OUTPUT:

- returns the H Macdonald basis of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: H = Sym.macdonald().H(); H
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q, t over Rationa
sage: s = Sym.schur()
sage: s(H([2]))
q*s[1, 1] + s[2]
sage: s(H([1,1]))
s[1, 1] + t*s[2]
```

Coercions to/from the Schur basis are implemented:

```
sage: H = Sym.macdonald().H()
sage: s = Sym.schur()
sage: H(s([2]))
(q/(q*t-1))*McdH[1, 1] - (1/(q*t-1))*McdH[2]
```

Ht()

Returns the Macdonald polynomials on the Ht basis. The elements of the Ht basis are eigenvectors of the $nabla$ operator. When expanded on the Schur basis, the coefficients are the modified qt-Kostka numbers.

INPUT:

- `self` – a family of Macdonald symmetric function bases

OUTPUT:

- returns the Ht Macdonald basis of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: Ht = Sym.macdonald().Ht(); Ht
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q, t over Rationa
sage: [Ht(p).nabla() for p in Partitions(3)]
[q^3*McdHt[3], q*t*McdHt[2, 1], t^3*McdHt[1, 1, 1]]

sage: s = Sym.schur()
sage: from sage.combinat.sf.macdonald import qt_kostka
sage: q,t = Ht.base_ring().gens()
sage: s(Ht([2,1]))
q*t*s[1, 1, 1] + (q+t)*s[2, 1] + s[3]
sage: qt_kostka([1,1,1],[2,1]).subs(t=1/t)*t^Partition([2,1]).weighted_size()
q*t
sage: qt_kostka([2,1],[2,1]).subs(t=1/t)*t^Partition([2,1]).weighted_size()
q + t
sage: qt_kostka([3],[2,1]).subs(t=1/t)*t^Partition([2,1]).weighted_size()
1
```

Coercions to/from the Schur basis are implemented:

```
sage: Ht = Sym.macdonald().Ht()
sage: s = Sym.schur()
sage: Ht(s([2,1]))
((-q)/(-q*t^2+t^3+q^2-q*t))*McdHt[1, 1, 1] + ((q^2+q*t+t^2)/(-q^2*t^2+q^3+t^3-q*t))*McdHt[2,
sage: Ht(s([2]))
((-q)/(-q+t))*McdHt[1, 1] + (t/(-q+t))*McdHt[2]
```

J()

Returns the Macdonald polynomials on the J basis also known as the integral form of the Macdonald polynomials. These are scalar multiples of both the P and Q bases. When expressed in the P or Q basis, the scaling coefficients are polynomials in q and t rather than rational functions.

The J basis is calculated using determinantal formulas of Lapointe-Lascoux-Morse giving the action on the S-basis [LLM1998].

INPUT:

- `self` – a family of Macdonald symmetric function bases

OUTPUT:

- returns the J Macdonald basis of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: J = Sym.macdonald().J(); J
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q, t over Rationa
sage: P = Sym.macdonald().P()
sage: Q = Sym.macdonald().Q()
sage: P(J([2]))
(q*t^2-q*t-t+1)*McdP[2]
sage: P(J([1,1]))
(t^3-t^2-t+1)*McdP[1, 1]
sage: Q(J([2]))
(q^3-q^2-q+1)*McdQ[2]
sage: Q(J([1,1]))
(q^2*t-q*t-q+1)*McdQ[1, 1]
```

Coercions from the Q and J basis (proportional) and to/from the Schur basis are implemented:

```
sage: P = Sym.macdonald().P()
sage: Q = Sym.macdonald().Q()
sage: J = Sym.macdonald().J()
sage: s = Sym.schur()

sage: J(P([2]))
(1/(q*t^2-q*t-t+1))*McdJ[2]

sage: J(Q([2]))
(1/(q^3-q^2-q+1))*McdJ[2]

sage: s(J([2]))
(-q*t+t^2+q-t)*s[1, 1] + (q*t^2-q*t-t+1)*s[2]
sage: J(s([2]))
((q-t)/(q*t^4-q*t^3-q*t^2-t^3+q*t+t^2+t-1))*McdJ[1, 1] + (1/(q*t^2-q*t-t+1))*McdJ[2]
```

P()

Returns Macdonald polynomials in P basis. The P basis is defined here as a normalized form of the J basis.

INPUT:

- `self` – a family of Macdonald symmetric function bases

OUTPUT:

- returns the P Macdonald basis of symmetric functions

EXAMPLES

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P(); P
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q, t over Rationa
sage: P[2]
McdP[2]
```

The P Macdonald basis is upper triangularly related to the monomial symmetric functions and are orthogonal with respect to the qt-Hall scalar product:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P(); P
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q, t over Rationa
sage: m = Sym.monomial()
sage: P.transition_matrix(m,2)
```

```
[                          1 (q*t - q + t - 1)/(q*t - 1)]
[                          0                         1]
sage: P([1,1]).scalar_qt(P([2]))
0
sage: P([2]).scalar_qt(P([2]))
(-q^3 + q^2 + q - 1)/(-q*t^2 + q*t + t - 1)
sage: P([1,1]).scalar_qt(P([1,1]))
(-q^2*t + q*t + q - 1)/(-t^3 + t^2 + t - 1)
```

When $q = 0$, the Macdonald polynomials on the P basis are the same as the Hall-Littlewood polynomials on the P basis.

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: P = Sym.macdonald(q=0).P(); P
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: P([2])^2
(t+1)*McdP[2, 2] + (-t+1)*McdP[3, 1] + McdP[4]
sage: HLP = Sym.hall_littlewood().P()
sage: HLP([2])^2
(t+1)*HLP[2, 2] + (-t+1)*HLP[3, 1] + HLP[4]
```

Coercions from the Q and J basis (proportional) are implemented:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P()
sage: Q = Sym.macdonald().Q()
sage: J = Sym.macdonald().J()
sage: s = Sym.schur()

sage: P(Q([2]))
((q*t^2-q*t-t+1)/(q^3-q^2-q+1))*McdP[2]
sage: P(Q([2,1]))
((-q*t^4+2*q*t^3-q*t^2+t^2-2*t+1)/(-q^4*t+2*q^3*t-q^2*t+q^2-2*q+1))*McdP[2, 1]

sage: P(J([2]))
(q*t^2-q*t-t+1)*McdP[2]
sage: P(J([2,1]))
(-q*t^4+2*q*t^3-q*t^2+t^2-2*t+1)*McdP[2, 1]
```

By transitivity, one get coercions from the classical bases:

```
sage: P(s([2]))
((q-t)/(q*t-1))*McdP[1, 1] + McdP[2]
sage: P(s([2,1]))
((q*t-t^2+q-t)/(q*t^2-1))*McdP[1, 1, 1] + McdP[2, 1]

sage: Sym = SymmetricFunctions(QQ['x','y','z'].fraction_field())
sage: (x,y,z) = Sym.base_ring().gens()
sage: Macxy = Sym.macdonald(q=x,t=y)
sage: Macyz = Sym.macdonald(q=y,t=z)
sage: Maczx = Sym.macdonald(q=z,t=x)
sage: P1 = Macxy.P()
sage: P2 = Macyz.P()
sage: P3 = Maczx.P()
sage: m(P1[2,1])
((-2*x*y^2+x*y-y^2+x-y+2)/(-x*y^2+1))*m[1, 1, 1] + m[2, 1]
sage: m(P2[2,1])
((-2*y*z^2+y*z-z^2+y-z+2)/(-y*z^2+1))*m[1, 1, 1] + m[2, 1]
sage: m(P1(P2(P3[2,1])))
```

```
((-2*x^2*z-x^2+x*z-x+z+2)/(-x^2*z+1))*m[1, 1, 1] + m[2, 1]
sage: P1(P2[2])
((-x*y^2+2*x*y*z-y^2*z-x+2*y-z)/(x*y^2*z-x*y-y*z+1))*McdP[1, 1] + McdP[2]
sage: m(z*P1[2]+x*P2[2])
((x^2*y^2*z+x*y^2*z^2-x^2*y^2+x^2*y*z-x*y*z^2+y^2*z^2-x^2*y-2*x*y*z-y*z^2+x*y-y*z+x+z)/(x*y^
```

Q()

Returns the Macdonald polynomials on the Q basis. These are dual to the Macdonald polynomials on the P basis with respect to the qt-Hall scalar product. The Q basis is defined to be a normalized form of the J basis.

INPUT:

- `self` – a family of Macdonald symmetric function bases

OUTPUT:

- returns the Q Macdonald basis of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: Q = Sym.macdonald().Q(); Q
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q, t over Rationa
sage: P = Sym.macdonald().P()
sage: Q([2]).scalar_qt(P([2]))
1
sage: Q([2]).scalar_qt(P([1,1]))
0
sage: Q([1,1]).scalar_qt(P([2]))
0
sage: Q([1,1]).scalar_qt(P([1,1]))
1
sage: Q(P([2]))
((q^3-q^2-q+1)/(q*t^2-q*t-t+1))*McdQ[2]
sage: Q(P([1,1]))
((q^2*t-q*t-q+1)/(t^3-t^2-t+1))*McdQ[1, 1]
```

Coercions from the P and J basis (proportional) are implemented:

```
sage: P = Sym.macdonald().P()
sage: Q = Sym.macdonald().Q()
sage: J = Sym.macdonald().J()
sage: s = Sym.schur()

sage: Q(J([2]))
(q^3-q^2-q+1)*McdQ[2]

sage: Q(P([2]))
((q^3-q^2-q+1)/(q*t^2-q*t-t+1))*McdQ[2]
sage: P(Q(P([2])))
McdP[2]
sage: Q(P(Q([2])))
McdQ[2]
```

By transitivity, one gets coercions from the classical bases:

```
sage: Q(s([2]))
((q^2-q*t-q+t)/(t^3-t^2-t+1))*McdQ[1, 1] + ((q^3-q^2-q+1)/(q*t^2-q*t-t+1))*McdQ[2]
```

S()

Returns the modified Schur functions defined by the plethystic substitution $S_\mu = s_\mu[X(1-t)/(1-q)]$. When the Macdonald polynomials in the J basis are expressed in terms of the modified Schur functions at $q = 0$, the coefficients are qt-Kostka numbers.

INPUT:

- `self` – a family of Macdonald symmetric function bases

OUTPUT:

- returns the S Macdonald basis of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: S = Sym.macdonald().S(); S
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q, t over Rationa
sage: p = Sym.power()
sage: p(S[2,1])
((1/3*t^3-t^2+t-1/3)/(q^3-3*q^2+3*q-1))*p[1, 1, 1] + ((-1/3*t^3+1/3)/(q^3-1))*p[3]
sage: J = Sym.macdonald().J()
sage: S(J([2]))
(q^3-q^2-q+1)*McdS[2]
sage: S(J([1,1]))
(q^2*t-q*t-q+1)*McdS[1, 1] + (q^2-q*t-q+t)*McdS[2]
sage: S = Sym.macdonald(q=0).S()
sage: S(J[1,1])
McdS[1, 1] + t*McdS[2]
sage: S(J[2])
q*McdS[1, 1] + McdS[2]
sage: p(S[2,1])
(-1/3*t^3+t^2-t+1/3)*p[1, 1, 1] + (1/3*t^3-1/3)*p[3]

sage: from sage.combinat.sf.macdonald import qt_kostka
sage: qt_kostka([2],[1,1])
t
sage: qt_kostka([1,1],[2])
q
```

Coercions to/from the Schur basis are implemented:

```
sage: S = Sym.macdonald().S()
sage: s = Sym.schur()
sage: S(s([2]))
((q^2-q*t-q+t)/(t^3-t^2-t+1))*McdS[1, 1] + ((-q^2*t+q*t+q-1)/(-t^3+t^2+t-1))*McdS[2]
sage: s(S([1,1]))
((-q*t^2+q*t+t-1)/(-q^3+q^2+q-1))*s[1, 1] + ((q*t-t^2-q+t)/(-q^3+q^2+q-1))*s[2]
```

base_ring()

Returns the base ring of the symmetric functions where the Macdonald symmetric functions live

INPUT:

- `self` – a family of Macdonald symmetric function bases

OUTPUT:

- the base ring associated to the corresponding symmetric function ring

EXAMPLES

```
sage: Sym = SymmetricFunctions(QQ['q'].fraction_field())
sage: Mac = Sym.macdonald(t=0)
sage: Mac.base_ring()
Fraction Field of Univariate Polynomial Ring in q over Rational Field
```

symmetric_function_ring()

Returns the base ring of the symmetric functions where the Macdonald symmetric functions live

INPUT:

- self – a family of Macdonald symmetric function bases

OUTPUT:

- the symmetric function ring associated to the Macdonald bases

EXAMPLES

```
sage: Mac = SymmetricFunctions(QQ['q'].fraction_field()).macdonald(t=0)
sage: Mac.symmetric_function_ring()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in q over Rational Fie
```

class sage.combinat.sf.macdonald.**MacdonaldPolynomials_generic**(*macdonald*)

Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic

A class for methods for one of the Macdonald bases of the symmetric functions

INPUT:

- self – a Macdonald basis
- macdonald – a family of Macdonald symmetric function bases

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q,t'])); Sym.rename("Sym"); Sym
Sym
sage: Sym.macdonald().P()
Sym in the Macdonald P basis
sage: Sym.macdonald(t=2).P()
Sym in the Macdonald P with t=2 basis
sage: Sym.rename()
```

TESTS:

```
sage: Sym.macdonald().P()._prefix
'McdP'
sage: Sym.macdonald().Ht()._prefix
'McdHt'
```

class Element(*M*, *x*)

Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic_Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

nabla (*q=None*, *t=None*, *power=1*)

Return the value of the nabla operator applied to `self`.

The eigenvectors of the nabla operator are the Macdonald polynomials in the Ht basis. For more information see: [BGHT1999].

The operator nabla acts on symmetric functions and has the Macdonald Ht basis as eigenfunctions and the eigenvalues are $q^{n(\mu')}t^{n(\mu)}$ where $n(\mu) = \sum_i (i-1)\mu_i$ and μ' is the conjugate shape of μ.

If the parameter `power` is an integer then it calculates nabla to that integer. The default value of `power` is 1.

INPUT:

- `self` – an element of a Macdonald basis
- `q`, `t` – optional parameters to specialize
- `power` – an integer (default: 1)

OUTPUT:

- returns the symmetric function of ∇ acting on `self`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P()
sage: P([1,1]).nabla()
((q^2*t+q*t^2-2*t)/(q*t-1))*McdP[1, 1] + McdP[2]
sage: P([1,1]).nabla(t=1)
((q^2*t+q*t-t-1)/(q*t-1))*McdP[1, 1] + McdP[2]
sage: H = Sym.macdonald().H()
sage: H([1,1]).nabla()
t*McdH[1, 1] + (-t^2+1)*McdH[2]
sage: H([1,1]).nabla(q=1)
((t^2+q-t-1)/(q*t-1))*McdH[1, 1] + ((-t^3+t^2+t-1)/(q*t-1))*McdH[2]
sage: H(0).nabla()
0
sage: H([2,2,1]).nabla(t=1/H.t)
((-q^2)/(-t^4))*McdH[2, 2, 1]
sage: H([2,2,1]).nabla(t=1/H.t,power=-1)
((-t^4)/(-q^2))*McdH[2, 2, 1]
```

MacdonaldPolynomials_generic.**c1** (*part*)

Returns the qt-Hall scalar product between `J(part)` and `P(part)`.

INPUT:

- `self` – a Macdonald basis
- `part` – a partition

OUTPUT:

- returns the qt-Hall scalar product between `J(part)` and `P(part)`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P()
sage: P.c1(Partition([2,1]))
-q^4*t + 2*q^3*t - q^2*t + q^2 - 2*q + 1
```

MacdonaldPolynomials_generic.**c2** (*part*)

Returns the qt-Hall scalar product between `J(part)` and `Q(part)`.

INPUT:

- `self` – a Macdonald basis
- `part` – a partition

OUTPUT:

- returns the qt-Hall scalar product between `J(part)` and `Q(part)`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P()
sage: P.c2(Partition([2,1]))
-q*t^4 + 2*q*t^3 - q*t^2 + t^2 - 2*t + 1
```

MacdonaldPolynomials_generic.**macdonald_family**()

Returns the family of Macdonald bases associated to the basis `self`

INPUT:

- `self` – a Macdonald basis

OUTPUT:

- the family of Macdonald symmetric functions associated to `self`

EXAMPLES

```
sage: MacP = SymmetricFunctions(QQ['q'].fraction_field()).macdonald(t=0).P()
sage: MacP.macdonald_family()
Macdonald polynomials with t=0 over Fraction Field of Univariate Polynomial Ring in q over R
```

class sage.combinat.sf.macdonald.**MacdonaldPolynomials_h**(*macdonald*)

Bases: sage.combinat.sf.macdonald.MacdonaldPolynomials_generic

The H basis is defined as $H_\mu = \sum_\lambda K_{\lambda\mu}(q,t)s_\lambda$ where $K_{\lambda\mu}(q,t)$ are the Macdonald Kostka coefficients.

In this implementation, it is calculated by using the Macdonald Ht basis and substituting $t \to 1/t$ and multiplying by $t^{n(\mu)}$.

INPUT:

- `self` – a Macdonald H basis
- `macdonald` – a family of Macdonald bases

TESTS:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: H = Sym.macdonald().H()
sage: TestSuite(H).run(skip=["_test_associativity","_test_distributivity","_test_prod"])
sage: TestSuite(H).run(elements = [H.t*H[1,1]+H.q*H[2], H[1]+(H.q+H.t)*H[1,1]])  # long time (26
```

class **Element**(*M*, *x*)

Bases: sage.combinat.sf.macdonald.MacdonaldPolynomials_generic.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
```

```
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.sf.macdonald.**MacdonaldPolynomials_ht**(*macdonald*)

Bases: sage.combinat.sf.macdonald.MacdonaldPolynomials_generic

The Ht basis is defined as $\tilde{H}_\mu = t^{n(\mu)} \sum_\lambda K_{\lambda\mu}(q, t^{-1}) s_\lambda$ where $K_{\lambda\mu}(q,t)$ are the Macdonald (q,t)-Kostka coefficients and $n(\mu) = \sum_i (i-1)\mu_i$.

It is implemented here by using a Pieri formula due to F. Bergeron and M. Haiman [BH2013].

INPUT:

- self – a Macdonald Ht basis
- macdonald – a family of Macdonald bases

TESTS:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: Ht = Sym.macdonald().Ht()
sage: TestSuite(Ht).run(skip=["_test_associativity","_test_distributivity","_test_prod"])  # lon
sage: TestSuite(Ht).run(elements = [Ht.t*Ht[1,1]+Ht.q*Ht[2], Ht[1]+(Ht.q+Ht.t)*Ht[1,1]])  # long
```

class **Element**(*M*, *x*)

Bases: sage.combinat.sf.macdonald.MacdonaldPolynomials_generic.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

nabla(*q=None*, *t=None*, *power=1*)

Returns the value of the nabla operator applied to self. The eigenvectors of the $nabla$ operator are the Macdonald polynomials in the Ht basis. For more information see: [BGHT1999].

The operator $nabla$ acts on symmetric functions and has the Macdonald Ht basis as eigenfunctions and the eigenvalues are $q^{n(\mu')}t^{n(\mu)}$ where $n(\mu) = \sum_i (i-1)\mu_i$.

If the parameter power is an integer then it calculates nabla to that integer. The default value of power is 1.

INPUT:

- self – an element of the Macdonald Ht basis
- q, t – optional parameters to specialize
- power – an integer (default: 1)

OUTPUT:

- returns the symmetric function of ∇ acting on self

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: Ht = Sym.macdonald().Ht()
sage: t = Ht.t; q = Ht.q;
sage: s = Sym.schur()
```

```
sage: a = sum(Ht(p) for p in Partitions(3))
sage: Ht(0).nabla()
0
sage: a.nabla() == t^3*Ht([1,1,1])+q*t*Ht([2,1]) + q^3*Ht([3])
True
sage: a.nabla(t=3) == 27*Ht([1,1,1])+3*q*Ht([2,1]) + q^3*Ht([3])
True
sage: a.nabla(q=3) == t^3*Ht([1,1,1])+3*t*Ht([2,1]) + 27*Ht([3])
True
sage: Ht[2,1].nabla(power=-1)
1/(q*t)*McdHt[2, 1]
sage: Ht[2,1].nabla(power=4)
q^4*t^4*McdHt[2, 1]
sage: s(a.nabla(q=3))
(t^6+27*q^3+3*q*t^2)*s[1, 1, 1] + (t^5+t^4+27*q^2+3*q*t+3*t^2+27*q)*s[2, 1] + (t^3+3*t+2
sage: Ht = Sym.macdonald(q=3).Ht()
sage: a = sum(Ht(p) for p in Partitions(3))
sage: s(a.nabla())
(t^6+9*t^2+729)*s[1, 1, 1] + (t^5+t^4+3*t^2+9*t+324)*s[2, 1] + (t^3+3*t+27)*s[3]
```

class sage.combinat.sf.macdonald.**MacdonaldPolynomials_j**(*macdonald*)

Bases: sage.combinat.sf.macdonald.MacdonaldPolynomials_generic

The J basis is calculated using determinantal formulas of Lapointe-Lascoux-Morse giving the action on the S-basis.

INPUT:

- self – a Macdonald J basis
- macdonald – a family of Macdonald bases

TESTS:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: J = Sym.macdonald().J()
sage: TestSuite(J).run(skip=["_test_associativity","_test_distributivity","_test_prod"])  # long
sage: TestSuite(J).run(elements = [J.t*J[1,1]+J.q*J[2], J[1]+(J.q+J.t)*J[1,1]])  # long time (de
```

class Element(*M*, *x*)

Bases: sage.combinat.sf.macdonald.MacdonaldPolynomials_generic.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.sf.macdonald.**MacdonaldPolynomials_p**(*macdonald*)

Bases: sage.combinat.sf.macdonald.MacdonaldPolynomials_generic

The P basis is defined here as the J basis times a normalizing coefficient $c2$.

INPUT:

- self – a Macdonald P basis

- macdonald – a family of Macdonald bases

TESTS:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P()
sage: TestSuite(P).run(skip=["_test_associativity","_test_distributivity","_test_prod"])  # long
sage: TestSuite(P).run(elements = [P.t*P[1,1]+P.q*P[2], P[1]+(P.q+P.t)*P[1,1]])  # long time (de
```

class Element(*M*, *x*)

Bases: sage.combinat.sf.macdonald.MacdonaldPolynomials_generic.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

MacdonaldPolynomials_p.**scalar_qt_basis**(*part1*, *part2=None*)

Returns the scalar product of $P(part1)$ and $P(part2)$ This scalar product formula is given in equation (4.11) p.323 and (6.19) p.339 of Macdonald's book [Macdonald1995].

INPUT:

- self – a Macdonald P basis
- part1, part2 – partitions

OUTPUT:

- returns the scalar product of P(part1) and P(part2)

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P()
sage: P.scalar_qt_basis(Partition([2,1]), Partition([1,1,1]))
0
sage: f = P.scalar_qt_basis(Partition([3,2,1]), Partition([3,2,1]))
sage: factor(f.numerator())
(q - 1)^3 * (q^2*t - 1)^2 * (q^3*t^2 - 1)
sage: factor(f.denominator())
(t - 1)^3 * (q*t^2 - 1)^2 * (q^2*t^3 - 1)
```

With a single argument, takes $part2 = part1$:

```
sage: P.scalar_qt_basis(Partition([2,1]), Partition([2,1]))
(-q^4*t + 2*q^3*t - q^2*t + q^2 - 2*q + 1)/(-q*t^4 + 2*q*t^3 - q*t^2 + t^2 - 2*t + 1)
```

class sage.combinat.sf.macdonald.**MacdonaldPolynomials_q**(*macdonald*)

Bases: sage.combinat.sf.macdonald.MacdonaldPolynomials_generic

The Q basis is defined here as the J basis times a normalizing coefficient.

INPUT:

- self – a Macdonald Q basis

- `macdonald` – a family of Macdonald bases

TESTS:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: Q = Sym.macdonald().Q()
sage: TestSuite(Q).run(skip=["_test_associativity","_test_distributivity","_test_prod"])  # long
sage: TestSuite(Q).run(elements = [Q.t*Q[1,1]+Q.q*Q[2], Q[1]+(Q.q+Q.t)*Q[1,1]])  # long time (de
```

class **Element**(M, x)

Bases: `sage.combinat.sf.macdonald.MacdonaldPolynomials_generic.Element`

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

class sage.combinat.sf.macdonald.**MacdonaldPolynomials_s**(*macdonald*)

Bases: `sage.combinat.sf.macdonald.MacdonaldPolynomials_generic`

An implementation of the basis $s_\lambda[(1-t)X/(1-q)]$

This is perhaps misnamed as a 'Macdonald' basis for the symmetric functions but is used in the calculation of the Macdonald J basis (see method 'creation' below) but does use both of the two parameters and can be specialized to $s_\lambda[(1-t)X]$ and $s_\lambda[X/(1-t)]$.

INPUT:

- `self` – a Macdonald S basis
- `macdonald` – a family of Macdonald bases

TESTS:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: S = Sym.macdonald().S()
sage: TestSuite(S).run(skip=["_test_associativity","_test_distributivity","_test_prod"])
sage: TestSuite(S).run(elements = [S.t*S[1,1]+S.q*S[2], S[1]+(S.q+S.t)*S[1,1]])
```

class **Element**(M, x)

Bases: `sage.combinat.sf.macdonald.MacdonaldPolynomials_generic.Element`

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

creation(k)

This function is a creation operator for the J-basis for which the action is known on the 'Macdonald' S-basis by formula from [LLM1998].

INPUT:

- self – an element of the Macdonald S basis
- k – a positive integer

OUTPUT:

- returns the column adding operator on the J basis on self

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: S = Sym.macdonald().S()
sage: a = S(1)
sage: a.creation(1)
(-q+1)*McdS[1]
sage: a.creation(2)
(q^2*t-q*t-q+1)*McdS[1, 1] + (q^2-q*t-q+t)*McdS[2]
```

sage.combinat.sf.macdonald.**c1**(*part*, *q*, *t*)

This function returns the qt-Hall scalar product between J(part) and P(part).

This coefficient is c_λ in equation (8.1') p. 352 of Macdonald's book [Macdonald1995].

INPUT:

- part – a partition
- q, t – parameters

OUTPUT:

- returns a polynomial of the scalar product between the J and P bases

EXAMPLES:

```
sage: from sage.combinat.sf.macdonald import c1
sage: R.<q,t> = QQ[]
sage: c1(Partition([2,1]),q,t)
-q^4*t + 2*q^3*t - q^2*t + q^2 - 2*q + 1
sage: c1(Partition([1,1]),q,t)
q^2*t - q*t - q + 1
```

sage.combinat.sf.macdonald.**c2**(*part*, *q*, *t*)

This function returns the qt-Hall scalar product between J(part) and Q(part).

This coefficient is c_λ in equation (8.1) p. 352 of Macdonald's book [Macdonald1995].

INPUT:

- part – a partition
- q, t – parameters

OUTPUT:

- returns a polynomial of the scalar product between the J and P bases

EXAMPLES:

```
sage: from sage.combinat.sf.macdonald import c2
sage: R.<q,t> = QQ[]
sage: c2(Partition([1,1]),q,t)
t^3 - t^2 - t + 1
sage: c2(Partition([2,1]),q,t)
-q*t^4 + 2*q*t^3 - q*t^2 + t^2 - 2*t + 1
```

sage.combinat.sf.macdonald.**cmunu**(*mu*, *nu*)

Return the coefficient of $\tilde{H}_\nu$ in $h_r^\perp \tilde{H}_\mu$.

Proposition 5 of F. Bergeron and M. Haiman [BH2013] states

$$c_{\mu\nu} = \sum_{\alpha \leftarrow \nu} c_{\mu\alpha} c_{\alpha\nu} B_{\alpha/\nu} / B_{\mu/\nu}$$

where $c_{\mu\nu}$ is the coefficient of $\tilde{H}_\nu$ in $h_r^\perp \tilde{H}_\mu$ and $B_{\mu/\nu}$ is the bi-exponent generator implemented in the function `sage.combinat.sf.macdonald.Bmu()`.

INPUT:

- `mu`, `nu` – partitions with `nu` contained in `mu`

OUTPUT:

- an element of the fraction field of polynomials in q and t

EXAMPLES:

```
sage: from sage.combinat.sf.macdonald import cmunu
sage: cmunu(Partition([2,1]),Partition([1]))
q + t + 1
sage: cmunu(Partition([2,2]),Partition([1,1]))
(-q^3 - q^2 + q*t + t)/(-q + t)
sage: Sym = SymmetricFunctions(QQ['q','t'].fraction_field())
sage: h = Sym.h()
sage: Ht = Sym.macdonald().Ht()
sage: all(Ht[2,2].skew_by(h[r]).coefficient(nu)
....:     == cmunu(Partition([2,2]),nu)
....:     for r in range(1,5) for nu in Partitions(4-r))
True
```

`sage.combinat.sf.macdonald.`**`cmunu1`**(*mu*, *nu*)

Return the coefficient of $\tilde{H}_\nu$ in $h_1^\perp \tilde{H}_\mu$.

INPUT:

- `mu`, `nu` – partitions with `nu` preceeds `mu`

OUTPUT:

- an element of the fraction field of polynomials in q and t

EXAMPLES:

```
sage: from sage.combinat.sf.macdonald import cmunu1
sage: cmunu1(Partition([2,1]),Partition([2]))
(-t^2 + q)/(q - t)
sage: cmunu1(Partition([2,1]),Partition([1,1]))
(-q^2 + t)/(-q + t)
sage: Sym = SymmetricFunctions(QQ['q','t'].fraction_field())
sage: h = Sym.h()
sage: Ht = Sym.macdonald().Ht()
sage: all(Ht[3,2,1].skew_by(h[1]).coefficient(nu)
....:     == cmunu1(Partition([3,2,1]),nu)
....:     for nu in Partition([3,2,1]).down_list())
True
```

`sage.combinat.sf.macdonald.`**`qt_kostka`**(*lam*, *mu*)

Returns the $K_{\lambda\mu}(q,t)$ by computing the change of basis from the Macdonald H basis to the Schurs.

INPUT:

- `lam`, `mu` – partitions of the same size

OUTPUT:

- returns the q,t-Kostka polynomial indexed by the partitions `lam` and `mu`

EXAMPLES:

```
sage: from sage.combinat.sf.macdonald import qt_kostka
sage: qt_kostka([2,1,1],[1,1,1,1])
t^3 + t^2 + t
sage: qt_kostka([1,1,1,1],[2,1,1])
q
sage: qt_kostka([1,1,1,1],[3,1])
q^3
sage: qt_kostka([1,1,1,1],[1,1,1,1])
1
sage: qt_kostka([2,1,1],[2,2])
q^2*t + q*t + q
sage: qt_kostka([2,2],[2,2])
q^2*t^2 + 1
sage: qt_kostka([4],[3,1])
t
sage: qt_kostka([2,2],[3,1])
q^2*t + q
sage: qt_kostka([3,1],[2,1,1])
q*t^3 + t^2 + t
sage: qt_kostka([2,1,1],[2,1,1])
q*t^2 + q*t + 1
sage: qt_kostka([2,1],[1,1,1,1])
0
```

5.1.258 Monomial symmetric functions

class sage.combinat.sf.monomial.**SymmetricFunctionAlgebra_monomial**(*Sym*)

Bases: sage.combinat.sf.classical.SymmetricFunctionAlgebra_classical

A class for methods related to monomial symmetric functions

INPUT:

- `self` – a monomial symmetric function basis
- `Sym` – an instance of the ring of the symmetric functions

TESTS:

```
sage: m = SymmetricFunctions(QQ).m()
sage: m == loads(dumps(m))
True
sage: TestSuite(m).run(skip=['_test_associativity', '_test_distributivity', '_test_prod'])
sage: TestSuite(m).run(elements = [m[1,1]+m[2], m[1]+2*m[1,1]])
```

class **Element**(*M*, *x*)

Bases: sage.combinat.sf.classical.SymmetricFunctionAlgebra_classical.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
```

```
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

expand(*n*, *alphabet='x'*)

Expand the symmetric function `self` as a symmetric polynomial in `n` variables.

INPUT:

- `n` – a nonnegative integer
- `alphabet` – (default: `'x'`) a variable for the expansion

OUTPUT:

A monomial expansion of `self` in the n variables labelled by `alphabet`.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
sage: m([2,1]).expand(3)
x0^2*x1 + x0*x1^2 + x0^2*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
sage: m([1,1,1]).expand(2)
0
sage: m([2,1]).expand(3,alphabet='z')
z0^2*z1 + z0*z1^2 + z0^2*z2 + z1^2*z2 + z0*z2^2 + z1*z2^2
sage: m([2,1]).expand(3,alphabet='x,y,z')
x^2*y + x*y^2 + x^2*z + y^2*z + x*z^2 + y*z^2
sage: m([1]).expand(0)
0
sage: (3*m([])).expand(0)
3
```

SymmetricFunctionAlgebra_monomial.**antipode_by_coercion**(*element*)

The antipode of `element` via coercion to and from the power-sum basis or the Schur basis (depending on whether the power sums really form a basis over the given ground ring).

INPUT:

- `element` – element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.monomial()
sage: m[3,2].antipode()
m[3, 2] + 2*m[5]
sage: m.antipode_by_coercion(m[3,2])
m[3, 2] + 2*m[5]

sage: Sym = SymmetricFunctions(ZZ)
sage: m = Sym.monomial()
sage: m[3,2].antipode()
m[3, 2] + 2*m[5]
sage: m.antipode_by_coercion(m[3,2])
m[3, 2] + 2*m[5]
```

Todo

Is there a not too difficult way to get the power-sum computations to work over any ring, not just one with coercion from **Q**?

SymmetricFunctionAlgebra_monomial.**from_polynomial**(*f*, *check=True*)

Returns the symmetric function in the monomial basis corresponding to the polynomial `f`.

INPUT:

- `self` – a monomial symmetric function basis
- `f` – a polynomial in finitely many variables over the same base ring as `self`. It is assumed that this polynomial is symmetric.
- `check` – boolean (default: `True`), checks whether the polynomial is indeed symmetric

OUTPUT:

- This function converts a symmetric polynomial f in a polynomial ring in finitely many variables to a symmetric function in the monomial basis of the ring of symmetric functions over the same base ring.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
sage: P = PolynomialRing(QQ, 'x', 3)
sage: x = P.gens()
sage: f = x[0] + x[1] + x[2]
sage: m.from_polynomial(f)
m[1]
sage: f = x[0]**2+x[1]**2+x[2]**2
sage: m.from_polynomial(f)
m[2]
sage: f=x[0]^2+x[1]
sage: m.from_polynomial(f)
Traceback (most recent call last):
...
ValueError: x0^2 + x1 is not a symmetric polynomial
sage: f = (m[2,1]+m[1,1]).expand(3)
sage: m.from_polynomial(f)
m[1, 1] + m[2, 1]
sage: f = (2*m[2,1]+m[1,1]+3*m[3]).expand(3)
sage: m.from_polynomial(f)
m[1, 1] + 2*m[2, 1] + 3*m[3]
```

SymmetricFunctionAlgebra_monomial.**from_polynomial_exp**(*p*)

Conversion from polynomial in exponential notation

INPUT:

- `self` – a monomial symmetric function basis
- `p` – a multivariate polynomial over the same base ring as `self`

OUTPUT:

- This returns a symmetric function by mapping each monomial of p with exponents `exp` into m_λ where λ is the partition with exponential notation `exp`.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
sage: P = PolynomialRing(QQ,'x',5)
sage: x = P.gens()
```

The exponential notation of the partition $(5,5,5,3,1,1)$ is:

```
sage: Partition([5,5,5,3,1,1]).to_exp()
[2, 0, 1, 0, 3]
```

Therefore, the monomial:

```
sage: f = x[0]^2 * x[2] * x[4]^3
```

is mapped to:

```
sage: m.from_polynomial_exp(f)
m[5, 5, 5, 3, 1, 1]
```

Furthermore, this function is linear:

```
sage: f = 3 * x[3] + 2 * x[0]^2 * x[2] * x[4]^3
sage: m.from_polynomial_exp(f)
3*m[4] + 2*m[5, 5, 5, 3, 1, 1]
```

..SEEALSO:: `Partition()`, `Partition.to_exp()`

5.1.259 Multiplicative symmetric functions

A realization h of the ring of symmetric functions is multiplicative if for a partition $\lambda = (\lambda_1, \lambda_2, \ldots)$ we have $h_\lambda = h_{\lambda_1} h_{\lambda_2} \cdots$.

class sage.combinat.sf.multiplicative.**SymmetricFunctionAlgebra_multiplicative**(*Sym*, *basis_name=None*, *prefix=None*, *graded=True*)

Bases: `sage.combinat.sf.classical.SymmetricFunctionAlgebra_classical`

The class of multiplicative bases of the ring of symmetric functions.

A realization q of the ring of symmetric functions is multiplicative if for a partition $\lambda = (\lambda_1, \lambda_2, \ldots)$ we have $q_\lambda = q_{\lambda_1} q_{\lambda_2} \cdots$ (with q_0 meaning 1).

Examples of multiplicative realizations are the elementary symmetric basis, the complete homogeneous basis, the powersum basis (if the base ring is a $\mathbf{Q}$-algebra), and the Witt basis (but not the Schur basis or the monomial basis).

coproduct_on_basis(*mu*)

Return the coproduct on a basis element for multiplicative bases.

INPUT:

- `mu` – a partition

OUTPUT:

- the image of `self[mu]` under comultiplication; this is an element of the tensor square of `self`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.powersum()
sage: p.coproduct_on_basis([2,1])
p[] # p[2, 1] + p[1] # p[2] + p[2] # p[1] + p[2, 1] # p[]

sage: e = Sym.elementary()
sage: e.coproduct_on_basis([3,1])
e[] # e[3, 1] + e[1] # e[2, 1] + e[1] # e[3] + e[1, 1] # e[2] + e[2] # e[1, 1] + e[2, 1] # e

sage: h = Sym.homogeneous()
```

```
sage: h.coproduct_on_basis([3,1])
h[] # h[3, 1] + h[1] # h[2, 1] + h[1] # h[3] + h[1, 1] # h[2] + h[2] # h[1, 1] + h[2, 1] # h
```

5.1.260 k-Schur Functions

class sage.combinat.sf.new_kschur.**KBoundedSubspace**(*Sym*, *k*, *t='t'*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

This class implements the subspace of the ring of symmetric functions spanned by $\{s_\lambda[X/(1-t)]\}_{\lambda_1 \leq k} = \{s^{(k)}_\lambda[X,t]\}_{\lambda_1 \leq k}$ over the base ring $\mathbb{Q}[t]$. When $t = 1$, this space is in fact a subring of the ring of symmetric functions generated by the complete homogeneous symmetric functions h_i for $1 \leq i \leq k$.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: KB = Sym.kBoundedSubspace(3,1); KB
3-bounded Symmetric Functions over Rational Field with t=1

sage: Sym = SymmetricFunctions(QQ['t'])
sage: KB = Sym.kBoundedSubspace(3); KB
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational Field
```

The k-Schur function basis can be constructed as follows:

```
sage: ks = KB.kschur(); ks
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational Field in the 3-
```

K_kschur()

Returns the k-bounded basis called the K-k-Schur basis. See [Morse11] and [LamSchillingShimozono10].

REFERENCES:

EXAMPLES:

```
sage: kB = SymmetricFunctions(QQ).kBoundedSubspace(3,1)
sage: g = kB.K_kschur()
sage: g
3-bounded Symmetric Functions over Rational Field with t=1 in the K-3-Schur basis
sage: kB = SymmetricFunctions(QQ['t']).kBoundedSubspace(3)
sage: g = kB.K_kschur()
Traceback (most recent call last):
...
ValueError: This basis only exists for t=1
```

khomogeneous()

The homogeneous basis of this algebra.

See also:

kHomogeneous()

EXAMPLES:

```
sage: kh3 = SymmetricFunctions(QQ).kBoundedSubspace(3,1).khomogeneous()
sage: TestSuite(kh3).run()
```

kschur()

The k-Schur basis of this algebra.

See also:

kSchur()

EXAMPLES:

```
sage: ks3 = SymmetricFunctions(QQ).kBoundedSubspace(3,1).kschur()
sage: TestSuite(ks3).run()
```

realizations()

A list of realizations of this algebra.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).kBoundedSubspace(3,1).realizations()
[3-bounded Symmetric Functions over Rational Field with t=1 in the 3-Schur basis also with t

sage: SymmetricFunctions(QQ['t']).kBoundedSubspace(3).realizations()
[3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational Field in t
```

retract(*sym*)

Return the retract of sym from the ring of symmetric functions to self.

INPUT:

- sym – a symmetric function

OUTPUT:

- the analogue of the symmetric function in the k-bounded subspace (if possible)

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: KB = Sym.kBoundedSubspace(3,1); KB
3-bounded Symmetric Functions over Rational Field with t=1
sage: KB.retract(s[2]+s[3])
ks3[2] + ks3[3]
sage: KB.retract(s[2,1,1])
Traceback (most recent call last):
...
ValueError: s[2, 1, 1] is not in the image
```

class sage.combinat.sf.new_kschur.**KBoundedSubspaceBases**(*base*, *t='t'*)

Bases: sage.categories.realizations.Category_realization_of_parent

The category of bases for the k-bounded subspace of symmetric functions.

class ElementMethods

expand(**args*, ***kwargs*)

Returns the monomial expansion of self in n variables.

INPUT:

- n – positive integer

OUTPUT: monomial expansion of self in n variables

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks = Sym.kschur(3,1)
sage: ks[3,1].expand(2)
```

```
x0^4 + 2*x0^3*x1 + 2*x0^2*x1^2 + 2*x0*x1^3 + x1^4
sage: s = Sym.schur()
sage: ks[3,1].expand(2) == s(ks[3,1]).expand(2)
True

sage: Sym = SymmetricFunctions(QQ['t'])
sage: ks = Sym.kschur(3)
sage: f = ks[3,2]-ks[1]
sage: f.expand(2)
t^2*x0^5 + (t^2 + t)*x0^4*x1 + (t^2 + t + 1)*x0^3*x1^2 + (t^2 + t + 1)*x0^2*x1^3 + (t^2 +
```

hl_creation_operator(*nu*, *t=None*)

This is the vertex operator that generalizes Jing's operator.

It is a linear operator that raises the degree by $|\nu|$. This creation operator is a t-analogue of multiplication by `s(nu)` .

See also:

Proposition 5 in [SZ.2001].

INPUT:

- `nu` – a partition
- `t` – a parameter (default: None, in this case t is used)

REFERENCES:

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: ks = Sym.kschur(4)
sage: s = Sym.schur()
sage: s(ks([3,1,1]).hl_creation_operator([1]))
(t-1)*s[2, 2, 1, 1] + t^2*s[3, 1, 1, 1] + (t^3+t^2-t)*s[3, 2, 1] + (t^3-t^2)*s[3, 3] + (
sage: ks([3,1,1]).hl_creation_operator([1])
(t-1)*ks4[2, 2, 1, 1] + t^2*ks4[3, 1, 1, 1] + t^3*ks4[3, 2, 1] + (t^3-t^2)*ks4[3, 3] + t

sage: Sym = SymmetricFunctions(QQ)
sage: ks = Sym.kschur(4,t=1)
sage: ks([3,1,1]).hl_creation_operator([1])
ks4[3, 1, 1, 1] + ks4[3, 2, 1] + ks4[4, 1, 1]
```

is_schur_positive(**args*, ***kwargs*)

Returns whether `self` is Schur positive.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks = Sym.kschur(3,1)
sage: f = ks[3,2]+ks[1]
sage: f.is_schur_positive()
True
sage: f = ks[3,2]-ks[1]
sage: f.is_schur_positive()
False

sage: Sym = SymmetricFunctions(QQ['t'])
sage: ks = Sym.kschur(3)
sage: f = ks[3,2]+ks[1]
sage: f.is_schur_positive()
True
sage: f = ks[3,2]-ks[1]
```

```
sage: f.is_schur_positive()
False
```

omega()

Returns the ω operator on `self`.

At $t = 1$, ω maps the k-Schur function $s_\lambda^{(k)}$ to $s_{\lambda^{(k)}}^{(k)}$, where $\lambda^{(k)}$ is the k-conjugate of the partition λ.

See also:

`k_conjugate()`.

For generic t, ω sends $s_\lambda^{(k)}[X;t]$ to $t^d s_{\lambda^{(k)}}^{(k)}[X;1/t]$, where d is the size of the core of λ minus the size of λ. Most of the time, this result is not in the k-bounded subspace.

See also:

`omega_t_inverse()`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks = Sym.kschur(3,1)
sage: ks[2,2,1,1].omega()
ks3[2, 2, 2]
sage: kh = Sym.khomogeneous(3)
sage: kh[3].omega()
h3[1, 1, 1] - 2*h3[2, 1] + h3[3]

sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: ks = Sym.kschur(3)
sage: ks[3,1,1].omega()
Traceback (most recent call last):
...
ValueError: t*s[2, 1, 1, 1] + s[3, 1, 1] is not in the image
```

omega_t_inverse()

Returns the map $t \to 1/t$ composed with ω on `self`.

Unlike the map `omega()`, the result of `omega_t_inverse()` lives in the k-bounded subspace and hence will return an element even for generic t. For $t = 1$, `omega()` and `omega_t_inverse()` return the same result.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: ks = Sym.kschur(3)
sage: ks[3,1,1].omega_t_inverse()
1/t*ks3[2, 1, 1, 1]
sage: ks[3,2].omega_t_inverse()
1/t^2*ks3[1, 1, 1, 1, 1]
```

scalar(*x*, *zee=None*)

Return standard scalar product between `self` and `x`.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`
- `zee` – an optional function on partitions giving the value for the scalar product between p_μ and p_μ (default is to use the standard `zee()` function)

See also:

`scalar()`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'])
sage: ks3 = Sym.kschur(3)
sage: ks3[3,2,1].scalar( ks3[2,2,2] )
t^3 + t
sage: dks3 = Sym.kBoundedQuotient(3).dks()
sage: [ks3[3,2,1].scalar(dks3(la)) for la in Partitions(6, max_part=3)]
[0, 1, 0, 0, 0, 0, 0]
sage: dks3 = Sym.kBoundedQuotient(3,t=1).dks()
sage: [ks3[2,2,2].scalar(dks3(la)) for la in Partitions(6, max_part=3)]
[0, t - 1, 0, 1, 0, 0, 0]
sage: ks3 = Sym.kschur(3,t=1)
sage: [ks3[2,2,2].scalar(dks3(la)) for la in Partitions(6, max_part=3)]
[0, 0, 0, 1, 0, 0, 0]
sage: kH = Sym.khomogeneous(4)
sage: kH([2,2,1]).scalar(ks3[2,2,1])
3
```

TESTS:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks3 = Sym.kschur(3,1)
sage: ks3(1).scalar(ks3([]))
1
```

class KBoundedSubspaceBases.**ParentMethods**

an_element()

Return an element of `self`.

EXAMPLES:

```
sage: SymmetricFunctions(QQ['t']).kschur(3).an_element()
2*ks3[] + 2*ks3[1] + 3*ks3[2]
```

antipode(*element*)

Return the antipode on `self` by lifting to the space of symmetric functions, computing the antipode, and then converting to `self.parent()`. This is only the antipode for $t = 1$ and for other values of t the result may not be in the space where the k-Schur functions live.

INPUT:

- `element` – an element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks3 = Sym.kschur(3,1)
sage: ks3[3,2].antipode()
-ks3[1, 1, 1, 1, 1]
sage: ks3.antipode(ks3[3,2])
-ks3[1, 1, 1, 1, 1]
```

coproduct(*element*)

Return the coproduct operation on `element`.

The coproduct is first computed on the homogeneous basis if $t = 1$ and on the Hall-Littlewood `Qp` basis otherwise. The result is computed then converted to the tensor squared of `self.parent()`.

INPUT:

- `element` – an element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks3 = Sym.kschur(3,1)
```

```
sage: ks3[2,1].coproduct()
ks3[] # ks3[2, 1] + ks3[1] # ks3[1, 1] + ks3[1] # ks3[2] + ks3[1, 1] # ks3[1] + ks3[2] #
sage: h3 = Sym.khomogeneous(3)
sage: h3[2,1].coproduct()
h3[] # h3[2, 1] + h3[1] # h3[1, 1] + h3[1] # h3[2] + h3[1, 1] # h3[1] + h3[2] # h3[1] + h
sage: ks3t = SymmetricFunctions(FractionField(QQ['t'])).kschur(3)
sage: ks3t[2,1].coproduct()
ks3[] # ks3[2, 1] + ks3[1] # ks3[1, 1] + ks3[1] # ks3[2] + ks3[1, 1] # ks3[1] + ks3[2] #
sage: ks3t[3,1].coproduct()
ks3[] # ks3[3, 1] + ks3[1] # ks3[2, 1] + (t+1)*ks3[1] # ks3[3] + ks3[1, 1] # ks3[2] + ks
+ (t+1)*ks3[2] # ks3[2] + ks3[2, 1] # ks3[1] + (t+1)*ks3[3] # ks3[1] + ks3[3, 1] # ks3[]
sage: h3.coproduct(h3[2,1])
h3[] # h3[2, 1] + h3[1] # h3[1, 1] + h3[1] # h3[2] + h3[1, 1] # h3[1] + h3[2] # h3[1] + h
```

counit(*element*)

Return the counit of `element`.

The counit is the constant term of `element`.

INPUT:

- `element` – an element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks3 = Sym.kschur(3,1)
sage: f = 2*ks3[2,1] + 3*ks3[[]]
sage: f.counit()
3
sage: ks3.counit(f)
3
```

degree_on_basis(*b*)

Return the degree of the basis element indexed by b.

INPUT: - `b` – a partition

EXAMPLES:

```
sage: ks3 = SymmetricFunctions(QQ).kschur(3,1)
sage: ks3.degree_on_basis(Partition([3,2]))
5
```

one_basis()

Return the basis element indexing `1`.

EXAMPLES:

```
sage: ks3 = SymmetricFunctions(QQ).kschur(3,1)
sage: ks3.one()  # indirect doctest
ks3[]
```

transition_matrix(*other*, *n*)

Return the degree `n` transition matrix between `self` and `other`.

INPUT:

- `other` – a basis in the ring of symmetric functions
- `n` – a positive integer

The entry in the i^{th} row and j^{th} column is the coefficient obtained by writing the i^{th} element of the basis of `self` in terms of the basis `other`, and extracting the j^{th} coefficient.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ); s = Sym.schur()
sage: ks3 = Sym.kschur(3,1)
sage: ks3.transition_matrix(s,5)
[1 1 1 0 0 0 0]
[0 1 0 1 0 0 0]
[0 0 1 0 1 0 0]
[0 0 0 1 0 1 0]
[0 0 0 0 1 1 1]

sage: Sym = SymmetricFunctions(QQ['t'])
sage: s = Sym.schur()
sage: ks = Sym.kschur(3)
sage: ks.transition_matrix(s,5)
[t^2   t   1   0   0   0   0]
[  0   t   0   1   0   0   0]
[  0   0   t   0   1   0   0]
[  0   0   0   t   0   1   0]
[  0   0   0   0 t^2   t   1]
```

KBoundedSubspaceBases.**super_categories**()

The super categories of `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'])
sage: from sage.combinat.sf.new_kschur import KBoundedSubspaceBases
sage: KB = Sym.kBoundedSubspace(3)
sage: KBB = KBoundedSubspaceBases(KB); KBB
Category of k bounded subspace bases of 3-bounded Symmetric Functions over Univariate Polync
sage: KBB.super_categories()
[Category of realizations of 3-bounded Symmetric Functions over Univariate Polynomial Ring i
 Join of Category of graded modules with basis over Univariate Polynomial Ring in t over Rat
     and Category of coalgebras with basis over Univariate Polynomial Ring in t over Rationa
     and Category of subobjects of sets]
```

class sage.combinat.sf.new_kschur.**K_kSchur**(*kBoundedRing*)

Bases: sage.combinat.free_module.CombinatorialFreeModule

This class implements the basis of the k-bounded subspace called the K-k-Schur basis. See [Morse2011], [LamSchillingShimozono2010].

REFERENCES:

K_k_Schur_non_commutative_variables(*la*)

Returns the K-k-Schur function, as embedded inside the affine zero Hecke algebra.

INPUT:

- `la` – A k-bounded Partition

OUTPUT:

- An element of the affine zero Hecke algebra.

EXAMPLES:

```
sage: g = SymmetricFunctions(QQ).kBoundedSubspace(3,1).K_kschur()
sage: g.K_k_Schur_non_commutative_variables([2,1])
T[3,1,0] + T[1,2,0] + T[3,2,0] + T[0,1,0] + T[2,0,1] + T[0,3,0] + T[2,0,3] + T[0,3,1] + T[2,
sage: g.K_k_Schur_non_commutative_variables([])
1
sage: g.K_k_Schur_non_commutative_variables([4,1])
```

```
Traceback (most recent call last):
...
ValueError: Partition should be 3-bounded
```

homogeneous_basis_noncommutative_variables_zero_Hecke(*la*)

Returns the homogeneous basis element indexed by `la`, viewed as an element inside the affine zero Hecke algebra. For the code, see method _homogeneous_basis.

INPUT:

- `la` – A k-bounded partition

OUTPUT:

- An element of the affine zero Hecke algebra.

EXAMPLES:

```
sage: g = SymmetricFunctions(QQ).kBoundedSubspace(3,1).K_kschur()
sage: g.homogeneous_basis_noncommutative_variables_zero_Hecke([2,1])
T[2,1,0] + T[3,1,0] + T[1,2,0] + T[3,2,0] + T[0,1,0] + T[2,0,1] + T[1,0,3] + T[0,3,0] + T[2,
sage: g.homogeneous_basis_noncommutative_variables_zero_Hecke([])
1
```

lift(*x*)

Returns the lift of a k-bounded symmetric function.

INPUT:

- **x – An expression in the K-k-Schur basis. Equivalently, x can be a** k-bounded partition (then `x` corresponds to the basis element indexed by `x`)

OUTPUT:

- A symmetric function.

EXAMPLES:

```
sage: g = SymmetricFunctions(QQ).kBoundedSubspace(3,1).K_kschur()
sage: g.lift([2,1])
h[2] + h[2, 1] - h[3]
sage: g.lift([])
h[]
sage: g.lift([4,1])
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 1]) an element of self (=3-bounded Symmetric Fur
```

product(*x*, *y*)

Returns the product of the two K-k-Schur functions.

INPUT:

- `x`, `y` – elements of the k-bounded subspace, in the K-k-Schur basis.

OUTPUT:

- An element of the k-bounded subspace, in the K-k-Schur basis

EXAMPLES:

```
sage: g = SymmetricFunctions(QQ).kBoundedSubspace(3,1).K_kschur()
sage: g.product(g([2,1]), g[1])
-2*Kks3[2, 1] + Kks3[2, 1, 1] + Kks3[2, 2]
```

```
sage: g.product(g([2,1]), g([]))
Kks3[2, 1]
```

retract(x)

Returns the retract of a symmetric function.

INPUT:

- x – A symmetric function.

OUTPUT:

- A k-bounded symmetric function in the K-k-Schur basis.

EXAMPLES:

```
sage: g = SymmetricFunctions(QQ).kBoundedSubspace(3,1).K_kschur()
sage: m = SymmetricFunctions(QQ).m()
sage: g.retract(m[2,1])
-2*Kks3[1] + 4*Kks3[1, 1] - 2*Kks3[1, 1, 1] - Kks3[2] + Kks3[2, 1]
sage: g.retract(m([]))
Kks3[]
```

class sage.combinat.sf.new_kschur.**kHomogeneous**(*kBoundedRing*)

Bases: sage.combinat.free_module.CombinatorialFreeModule

Space of k-bounded homogeneous symmetric functions.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: kH = Sym.khomogeneous(3)
sage: kH[2]
h3[2]
sage: kH[2].lift()
h[2]
```

class sage.combinat.sf.new_kschur.**kSchur**(*kBoundedRing*)

Bases: sage.combinat.free_module.CombinatorialFreeModule

Space of k-Schur functions.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'])
sage: KB = Sym.kBoundedSubspace(3); KB
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational Field
```

The k-Schur function basis can be constructed as follows:

```
sage: ks3 = KB.kschur(); ks3
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational Field in the 3-
```

We can convert to any basis of the ring of symmetric functions and, whenever it makes sense, also the other way round:

```
sage: s = Sym.schur()
sage: s(ks3([3,2,1]))
s[3, 2, 1] + t*s[4, 1, 1] + t*s[4, 2] + t^2*s[5, 1]
sage: t = Sym.base_ring().gen()
sage: ks3(s([3, 2, 1]) + t*s([4, 1, 1]) + t*s([4, 2]) + t^2*s([5, 1]))
ks3[3, 2, 1]
sage: s(ks3[2, 1, 1])
```

```
s[2, 1, 1] + t*s[3, 1]
sage: ks3(s[2, 1, 1] + t*s[3, 1])
ks3[2, 1, 1]
```

k-Schur functions are indexed by partitions with first part $\leq k$. Constructing a k-Schur function for a larger partition raises an error:

```
sage: ks3([4,3,2,1]) #
Traceback (most recent call last):
...
TypeError: do not know how to make x (= [4, 3, 2, 1]) an element of self (=3-bounded Symmetric F
```

Similarly, attempting to convert a function that is not in the linear span of the k-Schur functions raises an error:

```
sage: ks3(s([4]))
Traceback (most recent call last):
...
ValueError: s[4] is not in the image
```

Note that the product of k-Schur functions is not guaranteed to be in the space spanned by the k-Schurs. In general, we only have that a k-Schur times a j-Schur function is in the $(k+j)$-bounded subspace. The multiplication of two k-Schur functions thus generally returns the product of the lift of the functions to the ambient symmetric function space. If the result happens to lie in the k-bounded subspace, then the result is cast into the k-Schur basis:

```
sage: ks2 = Sym.kBoundedSubspace(2).kschur()
sage: ks2[1] * ks2[1]
ks2[1, 1] + ks2[2]
sage: ks2[1] * ks2[2]
s[2, 1] + s[3]
```

Because the target space of the product of a k-Schur and a j-Schur has several possibilities, the product of a k-Schur and j-Schur function is not implemented for distinct k and j. Let us show how to get around this 'manually':

```
sage: ks3 = Sym.kBoundedSubspace(3).kschur()
sage: ks2([2,1]) * ks3([3,1])
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '*': '2-bounded Symmetric Functions over Univariate
```

The workaround:

```
sage: f = s(ks2([2,1])) * s(ks3([3,1])); f # Convert to Schur functions first and multiply there
s[3, 2, 1, 1] + s[3, 2, 2] + (t+1)*s[3, 3, 1] + s[4, 1, 1, 1]
+ (2*t+2)*s[4, 2, 1] + (t^2+t+1)*s[4, 3] + (2*t+1)*s[5, 1, 1]
+ (t^2+2*t+1)*s[5, 2] + (t^2+2*t)*s[6, 1] + t^2*s[7]
```

or:

```
sage: f = ks2[2,1].lift() * ks3[3,1].lift()
sage: ks5 = Sym.kBoundedSubspace(5).kschur()
sage: ks5(f) # The product of a 'ks2' with a 'ks3' is a 'ks5'.
ks5[3, 2, 1, 1] + ks5[3, 2, 2] + (t+1)*ks5[3, 3, 1] + ks5[4, 1, 1, 1]
+ (t+2)*ks5[4, 2, 1] + (t^2+t+1)*ks5[4, 3] + (t+1)*ks5[5, 1, 1] + ks5[5, 2]
```

For other technical reasons, taking powers of k-Schur functions is not implemented, even when the answer is still in the k-bounded subspace:

```
sage: ks2([1])^2
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for ** or pow(): 'kSchur_with_category.element_class' and
```

Todo

Get rid of said technical "reasons".

However, at $t = 1$, the product of k-Schur functions is in the span of the k-Schur functions always. Below are some examples at $t = 1$

```
sage: ks3 = Sym.kBoundedSubspace(3, t=1).kschur(); ks3
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational Field with t=1
sage: s = SymmetricFunctions(ks3.base_ring()).schur()
sage: ks3(s([3]))
ks3[3]
sage: s(ks3([3,2,1]))
s[3, 2, 1] + s[4, 1, 1] + s[4, 2] + s[5, 1]
sage: ks3([2,1])^2    # taking powers works for t=1
ks3[2, 2, 1, 1] + ks3[2, 2, 2] + ks3[3, 1, 1, 1]
```

TESTS:

Check that trac ticket #13743 is fixed:

```
sage: ks3 = SymmetricFunctions(QQ).kschur(3, 1)
sage: f = ks3[2,1]
sage: f.coefficient(f.support()[0])
1
```

product (*left*, *right*)

Take the product of two k-Schur functions.

If $t \neq 1$, then take the product by lifting to the Schur functions and then retracting back into the the k-bounded subspace (if possible).

If $t = 1$, then the product is done using `_product_from_combinatorial_algebra_multiply()` and this method calls `_multiply_basis()`.

INPUT:

- `left`, `right` – elements of the k-Schur functions

OUTPUT:

- an element of the k-Schur functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'])
sage: ks3 = Sym.kschur(3,1)
sage: kH = Sym.khomogeneous(3)
sage: ks3(kH[2,1,1])
ks3[2, 1, 1] + ks3[2, 2] + ks3[3, 1]
sage: ks3([])*kH[2,1,1]
ks3[2, 1, 1] + ks3[2, 2] + ks3[3, 1]
sage: ks3([3,3,3,2,2,1,1,1])^2
ks3[3, 3, 3, 3, 3, 3, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1]
sage: ks3([3,3,3,2,2,1,1,1])*ks3([2,2,2,2,2,1,1,1,1])
ks3[3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1]
```

```
sage: ks3([2,2,1,1,1,1])*ks3([2,2,2,1,1,1,1])
ks3[2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1] + ks3[2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1]
sage: ks3[2,1]^2
ks3[2, 2, 1, 1] + ks3[2, 2, 2] + ks3[3, 1, 1, 1]
sage: ks3 = Sym.kschur(3)
sage: ks3[2,1]*ks3[2,1]
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[3, 3] + s[4, 1, 1] + s[4, 2]
```

TESTS:

```
sage: Sym = SymmetricFunctions(QQ['t'])
sage: ks3 = Sym.kschur(3,1)
sage: kH = Sym.khomogeneous(3)
sage: ks3.product( ks3([]), ks3([]) )
ks3[]
sage: ks3.product( ks3([]), kH([]) )
ks3[]
sage: ks3 = Sym.kschur(3)
sage: ks3.product( ks3([]), ks3([]) )
ks3[]
```

5.1.261 Non-symmetric Macdonald Polynomials

class sage.combinat.sf.ns_macdonald.**AugmentedLatticeDiagramFilling**(*l*, *pi=None*)

Bases: sage.combinat.combinat.CombinatorialObject

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a == loads(dumps(a))
True
sage: pi = Permutation([2,3,1]).to_permutation_group_element()
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]],pi)
sage: a == loads(dumps(a))
True
```

are_attacking(*i*, *j*, *ii*, *jj*)

Return `True` if the boxes `(i,j)` and `(ii,jj)` in `self` are attacking.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: all( a.are_attacking(i,j,ii,jj) for (i,j),(ii,jj) in a.attacking_boxes())
True
sage: a.are_attacking(1,1,3,2)
False
```

attacking_boxes()

Returns a list of pairs of boxes in `self` that are attacking.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.attacking_boxes()[:5]
[((1, 1), (2, 1)),
 ((1, 1), (3, 1)),
 ((1, 1), (6, 1)),
 ((1, 1), (2, 0)),
 ((1, 1), (3, 0))]
```

boxes()

Return a list of the coordinates of the boxes of `self`, including the 'basement row'.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.boxes()
[(1, 1),
 (1, 2),
 (2, 1),
 (3, 1),
 (3, 2),
 (3, 3),
 (6, 1),
 (6, 2),
 (1, 0),
 (2, 0),
 (3, 0),
 (4, 0),
 (5, 0),
 (6, 0)]
```

coeff(q, t)

Return the coefficient in front of `self` in the HHL formula for the expansion of the non-symmetric Macdonald polynomial E(self.shape()).

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: q,t = var('q,t')
sage: a.coeff(q,t)
(t - 1)^4/((q^2*t^3 - 1)^2*(q*t^2 - 1)^2)
```

coeff_integral(q, t)

Return the coefficient in front of `self` in the HHL formula for the expansion of the integral non-symmetric Macdonald polynomial E(self.shape())

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: q,t = var('q,t')
sage: a.coeff_integral(q,t)
(q^2*t^3 - 1)^2*(q*t^2 - 1)^2*(t - 1)^4
```

coinv()

Return `self`'s co-inversion statistic.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.coinv()
2
```

descents()

Return a list of the descents of `self`.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.descents()
```

```
[(1, 2), (3, 2)]
```

inv()
Return `self`'s inversion statistic.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.inv()
15
```

inversions()
Return a list of the inversions of `self`.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.inversions()[:5]
[((6, 2), (3, 2)),
 ((1, 2), (6, 1)),
 ((1, 2), (3, 1)),
 ((1, 2), (2, 1)),
 ((6, 1), (3, 1))]
sage: len(a.inversions())
25
```

is_non_attacking()
Return `True` if `self` is non-attacking.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.is_non_attacking()
True
sage: a = AugmentedLatticeDiagramFilling([[1, 1, 1], [2, 3], [3]])
sage: a.is_non_attacking()
False
sage: a = AugmentedLatticeDiagramFilling([[2,2],[1]])
sage: a.is_non_attacking()
False
sage: pi = Permutation([2,1]).to_permutation_group_element()
sage: a = AugmentedLatticeDiagramFilling([[2,2],[1]],pi)
sage: a.is_non_attacking()
True
```

maj()
Return the major index of `self`.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.maj()
3
```

permuted_filling(*sigma*)
EXAMPLES:

```
sage: pi=Permutation([2,1,4,3]).to_permutation_group_element()
sage: fill=[[2],[1,2,3],[],[3,1]]
sage: AugmentedLatticeDiagramFilling(fill).permuted_filling(pi)
[[2, 1], [1, 2, 1, 4], [4], [3, 4, 2]]
```

reading_order()
Return a list of coordinates of the boxes in `self`, starting from the top right, and reading from right to left. Note that this includes the 'basement row' of `self`.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.reading_order()
[(3, 3),
 (6, 2),
 (3, 2),
 (1, 2),
 (6, 1),
 (3, 1),
 (2, 1),
 (1, 1),
 (6, 0),
 (5, 0),
 (4, 0),
 (3, 0),
 (2, 0),
 (1, 0)]
```

reading_word()
Return the reading word of `self`, obtained by reading the boxes entries of self from right to left, starting in the upper right.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.reading_word()
word: 25465321
```

shape()
Return the shape of `self`.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.shape()
[2, 1, 3, 0, 0, 2]
```

weight()
Return the weight of `self`.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.weight()
[1, 2, 1, 1, 2, 1]
```

sage.combinat.sf.ns_macdonald.**E**(*mu*, *q=None*, *t=None*, *pi=None*)
Returns the non-symmetric Macdonald polynomial in type A corresponding to a shape `mu`, with basement permuted according to `pi`.

Note that if both q and t are specified, then they must have the same parent.

REFERENCE:

- J. Haglund, M. Haiman, N. Loehr. *A combinatorial formula for non-symmetric Macdonald polynomials*. Arxiv math/0601693v3.

See also:

NonSymmetricMacdonaldPolynomials for a type free implementation where the polynomials are constructed recursively by the application of intertwining operators.

EXAMPLES:

```
sage: from sage.combinat.sf.ns_macdonald import E
sage: E([0,0,0])
1
sage: E([1,0,0])
x0
sage: E([0,1,0])
((-t + 1)/(-q*t^2 + 1))*x0 + x1
sage: E([0,0,1])
((-t + 1)/(-q*t + 1))*x0 + ((-t + 1)/(-q*t + 1))*x1 + x2
sage: E([1,1,0])
x0*x1
sage: E([1,0,1])
((-t + 1)/(-q*t^2 + 1))*x0*x1 + x0*x2
sage: E([0,1,1])
((-t + 1)/(-q*t + 1))*x0*x1 + ((-t + 1)/(-q*t + 1))*x0*x2 + x1*x2
sage: E([2,0,0])
x0^2 + ((-q*t + q)/(-q*t + 1))*x0*x1 + ((-q*t + q)/(-q*t + 1))*x0*x2
sage: E([0,2,0])
((-t + 1)/(-q^2*t^2 + 1))*x0^2 + ((-q^2*t^3 + q^2*t^2 - q*t^2 + 2*q*t - q + t - 1)/(-q^3*t^3 + q
```

sage.combinat.sf.ns_macdonald.**E_integral**(*mu*, *q=None*, *t=None*, *pi=None*)

Returns the integral form for the non-symmetric Macdonald polynomial in type A corresponding to a shape mu.

Note that if both q and t are specified, then they must have the same parent.

REFERENCE:

- J. Haglund, M. Haiman, N. Loehr. *A combinatorial formula for non-symmetric Macdonald polynomials*. Arxiv math/0601693v3.

EXAMPLES:

```
sage: from sage.combinat.sf.ns_macdonald import E_integral
sage: E_integral([0,0,0])
1
sage: E_integral([1,0,0])
(-t + 1)*x0
sage: E_integral([0,1,0])
(-q*t^2 + 1)*x0 + (-t + 1)*x1
sage: E_integral([0,0,1])
(-q*t + 1)*x0 + (-q*t + 1)*x1 + (-t + 1)*x2
sage: E_integral([1,1,0])
(t^2 - 2*t + 1)*x0*x1
sage: E_integral([1,0,1])
(q*t^3 - q*t^2 - t + 1)*x0*x1 + (t^2 - 2*t + 1)*x0*x2
sage: E_integral([0,1,1])
(q^2*t^3 + q*t^4 - q*t^3 - q*t^2 - q*t - t^2 + t + 1)*x0*x1 + (q*t^2 - q*t - t + 1)*x0*x2 + (t^2
sage: E_integral([2,0,0])
(t^2 - 2*t + 1)*x0^2 + (q^2*t^2 - q^2*t - q*t + q)*x0*x1 + (q^2*t^2 - q^2*t - q*t + q)*x0*x2
sage: E_integral([0,2,0])
(q^2*t^3 - q^2*t^2 - t + 1)*x0^2 + (q^4*t^3 - q^3*t^2 - q^2*t + q*t^2 - q*t + q - t + 1)*x0*x1 +
```

sage.combinat.sf.ns_macdonald.**Ht**(*mu*, *q=None*, *t=None*, *pi=None*)

Returns the symmetric Macdonald polynomial using the Haiman, Haglund, and Loehr formula.

Note that if both q and t are specified, then they must have the same parent.

REFERENCE:

- J. Haglund, M. Haiman, N. Loehr. *A combinatorial formula for non-symmetric Macdonald polynomials*. Arxiv math/0601693v3.

EXAMPLES:

```
sage: from sage.combinat.sf.ns_macdonald import Ht
sage: HHt = SymmetricFunctions(QQ['q','t'].fraction_field()).macdonald().Ht()
sage: Ht([0,0,1])
x0 + x1 + x2
sage: HHt([1]).expand(3)
x0 + x1 + x2
sage: Ht([0,0,2])
x0^2 + (q + 1)*x0*x1 + x1^2 + (q + 1)*x0*x2 + (q + 1)*x1*x2 + x2^2
sage: HHt([2]).expand(3)
x0^2 + (q + 1)*x0*x1 + x1^2 + (q + 1)*x0*x2 + (q + 1)*x1*x2 + x2^2
```

class sage.combinat.sf.ns_macdonald.**LatticeDiagram**(*l*, *copy=True*)
Bases: sage.combinat.combinat.CombinatorialObject

CombinatorialObject provides a thin wrapper around a list. The main differences are that __setitem__ is disabled so that CombinatorialObjects are shallowly immutable, and the intention is that they are semantically immutable.

Because of this, CombinatorialObjects provide a __hash__ function which computes the hash of the string representation of a list and the hash of its parent's class. Thus, each CombinatorialObject should have a unique string representation.

See also:

`CombinatorialElement` if you want a combinatorial object which is an element of a parent.

Warning: This class is slowly being deprecated. Use `ClonableList` instead.

INPUT:

- `l` – a list or any object that can be converted to a list by calling `list()`.
- `copy` – (boolean, default `True`) if `False`, then `l` must be a `list`, which is assigned to `self._list` without copying.

EXAMPLES:

```
sage: c = CombinatorialObject([1,2,3])
sage: c == loads(dumps(c))
True
sage: c._list
[1, 2, 3]
sage: c._hash is None
True
```

For efficiency, you can specify `copy=False` if you know what you are doing:

```
sage: from sage.combinat.combinat import CombinatorialObject
sage: x = [3, 2, 1]
sage: C = CombinatorialObject(x, copy=False)
sage: C
[3, 2, 1]
sage: x[0] = 5
```

```
sage: C
[5, 2, 1]
```

TESTS:

Test indirectly that we copy the input (see trac ticket #18184):

```
sage: L = IntegerListsLex(element_class=Partition)
sage: x = [3, 2, 1]
sage: P = L(x)
sage: x[0] = 5
sage: list(P)
[3, 2, 1]
```

a(*i*, *j*)

Return the length of the arm of the box `(i, j)` in `self`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.a(5,2)
3
```

arm(*i*, *j*)

Return the arm of the box `(i, j)` in `self`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.arm(5,2)
[(1, 2), (3, 2), (8, 1)]
```

arm_left(*i*, *j*)

Return the left arm of the box `(i, j)` in `self`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.arm_left(5,2)
[(1, 2), (3, 2)]
```

arm_right(*i*, *j*)

Return the right arm of the box `(i, j)` in `self`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.arm_right(5,2)
[(8, 1)]
```

boxes()

EXAMPLES:

```
sage: a = LatticeDiagram([3,0,2])
sage: a.boxes()
[(1, 1), (1, 2), (1, 3), (3, 1), (3, 2)]
sage: a = LatticeDiagram([2, 1, 3, 0, 0, 2])
sage: a.boxes()
[(1, 1), (1, 2), (2, 1), (3, 1), (3, 2), (3, 3), (6, 1), (6, 2)]
```

boxes_same_and_lower_right(*ii*, *jj*)

Return a list of the boxes of `self` that are in row `jj` but not identical with `(ii, jj)`, or lie in the row `jj - 1` (the row directly below `jj`; this might be the basement) and strictly to the right of `(ii, jj)`.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a = a.shape()
sage: a.boxes_same_and_lower_right(1,1)
[(2, 1), (3, 1), (6, 1), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0)]
sage: a.boxes_same_and_lower_right(1,2)
[(3, 2), (6, 2), (2, 1), (3, 1), (6, 1)]
sage: a.boxes_same_and_lower_right(3,3)
[(6, 2)]
sage: a.boxes_same_and_lower_right(2,3)
[(3, 3), (3, 2), (6, 2)]
```

flip()

Return the flip of `self`, where flip is defined as follows. Let `r = max(self)`. Then `self.flip()[i] = r - self[i]`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,0,2])
sage: a.flip()
[0, 3, 1]
```

l(*i*, *j*)

Return `self[i] - j`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.l(5,2)
1
```

leg(*i*, *j*)

Return the leg of the box `(i,j)` in `self`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.leg(5,2)
[(5, 3)]
```

size()

Return the number of boxes in `self`.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.size()
22
```

class sage.combinat.sf.ns_macdonald.**NonattackingBacktracker**(*shape*, *pi=None*)

Bases: sage.combinat.backtrack.GenericBacktracker

EXAMPLES:

```
sage: from sage.combinat.sf.ns_macdonald import NonattackingBacktracker
sage: n = NonattackingBacktracker(LatticeDiagram([0,1,2]))
sage: n._ending_position
(3, 2)
```

```
sage: n._initial_state
(2, 1)
```

get_next_pos(*ii*, *jj*)

EXAMPLES:

```
sage: from sage.combinat.sf.ns_macdonald import NonattackingBacktracker
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: n = NonattackingBacktracker(a.shape())
sage: n.get_next_pos(1, 1)
(2, 1)
sage: n.get_next_pos(6, 1)
(1, 2)
sage: n = NonattackingBacktracker(LatticeDiagram([2,2,2]))
sage: n.get_next_pos(3, 1)
(1, 2)
```

sage.combinat.sf.ns_macdonald.**NonattackingFillings**(*shape*, *pi=None*)

Returning the combinatorial class of nonattacking fillings of a given shape.

EXAMPLES:

```
sage: NonattackingFillings([0,1,2])
Nonattacking fillings of [0, 1, 2]
sage: NonattackingFillings([0,1,2]).list()
[[[1], [2, 1], [3, 2, 1]],
 [[1], [2, 1], [3, 2, 2]],
 [[1], [2, 1], [3, 2, 3]],
 [[1], [2, 1], [3, 3, 1]],
 [[1], [2, 1], [3, 3, 2]],
 [[1], [2, 1], [3, 3, 3]],
 [[1], [2, 2], [3, 1, 1]],
 [[1], [2, 2], [3, 1, 2]],
 [[1], [2, 2], [3, 1, 3]],
 [[1], [2, 2], [3, 3, 1]],
 [[1], [2, 2], [3, 3, 2]],
 [[1], [2, 2], [3, 3, 3]]]
```

class sage.combinat.sf.ns_macdonald.**NonattackingFillings_shape**(*shape*, *pi=None*)

Bases: sage.combinat.combinat.CombinatorialClass

EXAMPLES:

```
sage: n = NonattackingFillings([0,1,2])
sage: n == loads(dumps(n))
True
```

flip()

Returns the nonattacking fillings of the the flipped shape.

EXAMPLES:

```
sage: NonattackingFillings([0,1,2]).flip()
Nonattacking fillings of [2, 1, 0]
```

5.1.262 Orthogonal Symmetric Functions

AUTHORS:

- Travis Scrimshaw (2013-11-10): Initial version

class sage.combinat.sf.orthogonal.**SymmetricFunctionAlgebra_orthogonal**(*Sym*)
Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic

The orthogonal symmetric function basis (or orthogonal basis, to be short).

The orthogonal basis $\{o_\lambda\}$ where λ is taken over all partitions is defined by the following change of basis with the Schur functions:

$$s_\lambda = \sum_\mu \left(\sum_{\nu \in H} c^\lambda_{\mu\nu} \right) o_\mu$$

where H is the set of all partitions with even-width rows and $c^\lambda_{\mu\nu}$ is the usual Littlewood-Richardson (LR) coefficients. By the properties of LR coefficients, this can be shown to be a upper unitriangular change of basis.

Note: This is only a filtered basis, not a $\mathbf{Z}$-graded basis. However this does respect the induced $(\mathbf{Z}/2\mathbf{Z})$-grading.

INPUT:

- Sym – an instance of the ring of the symmetric functions

REFERENCES:

- [ChariKleber2000]
- [KoikeTerada1987]
- [ShimozonoZabrocki2006]

EXAMPLES:

Here are the first few orthogonal symmetric functions, in various bases:

```
sage: Sym = SymmetricFunctions(QQ)
sage: o = Sym.o()
sage: e = Sym.e()
sage: h = Sym.h()
sage: p = Sym.p()
sage: s = Sym.s()
sage: m = Sym.m()

sage: p(o([1]))
p[1]
sage: m(o([1]))
m[1]
sage: e(o([1]))
e[1]
sage: h(o([1]))
h[1]
sage: s(o([1]))
s[1]

sage: p(o([2]))
-p[] + 1/2*p[1, 1] + 1/2*p[2]
sage: m(o([2]))
-m[] + m[1, 1] + m[2]
sage: e(o([2]))
-e[] + e[1, 1] - e[2]
sage: h(o([2]))
-h[] + h[2]
```

```
sage: s(o([2]))
-s[] + s[2]

sage: p(o([3]))
-p[1] + 1/6*p[1, 1, 1] + 1/2*p[2, 1] + 1/3*p[3]
sage: m(o([3]))
-m[1] + m[1, 1, 1] + m[2, 1] + m[3]
sage: e(o([3]))
-e[1] + e[1, 1, 1] - 2*e[2, 1] + e[3]
sage: h(o([3]))
-h[1] + h[3]
sage: s(o([3]))
-s[1] + s[3]

sage: Sym = SymmetricFunctions(ZZ)
sage: o = Sym.o()
sage: e = Sym.e()
sage: h = Sym.h()
sage: s = Sym.s()
sage: m = Sym.m()
sage: p = Sym.p()
sage: m(o([4]))
-m[1, 1] + m[1, 1, 1, 1] - m[2] + m[2, 1, 1] + m[2, 2] + m[3, 1] + m[4]
sage: e(o([4]))
-e[1, 1] + e[1, 1, 1, 1] + e[2] - 3*e[2, 1, 1] + e[2, 2] + 2*e[3, 1] - e[4]
sage: h(o([4]))
-h[2] + h[4]
sage: s(o([4]))
-s[2] + s[4]
```

Some examples of conversions the other way:

```
sage: o(h[3])
o[1] + o[3]
sage: o(e[3])
o[1, 1, 1]
sage: o(m[2,1])
o[1] - 2*o[1, 1, 1] + o[2, 1]
sage: o(p[3])
o[1, 1, 1] - o[2, 1] + o[3]
```

Some multiplication:

```
sage: o([2]) * o([1,1])
o[1, 1] + o[2] + o[2, 1, 1] + o[3, 1]
sage: o([2,1,1]) * o([2])
o[1, 1] + o[1, 1, 1, 1] + 2*o[2, 1, 1] + o[2, 2] + o[2, 2, 1, 1]
 + o[3, 1] + o[3, 1, 1, 1] + o[3, 2, 1] + o[4, 1, 1]
sage: o([1,1]) * o([2,1])
o[1] + o[1, 1, 1] + 2*o[2, 1] + o[2, 1, 1, 1] + o[2, 2, 1]
 + o[3] + o[3, 1, 1] + o[3, 2]
```

Examples of the Hopf algebra structure:

```
sage: o([1]).antipode()
-o[1]
sage: o([2]).antipode()
-o[] + o[1, 1]
sage: o([1]).coproduct()
```

```
o[] # o[1] + o[1] # o[]
sage: o([2]).coproduct()
o[] # o[] + o[] # o[2] + o[1] # o[1] + o[2] # o[]
sage: o([1]).counit()
0
sage: o.one().counit()
1
```

5.1.263 Symmetric functions defined by orthogonality and triangularity.

One characterization of Schur functions is that they are upper triangularly related to the monomial symmetric functions and orthogonal with respect to the Hall scalar product. We can use the class SymmetricFunctionAlgebra_orthotriang to obtain the Schur functions from this definition.

```
sage: from sage.combinat.sf.sfa import zee
sage: from sage.combinat.sf.orthotriang import SymmetricFunctionAlgebra_orthotriang
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.m()
sage: s = SymmetricFunctionAlgebra_orthotriang(Sym, m, zee, 's', 'Schur functions')
sage: s([2,1])^2
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[3, 3] + s[4, 1, 1] + s[4, 2]

sage: s2 = SymmetricFunctions(QQ).s()
sage: s2([2,1])^2
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[3, 3] + s[4, 1, 1] + s[4, 2]
```

class sage.combinat.sf.orthotriang.**SymmetricFunctionAlgebra_orthotriang**(*Sym*, *base*, *scalar*, *prefix*, *basis_name*, *leading_coeff=None*)

Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic

Initialization of the symmetric function algebra defined via orthotriangular rules.

INPUT:

- self – a basis determined by an orthotriangular definition
- Sym – ring of symmetric functions
- base – an instance of a basis of the ring of symmetric functions (e.g. the Schur functions)
- scalar – a function zee on partitions. The function zee determines the scalar product on the power sum basis with normalization $\langle p_\mu, p_\mu \rangle = \mathrm{zee}(\mu)$.
- prefix – the prefix used to display the basis
- basis_name – the name used for the basis

Note: The base ring is required to be a **Q**-algebra for this method to be useable, since the scalar product is defined by its values on the power sum basis.

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import zee
sage: from sage.combinat.sf.orthotriang import SymmetricFunctionAlgebra_orthotriang
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.m()
sage: s = SymmetricFunctionAlgebra_orthotriang(Sym, m, zee, 's', 'Schur'); s
Symmetric Functions over Rational Field in the Schur basis
```

TESTS:

```
sage: TestSuite(s).run(elements = [s[1,1]+2*s[2], s[1]+3*s[1,1]])
sage: TestSuite(s).run(skip = ["_test_associativity", "_test_prod"])  # long time (7s on sage.ma
```

Note: `s.an_element()` is of degree 4; so we skip `_test_associativity` and `_test_prod` which involve (currently?) expensive calculations up to degree 12.

class Element(*M*, *x*)

Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic_Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

5.1.264 Power sum symmetric functions

class sage.combinat.sf.powersum.**SymmetricFunctionAlgebra_power**(*Sym*)

Bases: sage.combinat.sf.multiplicative.SymmetricFunctionAlgebra_multiplicative

A class for methods associated to the power sum basis of the symmetric functions

INPUT:

- `self` – the power sum basis of the symmetric functions
- `Sym` – an instance of the ring of symmetric functions

TESTS:

```
sage: p = SymmetricFunctions(QQ).p()
sage: p == loads(dumps(p))
True
sage: TestSuite(p).run(skip=['_test_associativity', '_test_distributivity', '_test_prod'])
sage: TestSuite(p).run(elements = [p[1,1]+p[2], p[1]+2*p[1,1]])
```

class Element(*M*, *x*)

Bases: sage.combinat.sf.classical.SymmetricFunctionAlgebra_classical.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

adams_operation(*n*)

Return the image of the symmetric function `self` under the n-th Frobenius operator.

The n-th Frobenius operator $\mathbf{f}_n$ is defined to be the map from the ring of symmetric functions to itself that sends every symmetric function $P(x_1, x_2, x_3, \ldots)$ to $P(x_1^n, x_2^n, x_3^n, \ldots)$. This operator $\mathbf{f}_n$ is a Hopf algebra endomorphism, and satisfies

$$\mathbf{f}_n m_{(\lambda_1,\lambda_2,\lambda_3,\ldots)} = m_{(n\lambda_1,n\lambda_2,n\lambda_3,\ldots)}$$

for every partition $(\lambda_1, \lambda_2, \lambda_3, \ldots)$ (where m means the monomial basis). Moreover, $\mathbf{f}_n(p_r) = p_{nr}$ for every positive integer r (where p_k denotes the k-th powersum symmetric function).

The n-th Frobenius operator is also called the n-th Frobenius endomorphism. It is not related to the Frobenius map which connects the ring of symmetric functions with the representation theory of the symmetric group.

The n-th Frobenius operator is also the n-th Adams operator of the Λ-ring of symmetric functions over the integers.

The n-th Frobenius operator can also be described via plethysm: Every symmetric function P satisfies $\mathbf{f}_n(P) = p_n \circ P = P \circ p_n$, where p_n is the n-th powersum symmetric function, and $\circ$ denotes (outer) plethysm.

INPUT:

- `n` – a positive integer

OUTPUT:

The result of applying the n-th Frobenius operator (on the ring of symmetric functions) to `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: p = Sym.p()
sage: p[3].frobenius(2)
p[6]
sage: p[4,2,1].frobenius(3)
p[12, 6, 3]
sage: p([]).frobenius(4)
p[]
sage: p[3].frobenius(1)
p[3]
sage: (p([3]) - p([2]) + p([])).frobenius(3)
p[] - p[6] + p[9]
```

TESTS:

Let us check that this method on the powersum basis gives the same result as the implementation in `sage.combinat.sf.sfa` on the complete homogeneous basis:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.p(); h = Sym.h()
sage: all( h(p(lam)).frobenius(3) == h(p(lam).frobenius(3))
....:      for lam in Partitions(3) )
True
```

```
sage: all( p(h(lam)).frobenius(2) == p(h(lam).frobenius(2))
....:      for lam in Partitions(4) )
True
```

See also:

plethysm()

eval_at_permutation_roots(*rho*)

Evaluate at eigenvalues of a permutation matrix.

Evaluate an element of the power sum basis at the eigenvalues of a permutation matrix with cycle structure ρ.

This function evaluates an element at the roots of unity

$$\Xi_{\rho_1}, \Xi_{\rho_2}, \ldots, \Xi_{\rho_\ell}$$

where

$$\Xi_m = 1, \zeta_m, \zeta_m^2, \ldots, \zeta_m^{m-1}$$

and ζ_m is an m root of unity. These roots of unity represent the eigenvalues of permutation matrix with cycle structure ρ.

INPUT:

- rho – a partition or a list of non-negative integers

OUTPUT:

- an element of the base ring

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: p([3,3]).eval_at_permutation_roots([6])
0
sage: p([3,3]).eval_at_permutation_roots([3])
9
sage: p([3,3]).eval_at_permutation_roots([1])
1
sage: p([3,3]).eval_at_permutation_roots([3,3])
36
sage: p([3,3]).eval_at_permutation_roots([1,1,1,1,1])
25
sage: (p[1]+p[2]+p[3]).eval_at_permutation_roots([3,2])
5
```

expand(*n*, *alphabet='x'*)

Expand the symmetric function self as a symmetric polynomial in n variables.

INPUT:

- n – a nonnegative integer
- alphabet – (default: 'x') a variable for the expansion

OUTPUT:

A monomial expansion of self in the n variables labelled by alphabet.

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: a = p([2])
sage: a.expand(2)
x0^2 + x1^2
sage: a.expand(3, alphabet=['a','b','c'])
```

```
a^2 + b^2 + c^2
sage: p([2,1,1]).expand(2)
x0^4 + 2*x0^3*x1 + 2*x0^2*x1^2 + 2*x0*x1^3 + x1^4
sage: p([7]).expand(4)
x0^7 + x1^7 + x2^7 + x3^7
sage: p([7]).expand(4,alphabet='t')
t0^7 + t1^7 + t2^7 + t3^7
sage: p([7]).expand(4,alphabet='x,y,z,t')
x^7 + y^7 + z^7 + t^7
sage: p(1).expand(4)
1
sage: p(0).expand(4)
0
sage: (p([]) + 2*p([1])).expand(3)
2*x0 + 2*x1 + 2*x2 + 1
sage: p([1]).expand(0)
0
sage: (3*p([])).expand(0)
3
```

frobenius(*n*)

Return the image of the symmetric function `self` under the n-th Frobenius operator.

The n-th Frobenius operator $\mathbf{f}_n$ is defined to be the map from the ring of symmetric functions to itself that sends every symmetric function $P(x_1, x_2, x_3, \ldots)$ to $P(x_1^n, x_2^n, x_3^n, \ldots)$. This operator $\mathbf{f}_n$ is a Hopf algebra endomorphism, and satisfies

$$\mathbf{f}_n m_{(\lambda_1,\lambda_2,\lambda_3,\ldots)} = m_{(n\lambda_1,n\lambda_2,n\lambda_3,\ldots)}$$

for every partition $(\lambda_1, \lambda_2, \lambda_3, \ldots)$ (where m means the monomial basis). Moreover, $\mathbf{f}_n(p_r) = p_{nr}$ for every positive integer r (where p_k denotes the k-th powersum symmetric function).

The n-th Frobenius operator is also called the n-th Frobenius endomorphism. It is not related to the Frobenius map which connects the ring of symmetric functions with the representation theory of the symmetric group.

The n-th Frobenius operator is also the n-th Adams operator of the Λ-ring of symmetric functions over the integers.

The n-th Frobenius operator can also be described via plethysm: Every symmetric function P satisfies $\mathbf{f}_n(P) = p_n \circ P = P \circ p_n$, where p_n is the n-th powersum symmetric function, and $\circ$ denotes (outer) plethysm.

INPUT:

- `n` – a positive integer

OUTPUT:

The result of applying the n-th Frobenius operator (on the ring of symmetric functions) to `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: p = Sym.p()
sage: p[3].frobenius(2)
p[6]
sage: p[4,2,1].frobenius(3)
p[12, 6, 3]
sage: p([]).frobenius(4)
p[]
sage: p[3].frobenius(1)
p[3]
```

```
sage: (p([3]) - p([2]) + p([])).frobenius(3)
p[] - p[6] + p[9]
```

TESTS:

Let us check that this method on the powersum basis gives the same result as the implementation in `sage.combinat.sf.sfa` on the complete homogeneous basis:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.p(); h = Sym.h()
sage: all( h(p(lam)).frobenius(3) == h(p(lam).frobenius(3))
....:      for lam in Partitions(3) )
True
sage: all( p(h(lam)).frobenius(2) == p(h(lam).frobenius(2))
....:      for lam in Partitions(4) )
True
```

See also:

`plethysm()`

omega()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1}p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda|-\ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the `omega()` method.

OUTPUT:

- the image of `self` under the omega automorphism

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: a = p([2,1]); a
p[2, 1]
sage: a.omega()
-p[2, 1]
sage: p([]).omega()
p[]
sage: p(0).omega()
0
sage: p = SymmetricFunctions(ZZ).p()
sage: (p([3,1,1]) - 2 * p([2,1])).omega()
2*p[2, 1] + p[3, 1, 1]
```

omega_involution()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th

elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1}p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda|-\ell(\lambda)}p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the `omega()` method.

OUTPUT:

- the image of `self` under the omega automorphism

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: a = p([2,1]); a
p[2, 1]
sage: a.omega()
-p[2, 1]
sage: p([]).omega()
p[]
sage: p(0).omega()
0
sage: p = SymmetricFunctions(ZZ).p()
sage: (p([3,1,1]) - 2 * p([2,1])).omega()
2*p[2, 1] + p[3, 1, 1]
```

scalar(*x*, *zee=None*)

Return the standard scalar product of `self` and `x`.

INPUT:

- `x` – a power sum symmetric function
- `zee` – (default: uses standard `zee` function) optional input specifying the scalar product on the power sum basis with normalization $\langle p_\mu, p_\mu \rangle = \mathrm{zee}(\mu)$. `zee` should be a function on partitions.

Note that the power-sum symmetric functions are orthogonal under this scalar product. With the default value of `zee`, the value of $\langle p_\lambda, p_\lambda \rangle$ is given by the size of the centralizer in S_n of a permutation of cycle type λ.

OUTPUT:

- the standard scalar product between `self` and `x`, or, if the optional parameter `zee` is specified, then the scalar product with respect to the normalization $\langle p_\mu, p_\mu \rangle = \mathrm{zee}(\mu)$ with the power sum basis elements being orthogonal

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: p4 = Partitions(4)
sage: matrix([ [p(a).scalar(p(b)) for a in p4] for b in p4])
[ 4  0  0  0  0]
[ 0  3  0  0  0]
[ 0  0  8  0  0]
[ 0  0  0  4  0]
[ 0  0  0  0 24]
sage: p(0).scalar(p(1))
0
sage: p(1).scalar(p(2))
2
```

```
sage: zee = lambda x : 1
sage: matrix( [[p[la].scalar(p[mu], zee) for la in Partitions(3)] for mu in Partitions(3
[1 0 0]
[0 1 0]
[0 0 1]
```

verschiebung(*n*)

Return the image of the symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the unique algebra endomorphism V of the ring of symmetric functions that satisfies $V(h_r) = h_{r/n}$ for every positive integer r divisible by n, and satisfies $V(h_r) = 0$ for every positive integer r not divisible by n. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every nonnegative integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(h_r) = h_{r/n}, \quad \mathbf{V}_n(p_r) = np_{r/n}, \quad \mathbf{V}_n(e_r) = (-1)^{r-r/n} e_{r/n}$$

(where h is the complete homogeneous basis, p is the powersum basis, and e is the elementary basis). For every nonnegative integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(h_r) = \mathbf{V}_n(p_r) = \mathbf{V}_n(e_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism. Its name derives from the Verschiebung (German for "shift") endomorphism of the Witt vectors.

The n-th Verschiebung operator is adjoint to the n-th Frobenius operator (see `frobenius()` for its definition) with respect to the Hall scalar product (`scalar()`).

The action of the n-th Verschiebung operator on the Schur basis can also be computed explicitly. The following (probably clumsier than necessary) description can be obtained by solving exercise 7.61 in Stanley's [STA].

Let λ be a partition. Let n be a positive integer. If the n-core of λ is nonempty, then $\mathbf{V}_n(s_\lambda) = 0$. Otherwise, the following method computes $\mathbf{V}_n(s_\lambda)$: Write the partition λ in the form $(\lambda_1, \lambda_2, \ldots, \lambda_{ns})$ for some nonnegative integer s. (If n does not divide the length of λ, then this is achieved by adding trailing zeroes to λ.) Set $\beta_i = \lambda_i + ns - i$ for every $s \in \{1, 2, \ldots, ns\}$. Then, $(\beta_1, \beta_2, \ldots, \beta_{ns})$ is a strictly decreasing sequence of nonnegative integers. Stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $-1 - \beta_i$ modulo n. Let ξ be the sign of the permutation that is used for this sorting. Let ψ be the sign of the permutation that is used to stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $i - 1$ modulo n. (Notice that $\psi = (-1)^{n(n-1)s(s-1)/4}$.) Then, $\mathbf{V}_n(s_\lambda) = \xi\psi \prod_{i=0}^{n-1} s_{\lambda^{(i)}}$, where $(\lambda^{(0)}, \lambda^{(1)}, \ldots, \lambda^{(n-1)})$ is the n-quotient of λ.

INPUT:

- `n` – a positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of symmetric functions) to `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: p = Sym.p()
sage: p[3].verschiebung(2)
0
sage: p[4].verschiebung(4)
4*p[1]
```

The Verschiebung endomorphisms are multiplicative:

```
sage: all( all( p(lam).verschiebung(2) * p(mu).verschiebung(2)
....:           == (p(lam) * p(mu)).verschiebung(2)
....:           for mu in Partitions(4) )
```

```
....:       for lam in Partitions(4) )
True
```

Testing the adjointness between the Frobenius operators $\mathbf{f}_n$ and the Verschiebung operators $\mathbf{V}_n$:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.p()
sage: all( all( p(lam).verschiebung(2).scalar(p(mu))
....:           == p(lam).scalar(p(mu).frobenius(2))
....:           for mu in Partitions(2) )
....:       for lam in Partitions(4) )
True
```

TESTS:

Let us check that this method on the powersum basis gives the same result as the implementation in `sage.combinat.sf.sfa` on the monomial basis:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.p(); m = Sym.m()
sage: all( m(p(lam)).verschiebung(3) == m(p(lam).verschiebung(3))
....:       for lam in Partitions(6) )
True
sage: all( p(m(lam)).verschiebung(2) == p(m(lam).verschiebung(2))
....:       for lam in Partitions(4) )
True
```

`SymmetricFunctionAlgebra_power.`**`antipode_on_basis`**(*partition*)

Return the antipode of `self[partition]`.

The antipode on the generator p_i (for $i > 0$) is $-p_i$, and the antipode on p_μ is $(-1)^{length(\mu)} p_\mu$.

INPUT:

- `self` – the power sum basis of the symmetric functions
- `partition` – a partition

OUTPUT:

- the result of the antipode on `self(partition)`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.p()
sage: p.antipode_on_basis([2])
-p[2]
sage: p.antipode_on_basis([3])
-p[3]
sage: p.antipode_on_basis([2,2])
p[2, 2]
sage: p.antipode_on_basis([])
p[]
```

`SymmetricFunctionAlgebra_power.`**`bottom_schur_function`**(*partition*, *degree=None*)

Return the least-degree component of `s[partition]`, where `s` denotes the Schur basis of the symmetric functions, and the grading is not the usual grading on the symmetric functions but rather the grading which gives every p_i degree 1.

This least-degree component has its degree equal to the Frobenius rank of `partition`, while the degree with respect to the usual grading is still the size of `partition`.

This method requires the base ring to be a (commutative) $\mathbf{Q}$-algebra. This restriction is unavoidable, since the least-degree component (in general) has noninteger coefficients in all classical bases of the symmetric functions.

The optional keyword `degree` allows taking any homogeneous component rather than merely the least-degree one. Specifically, if `degree` is set, then the `degree`-th component will be returned.

REFERENCES:

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.p()
sage: p.bottom_schur_function([2,2,1])
-1/6*p[3, 2] + 1/4*p[4, 1]
sage: p.bottom_schur_function([2,1])
-1/3*p[3]
sage: p.bottom_schur_function([3])
1/3*p[3]
sage: p.bottom_schur_function([1,1,1])
1/3*p[3]
sage: p.bottom_schur_function(Partition([1,1,1]))
1/3*p[3]
sage: p.bottom_schur_function([2,1], degree=1)
-1/3*p[3]
sage: p.bottom_schur_function([2,1], degree=2)
0
sage: p.bottom_schur_function([2,1], degree=3)
1/3*p[1, 1, 1]
sage: p.bottom_schur_function([2,2,1], degree=3)
1/8*p[2, 2, 1] - 1/6*p[3, 1, 1]
```

SymmetricFunctionAlgebra_power.**coproduct_on_generators**(i)

Return coproduct on generators for power sums p_i (for $i > 0$).

The elements p_i are primitive elements.

INPUT:

- `self` – the power sum basis of the symmetric functions
- `i` – a positive integer

OUTPUT:

- the result of the coproduct on the generator $p(i)$

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.powersum()
sage: p.coproduct_on_generators(2)
p[] # p[2] + p[2] # p[]
```

SymmetricFunctionAlgebra_power.**eval_at_permutation_roots_on_generators**(*k*, *rho*)

Evaluate p_k at eigenvalues of permutation matrix.

This function evaluates a symmetric function `p([k])` at the eigenvalues of a permutation matrix with cycle structure `\rho`.

This function evaluates a p_k at the roots of unity

$$\Xi_{\rho_1}, \Xi_{\rho_2}, \ldots, \Xi_{\rho_\ell}$$

where

$$\Xi_m = 1, \zeta_m, \zeta_m^2, \ldots, \zeta_m^{m-1}$$

and ζ_m is an m root of unity. This is characterized by $p_k[A, B] = p_k[A] + p_k[B]$ and $p_k[\Xi_m] = 0$ unless m divides k and $p_{rm}[\Xi_m] = m$.

INPUT:

- k – a non-negative integer
- rho – a partition or a list of non-negative integers

OUTPUT:

- an element of the base ring

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: p.eval_at_permutation_roots_on_generators(3, [6])
0
sage: p.eval_at_permutation_roots_on_generators(3, [3])
3
sage: p.eval_at_permutation_roots_on_generators(3, [1])
1
sage: p.eval_at_permutation_roots_on_generators(3, [3,3])
6
sage: p.eval_at_permutation_roots_on_generators(3, [1,1,1,1,1])
5
```

5.1.265 Schur symmetric functions

class sage.combinat.sf.schur.**SymmetricFunctionAlgebra_schur**(*Sym*)

Bases: sage.combinat.sf.classical.SymmetricFunctionAlgebra_classical

A class for methods related to the Schur symmetric function basis

INPUT:

- self – a Schur symmetric function basis
- Sym – an instance of the ring of the symmetric functions

TESTS:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s == loads(dumps(s))
True
sage: TestSuite(s).run(skip=['_test_associativity', '_test_distributivity', '_test_prod'])
sage: TestSuite(s).run(elements = [s[1,1]+s[2], s[1]+2*s[1,1]])
```

class Element(*M*, *x*)

Bases: sage.combinat.sf.classical.SymmetricFunctionAlgebra_classical.Element

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
```

```
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

expand(*n*, *alphabet='x'*)

Expand the symmetric function `self` as a symmetric polynomial in `n` variables.

INPUT:

- `n` – a nonnegative integer
- `alphabet` – (default: `'x'`) a variable for the expansion

OUTPUT:

A monomial expansion of `self` in the n variables labelled by `alphabet`.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: a.expand(2)
x0^2*x1 + x0*x1^2
sage: a.expand(3)
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
sage: a.expand(4)
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2 + x0^2*x3 + 2*x0*
sage: a.expand(2, alphabet='y')
y0^2*y1 + y0*y1^2
sage: a.expand(2, alphabet=['a','b'])
a^2*b + a*b^2
sage: s([1,1,1,1]).expand(3)
0
sage: (s([]) + 2*s([1])).expand(3)
2*x0 + 2*x1 + 2*x2 + 1
sage: s([1]).expand(0)
0
sage: (3*s([])).expand(0)
3
```

omega()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1}p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda|-\ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the `omega()` method.

OUTPUT:

- the image of `self` under the omega automorphism

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([2,1]).omega()
s[2, 1]
sage: s([2,1,1]).omega()
s[3, 1]
```

omega_involution()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1}p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda|-\ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

`omega_involution()` is a synonym for the `omega()` method.

OUTPUT:

- the image of `self` under the omega automorphism

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([2,1]).omega()
s[2, 1]
sage: s([2,1,1]).omega()
s[3, 1]
```

scalar(*x*, *zee=None*)

Return the standard scalar product between `self` and x.

Note that the Schur functions are self-dual with respect to this scalar product. They are also lower-triangularly related to the monomial symmetric functions with respect to this scalar product.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`
- `zee` – an optional function on partitions giving the value for the scalar product between the power-sum symmetric function p_μ and itself (the default value is the standard `zee()` function)

OUTPUT:

- the scalar product between `self` and `x`

EXAMPLES:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: a = s([2,1])
sage: b = s([1,1,1])
sage: c = 2*s([1,1,1])
sage: d = a + b
sage: a.scalar(a)
1
sage: b.scalar(b)
1
sage: b.scalar(a)
0
```

```
sage: b.scalar(c)
2
sage: c.scalar(c)
4
sage: d.scalar(a)
1
sage: d.scalar(b)
1
sage: d.scalar(c)
2

sage: m = SymmetricFunctions(ZZ).monomial()
sage: p4 = Partitions(4)
sage: l = [ [s(p).scalar(m(q)) for q in p4] for p in p4]
sage: matrix(l)
[ 1  0  0  0  0]
[-1  1  0  0  0]
[ 0 -1  1  0  0]
[ 1 -1 -1  1  0]
[-1  2  1 -3  1]
```

verschiebung(*n*)

Return the image of the symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the unique algebra endomorphism V of the ring of symmetric functions that satisfies $V(h_r) = h_{r/n}$ for every positive integer r divisible by n, and satisfies $V(h_r) = 0$ for every positive integer r not divisible by n. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every nonnegative integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(h_r) = h_{r/n}, \quad \mathbf{V}_n(p_r) = np_{r/n}, \quad \mathbf{V}_n(e_r) = (-1)^{r-r/n} e_{r/n}$$

(where h is the complete homogeneous basis, p is the powersum basis, and e is the elementary basis). For every nonnegative integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(h_r) = \mathbf{V}_n(p_r) = \mathbf{V}_n(e_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism. Its name derives from the Verschiebung (German for "shift") endomorphism of the Witt vectors.

The n-th Verschiebung operator is adjoint to the n-th Frobenius operator (see `frobenius()` for its definition) with respect to the Hall scalar product (`scalar()`).

The action of the n-th Verschiebung operator on the Schur basis can also be computed explicitly. The following (probably clumsier than necessary) description can be obtained by solving exercise 7.61 in Stanley's [STA].

Let λ be a partition. Let n be a positive integer. If the n-core of λ is nonempty, then $\mathbf{V}_n(s_\lambda) = 0$. Otherwise, the following method computes $\mathbf{V}_n(s_\lambda)$: Write the partition λ in the form $(\lambda_1, \lambda_2, \ldots, \lambda_{ns})$ for some nonnegative integer s. (If n does not divide the length of λ, then this is achieved by adding trailing zeroes to λ.) Set $\beta_i = \lambda_i + ns - i$ for every $s \in \{1, 2, \ldots, ns\}$. Then, $(\beta_1, \beta_2, \ldots, \beta_{ns})$ is a strictly decreasing sequence of nonnegative integers. Stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $-1 - \beta_i$ modulo n. Let ξ be the sign of the permutation that is used for this sorting. Let ψ be the sign of the permutation that is used to stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $i - 1$ modulo n. (Notice that $\psi = (-1)^{n(n-1)s(s-1)/4}$.) Then, $\mathbf{V}_n(s_\lambda) = \xi\psi \prod_{i=0}^{n-1} s_{\lambda^{(i)}}$, where $(\lambda^{(0)}, \lambda^{(1)}, \ldots, \lambda^{(n-1)})$ is the n-quotient of λ.

INPUT:

- `n` – a positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of symmetric functions) to `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: s = Sym.s()
sage: s[5].verschiebung(2)
0
sage: s[6].verschiebung(6)
s[1]
sage: s[6,3].verschiebung(3)
s[2, 1] + s[3]
sage: s[6,3,1].verschiebung(2)
-s[3, 2]
sage: s[3,2,1].verschiebung(1)
s[3, 2, 1]
sage: s([]).verschiebung(1)
s[]
sage: s([]).verschiebung(4)
s[]
```

TESTS:

Let us check that this method on the powersum basis gives the same result as the implementation in sfa.py on the monomial basis:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s(); h = Sym.h()
sage: all( h(s(lam)).verschiebung(3) == h(s(lam).verschiebung(3))
....:      for lam in Partitions(6) )
True
sage: all( s(h(lam)).verschiebung(2) == s(h(lam).verschiebung(2))
....:      for lam in Partitions(4) )
True
sage: all( s(h(lam)).verschiebung(5) == s(h(lam).verschiebung(5))
....:      for lam in Partitions(10) )
True
sage: all( s(h(lam)).verschiebung(2) == s(h(lam).verschiebung(2))
....:      for lam in Partitions(8) )
True
sage: all( s(h(lam)).verschiebung(3) == s(h(lam).verschiebung(3))
....:      for lam in Partitions(12) )
True
sage: all( s(h(lam)).verschiebung(3) == s(h(lam).verschiebung(3))
....:      for lam in Partitions(9) )
True
```

SymmetricFunctionAlgebra_schur.**coproduct_on_basis**(*mu*)

Returns the coproduct of `self(mu)`.

Here `self` is the basis of Schur functions in the ring of symmetric functions.

INPUT:

- `self` – a Schur symmetric function basis
- `mu` – a partition

OUTPUT:

- the image of the `mu`-th Schur function under the comultiplication of the Hopf algebra of symmetric functions; this is an element of the tensor square of the Schur basis

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: s.coproduct_on_basis([2])
s[] # s[2] + s[1] # s[1] + s[2] # s[]
```

5.1.266 Symplectic Symmetric Functions

AUTHORS:

- Travis Scrimshaw (2013-11-10): Initial version

class sage.combinat.sf.symplectic.**SymmetricFunctionAlgebra_symplectic**(*Sym*)
Bases: sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic

The symplectic symmetric function basis (or symplectic basis, to be short).

The symplectic basis $\{sp_\lambda\}$ where λ is taken over all partitions is defined by the following change of basis with the Schur functions:

$$s_\lambda = \sum_\mu \left(\sum_{\nu \in V} c^\lambda_{\mu\nu} \right) sp_\mu$$

where V is the set of all partitions with even-height columns and $c^\lambda_{\mu\nu}$ is the usual Littlewood-Richardson (LR) coefficients. By the properties of LR coefficients, this can be shown to be a upper unitriangular change of basis.

Note: This is only a filtered basis, not a $\mathbf{Z}$-graded basis. However this does respect the induced $(\mathbf{Z}/2\mathbf{Z})$-grading.

INPUT:

- Sym – an instance of the ring of the symmetric functions

REFERENCES:

EXAMPLES:

Here are the first few symplectic symmetric functions, in various bases:

```
sage: Sym = SymmetricFunctions(QQ)
sage: sp = Sym.sp()
sage: e = Sym.e()
sage: h = Sym.h()
sage: p = Sym.p()
sage: s = Sym.s()
sage: m = Sym.m()

sage: p(sp([1]))
p[1]
sage: m(sp([1]))
m[1]
sage: e(sp([1]))
e[1]
sage: h(sp([1]))
h[1]
sage: s(sp([1]))
s[1]

sage: p(sp([2]))
```

```
1/2*p[1, 1] + 1/2*p[2]
sage: m(sp([2]))
m[1, 1] + m[2]
sage: e(sp([2]))
e[1, 1] - e[2]
sage: h(sp([2]))
h[2]
sage: s(sp([2]))
s[2]

sage: p(sp([3]))
1/6*p[1, 1, 1] + 1/2*p[2, 1] + 1/3*p[3]
sage: m(sp([3]))
m[1, 1, 1] + m[2, 1] + m[3]
sage: e(sp([3]))
e[1, 1, 1] - 2*e[2, 1] + e[3]
sage: h(sp([3]))
h[3]
sage: s(sp([3]))
s[3]

sage: Sym = SymmetricFunctions(ZZ)
sage: sp = Sym.sp()
sage: e = Sym.e()
sage: h = Sym.h()
sage: s = Sym.s()
sage: m = Sym.m()
sage: p = Sym.p()
sage: m(sp([4]))
m[1, 1, 1, 1] + m[2, 1, 1] + m[2, 2] + m[3, 1] + m[4]
sage: e(sp([4]))
e[1, 1, 1, 1] - 3*e[2, 1, 1] + e[2, 2] + 2*e[3, 1] - e[4]
sage: h(sp([4]))
h[4]
sage: s(sp([4]))
s[4]
```

Some examples of conversions the other way:

```
sage: sp(h[3])
sp[3]
sage: sp(e[3])
sp[1] + sp[1, 1, 1]
sage: sp(m[2,1])
-sp[1] - 2*sp[1, 1, 1] + sp[2, 1]
sage: sp(p[3])
sp[1, 1, 1] - sp[2, 1] + sp[3]
```

Some multiplication:

```
sage: sp([2]) * sp([1,1])
sp[1, 1] + sp[2] + sp[2, 1, 1] + sp[3, 1]
sage: sp([2,1,1]) * sp([2])
sp[1, 1] + sp[1, 1, 1, 1] + 2*sp[2, 1, 1] + sp[2, 2] + sp[2, 2, 1, 1]
 + sp[3, 1] + sp[3, 1, 1, 1] + sp[3, 2, 1] + sp[4, 1, 1]
sage: sp([1,1]) * sp([2,1])
sp[1] + sp[1, 1, 1] + 2*sp[2, 1] + sp[2, 1, 1, 1] + sp[2, 2, 1]
 + sp[3] + sp[3, 1, 1] + sp[3, 2]
```

Examples of the Hopf algebra structure:

```
sage: sp([1]).antipode()
-sp[1]
sage: sp([2]).antipode()
sp[] + sp[1, 1]
sage: sp([1]).coproduct()
sp[] # sp[1] + sp[1] # sp[]
sage: sp([2]).coproduct()
sp[] # sp[2] + sp[1] # sp[1] + sp[2] # sp[]
sage: sp([1]).counit()
0
sage: sp.one().counit()
1
```

5.1.267 Symmetric functions, with their multiple realizations

class sage.combinat.sf.sf.**SymmetricFunctions**(*R*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The abstract algebra of commutative symmetric functions

Symmetric Functions in Sage

This document is an introduction to working with symmetric function theory in Sage. It is not intended to be an introduction to the theory of symmetric functions ([MAC] and [STA], Chapter 7, are two excellent references.) The reader is also expected to be familiar with Sage.

The algebra of symmetric functions

The algebra of symmetric functions is the unique free commutative graded connected algebra over the given ring, with one generator in each degree. It can also be thought of as the inverse limit (in the category of graded algebras) of the algebra of symmetric polynomials in n variables as $n \to \infty$. Sage allows us to construct the algebra of symmetric functions over any ring. We will use a base ring of rational numbers in these first examples:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Sym
Symmetric Functions over Rational Field
```

Sage knows certain categorical information about this algebra:

```
sage: Sym.category()
Join of Category of hopf algebras over Rational Field
    and Category of graded algebras over Rational Field
    and Category of monoids with realizations
    and Category of coalgebras over Rational Field with realizations
```

Notice that `Sym` is an *abstract* algebra. This reflects the fact that there are multiple natural bases. To work with specific elements, we need a *realization* of this algebra. In practice, this means we need to specify a basis.

An example basis - power sums

Here is an example of how one might use the power sum realization:

```
sage: p = Sym.powersum()
sage: p
Symmetric Functions over Rational Field in the powersum basis
```

p now represents the realization of the symmetric function algebra on the power sum basis. The basis itself is accessible through:

```
sage: p.basis()
Lazy family (Term map from Partitions to Symmetric Functions over Rational Field in the powersum
sage: p.basis().keys()
Partitions
```

This last line means that `p.basis()` is an association between the set of Partitions and the basis elements of the algebra p. To construct a specific element one can therefore do:

```
sage: p.basis()[Partition([2,1,1])]
p[2, 1, 1]
```

As this is rather cumbersome, realizations of the symmetric function algebra allow for the following abuses of notation:

```
sage: p[Partition([2, 1, 1])]
p[2, 1, 1]
sage: p[[2, 1, 1]]
p[2, 1, 1]
sage: p[2, 1, 1]
p[2, 1, 1]
```

or even:

```
sage: p[(i for i in [2, 1, 1])]
p[2, 1, 1]
```

In the special case of the empty partition, due to a limitation in Python syntax, one cannot use:

```
sage: p[]       # todo: not implemented
```

Please use instead:

```
sage: p[[]]
p[]
```

Note: When elements are constructed using the `p[something ]` syntax , an error will be raised if the input cannot be interpreted as a partition. This is *not* the case when `p.basis()` is used:

```
sage: p['something']
Traceback (most recent call last):
...
ValueError: ['s', 'o', 'm', 'e', 't', 'h', 'i', 'n', 'g'] is not an element of Partitions
sage: p.basis()['something']
p'something'
```

Elements of p are linear combinations of such compositions:

```
sage: p.an_element()
2*p[] + 2*p[1] + 3*p[2]
```

Algebra structure

Algebraic combinations of basis elements can be entered in a natural way:

```
sage: p[2,1,1] + 2 * p[1] * (p[4] + p[2,1])
3*p[2, 1, 1] + 2*p[4, 1]
```

Let us explore the other operations of p. We can ask for the mathematical properties of p:

```
sage: p.categories()
[Category of graded bases of Symmetric Functions over Rational Field,
 Category of filtered bases of Symmetric Functions over Rational Field,
 Category of bases of Symmetric Functions over Rational Field,
 Category of graded hopf algebras with basis over Rational Field,
 ...]
```

To start with, p is a graded algebra, the grading being induced by the size of the partitions. Due to this, the one is the basis element indexed by the empty partition:

```
sage: p.one()
p[]
```

The p basis is multiplicative; that is, multiplication is induced by linearity from the (nonincreasingly sorted) concatenation of partitions:

```
sage: p[3,1] * p[2,1]
p[3, 2, 1, 1]

sage: (p.one() + 2 * p[3,1]) * p[4, 2]
p[4, 2] + 2*p[4, 3, 2, 1]
```

The classical bases

In addition to the power sum basis, the other classical bases of the symmetric function algebra are the elementary, complete homogeneous, monomial, and Schur bases. These can be defined as follows:

```
sage: e = Sym.elementary()
sage: h = Sym.homogeneous()
sage: m = Sym.monomial()
sage: s = Sym.schur()
```

These can be defined all at once with the single command

```
sage: Sym.inject_shorthands()
doctest:...: RuntimeWarning: redefining global value `h`
doctest:...: RuntimeWarning: redefining global value `s`
doctest:...: RuntimeWarning: redefining global value `e`
doctest:...: RuntimeWarning: redefining global value `m`
doctest:...: RuntimeWarning: redefining global value `p`
```

We can then do conversions from one basis to another:

```
sage: s(p[2,1])
-s[1, 1, 1] + s[3]

sage: m(p[3])
m[3]
sage: m(p[3,2])
m[3, 2] + m[5]
```

For computations which mix bases, Sage will return a result with respect to a single (not necessarily predictable) basis:

```
sage: p[2] * s[2] - m[4]
1/2*p[2, 1, 1] + 1/2*p[2, 2] - p[4]

sage: p( m[1] * ( e[3]*s[2] + 1 ))
p[1] + 1/12*p[1, 1, 1, 1, 1, 1] - 1/6*p[2, 1, 1, 1, 1] - 1/4*p[2, 2, 1, 1] + 1/6*p[3, 1, 1, 1] +
```

The one for different bases such as the power sum and Schur function is the same:

```
sage: s.one() == p.one()
True
```

Basic computations

In this section, we explore some of the many methods that can be applied to an arbitrary symmetric function:

```
sage: f = s[2]^2; f
s[2, 2] + s[3, 1] + s[4]
```

For more methods than discussed here, create a symmetric function as above, and use `f.<tab>`.

Representation theory of the symmetric group

The Schur functions s_λ can also be interpreted as irreducible characters of the symmetric group S_n, where n is the size of the partition λ. Since the Schur functions of degree n form a basis of the symmetric functions of degree n, it follows that an arbitrary symmetric function (homogeneous of degree n) may be interpreted as a function on the symmetric group. In this interpretation the power sum symmetric function p_λ is the characteristic function of the conjugacy class with shape λ, multiplied by the order of the centralizer of an element. Hence the irreducible characters can be computed as follows:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: p = Sym.power()
sage: P = Partitions(5).list()
sage: P = [P[i] for i in range(len(P)-1,-1,-1)]
sage: M = matrix([[s[P[i]].scalar(p[P[j]]) for j in range(len(P))] for i in range(len(P))])
sage: M
[ 1 -1  1  1 -1 -1  1]
[ 4 -2  0  1  1  0 -1]
[ 5 -1  1 -1 -1  1  0]
[ 6  0 -2  0  0  0  1]
[ 5  1  1 -1  1 -1  0]
[ 4  2  0  1 -1  0 -1]
[ 1  1  1  1  1  1  1]
```

We can indeed check that this agrees with the character table of S_5:

```
sage: SymmetricGroup(5).character_table() == M
True
```

In this interpretation of symmetric functions as characters on the symmetric group, the multiplication and co-multiplication are interpreted as induction (from $S_n \times S_m$ to S_{n+m}) and restriction, respectively. The Schur functions can also be interpreted as characters of GL_n, see Partitions and Schur functions.

The omega involution

The ω involution is the linear extension of the map which sends e_λ to h_λ:

```
sage: h(f)
h[2, 2]
sage: e(f.omega())
e[2, 2]
```

The Hall scalar product

The Hall scalar product on the algebra of symmetric functions makes the Schur functions into an orthonormal basis:

```
sage: f.scalar(f)
3
```

Skewing

Skewing is the adjoint operation to multiplication with respect to this scalar product:

```
sage: f.skew_by(s[1])
2*s[2, 1] + 2*s[3]
```

In general, `s[la].skew_by(s[mu])` is the symmetric function typically denoted $s_{\lambda\backslash\mu}$ or $s_{\lambda/\mu}$.

Expanding into variables

We can expand a symmetric function into a symmetric polynomial in a specified number of variables:

```
sage: f.expand(2)
x0^4 + 2*x0^3*x1 + 3*x0^2*x1^2 + 2*x0*x1^3 + x1^4
```

See the documentation for `expand` for more examples.

The Kronecker product

As in the section on the Representation theory of the symmetric group, a symmetric function may be considered as a class function on the symmetric group where the elements p_μ/z_μ are the indicators of a permutation having cycle structure μ. The Kronecker product of two symmetric functions corresponds to the pointwise product of these class functions.

Since the Schur functions are the irreducible characters of the symmetric group under this identification, the Kronecker product of two Schur functions corresponds to the internal tensor product of two irreducible symmetric group representations.

Under this identification, the Kronecker product of p_μ/z_μ and p_ν/z_ν is p_μ/z_μ if $\mu = \nu$, and the result is equal to 0 otherwise.

`internal_product`, `kronecker_product`, `inner_tensor` and `itensor` are different names for the same function.

```
sage: f.kronecker_product(f)
s[1, 1, 1, 1] + 3*s[2, 1, 1] + 4*s[2, 2] + 5*s[3, 1] + 3*s[4]
```

Plethysm

The *plethysm* of symmetric functions is the operation corresponding to composition of representations of the general linear group. See [STA] Chapter 7, Appendix 2 for details.

```
sage: s[2].plethysm(s[2])
s[2, 2] + s[4]
```

Plethysm can also be written as a composition of functions:

```
sage: s[2]( s[2] )
s[2, 2] + s[4]
```

If the coefficient ring contains degree 1 elements, these are handled properly by plethysm:

```
sage: R.<t> = QQ[]; s = SymmetricFunctions(R).schur()
sage: s[2]( (1-t)*s[1] )
(t^2-t)*s[1, 1] + (-t+1)*s[2]
```

See the documentation for `plethysm` for more information.

Inner plethysm

The operation of inner plethysm `f.inner_plethysm(g)` models the composition of the S_n representation represented by g with the GL_m representation whose character is f. See the documentation of `inner_plethysm`, [ST94] or [STA], exercise 7.74 solutions for more information:

```
sage: s = SymmetricFunctions(QQ).schur()
sage: f = s[2]^2
sage: f.inner_plethysm(s[2])
s[2]
```

Hopf algebra structure

The ring of symmetric functions is further endowed with a coalgebra structure. The coproduct is an algebra morphism, and therefore determined by its values on the generators; the power sum generators are primitive:

```
sage: p[1].coproduct()
p[] # p[1] + p[1] # p[]
sage: p[2].coproduct()
p[] # p[2] + p[2] # p[]
```

The coproduct, being cocommutative on the generators, is cocommutative everywhere:

```
sage: p[2, 1].coproduct()
p[] # p[2, 1] + p[1] # p[2] + p[2] # p[1] + p[2, 1] # p[]
```

This coproduct, along with the counit which sends every symmetric function to its 0-th homogeneous component, makes the ring of symmetric functions into a graded connected bialgebra. It is known that every graded connected bialgebra has an antipode. For the ring of symmetric functions, the antipode can be characterized explicitly: The antipode is an anti-algebra morphism (thus an algebra morphism, since our algebra is commutative) which sends p_λ to $(-1)^{\mathrm{length}(\lambda)} p_\lambda$ for every partition λ. Thus, in particular, it sends the generators on the p basis to their opposites:

```
sage: p[3].antipode()
-p[3]
sage: p[3,2,1].antipode()
-p[3, 2, 1]
```

The graded connected bialgebra of symmetric functions over a $\mathbf{Q}$-algebra has a rather simply-understood structure: It is (isomorphic to) the symmetric algebra of its space of primitives (which is spanned by the power-sum symmetric functions).

Here are further examples:

```
sage: f = s[2]^2
sage: f.antipode()
s[1, 1, 1, 1] + s[2, 1, 1] + s[2, 2]
sage: f.coproduct()
s[] # s[2, 2] + s[] # s[3, 1] + s[] # s[4] + 2*s[1] # s[2, 1] + 2*s[1] # s[3] + s[1, 1] # s[1, 1
+ s[1, 1] # s[2] + s[2] # s[1, 1] + 3*s[2] # s[2] + 2*s[2, 1] # s[1] + s[2, 2] # s[] + 2*s[3] #
+ s[3, 1] # s[] + s[4] # s[]
sage: f.coproduct().apply_multilinear_morphism( lambda x,y: x*y.antipode() )
0
```

Transformations of symmetric functions

There are many methods in Sage which make it easy to manipulate symmetric functions. For example, if we have some function which acts on partitions (say, conjugation), it is a simple matter to apply it to the support of a symmetric function. Here is an example:

```
sage: conj = lambda mu: mu.conjugate()
sage: f = h[4] + 2*h[3,1]
sage: f.map_support(conj)
h[1, 1, 1, 1] + 2*h[2, 1, 1]
```

We can also easily modify the coefficients:

```
sage: def foo(mu, coeff): return mu.conjugate(), -coeff
sage: f.map_item(foo)
-h[1, 1, 1, 1] - 2*h[2, 1, 1]
```

See also `map_coefficients`.

There are also methods for building functions directly:

```
sage: s.sum_of_monomials(mu for mu in Partitions(3))
s[1, 1, 1] + s[2, 1] + s[3]
sage: s.sum_of_monomials(Partitions(3))
s[1, 1, 1] + s[2, 1] + s[3]
sage: s.sum_of_terms( (mu, mu[0]) for mu in Partitions(3))
s[1, 1, 1] + 2*s[2, 1] + 3*s[3]
```

These are the preferred way to build elements within a program; the result will usually be faster than using `sum()`. It also guarantees that empty sums yields the zero of `s` (see also `s.sum`).

Note also that it is a good idea to use:

```
sage: s.one()
s[]
sage: s.zero()
0
```

instead of `s(1)` and `s(0)` within programs where speed is important, in order to prevent unnecessary coercions.

Different base rings

Depending on the base ring, the different realizations of the symmetric function algebra may not span the same space:

```
sage: SZ = SymmetricFunctions(ZZ)
sage: p = SZ.power(); s = SZ.schur()
sage: p(s[1,1,1])
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

Because of this, some functions may not behave as expected when working over the integers, even though they make mathematical sense:

```
sage: s[1,1,1].plethysm(s[1,1,1])
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

It is possible to work over different base rings simultaneously:

```
sage: s = SymmetricFunctions(QQ).schur()
sage: p = SymmetricFunctions(QQ).power()
sage: sz = SymmetricFunctions(ZZ).schur(); sz._prefix = 'sz'
sage: pz = SymmetricFunctions(ZZ).power(); pz._prefix = 'pz'
sage: p(sz[1,1,1])
1/6*p[1, 1, 1] - 1/2*p[2, 1] + 1/3*p[3]
sage: sz( 1/6*p[1, 1, 1] - 1/2*p[2, 1] + 1/3*p[3] )
sz[1, 1, 1]
```

As shown in this example, if you are working over multiple base rings simultaneously, it is a good idea to change the prefix in some cases, so that you can tell from the output which realization your result is in.

Let us change the notation back for the remainder of this tutorial:

```
sage: sz._prefix = 's'
sage: pz._prefix = 'p'
```

One can also use the Sage standard renaming idiom to get shorter outputs:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Sym.rename("Sym")
sage: Sym
Sym
sage: Sym.rename()
```

And we name it back:

```
sage: Sym.rename("Symmetric Functions over Rational Field"); Sym
Symmetric Functions over Rational Field
```

Other bases

There are two additional basis of the symmetric functions which are not considered as classical bases:

- forgotten basis
- Witt basis

The forgotten basis is the dual basis of the elementary symmetric functions basis with respect to the Hall scalar product. The Witt basis can be constructed by

$$\prod_{d=1}^{\infty}(1-w_d t^d)^{-1} = \sum_{n=0}^{\infty} h_n t^n$$

where t is a formal variable.

There are further bases of the ring of symmetric functions, in general over fields with parameters such as q and t:

- Hall-Littlewood bases
- Jack bases
- Macdonald bases
- k-Schur functions

We briefly demonstrate how to access these bases. For more information, see the documentation of the individual bases.

The *Jack polynomials* can be obtained as:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: Jack = Sym.jack()
sage: P = Jack.P(); J = Jack.J(); Q = Jack.Q()
sage: J(P[2,1])
(1/(t+2))*JackJ[2, 1]
```

The parameter t can be specialized as follows:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Jack = Sym.jack(t = 1)
sage: P = Jack.P(); J = Jack.J(); Q = Jack.Q()
sage: J(P[2,1])
1/3*JackJ[2, 1]
```

Similarly one can access the Hall-Littlewood and Macdonald polynomials, etc:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: Mcd = Sym.macdonald()
sage: P = Mcd.P(); J = Mcd.J(); Q = Mcd.Q()
sage: J(P[2,1])
(1/(-q*t^4+2*q*t^3-q*t^2+t^2-2*t+1))*McdJ[2, 1]
```

k-Schur functions

The k-Schur functions live in the k-bounded subspace of the ring of symmetric functions. It is possible to compute in the k-bounded subspace directly:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks = Sym.kschur(3,1)
sage: f = ks[2,1]*ks[2,1]; f
ks3[2, 2, 1, 1] + ks3[2, 2, 2] + ks3[3, 1, 1, 1]
```

or to lift to the ring of symmetric functions:

```
sage: f.lift()
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[3, 3] + s[4, 1, 1] + s[4, 2]
```

However, it is not always possible to convert a symmetric function to the k-bounded subspace:

```
sage: s = Sym.schur()
sage: ks(s[2,1,1])
Traceback (most recent call last):
...
ValueError: s[2, 1, 1] is not in the image
```

The k-Schur functions are more generally defined with a parameter t and they are a basis of the subspace spanned by the Hall-Littlewood `Qp` symmetric functions indexed by partitions whose first part is less than or equal to k:

```
sage: Sym = SymmetricFunctions(QQ['t'].fraction_field())
sage: SymS3 = Sym.kBoundedSubspace(3) # default t='t'
sage: ks = SymS3.kschur()
sage: Qp = Sym.hall_littlewood().Qp()
sage: ks(Qp[2,1,1,1])
ks3[2, 1, 1, 1] + (t^2+t)*ks3[2, 2, 1] + (t^3+t^2)*ks3[3, 1, 1] + t^4*ks3[3, 2]
```

The subspace spanned by the k-Schur functions with a parameter t are not known to form a natural algebra. However it is known that the product of a k-Schur function and an ℓ-Schur function is in the linear span of the $k + \ell$-Schur functions:

```
sage: ks(ks[2,1]*ks[1,1])
Traceback (most recent call last):
...
ValueError: s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2] is not in the image
sage: ks[2,1]*ks[1,1]
s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2]
sage: ks6 = Sym.kBoundedSubspace(6).kschur()
sage: ks6(ks[3,1,1]*ks[3])
ks6[3, 3, 1, 1] + ks6[4, 2, 1, 1] + (t+1)*ks6[4, 3, 1] + t*ks6[4, 4]
+ ks6[5, 1, 1, 1] + ks6[5, 2, 1] + t*ks6[5, 3] + ks6[6, 1, 1]
```

dual k-Schur functions

The dual space to the subspace spanned by the k-Schur functions is most naturally realized as a quotient of the ring of symmetric functions by an ideal. When $t = 1$ the ideal is generated by the monomial symmetric functions indexed by partitions whose first part is greater than k.:

```
sage: Sym = SymmetricFunctions(QQ)
sage: SymQ3 = Sym.kBoundedQuotient(3,t=1)
sage: km = SymQ3.kmonomial()
sage: km[2,1]*km[2,1]
4*m3[2, 2, 1, 1] + 6*m3[2, 2, 2] + 2*m3[3, 2, 1] + 2*m3[3, 3]
sage: F = SymQ3.affineSchur()
sage: F[2,1]*F[2,1]
2*F3[1, 1, 1, 1, 1, 1] + 4*F3[2, 1, 1, 1, 1] + 4*F3[2, 2, 1, 1] + 4*F3[2, 2, 2]
+ 2*F3[3, 1, 1, 1] + 4*F3[3, 2, 1] + 2*F3[3, 3]
```

When t is not equal to 1, the subspace spanned by the k-Schur functions is realized as a quotient of the ring of symmetric functions by the ideal generated by the Hall-Littlewood symmetric functions in the P basis indexed by partitions with first part greater than k.:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: SymQ3 = Sym.kBoundedQuotient(3)
sage: kHLP = SymQ3.kHallLittlewoodP()
sage: kHLP[2,1]*kHLP[2,1]
(t^2+2*t+1)*HLP3[2, 2, 1, 1] + (t^3+2*t^2+2*t+1)*HLP3[2, 2, 2]
+ (-t^4-t^3+t+1)*HLP3[3, 1, 1, 1] + (-t^2+t+2)*HLP3[3, 2, 1] + (t+1)*HLP3[3, 3]
sage: HLP = Sym.hall_littlewood().P()
sage: kHLP(HLP[3,1])
HLP3[3, 1]
sage: kHLP(HLP[4])
0
```

In this space, the basis which is dual to the k-Schur functions conjecturally expands positively in the k-bounded Hall-Littlewood functions and has positive structure coefficients.:

```
sage: dks = SymQ3.dual_k_Schur()
sage: kHLP(dks[2,2])
(t^4+t^2)*HLP3[1, 1, 1, 1] + t*HLP3[2, 1, 1] + HLP3[2, 2]
sage: dks[2,1]*dks[1,1]
(t^2+t)*dks3[1, 1, 1, 1, 1] + (t+1)*dks3[2, 1, 1, 1] + (t+1)*dks3[2, 2, 1]
+ dks3[3, 1, 1] + dks3[3, 2]
```

At $t = 1$ the k-bounded Hall-Littlewood basis is equal to the k-bounded monomial basis and the dual k-Schur elements are equal to the affine Schur basis. The k-bounded monomial basis and affine Schur functions are faster and should be used instead of the k-bounded Hall-Littlewood P basis and dual k-Schur functions when $t = 1$.:

```
sage: SymQ3 = Sym.kBoundedQuotient(3,t=1)
sage: dks = SymQ3.dual_k_Schur()
sage: F = SymQ3.affineSchur()
sage: F[3,1]==dks[3,1]
True
```

Implementing new bases

In order to implement a new symmetric function basis, Sage will need to know at a minimum how to change back and forth between at least one other basis (although they do not necessarily have to be the same basis). All of the standard functions associated with the basis will have a default implementation (although a more specific implementation may be more efficient).

To present an idea of how this is done, we will create here the example of how to implement the basis $s_\mu[X(1-t)]$.

To begin, we import the class `sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic()`. Our new basis will inherit all of the default methods from this class:

```
sage: from sage.combinat.sf.sfa import SymmetricFunctionAlgebra_generic as SFA_generic
```

Now the basis we are creating has a parameter t which is possible to specialize. In this example we will convert to and from the Schur basis. For this we implement methods `_self_to_s` and `_s_to_self`. By registering these two functions as coercions, Sage then knows automatically how it possible to change between any two bases for which there is a path of changes of bases.

```
sage: from sage.categories.morphism import SetMorphism
sage: class SFA_st(SFA_generic):
....:     def __init__(self, Sym, t):
....:         SFA_generic.__init__(self, Sym, basis_name=
....:           "Schur functions with a plethystic substitution of X -> X(1-t)",
....:           prefix='st')
....:         self._s = Sym.s()
....:         self.t = Sym.base_ring()(t)
....:         cat = HopfAlgebras(Sym.base_ring()).WithBasis()
....:         self.register_coercion(
....:           SetMorphism(Hom(self._s, self, cat), self._s_to_self))
....:         self._s.register_coercion(
....:           SetMorphism(Hom(self, self._s, cat), self._self_to_s))
....:     def _s_to_self(self, f):
....:         # f is a Schur function and the output is in the st basis
....:         return self._from_dict(f.theta_qt(0,self.t)._monomial_coefficients)
....:     def _self_to_s(self, f):
....:         # f is in the st basis and the output is in the Schur basis
....:         return self._s.sum(cmu*self._s(mu).theta_qt(self.t,0) for mu,cmu in f)
....:     class Element(SFA_generic.Element):
....:         pass
```

An instance of this basis is created by calling it with a symmetric function ring `Sym` and a parameter `t` which is in the base ring of `Sym`. The `Element` class inherits all of the methods from `sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic_Element`.

In the reference [MAC] on page 354, this basis is denoted $S_\lambda(x;t)$ and the change of basis coefficients of the Macdonald `J` basis are the coefficients $K_{\lambda\mu}(q,t)$. Here is an example of its use:

```
sage: QQqt = QQ['q','t'].fraction_field()
sage: (q,t) = QQqt.gens()
sage: st = SFA_st(SymmetricFunctions(QQqt),t)
sage: st
Symmetric Functions over Fraction Field of Multivariate Polynomial
 Ring in q, t over Rational Field in the Schur functions with a
 plethystic substitution of X -> X(1-t) basis
sage: st[2,1] * st[1]
st[2, 1, 1] + st[2, 2] + st[3, 1]
sage: st([2]).coproduct()
st[] # st[2] + st[1] # st[1] + st[2] # st[]
sage: J = st.symmetric_function_ring().macdonald().J()
sage: st(J[2,1])
q*st[1, 1, 1] + (q*t+1)*st[2, 1] + t*st[3]
```

Acknowledgements

The design is heavily inspired from the implementation of symmetric functions in MuPAD-Combinat (see [HT04] and [FD06]).

REFERENCES:

Further tests

TESTS:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Sym
Symmetric Functions over Rational Field
sage: h = Sym.h(); e = Sym.e(); s = Sym.s(); m = Sym.m(); p = Sym.p()
sage: ( ( h[2,1] * ( 1 + 3 * h[2,1]) ) + s[2]. antipode()) . coproduct()
h[] # h[1, 1] - h[] # h[2] + h[] # h[2, 1] + 3*h[] # h[2, 2, 1, 1] + h[1] # h[1] + h[1] # h[1, 1
+ h[1] # h[2] + 6*h[1] # h[2, 1, 1, 1] + 6*h[1] # h[2, 2, 1] + h[1, 1] # h[] + h[1, 1] # h[1]
+ 3*h[1, 1] # h[1, 1, 1, 1] + 12*h[1, 1] # h[2, 1, 1] + 3*h[1, 1] # h[2, 2] + 6*h[1, 1, 1] # h[1
+ 6*h[1, 1, 1] # h[2, 1] + 3*h[1, 1, 1, 1] # h[1, 1] - h[2] # h[] + h[2] # h[1] + 6*h[2] # h[2,
+ h[2, 1] # h[] + 6*h[2, 1] # h[1, 1, 1] + 12*h[2, 1] # h[2, 1] + 12*h[2, 1, 1] # h[1, 1]
+ 6*h[2, 1, 1] # h[2] + 6*h[2, 1, 1, 1] # h[1] + 3*h[2, 2] # h[1, 1] + 6*h[2, 2, 1] # h[1] + 3*h
```

Todo

- Introduce fields with degree 1 elements as in MuPAD-Combinat, to get proper plethysm.
- Use UniqueRepresentation to get rid of all the manual cache handling for the bases
- Devise a mechanism so that pickling bases of symmetric functions pickles the coercions which have a cache.

Schur()

The Schur basis of the Symmetric Functions

EXAMPLES:

```
sage: SymmetricFunctions(QQ).schur()
Symmetric Functions over Rational Field in the Schur basis
```

Witt(*coerce_h=True*, *coerce_e=False*, *coerce_p=False*)

The Witt basis of the symmetric functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).witt()
Symmetric Functions over Rational Field in the Witt basis
sage: SymmetricFunctions(QQ).witt(coerce_p=True)
Symmetric Functions over Rational Field in the Witt basis
sage: SymmetricFunctions(QQ).witt(coerce_h=False, coerce_e=True, coerce_p=True)
Symmetric Functions over Rational Field in the Witt basis
```

a_realization()

Return a particular realization of `self` (the Schur basis).

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: Sym.a_realization()
Symmetric Functions over Rational Field in the Schur basis
```

complete()

The complete basis of the Symmetric Functions

EXAMPLES:

```
sage: SymmetricFunctions(QQ).complete()
Symmetric Functions over Rational Field in the homogeneous basis
```

e()

The elementary basis of the Symmetric Functions

EXAMPLES:

```
sage: SymmetricFunctions(QQ).elementary()
Symmetric Functions over Rational Field in the elementary basis
```

elementary()

The elementary basis of the Symmetric Functions

EXAMPLES:

```
sage: SymmetricFunctions(QQ).elementary()
Symmetric Functions over Rational Field in the elementary basis
```

f()

The forgotten basis of the Symmetric Functions (or the basis dual to the elementary basis with respect to the Hall scalar product).

EXAMPLES:

```
sage: SymmetricFunctions(QQ).forgotten()
Symmetric Functions over Rational Field in the forgotten basis
```

TESTS:

Over the rationals:

```
sage: Sym = SymmetricFunctions(QQ)
sage: e = Sym.e()
sage: f = Sym.f()
sage: h = Sym.h()
sage: p = Sym.p()
sage: s = Sym.s()
sage: m = Sym.m()
sage: e(f([2,1]))
-2*e[1, 1, 1] + 5*e[2, 1] - 3*e[3]
sage: f(e([2,1]))
3*f[1, 1, 1] + 2*f[2, 1] + f[3]
sage: h(f([2,1]))
h[2, 1] - 3*h[3]
sage: f(h([2,1]))
3*f[1, 1, 1] + f[2, 1]
sage: p(f([2,1]))
-p[2, 1] - p[3]
sage: f(p([2,1]))
-f[2, 1] - f[3]
sage: s(f([2,1]))
s[2, 1] - 2*s[3]
sage: f(s([2,1]))
2*f[1, 1, 1] + f[2, 1]
sage: m(f([2,1]))
-m[2, 1] - 2*m[3]
sage: f(m([2,1]))
-f[2, 1] - 2*f[3]
```

Over the integers:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: e = Sym.e()
sage: f = Sym.f()
sage: h = Sym.h()
sage: p = Sym.p()
sage: s = Sym.s()
```

```
sage: m = Sym.m()
sage: e(f([2,1]))
-2*e[1, 1, 1] + 5*e[2, 1] - 3*e[3]
sage: f(e([2,1]))
3*f[1, 1, 1] + 2*f[2, 1] + f[3]
sage: h(f([2,1]))
h[2, 1] - 3*h[3]
sage: f(h([2,1]))
3*f[1, 1, 1] + f[2, 1]
sage: f(p([2,1]))
-f[2, 1] - f[3]
sage: s(f([2,1]))
s[2, 1] - 2*s[3]
sage: f(s([2,1]))
2*f[1, 1, 1] + f[2, 1]
sage: m(f([2,1]))
-m[2, 1] - 2*m[3]
sage: f(m([2,1]))
-f[2, 1] - 2*f[3]
```

Conversion from the forgotten basis to the power-sum basis over the integers is not well-defined in general, even if the result happens to have integral coefficients:

```
sage: p(f([2,1]))
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

Fun exercise: prove that $p(f_\lambda)$ and $p(m_\lambda)$ have integral coefficients whenever λ is a strict partition.

forgotten()

The forgotten basis of the Symmetric Functions (or the basis dual to the elementary basis with respect to the Hall scalar product).

EXAMPLES:

```
sage: SymmetricFunctions(QQ).forgotten()
Symmetric Functions over Rational Field in the forgotten basis
```

TESTS:

Over the rationals:

```
sage: Sym = SymmetricFunctions(QQ)
sage: e = Sym.e()
sage: f = Sym.f()
sage: h = Sym.h()
sage: p = Sym.p()
sage: s = Sym.s()
sage: m = Sym.m()
sage: e(f([2,1]))
-2*e[1, 1, 1] + 5*e[2, 1] - 3*e[3]
sage: f(e([2,1]))
3*f[1, 1, 1] + 2*f[2, 1] + f[3]
sage: h(f([2,1]))
h[2, 1] - 3*h[3]
sage: f(h([2,1]))
3*f[1, 1, 1] + f[2, 1]
sage: p(f([2,1]))
-p[2, 1] - p[3]
```

```
sage: f(p([2,1]))
-f[2, 1] - f[3]
sage: s(f([2,1]))
s[2, 1] - 2*s[3]
sage: f(s([2,1]))
2*f[1, 1, 1] + f[2, 1]
sage: m(f([2,1]))
-m[2, 1] - 2*m[3]
sage: f(m([2,1]))
-f[2, 1] - 2*f[3]
```

Over the integers:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: e = Sym.e()
sage: f = Sym.f()
sage: h = Sym.h()
sage: p = Sym.p()
sage: s = Sym.s()
sage: m = Sym.m()
sage: e(f([2,1]))
-2*e[1, 1, 1] + 5*e[2, 1] - 3*e[3]
sage: f(e([2,1]))
3*f[1, 1, 1] + 2*f[2, 1] + f[3]
sage: h(f([2,1]))
h[2, 1] - 3*h[3]
sage: f(h([2,1]))
3*f[1, 1, 1] + f[2, 1]
sage: f(p([2,1]))
-f[2, 1] - f[3]
sage: s(f([2,1]))
s[2, 1] - 2*s[3]
sage: f(s([2,1]))
2*f[1, 1, 1] + f[2, 1]
sage: m(f([2,1]))
-m[2, 1] - 2*m[3]
sage: f(m([2,1]))
-f[2, 1] - 2*f[3]
```

Conversion from the forgotten basis to the power-sum basis over the integers is not well-defined in general, even if the result happens to have integral coefficients:

```
sage: p(f([2,1]))
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

Fun exercise: prove that $p(f_\lambda)$ and $p(m_\lambda)$ have integral coefficients whenever λ is a strict partition.

from_polynomial(f)

Converts a symmetric polynomial `f` to a symmetric function.

INPUT:

- `f` – a symmetric polynomial

This function converts a symmetric polynomial f in a polynomial ring in finitely many variables to a symmetric function in the monomial basis of the ring of symmetric functions over the same base ring.

EXAMPLES:

```
sage: P = PolynomialRing(QQ, 'x', 3)
sage: x = P.gens()
sage: f = x[0] + x[1] + x[2]
sage: S = SymmetricFunctions(QQ)
sage: S.from_polynomial(f)
m[1]

sage: f = x[0] + 2*x[1] + x[2]
sage: S.from_polynomial(f)
Traceback (most recent call last):
...
ValueError: x0 + 2*x1 + x2 is not a symmetric polynomial
```

h()

The complete basis of the Symmetric Functions

EXAMPLES:

```
sage: SymmetricFunctions(QQ).complete()
Symmetric Functions over Rational Field in the homogeneous basis
```

hall_littlewood(*t='t'*)

Returns the entry point for the various Hall-Littlewood bases.

INPUT:

- t – parameter

Hall-Littlewood symmetric functions including bases P, Q, Qp. The Hall-Littlewood P and Q functions at $t = -1$ are the Schur-P and Schur-Q functions when indexed by strict partitions.

The parameter t must be in the base ring of parent.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: P = Sym.hall_littlewood().P(); P
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: P[2]
HLP[2]
sage: Q = Sym.hall_littlewood().Q(); Q
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: Q[2]
HLQ[2]
sage: Qp = Sym.hall_littlewood().Qp(); Qp
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: Qp[2]
HLQp[2]
```

homogeneous()

The complete basis of the Symmetric Functions

EXAMPLES:

```
sage: SymmetricFunctions(QQ).complete()
Symmetric Functions over Rational Field in the homogeneous basis
```

ht()

The induced trivial character basis of the Symmetric Functions.

The trivial character of

$$S_{n-|\lambda|} \times S_{\lambda_1} \times S_{\lambda_2} \times \cdots \times S_{\lambda_{\ell(\lambda)}}$$

induced to the group S_n is a symmetric function in the eigenvalues of a permutation matrix. This basis is that character.

It has the property that if the element indexed by the partition λ is evaluated at the roots of a permutation of cycle structure ρ then the value is the coefficient $\langle h_{(n-|\lambda|,\lambda)}, p_\rho \rangle$.

In terms of methods that are implemented in Sage, if `n` is a sufficiently large integer, then `ht(lam).character_to_frobenius_image(n)` is equal the complete function indexed by `[n-sum(lam)]+lam`.

This basis is introduced in [OZ2015].

See also:

`character_to_frobenius_image()`, `eval_at_permutation_roots()`

EXAMPLES:

```
sage: SymmetricFunctions(QQ).induced_trivial_character()
Symmetric Functions over Rational Field in the induced trivial character basis
sage: ht = SymmetricFunctions(QQ).ht()
sage: h = SymmetricFunctions(QQ).h()
sage: h(ht([3,2]).character_to_frobenius_image(9))
h[4, 3, 2]
sage: h(ht([3,2]).character_to_frobenius_image(7))
h[3, 2, 2]
sage: h(ht([3,2]).character_to_frobenius_image(5))
h[3, 2]
sage: h(ht([3,2]).character_to_frobenius_image(4))
0
sage: p = SymmetricFunctions(QQ).p()
sage: [h([4,1]).scalar(p(rho)) for rho in Partitions(5)]
[0, 1, 0, 2, 1, 3, 5]
sage: [ht([1]).eval_at_permutation_roots(rho) for rho in Partitions(5)]
[0, 1, 0, 2, 1, 3, 5]
```

induced_trivial_character()

The induced trivial character basis of the Symmetric Functions.

The trivial character of

$$S_{n-|\lambda|} \times S_{\lambda_1} \times S_{\lambda_2} \times \cdots \times S_{\lambda_{\ell(\lambda)}}$$

induced to the group S_n is a symmetric function in the eigenvalues of a permutation matrix. This basis is that character.

It has the property that if the element indexed by the partition λ is evaluated at the roots of a permutation of cycle structure ρ then the value is the coefficient $\langle h_{(n-|\lambda|,\lambda)}, p_\rho \rangle$.

In terms of methods that are implemented in Sage, if `n` is a sufficiently large integer, then `ht(lam).character_to_frobenius_image(n)` is equal the complete function indexed by `[n-sum(lam)]+lam`.

This basis is introduced in [OZ2015].

See also:

`character_to_frobenius_image()`, `eval_at_permutation_roots()`

EXAMPLES:

```
sage: SymmetricFunctions(QQ).induced_trivial_character()
Symmetric Functions over Rational Field in the induced trivial character basis
sage: ht = SymmetricFunctions(QQ).ht()
sage: h = SymmetricFunctions(QQ).h()
sage: h(ht([3,2]).character_to_frobenius_image(9))
h[4, 3, 2]
sage: h(ht([3,2]).character_to_frobenius_image(7))
h[3, 2, 2]
sage: h(ht([3,2]).character_to_frobenius_image(5))
h[3, 2]
sage: h(ht([3,2]).character_to_frobenius_image(4))
0
sage: p = SymmetricFunctions(QQ).p()
sage: [h([4,1]).scalar(p(rho)) for rho in Partitions(5)]
[0, 1, 0, 2, 1, 3, 5]
sage: [ht([1]).eval_at_permutation_roots(rho) for rho in Partitions(5)]
[0, 1, 0, 2, 1, 3, 5]
```

inject_shorthands(*shorthands=set(['h', 's', 'e', 'm', 'p'])*)

Imports standard shorthands into the global namespace

INPUT:

- `shorthands` – a list (or iterable) of strings (default: ['e', 'h', 'm', 'p', 's'])

EXAMPLES:

```
sage: S = SymmetricFunctions(ZZ)
sage: S.inject_shorthands()
sage: s[1] + e[2] * p[1,1] + 2*h[3] + m[2,1]
s[1] - 2*s[1, 1, 1] + s[1, 1, 1, 1] + s[2, 1] + 2*s[2, 1, 1] + s[2, 2] + 2*s[3] + s[3, 1]
sage: e
Symmetric Functions over Integer Ring in the elementary basis
sage: p
Symmetric Functions over Integer Ring in the powersum basis
sage: s
Symmetric Functions over Integer Ring in the Schur basis

sage: e == S.e(), h == S.h(), m == S.m(), p == S.p(), s == S.s()
(True, True, True, True, True)
```

One can also just import a subset of the shorthands:

```
sage: S = SymmetricFunctions(QQ)
sage: S.inject_shorthands(['p', 's'])
sage: p
Symmetric Functions over Rational Field in the powersum basis
sage: s
Symmetric Functions over Rational Field in the Schur basis
```

Note that `e` is left unchanged:

```
sage: e
Symmetric Functions over Integer Ring in the elementary basis
```

irreducible_symmetric_group_character()

The irreducible S_n character basis of the Symmetric Functions.

This basis has the property that if the element indexed by the partition λ is evaluated at the roots of a permutation of cycle structure ρ then the value is the irreducible character $\chi^{(|\rho|-|\lambda|,\lambda)}(\rho)$.

In terms of methods that are implemented in Sage, if `n` is a sufficiently large integer, then `st(lam).character_to_frobenius_image(n)` is equal the Schur function indexed by `[n-sum(lam)]+lam`.

This basis is introduced in [OZ2015].

See also:

`character_to_frobenius_image()`, `eval_at_permutation_roots()`

EXAMPLES:

```
sage: SymmetricFunctions(QQ).irreducible_symmetric_group_character()
Symmetric Functions over Rational Field in the irreducible symmetric group character basis
sage: st = SymmetricFunctions(QQ).st()
sage: s = SymmetricFunctions(QQ).s()
sage: s(st([3,2]).character_to_frobenius_image(9))
s[4, 3, 2]
sage: s(st([3,2]).character_to_frobenius_image(7))
0
sage: s(st([3,2]).character_to_frobenius_image(6))
-s[2, 2, 2]
sage: list(SymmetricGroup(5).character_table()[-2])
[4, 2, 0, 1, -1, 0, -1]
sage: list(reversed([st([1]).eval_at_permutation_roots(rho) \
....:   for rho in Partitions(5)]))
[4, 2, 0, 1, -1, 0, -1]
```

jack(*t='t'*)

Returns the entry point for the various Jack bases.

INPUT:

- `t` – parameter

Jack symmetric functions including bases P, Q, Qp.

The parameter t must be in the base ring of parent.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: JP = Sym.jack().P(); JP
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: JQ = Sym.jack().Q(); JQ
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: JJ = Sym.jack().J(); JJ
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: JQp = Sym.jack().Qp(); JQp
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
```

kBoundedQuotient(*k*, *t='t'*)

Returns the k-bounded quotient space of the ring of symmetric functions.

INPUT:

- `k` - a positive integer

The quotient of the ring of symmetric functions ...

See also:

`sage.combinat.sf.k_dual.KBoundedQuotient()`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: KQ = Sym.kBoundedQuotient(3); KQ
Traceback (most recent call last):
...
TypeError: unable to convert t to a rational
sage: KQ = Sym.kBoundedQuotient(3,t=1); KQ
3-Bounded Quotient of Symmetric Functions over Rational Field with t=1
sage: Sym = SymmetricFunctions(QQ['t'].fraction_field())
sage: KQ = Sym.kBoundedQuotient(3); KQ
3-Bounded Quotient of Symmetric Functions over Fraction Field of Univariate Polynomial Ring
```

kBoundedSubspace (k, t='t')

Return the k-bounded subspace of the ring of symmetric functions.

INPUT:

- k - a positive integer
- t a formal parameter; $t = 1$ yields a subring

The subspace of the ring of symmetric functions spanned by $\{s_\lambda[X/(1-t)]\}_{\lambda_1 \leq k} = \{s^{(k)}_\lambda[X,t]\}_{\lambda_1 \leq k}$ over the base ring $\mathbb{Q}[t]$. When $t = 1$, this space is in fact a subalgebra of the ring of symmetric functions generated by the complete homogeneous symmetric functions h_i for $1 \leq i \leq k$.

See also:

sage.combinat.sf.new_kschur.KBoundedSubspace()

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: KB = Sym.kBoundedSubspace(3,1); KB
3-bounded Symmetric Functions over Rational Field with t=1

sage: Sym = SymmetricFunctions(QQ['t'])
sage: Sym.kBoundedSubspace(3)
3-bounded Symmetric Functions over Univariate Polynomial Ring in t over Rational Field

sage: Sym = SymmetricFunctions(QQ['z'])
sage: z = Sym.base_ring().gens()[0]
sage: Sym.kBoundedSubspace(3,t=z)
3-bounded Symmetric Functions over Univariate Polynomial Ring in z over Rational Field with
```

khomogeneous (k)

Returns the homogeneous symmetric functions in the k-bounded subspace.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: kh = Sym.khomogeneous(4)
sage: kh[3]*kh[4]
h4[4, 3]
sage: kh[4].lift()
h[4]
```

kschur (k, t='t')

Returns the k-Schur functions.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: ks = Sym.kschur(3,1)
```

```
sage: ks[2]*ks[2]
ks3[2, 2] + ks3[3, 1]
sage: ks[2,1,1].lift()
s[2, 1, 1] + s[3, 1]

sage: Sym = SymmetricFunctions(QQ['t'])
sage: ks = Sym.kschur(3)
sage: ks[2,2,1].lift()
s[2, 2, 1] + t*s[3, 2]
```

llt(*k*, *t='t'*)

The LLT symmetric functions.

INPUT:

- k – a positive integer indicating the level
- t – a parameter (default: t)

LLT polynomials in $hspin$ and $hcospin$ bases.

EXAMPLES:

```
sage: llt3 = SymmetricFunctions(QQ['t'].fraction_field()).llt(3); llt3
level 3 LLT polynomials over Fraction Field of Univariate Polynomial Ring in t over Rational
sage: llt3.hspin()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: llt3.hcospin()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
sage: llt3.hcospin()
Symmetric Functions over Fraction Field of Univariate Polynomial Ring in t over Rational Fie
```

m()

The monomial basis of the Symmetric Functions

EXAMPLES:

```
sage: SymmetricFunctions(QQ).monomial()
Symmetric Functions over Rational Field in the monomial basis
```

macdonald(*q='q'*, *t='t'*)

Returns the entry point for the various Macdonald bases.

INPUT:

- q, t – parameters

Macdonald symmetric functions including bases P, Q, J, H, Ht. This also contains the S basis which is dual to the Schur basis with respect to the q, t scalar product.

The parameters q and t must be in the base_ring of parent.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P(); P
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q, t over Rationa
sage: P[2]
McdP[2]
sage: Q = Sym.macdonald().Q(); Q
Symmetric Functions over Fraction Field of Multivariate Polynomial Ring in q, t over Rationa
sage: S = Sym.macdonald().S()
sage: s = Sym.schur()
```

```
sage: matrix([[S(la).scalar_qt(s(mu)) for la in Partitions(3)] for mu in Partitions(3)])
[1 0 0]
[0 1 0]
[0 0 1]
sage: H = Sym.macdonald().H()
sage: s(H[2,2])
q^2*s[1, 1, 1, 1] + (q^2*t+q*t+q)*s[2, 1, 1] + (q^2*t^2+1)*s[2, 2] + (q*t^2+q*t+t)*s[3, 1] +

sage: Sym = SymmetricFunctions(QQ['z','q'].fraction_field())
sage: (z,q) = Sym.base_ring().gens()
sage: Hzq = Sym.macdonald(q=z,t=q).H()
sage: H1z = Sym.macdonald(q=1,t=z).H()
sage: s = Sym.schur()
sage: s(H1z([2,2]))
s[1, 1, 1, 1] + (2*z+1)*s[2, 1, 1] + (z^2+1)*s[2, 2] + (z^2+2*z)*s[3, 1] + z^2*s[4]
sage: s(Hzq[2,2])
z^2*s[1, 1, 1, 1] + (z^2*q+z*q+z)*s[2, 1, 1] + (z^2*q^2+1)*s[2, 2] + (z*q^2+z*q+q)*s[3, 1] +
sage: s(H1z(Hzq[2,2]))
z^2*s[1, 1, 1, 1] + (z^2*q+z*q+z)*s[2, 1, 1] + (z^2*q^2+1)*s[2, 2] + (z*q^2+z*q+q)*s[3, 1] +
```

monomial()

The monomial basis of the Symmetric Functions

EXAMPLES:

```
sage: SymmetricFunctions(QQ).monomial()
Symmetric Functions over Rational Field in the monomial basis
```

o()

The orthogonal basis of the symmetric functions.

See also:

SymmetricFunctionAlgebra_orthogonal

EXAMPLES:

```
sage: SymmetricFunctions(QQ).orthogonal()
Symmetric Functions over Rational Field in the orthogonal basis
```

orthogonal()

The orthogonal basis of the symmetric functions.

See also:

SymmetricFunctionAlgebra_orthogonal

EXAMPLES:

```
sage: SymmetricFunctions(QQ).orthogonal()
Symmetric Functions over Rational Field in the orthogonal basis
```

p()

The power sum basis of the Symmetric Functions

EXAMPLES:

```
sage: SymmetricFunctions(QQ).powersum()
Symmetric Functions over Rational Field in the powersum basis
```

power()

The power sum basis of the Symmetric Functions

EXAMPLES:

```
sage: SymmetricFunctions(QQ).powersum()
Symmetric Functions over Rational Field in the powersum basis
```

powersum()

The power sum basis of the Symmetric Functions

EXAMPLES:

```
sage: SymmetricFunctions(QQ).powersum()
Symmetric Functions over Rational Field in the powersum basis
```

register_isomorphism(*morphism*, *only_conversion=False*)

Register an isomorphism between two bases of `self`, as a canonical coercion (unless the optional keyword `only_conversion` is set to `True`, in which case the isomorphism is registered as conversion only).

EXAMPLES:

We override the canonical coercion from the Schur basis to the powersum basis by a (stupid!) map $s_\lambda \mapsto 2p_\lambda$.

```
sage: Sym = SymmetricFunctions(QQ['zorglub']) # make sure we are not going to screw up later
sage: s = Sym.s(); p = Sym.p().dual_basis()
sage: phi = s.module_morphism(diagonal = lambda t: 2, codomain = p)
sage: phi(s[2, 1])
2*d_p[2, 1]
sage: Sym.register_isomorphism(phi)
sage: p(s[2,1])
2*d_p[2, 1]
```

The map is supposed to implement the canonical isomorphism between the two bases. Otherwise, the results will be mathematically wrong, as above. Use with care!

s()

The Schur basis of the Symmetric Functions

EXAMPLES:

```
sage: SymmetricFunctions(QQ).schur()
Symmetric Functions over Rational Field in the Schur basis
```

schur()

The Schur basis of the Symmetric Functions

EXAMPLES:

```
sage: SymmetricFunctions(QQ).schur()
Symmetric Functions over Rational Field in the Schur basis
```

sp()

The symplectic basis of the symmetric functions.

See also:

`SymmetricFunctionAlgebra_symplectic`

EXAMPLES:

```
sage: SymmetricFunctions(QQ).symplectic()
Symmetric Functions over Rational Field in the symplectic basis
```

st()

The irreducible S_n character basis of the Symmetric Functions.

This basis has the property that if the element indexed by the partition λ is evaluated at the roots of a permutation of cycle structure ρ then the value is the irreducible character $\chi^{(|\rho|-|\lambda|,\lambda)}(\rho)$.

In terms of methods that are implemented in Sage, if `n` is a sufficiently large integer, then `st(lam).character_to_frobenius_image(n)` is equal the Schur function indexed by `[n-sum(lam)]+lam`.

This basis is introduced in [OZ2015].

See also:

`character_to_frobenius_image()`, `eval_at_permutation_roots()`

EXAMPLES:

```
sage: SymmetricFunctions(QQ).irreducible_symmetric_group_character()
Symmetric Functions over Rational Field in the irreducible symmetric group character basis
sage: st = SymmetricFunctions(QQ).st()
sage: s = SymmetricFunctions(QQ).s()
sage: s(st([3,2]).character_to_frobenius_image(9))
s[4, 3, 2]
sage: s(st([3,2]).character_to_frobenius_image(7))
0
sage: s(st([3,2]).character_to_frobenius_image(6))
-s[2, 2, 2]
sage: list(SymmetricGroup(5).character_table()[-2])
[4, 2, 0, 1, -1, 0, -1]
sage: list(reversed([st([1]).eval_at_permutation_roots(rho) \
....:   for rho in Partitions(5)]))
[4, 2, 0, 1, -1, 0, -1]
```

symplectic()

The symplectic basis of the symmetric functions.

See also:

`SymmetricFunctionAlgebra_symplectic`

EXAMPLES:

```
sage: SymmetricFunctions(QQ).symplectic()
Symmetric Functions over Rational Field in the symplectic basis
```

w(*coerce_h=True*, *coerce_e=False*, *coerce_p=False*)

The Witt basis of the symmetric functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).witt()
Symmetric Functions over Rational Field in the Witt basis
sage: SymmetricFunctions(QQ).witt(coerce_p=True)
Symmetric Functions over Rational Field in the Witt basis
sage: SymmetricFunctions(QQ).witt(coerce_h=False, coerce_e=True, coerce_p=True)
Symmetric Functions over Rational Field in the Witt basis
```

witt(*coerce_h=True*, *coerce_e=False*, *coerce_p=False*)

The Witt basis of the symmetric functions.

EXAMPLES:

```
sage: SymmetricFunctions(QQ).witt()
Symmetric Functions over Rational Field in the Witt basis
sage: SymmetricFunctions(QQ).witt(coerce_p=True)
Symmetric Functions over Rational Field in the Witt basis
sage: SymmetricFunctions(QQ).witt(coerce_h=False, coerce_e=True, coerce_p=True)
Symmetric Functions over Rational Field in the Witt basis
```

zonal()

The zonal basis of the Symmetric Functions

EXAMPLES:

```
sage: SymmetricFunctions(QQ).zonal()
Symmetric Functions over Rational Field in the zonal basis
```

class sage.combinat.sf.sf.**SymmetricaConversionOnBasis**(*t*, *domain*, *codomain*)

Initialization of `self`.

INPUT:

- **t – a function taking a monomial in CombinatorialFreeModule(QQ, Partitions()), and returning a (partition, coefficient) list.**
- `domain`, `codomain` – parents

Construct a function mapping a partition to an element of `codomain`.

This is a temporary quick hack to wrap around the existing symmetrica conversions, without changing their specs.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['x'])
sage: p = Sym.p(); s = Sym.s()
sage: def t(x) : [(p,c)] = x; return [ (p,2*c), (p.conjugate(), c) ]
sage: f = sage.combinat.sf.sf.SymmetricaConversionOnBasis(t, p, s)
sage: f(Partition([3,1]))
s[2, 1, 1] + 2*s[3, 1]
```

5.1.268 Symmetric Functions

For a comprehensive tutorial on how to use symmetric functions in Sage

See also:

`SymmetricFunctions()`

We define the algebra of symmetric functions in the Schur and elementary bases:

```
sage: s = SymmetricFunctions(QQ).schur()
sage: e = SymmetricFunctions(QQ).elementary()
```

Each is actually a graded Hopf algebra whose basis is indexed by integer partitions:

```
sage: s.category()
Category of graded bases of Symmetric Functions over Rational Field
sage: s.basis().keys()
Partitions
```

Let us compute with some elements in different bases:

```
sage: f1 = s([2,1]); f1
s[2, 1]
sage: f2 = e(f1); f2 # basis conversion
e[2, 1] - e[3]
sage: f1 == f2
True
sage: f1.expand(3, alphabet=['x','y','z'])
x^2*y + x*y^2 + x^2*z + 2*x*y*z + y^2*z + x*z^2 + y*z^2
sage: f2.expand(3, alphabet=['x','y','z'])
x^2*y + x*y^2 + x^2*z + 2*x*y*z + y^2*z + x*z^2 + y*z^2
```

```
sage: m = SymmetricFunctions(QQ).monomial()
sage: m([3,1])
m[3, 1]
sage: m(4) # This is the constant 4, not the partition 4.
4*m[]
sage: m([4]) # This is the partition 4.
m[4]
sage: 3*m([3,1])-1/2*m([4])
3*m[3, 1] - 1/2*m[4]
```

```
sage: p = SymmetricFunctions(QQ).power()
sage: f = p(3)
sage: f
3*p[]
sage: f.parent()
Symmetric Functions over Rational Field in the powersum basis
sage: f + p([3,2])
3*p[] + p[3, 2]
```

One can convert symmetric functions to symmetric polynomials and vice versa:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.powersum()
sage: h = Sym.homogeneous()
sage: f = h[2,1] + 2*p[3,1]
sage: poly = f.expand(3); poly
2*x0^4 + 2*x0^3*x1 + 2*x0*x1^3 + 2*x1^4 + 2*x0^3*x2 + 2*x1^3*x2 + 2*x0*x2^3 + 2*x1*x2^3 + 2*x2^4
+ x0^3 + 2*x0^2*x1 + 2*x0*x1^2 + x1^3 + 2*x0^2*x2 + 3*x0*x1*x2 + 2*x1^2*x2 + 2*x0*x2^2 + 2*x1*x2^2 +
sage: Sym.from_polynomial(poly)
3*m[1, 1, 1] + 2*m[2, 1] + m[3] + 2*m[3, 1] + 2*m[4]
sage: Sym.from_polynomial(poly) == f
True
sage: g = h[1,1,1,1]
sage: poly = g.expand(3)
sage: Sym.from_polynomial(poly) == g
False
```

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: h = Sym.h()
sage: p = Sym.p()
sage: e = Sym.e()
sage: m = Sym.m()
sage: a = s([3,1])
sage: s(a)
s[3, 1]
```

```
sage: h(a)
h[3, 1] - h[4]
sage: p(a)
1/8*p[1, 1, 1, 1] + 1/4*p[2, 1, 1] - 1/8*p[2, 2] - 1/4*p[4]
sage: e(a)
e[2, 1, 1] - e[2, 2] - e[3, 1] + e[4]
sage: m(a)
3*m[1, 1, 1, 1] + 2*m[2, 1, 1] + m[2, 2] + m[3, 1]
sage: a.expand(4)
x0^3*x1 + x0^2*x1^2 + x0*x1^3 + x0^3*x2 + 2*x0^2*x1*x2 + 2*x0*x1^2*x2 + x1^3*x2 + x0^2*x2^2 + 2*x0*x1
```

Here are further examples:

```
sage: h(m([1]))
h[1]
sage: h( m([2]) +m([1,1]) )
h[2]
sage: h( m([3]) + m([2,1]) + m([1,1,1]) )
h[3]
sage: h( m([4]) + m([3,1]) + m([2,2]) + m([2,1,1]) + m([1,1,1,1]) )
h[4]
sage: k = 5
sage: h( sum([ m(part) for part in Partitions(k)]) )
h[5]
sage: k = 10
sage: h( sum([ m(part) for part in Partitions(k)]) )
h[10]
```

```
sage: P3 = Partitions(3)
sage: P3.list()
[[3], [2, 1], [1, 1, 1]]
sage: m = SymmetricFunctions(QQ).monomial()
sage: f = sum([m(p) for p in P3])
sage: m.get_print_style()
'lex'
sage: f
m[1, 1, 1] + m[2, 1] + m[3]
sage: m.set_print_style('length')
sage: f
m[3] + m[2, 1] + m[1, 1, 1]
sage: m.set_print_style('maximal_part')
sage: f
m[1, 1, 1] + m[2, 1] + m[3]
sage: m.set_print_style('lex')
```

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: m = Sym.m()
sage: m([3])*s([2,1])
2*m[3, 1, 1, 1] + m[3, 2, 1] + 2*m[4, 1, 1] + m[4, 2] + m[5, 1]
sage: s(m([3])*s([2,1]))
s[2, 1, 1, 1, 1] - s[2, 2, 2] - s[3, 3] + s[5, 1]
sage: s(s([2,1])*m([3]))
s[2, 1, 1, 1, 1] - s[2, 2, 2] - s[3, 3] + s[5, 1]
sage: e = Sym.e()
sage: e([4])*e([3])*e([1])
e[4, 3, 1]
```

```
sage: s = SymmetricFunctions(QQ).s()
sage: z = s([2,1]) + s([1,1,1])
sage: z.coefficient([2,1])
1
sage: z.length()
2
sage: z.support()
[[1, 1, 1], [2, 1]]
sage: z.degree()
3
```

TESTS:

Check that we can handle large integers properly (trac ticket #13413):

```
sage: s = SymmetricFunctions(QQ).s()
sage: p = SymmetricFunctions(QQ).p()
sage: max(s(p([1]*36)).coefficients())  # long time (4s on sage.math, 2013)
40971642983700000000
```

BACKWARD INCOMPATIBLE CHANGES (trac ticket #5457):

The symmetric functions code has been refactored to take advantage of the coercion systems. This introduced a couple of glitches:

- On some bases changes, coefficients in Jack polynomials are not normalized
- Except in a few cases, conversions and coercions are only defined between symmetric functions over the same coefficient ring. E.g. the following does not work anymore:

  ```
  sage: s  = SymmetricFunctions(QQ)
  sage: s2 = SymmetricFunctions(QQ['t'])
  sage: s([1]) + s2([2]) # todo: not implemented
  ```

  This feature will probably come back at some point through improvements to the Sage coercion system.

Backward compatibility should be essentially retained.

AUTHORS:

- Mike Hansen (2007-06-15)
- Nicolas M. Thiery (partial refactoring)
- Mike Zabrocki, Anne Schilling (2012)
- Darij Grinberg (2013) Sym over rings that are not characteristic 0

class sage.combinat.sf.sfa.**FilteredSymmetricFunctionsBases**(*parent_with_realization*)
Bases: sage.categories.realizations.Category_realization_of_parent

The category of filtered bases of the ring of symmetric functions.

TESTS:

```
sage: from sage.combinat.sf.sfa import FilteredSymmetricFunctionsBases
sage: Sym = SymmetricFunctions(QQ)
sage: bases = FilteredSymmetricFunctionsBases(Sym); bases
Category of filtered bases of Symmetric Functions over Rational Field
sage: Sym.schur() in bases
True
sage: Sym.sp() in bases
True
```

super_categories()
The super categories of self.

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import FilteredSymmetricFunctionsBases
sage: Sym = SymmetricFunctions(QQ)
sage: bases = FilteredSymmetricFunctionsBases(Sym)
sage: bases.super_categories()
[Category of bases of Symmetric Functions over Rational Field,
 Join of Category of hopf algebras with basis over Rational Field
     and Category of filtered algebras with basis over Rational Field
     and Category of commutative algebras over Rational Field]
```

class sage.combinat.sf.sfa.**GradedSymmetricFunctionsBases**(*parent_with_realization*)
Bases: sage.categories.realizations.Category_realization_of_parent

The category of graded bases of the ring of symmetric functions.

These are further required to have the property that the basis element indexed by the empty partition is 1.

TESTS:

```
sage: from sage.combinat.sf.sfa import GradedSymmetricFunctionsBases
sage: Sym = SymmetricFunctions(QQ)
sage: bases = GradedSymmetricFunctionsBases(Sym); bases
Category of graded bases of Symmetric Functions over Rational Field
sage: Sym.schur() in bases
True
sage: Sym.sp() in bases
False
```

class ElementMethods

degree_negation()
Return the image of self under the degree negation automorphism of the ring of symmetric functions.

The degree negation is the automorphism which scales every homogeneous element of degree k by $(-1)^k$ (for all k).

Calling degree_negation(self) is equivalent to calling self.parent().degree_negation(self).

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: m = Sym.monomial()
sage: f = 2*m[2,1] + 4*m[1,1] - 5*m[1] - 3*m[[]]
sage: f.degree_negation()
-3*m[] + 5*m[1] + 4*m[1, 1] - 2*m[2, 1]
sage: x = m.zero().degree_negation(); x
0
sage: parent(x) is m
True
```

degree_zero_coefficient()
Returns the degree zero coefficient of self.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.monomial()
sage: f = 2*m[2,1] + 3*m[[]]
sage: f.degree_zero_coefficient()
3
```

class GradedSymmetricFunctionsBases.**ParentMethods**

antipode_by_coercion(*element*)

The antipode of `element`.

INPUT:

- `element` – element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.p()
sage: s = Sym.s()
sage: e = Sym.e()
sage: h = Sym.h()
sage: (h([]) + h([1])).antipode() # indirect doctest
h[] - h[1]
sage: (s([]) + s([1]) + s[2]).antipode()
s[] - s[1] + s[1, 1]
sage: (p([2]) + p([3])).antipode()
-p[2] - p[3]
sage: (e([2]) + e([3])).antipode()
e[1, 1] - e[1, 1, 1] - e[2] + 2*e[2, 1] - e[3]
sage: f = Sym.f()
sage: f([3,2,1]).antipode()
-f[3, 2, 1] - 4*f[3, 3] - 2*f[4, 2] - 2*f[5, 1] - 6*f[6]
```

The antipode is an involution:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: s = Sym.s()
sage: all( s[u].antipode().antipode() == s[u] for u in Partitions(4) )
True
```

The antipode is an algebra homomorphism:

```
sage: Sym = SymmetricFunctions(FiniteField(23))
sage: h = Sym.h()
sage: all( all( (s[u] * s[v]).antipode() == s[u].antipode() * s[v].antipode()
....:           for u in Partitions(3) )
....:      for v in Partitions(3) )
True
```

TESTS:

Everything works over **Z**:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: p = Sym.p()
sage: s = Sym.s()
sage: e = Sym.e()
sage: h = Sym.h()
sage: (h([]) + h([1])).antipode() # indirect doctest
h[] - h[1]
sage: (s([]) + s([1]) + s[2]).antipode()
s[] - s[1] + s[1, 1]
sage: (p([2]) + p([3])).antipode()
```

```
-p[2] - p[3]
sage: (e([2]) + e([3])).antipode()
e[1, 1] - e[1, 1, 1] - e[2] + 2*e[2, 1] - e[3]
```

counit(*element*)

Return the counit of `element`.

The counit is the constant term of `element`.

INPUT:

- `element` – element in a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.monomial()
sage: f = 2*m[2,1] + 3*m[[]]
sage: f.counit()
3
```

degree_negation(*element*)

Return the image of `element` under the degree negation automorphism of the ring of symmetric functions.

The degree negation is the automorphism which scales every homogeneous element of degree k by $(-1)^k$ (for all k).

INPUT:

- `element` – symmetric function written in `self`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: m = Sym.monomial()
sage: f = 2*m[2,1] + 4*m[1,1] - 5*m[1] - 3*m[[]]
sage: m.degree_negation(f)
-3*m[] + 5*m[1] + 4*m[1, 1] - 2*m[2, 1]
```

TESTS:

Using `degree_negation()` on an element of a different basis works correctly:

```
sage: e = Sym.elementary()
sage: m.degree_negation(e[3])
-m[1, 1, 1]
sage: m.degree_negation(m(e[3]))
-m[1, 1, 1]
```

GradedSymmetricFunctionsBases.**super_categories**()

The super categories of `self`.

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import GradedSymmetricFunctionsBases
sage: Sym = SymmetricFunctions(QQ)
sage: bases = GradedSymmetricFunctionsBases(Sym)
sage: bases.super_categories()
[Category of filtered bases of Symmetric Functions over Rational Field,
 Category of commutative graded hopf algebras with basis over Rational Field]
```

class sage.combinat.sf.sfa.**SymmetricFunctionAlgebra_generic**(*Sym*, *basis_name=None*, *prefix=None*, *graded=True*)

Bases: sage.combinat.free_module.CombinatorialFreeModule

Abstract base class for symmetric function algebras.

Todo

Most of the methods in this class are generic (manipulations of morphisms, ...) and should be generalized (or removed)

TESTS:

```
sage: s = SymmetricFunctions(QQ).s()
sage: m = SymmetricFunctions(ZZ).m()
sage: s(m([2,1]))
-2*s[1, 1, 1] + s[2, 1]
```

Element
alias of SymmetricFunctionAlgebra_generic_Element

basis_name()
Return the name of the basis of self.

This is used for output and, for the classical bases of symmetric functions, to connect this basis with Symmetrica.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: s.basis_name()
'Schur'
sage: p = Sym.p()
sage: p.basis_name()
'powersum'
sage: h = Sym.h()
sage: h.basis_name()
'homogeneous'
sage: e = Sym.e()
sage: e.basis_name()
'elementary'
sage: m = Sym.m()
sage: m.basis_name()
'monomial'
sage: f = Sym.f()
sage: f.basis_name()
'forgotten'
```

coproduct_by_coercion(*elt*)
Return the coproduct of the element elt by coercion to the Schur basis.

INPUT:

- elt – an instance of this basis

OUTPUT:

- The image of elt under the comultiplication (=coproduct) of the coalgebra of symmetric functions. The result is an element of the tensor squared of the basis self.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
sage: m[3,1,1].coproduct()
m[] # m[3, 1, 1] + m[1] # m[3, 1] + m[1, 1] # m[3] + m[3] # m[1, 1] + m[3, 1] # m[1] + m[3,
```

```
sage: m.coproduct_by_coercion(m[2,1])
m[] # m[2, 1] + m[1] # m[2] + m[2] # m[1] + m[2, 1] # m[]
sage: m.coproduct_by_coercion(m[2,1]) == m([2,1]).coproduct()
True
sage: McdH = SymmetricFunctions(QQ['q','t'].fraction_field()).macdonald().H()
sage: McdH[2,1].coproduct()
McdH[] # McdH[2, 1] + ((q^2*t-1)/(q*t-1))*McdH[1] # McdH[1, 1] + ((q*t^2-1)/(q*t-1))*McdH[1]
sage: HLQp = SymmetricFunctions(QQ['t'].fraction_field()).hall_littlewood().Qp()
sage: HLQp[2,1].coproduct()
HLQp[] # HLQp[2, 1] + HLQp[1] # HLQp[1, 1] + HLQp[1] # HLQp[2] + HLQp[1, 1] # HLQp[1] + HLQp
sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: LLT = Sym.llt(3)
sage: LLT.cospin([3,2,1]).coproduct()
(t+1)*m[] # m[1, 1] + m[] # m[2] + (t+1)*m[1] # m[1] + (t+1)*m[1, 1] # m[] + m[2] # m[]
sage: f = SymmetricFunctions(ZZ).f()
sage: f[3].coproduct()
f[] # f[3] + f[3] # f[]
sage: f[3,2,1].coproduct()
f[] # f[3, 2, 1] + f[1] # f[3, 2] + f[2] # f[3, 1] + f[2, 1] # f[3] + f[3] # f[2, 1] + f[3,
```

dual_basis(*scalar=None*, *scalar_name=''*, *basis_name=None*, *prefix=None*)

Return the dual basis of `self` with respect to the scalar product `scalar`.

INPUT:

- `scalar` – A function `zee` from partitions to the base ring which specifies the scalar product by $\langle p_\lambda, p_\lambda \rangle = \mathrm{zee}(\lambda)$. (Independently on the function chosen, the power sum basis will always be orthogonal; the function `scalar` only determines the norms of the basis elements.) If `scalar` is None, then the standard (Hall) scalar product is used.
- `scalar_name` – name of the scalar function
- `prefix` – prefix used to display the basis

EXAMPLES:

The duals of the elementary symmetric functions with respect to the Hall scalar product are the forgotten symmetric functions.

```
sage: e = SymmetricFunctions(QQ).e()
sage: f = e.dual_basis(prefix='f'); f
Dual basis to Symmetric Functions over Rational Field in the elementary basis with respect t
sage: f([2,1])^2
4*f[2, 2, 1, 1] + 6*f[2, 2, 2] + 2*f[3, 2, 1] + 2*f[3, 3] + 2*f[4, 1, 1] + f[4, 2]
sage: f([2,1]).scalar(e([2,1]))
1
sage: f([2,1]).scalar(e([1,1,1]))
0
```

Since the power-sum symmetric functions are orthogonal, their duals with respect to the Hall scalar product are scalar multiples of themselves.

```
sage: p = SymmetricFunctions(QQ).p()
sage: q = p.dual_basis(prefix='q'); q
Dual basis to Symmetric Functions over Rational Field in the powersum basis with respect to
sage: q([2,1])^2
4*q[2, 2, 1, 1]
sage: p([2,1]).scalar(q([2,1]))
1
sage: p([2,1]).scalar(q([1,1,1]))
0
```

from_polynomial(*poly*, *check=True*)

Convert polynomial to a symmetric function in the monomial basis and then to the basis `self`.

INPUT:

- `poly` – a symmetric polynomial
- `check` – (default: `True`) boolean, specifies whether the computation checks that the polynomial is indeed symmetric

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: h = Sym.homogeneous()
sage: f = (h([]) + h([2,1]) + h([3])).expand(3)
sage: h.from_polynomial(f)
h[] + h[2, 1] + h[3]
sage: s = Sym.s()
sage: g = (s([]) + s([2,1])).expand(3); g
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2 + 1
sage: s.from_polynomial(g)
s[] + s[2, 1]
```

get_print_style()

Return the value of the current print style for `self`.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s.get_print_style()
'lex'
sage: s.set_print_style('length')
sage: s.get_print_style()
'length'
sage: s.set_print_style('lex')
```

prefix()

Return the prefix on the elements of `self`.

EXAMPLES:

```
sage: schur = SymmetricFunctions(QQ).schur()
sage: schur([3,2,1])
s[3, 2, 1]
sage: schur.prefix()
's'
```

product_by_coercion(*left*, *right*)

Return the product of elements `left` and `right` by coercion to the Schur basis.

INPUT:

- `left`, `right` – instances of this basis

OUTPUT:

- the product of `left` and `right` expressed in the basis `self`

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: p.product_by_coercion(p[3,1,1], p[2,2])
```

```
p[3, 2, 2, 1, 1]
sage: m = SymmetricFunctions(QQ).m()
sage: m.product_by_coercion(m[2,1],m[1,1]) == m[2,1]*m[1,1]
True
```

set_print_style(*ps*)

Set the value of the current print style to ps.

INPUT:

- ps – a string specifying the printing style

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s.get_print_style()
'lex'
sage: s.set_print_style('length')
sage: s.get_print_style()
'length'
sage: s.set_print_style('lex')
```

symmetric_function_ring()

Return the family of symmetric functions associated to the basis self.

OUTPUT:

- returns an instance of the ring of symmetric functions

EXAMPLES:

```
sage: schur = SymmetricFunctions(QQ).schur()
sage: schur.symmetric_function_ring()
Symmetric Functions over Rational Field
sage: power = SymmetricFunctions(QQ['t']).power()
sage: power.symmetric_function_ring()
Symmetric Functions over Univariate Polynomial Ring in t over Rational Field
```

transition_matrix(*basis*, *n*)

Return the transition matrix between self and basis for the homogeneous component of degree n.

INPUT:

- basis – a basis of the ring of symmetric functions
- n – a nonnegative integer

OUTPUT:

- a matrix of coefficients giving the expansion of the homogeneous degree-n elements of self in the degree-n elements of basis

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: m = SymmetricFunctions(QQ).m()
sage: s.transition_matrix(m,5)
[1 1 1 1 1 1 1]
[0 1 1 2 2 3 4]
[0 0 1 1 2 3 5]
[0 0 0 1 1 3 6]
[0 0 0 0 1 2 5]
[0 0 0 0 0 1 4]
```

```
[0 0 0 0 0 0 1]
sage: s.transition_matrix(m,1)
[1]
sage: s.transition_matrix(m,0)
[1]

sage: p = SymmetricFunctions(QQ).p()
sage: s.transition_matrix(p, 4)
[ 1/4  1/3  1/8  1/4 1/24]
[-1/4    0 -1/8  1/4  1/8]
[   0 -1/3  1/4    0 1/12]
[ 1/4    0 -1/8 -1/4  1/8]
[-1/4  1/3  1/8 -1/4 1/24]
sage: StoP = s.transition_matrix(p,4)
sage: a = s([3,1])+5*s([1,1,1,1])-s([4])
sage: a
5*s[1, 1, 1, 1] + s[3, 1] - s[4]
sage: mon = sorted(a.support())
sage: coeffs = [a[i] for i in mon]
sage: coeffs
[5, 1, -1]
sage: mon
[[1, 1, 1, 1], [3, 1], [4]]
sage: cm = matrix([[-1,1,0,0,5]])
sage: cm * StoP
[-7/4  4/3  3/8 -5/4 7/24]
sage: p(a)
7/24*p[1, 1, 1, 1] - 5/4*p[2, 1, 1] + 3/8*p[2, 2] + 4/3*p[3, 1] - 7/4*p[4]

sage: h = SymmetricFunctions(QQ).h()
sage: e = SymmetricFunctions(QQ).e()
sage: s.transition_matrix(m,7) == h.transition_matrix(s,7).transpose()
True

sage: h.transition_matrix(m, 7) == h.transition_matrix(m, 7).transpose()
True

sage: h.transition_matrix(e, 7) == e.transition_matrix(h, 7)
True

sage: p.transition_matrix(s, 5)
[ 1 -1  0  1  0 -1  1]
[ 1  0 -1  0  1  0 -1]
[ 1 -1  1  0 -1  1 -1]
[ 1  1 -1  0 -1  1  1]
[ 1  0  1 -2  1  0  1]
[ 1  2  1  0 -1 -2 -1]
[ 1  4  5  6  5  4  1]

sage: e.transition_matrix(m,7) == e.transition_matrix(m,7).transpose()
True
```

class sage.combinat.sf.sfa.**SymmetricFunctionAlgebra_generic_Element**(M, x)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Class of generic elements for the symmetric function algebra.

TESTS:

```
sage: m = SymmetricFunctions(QQ).m()
sage: f = sum([m(p) for p in Partitions(3)])
sage: m.set_print_style('lex')
sage: f
m[1, 1, 1] + m[2, 1] + m[3]
sage: m.set_print_style('length')
sage: f
m[3] + m[2, 1] + m[1, 1, 1]
sage: m.set_print_style('maximal_part')
sage: f
m[1, 1, 1] + m[2, 1] + m[3]
sage: m.set_print_style('lex')
```

adams_operation(**args*, ***opts*)

arithmetic_product(*x*)

Return the arithmetic product of `self` and `x` in the basis of `self`.

The arithmetic product is a binary operation $\boxdot$ on the ring of symmetric functions which is bilinear in its two arguments and satisfies

$$p_\lambda \boxdot p_\mu = \prod_{i\geq 1, j\geq 1} p_{\mathrm{lcm}(\lambda_i,\mu_j)}^{\gcd(\lambda_i,\mu_j)}$$

for any two partitions $\lambda = (\lambda_1, \lambda_2, \lambda_3, \ldots)$ and $\mu = (\mu_1, \mu_2, \mu_3, \ldots)$ (where p_ν denotes the power-sum symmetric function indexed by the partition ν, and p_i denotes the i-th power-sum symmetric function). This is enough to define the arithmetic product if the base ring is torsion-free as a $\mathbf{Z}$-module; for all other cases the arithmetic product is uniquely determined by requiring it to be functorial in the base ring. See http://mathoverflow.net/questions/138148/ for a discussion of this arithmetic product.

If f and g are two symmetric functions which are homogeneous of degrees a and b, respectively, then $f \boxdot g$ is homogeneous of degree ab.

The arithmetic product is commutative and associative and has unity $e_1 = p_1 = h_1$.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

OUTPUT:

Arithmetic product of `self` with `x`; this is a symmetric function over the same base ring as `self`.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([2]).arithmetic_product(s([2]))
s[1, 1, 1, 1] + 2*s[2, 2] + s[4]
sage: s([2]).arithmetic_product(s([1,1]))
s[2, 1, 1] + s[3, 1]
```

The symmetric function `e[1]` is the unity for the arithmetic product:

```
sage: e = SymmetricFunctions(ZZ).e()
sage: all( e([1]).arithmetic_product(e(q)) == e(q) for q in Partitions(4) )
True
```

The arithmetic product is commutative:

```
sage: e = SymmetricFunctions(FiniteField(19)).e()
sage: m = SymmetricFunctions(FiniteField(19)).m()
sage: all( all( e(p).arithmetic_product(m(q)) == m(q).arithmetic_product(e(p))  # long time
```

```
....:             for q in Partitions(4) )
....:       for p in Partitions(4) )
True
```

Note: The currently existing implementation of this function is technically unsatisfactory. It distinguishes the case when the base ring is a $\mathbf{Q}$-algebra (in which case the arithmetic product can be easily computed using the power sum basis) from the case where it isn't. In the latter, it does a computation using universal coefficients, again distinguishing the case when it is able to compute the "corresponding" basis of the symmetric function algebra over $\mathbf{Q}$ (using the `corresponding_basis_over` hack) from the case when it isn't (in which case it transforms everything into the Schur basis, which is slow).

---

bernstein_creation_operator(*n*)

Return the image of `self` under the n-th Bernstein creation operator.

Let n be an integer. The n-th Bernstein creation operator $\mathbf{B}_n$ is defined as the endomorphism of the space Sym of symmetric functions which sends every f to

$$\sum_{i \geq 0} (-1)^i h_{n+i} e_i^{\perp},$$

where usual notations are in place (h stands for the complete homogeneous symmetric functions, e for the elementary ones, and $e_i^{\perp}$ means skewing (`skew_by()`) by e_i).

This has been studied in [BBSSZ2012], section 2.2, where the following rule is given for computing $\mathbf{B}_n$ on a Schur function: If $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ is an n-tuple of integers (positive or not), then

$$\mathbf{B}_n s_{(\alpha_1, \alpha_2, \ldots, \alpha_n)} = s_{(n, \alpha_1, \alpha_2, \ldots, \alpha_n)}.$$

Here, $s_{(\alpha_1, \alpha_2, \ldots, \alpha_n)}$ is the "Schur function" associated to the n-tuple $(\alpha_1, \alpha_2, \ldots, \alpha_n)$, and defined by literally applying the Jacobi-Trudi identity, i.e., by

$$s_{(\alpha_1, \alpha_2, \ldots, \alpha_n)} = \det\left((h_{\alpha_i - i + j})_{i,j=1,2,\ldots,n} \right).$$

This notion of a Schur function clearly extends the classical notion of Schur function corresponding to a partition, but is easily reduced to the latter (in fact, for any n-tuple α of integers, one easily sees that s_α is either 0 or minus-plus a Schur function corresponding to a partition; and it is easy to determine which of these is the case and find the partition by a combinatorial algorithm).

EXAMPLES:

Let us check that what this method computes agrees with the definition:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: e = Sym.e()
sage: h = Sym.h()
sage: s = Sym.s()
sage: def bernstein_creation_by_def(n, f):
....:     # `n`-th Bernstein creation operator applied to `f`
....:     # computed according to its definition.
....:     res = f.parent().zero()
....:     if not f:
....:         return res
....:     max_degree = max(sum(m) for m, c in f)
....:     for i in range(max_degree + 1):
....:         if n + i >= 0:
....:             res += (-1) ** i * h[n + i] * f.skew_by(e[i])
....:     return res
sage: all( bernstein_creation_by_def(n, s[l]) == s[l].bernstein_creation_operator(n)
```

```
....:       for n in range(-2, 3) for l in Partitions(4) )
True
sage: all( bernstein_creation_by_def(n, s[l]) == s[l].bernstein_creation_operator(n)
....:       for n in range(-3, 4) for l in Partitions(3) )
True
sage: all( bernstein_creation_by_def(n, e[l]) == e[l].bernstein_creation_operator(n)
....:       for n in range(-3, 4) for k in range(3) for l in Partitions(k) )
True
```

Some examples:

```
sage: s[3,2].bernstein_creation_operator(3)
s[3, 3, 2]
sage: s[3,2].bernstein_creation_operator(1)
-s[2, 2, 2]
sage: h[3,2].bernstein_creation_operator(-2)
h[2, 1]
sage: h[3,2].bernstein_creation_operator(-1)
h[2, 1, 1] - h[2, 2] - h[3, 1]
sage: h[3,2].bernstein_creation_operator(0)
-h[3, 1, 1] + h[3, 2]
sage: h[3,2].bernstein_creation_operator(1)
-h[2, 2, 2] + h[3, 2, 1]
sage: h[3,2].bernstein_creation_operator(2)
-h[3, 3, 1] + h[4, 2, 1]
```

character_to_frobenius_image(*n*)

Interpret `self` as a Gl_n character and then take the Frobenius image of this character of the permutation matrices S_n which naturally sit inside of Gl_n.

To know the value of this character at a permutation of cycle structure ρ the symmetric function `self` is evaluated at the eigenvalues of of a permutation of cycle structure ρ. The Frobenius image is then defined as $\sum_{\rho \vdash n} f[\Xi_\rho] p_\rho / z_\rho$.

See also:

`eval_at_permutation_roots()`

INPUT:

- `n` – a non-negative integer to interpret `self` as a character of Gl_n

OUTPUT:

- a symmetric function of degree `n`

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([1,1]).character_to_frobenius_image(5)
s[3, 1, 1] + s[4, 1]
sage: s([2,1]).character_to_frobenius_image(5)
s[2, 2, 1] + 2*s[3, 1, 1] + 2*s[3, 2] + 3*s[4, 1] + s[5]
sage: s([2,2,2]).character_to_frobenius_image(3)
s[3]
sage: s([2,2,2]).character_to_frobenius_image(4)
s[2, 2] + 2*s[3, 1] + 2*s[4]
sage: s([2,2,2]).character_to_frobenius_image(5)
2*s[2, 2, 1] + s[3, 1, 1] + 4*s[3, 2] + 3*s[4, 1] + 2*s[5]
```

degree()

Return the degree of `self` (which is defined to be 0 for the zero element).

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1]) + 3
sage: z.degree()
4
sage: s(1).degree()
0
sage: s(0).degree()
0
```

derivative_with_respect_to_p1(*n=1*)

Return the symmetric function obtained by taking the derivative of `self` with respect to the power-sum symmetric function p_1 when the expansion of `self` in the power-sum basis is considered as a polynomial in p_k's (with $k \geq 1$).

This is the same as skewing `self` by the first power-sum symmetric function p_1.

INPUT:

- n – (default: 1) nonnegative integer which determines which power of the derivative is taken

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ).p()
sage: a = p([1,1,1])
sage: a.derivative_with_respect_to_p1()
3*p[1, 1]
sage: a.derivative_with_respect_to_p1(1)
3*p[1, 1]
sage: a.derivative_with_respect_to_p1(2)
6*p[1]
sage: a.derivative_with_respect_to_p1(3)
6*p[]

sage: s = SymmetricFunctions(QQ).s()
sage: s([3]).derivative_with_respect_to_p1()
s[2]
sage: s([2,1]).derivative_with_respect_to_p1()
s[1, 1] + s[2]
sage: s([1,1,1]).derivative_with_respect_to_p1()
s[1, 1]
sage: s(0).derivative_with_respect_to_p1()
0
sage: s(1).derivative_with_respect_to_p1()
0
sage: s([1]).derivative_with_respect_to_p1()
s[]
```

Let us check that taking the derivative with respect to `p[1]` is equivalent to skewing by `p[1]`:

```
sage: p1 = s([1])
sage: all( s(lam).derivative_with_respect_to_p1()
....:      == s(lam).skew_by(p1) for lam in Partitions(4) )
True
```

eval_at_permutation_roots(*rho*)

Evaluate at eigenvalues of a permutation matrix.

Evaluate a symmetric function at the eigenvalues of a permutation matrix whose cycle structure is `rho`. This computation is computed by coercing to the power sum basis where the value may be computed on the generators.

This function evaluates an element at the roots of unity

$$\Xi_{\rho_1}, \Xi_{\rho_2}, \ldots, \Xi_{\rho_\ell}$$

where

$$\Xi_m = 1, \zeta_m, \zeta_m^2, \ldots, \zeta_m^{m-1}$$

and ζ_m is an m root of unity. These roots of unity represent the eigenvalues of permutation matrix with cycle structure ρ.

INPUT:

- `rho` – a partition or a list of non-negative integers

OUTPUT:

- an element of the base ring

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([3,3]).eval_at_permutation_roots([6])
0
sage: s([3,3]).eval_at_permutation_roots([3])
1
sage: s([3,3]).eval_at_permutation_roots([1])
0
sage: s([3,3]).eval_at_permutation_roots([3,3])
4
sage: s([3,3]).eval_at_permutation_roots([1,1,1,1,1])
175
sage: (s[1]+s[2]+s[3]).eval_at_permutation_roots([3,2])
2
```

expand(*n*, *alphabet='x'*)

Expand the symmetric function `self` as a symmetric polynomial in `n` variables.

INPUT:

- `n` – a nonnegative integer
- `alphabet` – (default: `'x'`) a variable for the expansion

OUTPUT:

A monomial expansion of `self` in the n variables labelled `x0`, `x1`, ..., `x{n-1}` (or just `x` if $n = 1$), where `x` is `alphabet`.

EXAMPLES:

```
sage: J = SymmetricFunctions(QQ).jack(t=2).J()
sage: J([2,1]).expand(3)
4*x0^2*x1 + 4*x0*x1^2 + 4*x0^2*x2 + 6*x0*x1*x2 + 4*x1^2*x2 + 4*x0*x2^2 + 4*x1*x2^2
sage: (2*J([2])).expand(0)
0
sage: (3*J([])).expand(0)
3
```

frobenius(*n*)

Return the image of the symmetric function `self` under the n-th Frobenius operator.

The n-th Frobenius operator $\mathbf{f}_n$ is defined to be the map from the ring of symmetric functions to itself that sends every symmetric function $P(x_1, x_2, x_3, \ldots)$ to $P(x_1^n, x_2^n, x_3^n, \ldots)$. This operator $\mathbf{f}_n$ is a Hopf algebra endomorphism, and satisfies

$$\mathbf{f}_n m_{(\lambda_1,\lambda_2,\lambda_3,\ldots)} = m_{(n\lambda_1,n\lambda_2,n\lambda_3,\ldots)}$$

for every partition $(\lambda_1, \lambda_2, \lambda_3, \ldots)$ (where m means the monomial basis). Moreover, $\mathbf{f}_n(p_r) = p_{nr}$ for every positive integer r (where p_k denotes the k-th powersum symmetric function).

The n-th Frobenius operator is also called the n-th Frobenius endomorphism. It is not related to the Frobenius map which connects the ring of symmetric functions with the representation theory of the symmetric group.

The n-th Frobenius operator is also the n-th Adams operator of the Λ-ring of symmetric functions over the integers.

The n-th Frobenius operator can also be described via plethysm: Every symmetric function P satisfies $\mathbf{f}_n(P) = p_n \circ P = P \circ p_n$, where p_n is the n-th powersum symmetric function, and $\circ$ denotes (outer) plethysm.

INPUT:

- `n` – a positive integer

OUTPUT:

The result of applying the n-th Frobenius operator (on the ring of symmetric functions) to `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: p = Sym.p()
sage: h = Sym.h()
sage: s = Sym.s()
sage: m = Sym.m()
sage: s[3].frobenius(2)
-s[3, 3] + s[4, 2] - s[5, 1] + s[6]
sage: m[4,2,1].frobenius(3)
m[12, 6, 3]
sage: p[4,2,1].frobenius(3)
p[12, 6, 3]
sage: h[4].frobenius(2)
h[4, 4] - 2*h[5, 3] + 2*h[6, 2] - 2*h[7, 1] + 2*h[8]
```

The Frobenius endomorphisms are multiplicative:

```
sage: all( all( s(lam).frobenius(3) * s(mu).frobenius(3) # long time
....:           == (s(lam) * s(mu)).frobenius(3)
....:           for mu in Partitions(3) )
....:       for lam in Partitions(3) )
True
sage: all( all( m(lam).frobenius(2) * m(mu).frobenius(2)
....:           == (m(lam) * m(mu)).frobenius(2)
....:           for mu in Partitions(4) )
....:       for lam in Partitions(4) )
True
sage: all( all( p(lam).frobenius(2) * p(mu).frobenius(2)
....:           == (p(lam) * p(mu)).frobenius(2)
....:           for mu in Partitions(3) )
```

```
....:       for lam in Partitions(4) )
True
```

Being Hopf algebra endomorphisms, the Frobenius operators commute with the antipode:

```
sage: all( p(lam).frobenius(4).antipode()
....:       == p(lam).antipode().frobenius(4)
....:       for lam in Partitions(3) )
True
```

Testing the $\mathbf{f}_n(P) = p_n \circ P = P \circ p_n$ equality (over $\mathbf{Q}$, since plethysm is currently not defined over $\mathbf{Z}$ in Sage):

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: p = Sym.p()
sage: all( s(lam).frobenius(3) == s(lam).plethysm(p[3])
....:       == s(p[3].plethysm(s(lam)))
....:       for lam in Partitions(4) )
True
```

By Exercise 7.61 in Stanley's EC2 [STA] (see the errata on his website), $\mathbf{f}_n(h_m)$ is a linear combination of Schur polynomials (of straight shapes) using coefficients 0, 1 and -1 only; moreover, all partitions whose Schur polynomials occur with coefficient $\neq 0$ in this combination have empty n-cores. Let us check this on examples:

```
sage: all( all( all( (coeff == -1 or coeff == 1)
....:                and lam.core(n) == Partition([])
....:                for lam, coeff in s([m]).frobenius(n) )
....:           for n in range(2, 4) )
....:       for m in range(4) )
True
```

See also:

`plethysm()`

Todo

This method is fast on the monomial and the powersum bases, while all other bases get converted to the monomial basis. For most bases, this is probably the quickest way to do, but at least the Schur basis should have a better option. (Quoting from Stanley's EC2 [STA]: "D. G. Duncan, J. London Math. Soc. 27 (1952), 235-236, or Y. M. Chen, A. M. Garsia, and J. B. Remmel, Contemp. Math. 34 (1984), 109-153".)

---

hl_creation_operator(*nu*, *t=None*)

This is the vertex operator that generalizes Jing's operator.

It is a linear operator that raises the degree by $|\nu|$. This creation operator is a t-analogue of multiplication by `s(nu)` .

See also:

Proposition 5 in [SZ2001].

INPUT:

- `nu` – a partition
- `t` – (default: `None`, in which case `t` is used) a parameter

REFERENCES:

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ['t']).s()
sage: s([2]).hl_creation_operator([3,2])
s[3, 2, 2] + t*s[3, 3, 1] + t*s[4, 2, 1] + t^2*s[4, 3] + t^2*s[5, 2]

sage: Sym = SymmetricFunctions(FractionField(QQ['t']))
sage: HLQp = Sym.hall_littlewood().Qp()
sage: s = Sym.s()
sage: HLQp(s([2]).hl_creation_operator([2]).hl_creation_operator([3]))
HLQp[3, 2, 2]
sage: s([2,2]).hl_creation_operator([2,1])
t*s[2, 2, 2, 1] + t^2*s[3, 2, 1, 1] + t^2*s[3, 2, 2] + t^3*s[3, 3, 1] + t^3*s[4, 2, 1] + t^4
sage: s(1).hl_creation_operator([2,1,1])
s[2, 1, 1]
sage: s(0).hl_creation_operator([2,1,1])
0
sage: s([3,2]).hl_creation_operator([2,1,1])
(t^2-t)*s[2, 2, 2, 2, 1] + t^3*s[3, 2, 2, 1, 1] + (t^3-t^2)*s[3, 2, 2, 2] + t^3*s[3, 3, 1, 1
```

TESTS:

```
sage: s(0).hl_creation_operator([1])
0
```

inner_plethysm(x)

Return the inner plethysm of `self` with `x`.

Whenever R is a $\mathbf{Q}$-algebra, and f and g are two symmetric functions over R such that the constant term of f is zero, the inner plethysm of f with g is a symmetric function over R, and the degree of this symmetric function is the same as the degree of g. We will denote the inner plethysm of f with g by $f\{g\}$ (in contrast to the notation of outer plethysm which is generally denoted $f[g]$); in Sage syntax, it is `f.inner_plethysm(g)`.

First we describe the axiomatic definition of the operation; see below for a representation-theoretic interpretation. In the following equations, we denote the outer product (i.e., the standard product on the ring of symmetric functions, `product()`) by $\cdot$ and the Kronecker product (`itensor()`) by $*$).

$$(f+g)\{h\} = f\{h\} + g\{h\}$$
$$(f\cdot g)\{h\} = (f\{h\}) * (g\{h\})$$
$$p_k\{f+g\} = p_k\{f\} + p_k\{g\}$$

where p_k is the k-th power-sum symmetric function for every $k > 0$.

Let σ be a permutation of cycle type μ and let μ^k be the cycle type of σ^k. Then,

$$p_k\{p_\mu/z_\mu\} = \sum_{\nu:\nu^k=\mu} p_\nu/z_\nu$$

Since $(p_\mu/z_\mu)_\mu$ is a basis for the symmetric functions, these four formulas define the symmetric function operation $f\{g\}$ for any symmetric functions f and g (where f has constant term 0) by expanding f in the power sum basis and g in the dual basis p_μ/z_μ.

See also:

`itensor()`, `partition_power()`, `plethysm()`

This operation admits a representation-theoretic interpretation in the case where f is a Schur function s_λ and g is a homogeneous degree n symmetric function with nonnegative integral coefficients in the

Schur basis. The symmetric function $f\{g\}$ is the Frobenius image of the S_n-representation constructed as follows.

The assumptions on g imply that g is the Frobenius image of a representation ρ of the symmetric group S_n:

$$\rho : S_n \to GL_N.$$

If the degree N of this representation is greater than or equal to the number of parts of λ, then f, which denotes s_λ, corresponds to the character of some irreducible GL_N-representation, say

$$\sigma : GL_N \to GL_M.$$

The composition $\sigma \circ \rho : S_n \to GL_M$ is a representation of S_n whose Frobenius image is precisely $f\{g\}$.

If N is less than the number of parts of λ, then $f\{g\}$ is 0 by definition.

When f is a symmetric function with constant term $\neq 0$, the inner plethysm $f\{g\}$ isn't well-defined in the ring of symmetric functions. Indeed, it is not clear how to define $1\{g\}$. The most sensible way to get around this probably is defining it as the infinite sum $h_0 + h_1 + h_2 + \cdots$ (where h_i means the i-th complete homogeneous symmetric function) in the completion of this ring with respect to its grading. This is how [SchaThi1994] defines $1\{g\}$. The present method, however, sets it to be the sum of h_i over all i for which the i-th homogeneous component of g is nonzero. This is rather a hack than a reasonable definition. Use with caution!

Note: If a symmetric function g is written in the form $g = g_0 + g_1 + g_2 + \cdots$ with each g_i homogeneous of degree i, then $f\{g\} = f\{g_0\} + f\{g_1\} + f\{g_2\} + \cdots$ for every f with constant term 0. But in general, inner plethysm is not linear in the second variable.

REFERENCES:

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

OUTPUT:

- an element of symmetric functions in the parent of `self`

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: p = Sym.power()
sage: h = Sym.complete()
sage: s([2,1]).inner_plethysm(s([1,1,1]))
0
sage: s([2]).inner_plethysm(s([2,1]))
s[2, 1] + s[3]
sage: s([1,1]).inner_plethysm(s([2,1]))
s[1, 1, 1]
sage: s[2,1].inner_tensor(s[2,1])
s[1, 1, 1] + s[2, 1] + s[3]

sage: f = s([2,1]) + 2*s([3,1])
sage: f.itensor(f)
s[1, 1, 1] + s[2, 1] + 4*s[2, 1, 1] + 4*s[2, 2] + s[3] + 4*s[3, 1] + 4*s[4]
sage: s( h([1,1]).inner_plethysm(f) )
s[1, 1, 1] + s[2, 1] + 4*s[2, 1, 1] + 4*s[2, 2] + s[3] + 4*s[3, 1] + 4*s[4]
```

```
sage: s([]).inner_plethysm(s([1,1]) + 2*s([2,1])+s([3]))
s[2] + s[3]
sage: [s([]).inner_plethysm(s(la)) for la in Partitions(4)]
[s[4], s[4], s[4], s[4], s[4]]
sage: s([3]).inner_plethysm(s([]))
s[]
sage: s[1,1,1,1].inner_plethysm(s[2,1])
0
sage: s[1,1,1,1].inner_plethysm(2*s[2,1])
s[3]

sage: p[3].inner_plethysm(p[3])
0
sage: p[3,3].inner_plethysm(p[3])
0
sage: p[3].inner_plethysm(p[1,1,1])
p[1, 1, 1] + 2*p[3]
sage: p[4].inner_plethysm(p[1,1,1,1]/24)
1/24*p[1, 1, 1, 1] + 1/4*p[2, 1, 1] + 1/8*p[2, 2] + 1/4*p[4]
sage: p[3,3].inner_plethysm(p[1,1,1])
6*p[1, 1, 1] + 12*p[3]
```

TESTS:

```
sage: s(0).inner_plethysm(s(0))
0
sage: s(1).inner_plethysm(s(0))
0
sage: s(0).inner_plethysm(s(1))
0
sage: s(1).inner_plethysm(s(1))
s[]
sage: s(2).inner_plethysm(s(1))
2*s[]
sage: s(1).inner_plethysm(s(2))
s[]
```

inner_tensor(*x*)

Return the internal (tensor) product of `self` and `x` in the basis of `self`.

The internal tensor product can be defined as the linear extension of the definition on power sums $p_\lambda * p_\mu = \delta_{\lambda,\mu} z_\lambda p_\lambda$, where $z_\lambda = (1^{r_1} r_1!)(2^{r_2} r_2!) \cdots$ for $\lambda = (1^{r_1} 2^{r_2} \cdots)$ and where $*$ denotes the internal tensor product. The internal tensor product is also known as the Kronecker product, or as the second multiplication on the ring of symmetric functions.

Note that the internal product of any two homogeneous symmetric functions of equal degrees is a homogeneous symmetric function of the same degree. On the other hand, the internal product of two homogeneous symmetric functions of distinct degrees is 0.

Note: The internal product is sometimes referred to as "inner product" in the literature, but unfortunately this name is shared by a different operation, namely the Hall inner product (see `scalar()`).

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

OUTPUT:

- the internal product of `self` with `x` (an element of the ring of symmetric functions in the same basis

as `self`)

The methods `itensor()`, `internal_product()`, `kronecker_product()`, `inner_tensor()` are all synonyms.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: b = s([3])
sage: a.itensor(b)
s[2, 1]
sage: c = s([3,2,1])
sage: c.itensor(c)
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2]
 + 4*s[3, 1, 1, 1] + 5*s[3, 2, 1] + 2*s[3, 3] + 4*s[4, 1, 1]
 + 3*s[4, 2] + 2*s[5, 1] + s[6]
```

There are few quantitative results pertaining to Kronecker products in general, which makes their computation so difficult. Let us test a few of them in different bases.

The Kronecker product of any homogeneous symmetric function f of degree n with the n-th complete homogeneous symmetric function `h[n]` (a.k.a. `s[n]`) is f:

```
sage: h = SymmetricFunctions(ZZ).h()
sage: all( h([5]).itensor(h(p)) == h(p) for p in Partitions(5) )
True
```

The Kronecker product of a Schur function s_λ with the n-th elementary symmetric function `e[n]`, where $n = |\lambda|$, is $s_{\lambda'}$ (where λ' is the conjugate partition of λ):

```
sage: F = CyclotomicField(12)
sage: s = SymmetricFunctions(F).s()
sage: e = SymmetricFunctions(F).e()
sage: all( e([5]).itensor(s(p)) == s(p.conjugate()) for p in Partitions(5) )
True
```

The Kronecker product is commutative:

```
sage: e = SymmetricFunctions(FiniteField(19)).e()
sage: m = SymmetricFunctions(FiniteField(19)).m()
sage: all( all( e(p).itensor(m(q)) == m(q).itensor(e(p)) for q in Partitions(4) )
....:      for p in Partitions(4) )
True

sage: F = FractionField(QQ['q','t'])
sage: mq = SymmetricFunctions(F).macdonald().Q()
sage: mh = SymmetricFunctions(F).macdonald().H()
sage: all( all( mq(p).itensor(mh(r)) == mh(r).itensor(mq(p))    # long time
....:           for r in Partitions(4) )
....:      for p in Partitions(3) )
True
```

Let us check (on examples) Proposition 5.2 of Gelfand, Krob, Lascoux, Leclerc, Retakh, Thibon, "Noncommutative symmetric functions", Arxiv hep-th/9407124, for $r = 2$:

```
sage: e = SymmetricFunctions(FiniteField(29)).e()
sage: s = SymmetricFunctions(FiniteField(29)).s()
sage: m = SymmetricFunctions(FiniteField(29)).m()
sage: def tensor_copr(u, v, w):  # computes \mu ((u \otimes v) * \Delta(w)) with
....:                            # * meaning Kronecker product and \mu meaning the
....:                            # usual multiplication.
```

```
....:     result = w.parent().zero()
....:     for partition_pair, coeff in w.coproduct():
....:         result += coeff * w.parent()(u).itensor(partition_pair[0]) * w.parent()(v).ite
....:     return result
sage: all( all( all( tensor_copr(e[u], s[v], m[w])   # long time
....:                == (e[u] * s[v]).itensor(m[w])
....:                for w in Partitions(5) )
....:           for v in Partitions(2) )
....:      for u in Partitions(3) )
True
```

Some examples from Briand, Orellana, Rosas, "The stability of the Kronecker products of Schur functions." Arxiv 0907.4652:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: s[2,2].itensor(s[2,2])
s[1, 1, 1, 1] + s[2, 2] + s[4]
sage: s[3,2].itensor(s[3,2])
s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2] + s[4, 1] + s[5]
sage: s[4,2].itensor(s[4,2])
s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[4, 1, 1] + 2*s[4, 2] + s[5, 1] + s[6]
```

An example from p. 220 of Thibon, "Hopf algebras of symmetric functions and tensor products of symmetric group representations", International Journal of Algebra and Computation, 1991:

```
sage: s = SymmetricFunctions(QQbar).s()
sage: s[2,1].itensor(s[2,1])
s[1, 1, 1] + s[2, 1] + s[3]
```

TESTS:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([8,8])
sage: a.itensor(a) # long time
s[4, 4, 4, 4] + s[5, 5, 3, 3] + s[5, 5, 5, 1] + s[6, 4, 4, 2]
 + s[6, 6, 2, 2] + s[6, 6, 4] + s[7, 3, 3, 3] + s[7, 5, 3, 1]
 + s[7, 7, 1, 1] + s[8, 4, 2, 2] + s[8, 4, 4] + s[8, 6, 2]
 + s[8, 8] + s[9, 3, 3, 1] + s[9, 5, 1, 1] + s[10, 2, 2, 2]
 + s[10, 4, 2] + s[10, 6] + s[11, 3, 1, 1] + s[12, 2, 2]
 + s[12, 4] + s[13, 1, 1, 1] + s[14, 2] + s[16]
sage: s[8].itensor(s[7])
0
sage: s(0).itensor(s(0))
0
sage: s(1).itensor(s(0))
0
sage: s(0).itensor(s(1))
0
sage: s(1).itensor(s(1))
s[]
```

Same over the ring of integers:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: a = s([8,8])
sage: a.itensor(a) # long time
s[4, 4, 4, 4] + s[5, 5, 3, 3] + s[5, 5, 5, 1] + s[6, 4, 4, 2]
 + s[6, 6, 2, 2] + s[6, 6, 4] + s[7, 3, 3, 3] + s[7, 5, 3, 1]
 + s[7, 7, 1, 1] + s[8, 4, 2, 2] + s[8, 4, 4] + s[8, 6, 2]
 + s[8, 8] + s[9, 3, 3, 1] + s[9, 5, 1, 1] + s[10, 2, 2, 2]
```

```
 + s[10, 4, 2] + s[10, 6] + s[11, 3, 1, 1] + s[12, 2, 2]
 + s[12, 4] + s[13, 1, 1, 1] + s[14, 2] + s[16]
sage: s[8].itensor(s[7])
0
sage: s(0).itensor(s(0))
0
sage: s(1).itensor(s(0))
0
sage: s(0).itensor(s(1))
0
sage: s(1).itensor(s(1))
s[]
```

Theorem 2.1 in Bessenrodt, van Willigenburg, Arxiv 1105.3170v2:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: all( all( max( r[0] for r in s(p).itensor(s(q)).monomial_coefficients().keys() )
....:           == sum( min(p[i], q.get_part(i)) for i in range(len(p)) )
....:           for p in Partitions(4) )
....:       for q in Partitions(4) )
True
sage: all( all( max( len(r) for r in s(p).itensor(s(q)).monomial_coefficients().keys() )
....:           == sum( min(p[i], q.conjugate().get_part(i)) for i in range(len(p)) )
....:           for p in Partitions(4) )
....:       for q in Partitions(4) )
True
```

Check that the basis and ground ring of `self` are preserved:

```
sage: F = CyclotomicField(12)
sage: s = SymmetricFunctions(F).s()
sage: e = SymmetricFunctions(F).e()
sage: e[3].itensor(s[3])
e[3]
sage: s[3].itensor(e[3])
s[1, 1, 1]
sage: parent(e[3].itensor(s[3]))
Symmetric Functions over Cyclotomic Field of order 12 and degree 4 in the elementary basis
sage: parent(s[3].itensor(e[3]))
Symmetric Functions over Cyclotomic Field of order 12 and degree 4 in the Schur basis
```

Note: The currently existing implementation of this function is technically unsatisfactory. It distinguishes the case when the base ring is a $\mathbf{Q}$-algebra (in which case the Kronecker product can be easily computed using the power sum basis) from the case where it isn't. In the latter, it does a computation using universal coefficients, again distinguishing the case when it is able to compute the "corresponding" basis of the symmetric function algebra over $\mathbf{Q}$ (using the `corresponding_basis_over` hack) from the case when it isn't (in which case it transforms everything into the Schur basis, which is slow).

---

internal_coproduct()

Return the inner coproduct of `self` in the basis of `self`.

The inner coproduct (also known as the Kronecker coproduct, as the internal coproduct, or as the second comultiplication on the ring of symmetric functions) is a ring homomorphism $\Delta^{\times}$ from the ring of symmetric functions to the tensor product (over the base ring) of this ring with itself. It is uniquely characterized by the formula

$$\Delta^{\times}(h_n) = \sum_{\lambda \vdash n} s_\lambda \otimes s_\lambda = \sum_{\lambda \vdash n} h_\lambda \otimes m_\lambda = \sum_{\lambda \vdash n} m_\lambda \otimes h_\lambda,$$

where $\lambda \vdash n$ means λ is a partition of n, and n is any nonnegative integer. It also satisfies

$$\Delta^{\times}(p_n) = p_n \otimes p_n$$

for any positive integer n. If the base ring is a $\mathbf{Q}$-algebra, it also satisfies

$$\Delta^{\times}(h_n) = \sum_{\lambda \vdash n} z_{\lambda}^{-1} p_{\lambda} \otimes p_{\lambda},$$

where

$$z_{\lambda} = \prod_{i=1}^{\infty} i^{m_i(\lambda)} m_i(\lambda)!$$

with $m_i(\lambda)$ meaning the number of appearances of i in λ (see `zee()`).

The method `kronecker_coproduct()` is a synonym of `internal_coproduct()`.

EXAMPLES:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: a = s([2,1])
sage: a.internal_coproduct()
s[1, 1, 1] # s[2, 1] + s[2, 1] # s[1, 1, 1] + s[2, 1] # s[2, 1] + s[2, 1] # s[3] + s[3] # s[

sage: e = SymmetricFunctions(QQ).e()
sage: b = e([2])
sage: b.internal_coproduct()
e[1, 1] # e[2] + e[2] # e[1, 1] - 2*e[2] # e[2]
```

The internal coproduct is adjoint to the internal product with respect to the Hall inner product: Any three symmetric functions f, g and h satisfy $\langle f*g, h\rangle = \sum_i \langle f, h'_i\rangle \langle g, h''_i\rangle$, where we write $\Delta^{\times}(h)$ as $\sum_i h'_i \otimes h''_i$. Let us check this in degree 4:

```
sage: e = SymmetricFunctions(FiniteField(29)).e()
sage: s = SymmetricFunctions(FiniteField(29)).s()
sage: m = SymmetricFunctions(FiniteField(29)).m()
sage: def tensor_incopr(f, g, h):  # computes \sum_i \left< f, h'_i \right> \left< g, h''_i
....:     result = h.base_ring().zero()
....:     for partition_pair, coeff in h.internal_coproduct():
....:         result += coeff * h.parent()(f).scalar(partition_pair[0]) * h.parent()(g).scal
....:     return result
sage: all( all( all( tensor_incopr(e[u], s[v], m[w]) == (e[u].itensor(s[v])).scalar(m[w])  #
....:                for w in Partitions(5) )
....:           for v in Partitions(2) )
....:      for u in Partitions(3) )
True
```

Let us check the formulas for $\Delta^{\times}(h_n)$ and $\Delta^{\times}(p_n)$ given in the description of this method:

```
sage: e = SymmetricFunctions(QQ).e()
sage: p = SymmetricFunctions(QQ).p()
sage: h = SymmetricFunctions(QQ).h()
sage: s = SymmetricFunctions(QQ).s()
sage: all( s(h([n])).internal_coproduct() == sum([tensor([s(lam), s(lam)]) for lam in Partit
....:      for n in range(6) )
True
sage: all( h([n]).internal_coproduct() == sum([tensor([h(lam), h(m(lam))]) for lam in Partit
....:      for n in range(6) )
True
sage: all( factorial(n) * h([n]).internal_coproduct()
```

```
....:       == sum([lam.conjugacy_class_size() * tensor([h(p(lam)), h(p(lam))])
....:               for lam in Partitions(n)])
....:       for n in range(6) )
True
```

TESTS:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([]).internal_coproduct()
s[] # s[]
```

internal_product(x)

Return the internal (tensor) product of `self` and `x` in the basis of `self`.

The internal tensor product can be defined as the linear extension of the definition on power sums $p_\lambda * p_\mu = \delta_{\lambda,\mu} z_\lambda p_\lambda$, where $z_\lambda = (1^{r_1} r_1!)(2^{r_2} r_2!) \cdots$ for $\lambda = (1^{r_1} 2^{r_2} \cdots)$ and where $*$ denotes the internal tensor product. The internal tensor product is also known as the Kronecker product, or as the second multiplication on the ring of symmetric functions.

Note that the internal product of any two homogeneous symmetric functions of equal degrees is a homogeneous symmetric function of the same degree. On the other hand, the internal product of two homogeneous symmetric functions of distinct degrees is 0.

Note: The internal product is sometimes referred to as "inner product" in the literature, but unfortunately this name is shared by a different operation, namely the Hall inner product (see `scalar()`).

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

OUTPUT:

- the internal product of `self` with `x` (an element of the ring of symmetric functions in the same basis as `self`)

The methods `itensor()`, `internal_product()`, `kronecker_product()`, `inner_tensor()` are all synonyms.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: b = s([3])
sage: a.itensor(b)
s[2, 1]
sage: c = s([3,2,1])
sage: c.itensor(c)
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2]
 + 4*s[3, 1, 1, 1] + 5*s[3, 2, 1] + 2*s[3, 3] + 4*s[4, 1, 1]
 + 3*s[4, 2] + 2*s[5, 1] + s[6]
```

There are few quantitative results pertaining to Kronecker products in general, which makes their computation so difficult. Let us test a few of them in different bases.

The Kronecker product of any homogeneous symmetric function f of degree n with the n-th complete homogeneous symmetric function `h[n]` (a.k.a. `s[n]`) is f:

```
sage: h = SymmetricFunctions(ZZ).h()
sage: all( h([5]).itensor(h(p)) == h(p) for p in Partitions(5) )
True
```

The Kronecker product of a Schur function s_λ with the n-th elementary symmetric function `e[n]`, where $n = |\lambda|$, is $s_{\lambda'}$ (where λ' is the conjugate partition of λ):

```
sage: F = CyclotomicField(12)
sage: s = SymmetricFunctions(F).s()
sage: e = SymmetricFunctions(F).e()
sage: all( e([5]).itensor(s(p)) == s(p.conjugate()) for p in Partitions(5) )
True
```

The Kronecker product is commutative:

```
sage: e = SymmetricFunctions(FiniteField(19)).e()
sage: m = SymmetricFunctions(FiniteField(19)).m()
sage: all( all( e(p).itensor(m(q)) == m(q).itensor(e(p)) for q in Partitions(4) )
....:      for p in Partitions(4) )
True

sage: F = FractionField(QQ['q','t'])
sage: mq = SymmetricFunctions(F).macdonald().Q()
sage: mh = SymmetricFunctions(F).macdonald().H()
sage: all( all( mq(p).itensor(mh(r)) == mh(r).itensor(mq(p))    # long time
....:           for r in Partitions(4) )
....:      for p in Partitions(3) )
True
```

Let us check (on examples) Proposition 5.2 of Gelfand, Krob, Lascoux, Leclerc, Retakh, Thibon, "Non-commutative symmetric functions", Arxiv hep-th/9407124, for $r = 2$:

```
sage: e = SymmetricFunctions(FiniteField(29)).e()
sage: s = SymmetricFunctions(FiniteField(29)).s()
sage: m = SymmetricFunctions(FiniteField(29)).m()
sage: def tensor_copr(u, v, w):  # computes \mu ((u \otimes v) * \Delta(w)) with
....:                            # * meaning Kronecker product and \mu meaning the
....:                            # usual multiplication.
....:     result = w.parent().zero()
....:     for partition_pair, coeff in w.coproduct():
....:         result += coeff * w.parent()(u).itensor(partition_pair[0]) * w.parent()(v).ite
....:     return result
sage: all( all( all( tensor_copr(e[u], s[v], m[w])    # long time
....:                == (e[u] * s[v]).itensor(m[w])
....:                for w in Partitions(5) )
....:           for v in Partitions(2) )
....:      for u in Partitions(3) )
True
```

Some examples from Briand, Orellana, Rosas, "The stability of the Kronecker products of Schur functions." Arxiv 0907.4652:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: s[2,2].itensor(s[2,2])
s[1, 1, 1, 1] + s[2, 2] + s[4]
sage: s[3,2].itensor(s[3,2])
s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2] + s[4, 1] + s[5]
sage: s[4,2].itensor(s[4,2])
s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[4, 1, 1] + 2*s[4, 2] + s[5, 1] + s[6]
```

An example from p. 220 of Thibon, "Hopf algebras of symmetric functions and tensor products of symmetric group representations", International Journal of Algebra and Computation, 1991:

```
sage: s = SymmetricFunctions(QQbar).s()
sage: s[2,1].itensor(s[2,1])
s[1, 1, 1] + s[2, 1] + s[3]
```

TESTS:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([8,8])
sage: a.itensor(a) # long time
s[4, 4, 4, 4] + s[5, 5, 3, 3] + s[5, 5, 5, 1] + s[6, 4, 4, 2]
 + s[6, 6, 2, 2] + s[6, 6, 4] + s[7, 3, 3, 3] + s[7, 5, 3, 1]
 + s[7, 7, 1, 1] + s[8, 4, 2, 2] + s[8, 4, 4] + s[8, 6, 2]
 + s[8, 8] + s[9, 3, 3, 1] + s[9, 5, 1, 1] + s[10, 2, 2, 2]
 + s[10, 4, 2] + s[10, 6] + s[11, 3, 1, 1] + s[12, 2, 2]
 + s[12, 4] + s[13, 1, 1, 1] + s[14, 2] + s[16]
sage: s[8].itensor(s[7])
0
sage: s(0).itensor(s(0))
0
sage: s(1).itensor(s(0))
0
sage: s(0).itensor(s(1))
0
sage: s(1).itensor(s(1))
s[]
```

Same over the ring of integers:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: a = s([8,8])
sage: a.itensor(a) # long time
s[4, 4, 4, 4] + s[5, 5, 3, 3] + s[5, 5, 5, 1] + s[6, 4, 4, 2]
 + s[6, 6, 2, 2] + s[6, 6, 4] + s[7, 3, 3, 3] + s[7, 5, 3, 1]
 + s[7, 7, 1, 1] + s[8, 4, 2, 2] + s[8, 4, 4] + s[8, 6, 2]
 + s[8, 8] + s[9, 3, 3, 1] + s[9, 5, 1, 1] + s[10, 2, 2, 2]
 + s[10, 4, 2] + s[10, 6] + s[11, 3, 1, 1] + s[12, 2, 2]
 + s[12, 4] + s[13, 1, 1, 1] + s[14, 2] + s[16]
sage: s[8].itensor(s[7])
0
sage: s(0).itensor(s(0))
0
sage: s(1).itensor(s(0))
0
sage: s(0).itensor(s(1))
0
sage: s(1).itensor(s(1))
s[]
```

Theorem 2.1 in Bessenrodt, van Willigenburg, Arxiv 1105.3170v2:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: all( all( max( r[0] for r in s(p).itensor(s(q)).monomial_coefficients().keys() )
....:           == sum( min(p[i], q.get_part(i)) for i in range(len(p)) )
....:           for p in Partitions(4) )
....:       for q in Partitions(4) )
True
sage: all( all( max( len(r) for r in s(p).itensor(s(q)).monomial_coefficients().keys() )
....:           == sum( min(p[i], q.conjugate().get_part(i)) for i in range(len(p)) )
....:           for p in Partitions(4) )
```

```
....:       for q in Partitions(4) )
True
```

Check that the basis and ground ring of `self` are preserved:

```
sage: F = CyclotomicField(12)
sage: s = SymmetricFunctions(F).s()
sage: e = SymmetricFunctions(F).e()
sage: e[3].itensor(s[3])
e[3]
sage: s[3].itensor(e[3])
s[1, 1, 1]
sage: parent(e[3].itensor(s[3]))
Symmetric Functions over Cyclotomic Field of order 12 and degree 4 in the elementary basis
sage: parent(s[3].itensor(e[3]))
Symmetric Functions over Cyclotomic Field of order 12 and degree 4 in the Schur basis
```

Note: The currently existing implementation of this function is technically unsatisfactory. It distinguishes the case when the base ring is a $\mathbf{Q}$-algebra (in which case the Kronecker product can be easily computed using the power sum basis) from the case where it isn't. In the latter, it does a computation using universal coefficients, again distinguishing the case when it is able to compute the "corresponding" basis of the symmetric function algebra over $\mathbf{Q}$ (using the `corresponding_basis_over` hack) from the case when it isn't (in which case it transforms everything into the Schur basis, which is slow).

---

is_schur_positive()

Return `True` if and only if `self` is Schur positive.

If s is the space of Schur functions over `self`'s base ring, then this is the same as `self._is_positive(s)`.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1]) + s([3])
sage: a.is_schur_positive()
True
sage: a = s([2,1]) - s([3])
sage: a.is_schur_positive()
False

sage: QQx = QQ['x']
sage: s = SymmetricFunctions(QQx).s()
sage: x = QQx.gen()
sage: a = (1+x)*s([2,1])
sage: a.is_schur_positive()
True
sage: a = (1-x)*s([2,1])
sage: a.is_schur_positive()
False
sage: s(0).is_schur_positive()
True
sage: s(1+x).is_schur_positive()
True
```

itensor(x)

Return the internal (tensor) product of `self` and `x` in the basis of `self`.

The internal tensor product can be defined as the linear extension of the definition on power sums $p_\lambda *$

$p_\mu = \delta_{\lambda,\mu} z_\lambda p_\lambda$, where $z_\lambda = (1^{r_1} r_1!)(2^{r_2} r_2!) \cdots$ for $\lambda = (1^{r_1} 2^{r_2} \cdots)$ and where $*$ denotes the internal tensor product. The internal tensor product is also known as the Kronecker product, or as the second multiplication on the ring of symmetric functions.

Note that the internal product of any two homogeneous symmetric functions of equal degrees is a homogeneous symmetric function of the same degree. On the other hand, the internal product of two homogeneous symmetric functions of distinct degrees is 0.

Note: The internal product is sometimes referred to as "inner product" in the literature, but unfortunately this name is shared by a different operation, namely the Hall inner product (see `scalar()`).

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

OUTPUT:

- the internal product of `self` with `x` (an element of the ring of symmetric functions in the same basis as `self`)

The methods `itensor()`, `internal_product()`, `kronecker_product()`, `inner_tensor()` are all synonyms.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: b = s([3])
sage: a.itensor(b)
s[2, 1]
sage: c = s([3,2,1])
sage: c.itensor(c)
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2]
 + 4*s[3, 1, 1, 1] + 5*s[3, 2, 1] + 2*s[3, 3] + 4*s[4, 1, 1]
 + 3*s[4, 2] + 2*s[5, 1] + s[6]
```

There are few quantitative results pertaining to Kronecker products in general, which makes their computation so difficult. Let us test a few of them in different bases.

The Kronecker product of any homogeneous symmetric function f of degree n with the n-th complete homogeneous symmetric function `h[n]` (a.k.a. `s[n]`) is f:

```
sage: h = SymmetricFunctions(ZZ).h()
sage: all( h([5]).itensor(h(p)) == h(p) for p in Partitions(5) )
True
```

The Kronecker product of a Schur function s_λ with the n-th elementary symmetric function `e[n]`, where $n = |\lambda|$, is $s_{\lambda'}$ (where λ' is the conjugate partition of λ):

```
sage: F = CyclotomicField(12)
sage: s = SymmetricFunctions(F).s()
sage: e = SymmetricFunctions(F).e()
sage: all( e([5]).itensor(s(p)) == s(p.conjugate()) for p in Partitions(5) )
True
```

The Kronecker product is commutative:

```
sage: e = SymmetricFunctions(FiniteField(19)).e()
sage: m = SymmetricFunctions(FiniteField(19)).m()
sage: all( all( e(p).itensor(m(q)) == m(q).itensor(e(p)) for q in Partitions(4) )
....:      for p in Partitions(4) )
True
```

```
sage: F = FractionField(QQ['q','t'])
sage: mq = SymmetricFunctions(F).macdonald().Q()
sage: mh = SymmetricFunctions(F).macdonald().H()
sage: all( all( mq(p).itensor(mh(r)) == mh(r).itensor(mq(p))    # long time
....:           for r in Partitions(4) )
....:       for p in Partitions(3) )
True
```

Let us check (on examples) Proposition 5.2 of Gelfand, Krob, Lascoux, Leclerc, Retakh, Thibon, "Non-commutative symmetric functions", Arxiv hep-th/9407124, for $r = 2$:

```
sage: e = SymmetricFunctions(FiniteField(29)).e()
sage: s = SymmetricFunctions(FiniteField(29)).s()
sage: m = SymmetricFunctions(FiniteField(29)).m()
sage: def tensor_copr(u, v, w):  # computes \mu ((u \otimes v) * \Delta(w)) with
....:                            # * meaning Kronecker product and \mu meaning the
....:                            # usual multiplication.
....:     result = w.parent().zero()
....:     for partition_pair, coeff in w.coproduct():
....:         result += coeff * w.parent()(u).itensor(partition_pair[0]) * w.parent()(v).ite
....:     return result
sage: all( all( all( tensor_copr(e[u], s[v], m[w])    # long time
....:                == (e[u] * s[v]).itensor(m[w])
....:                for w in Partitions(5) )
....:           for v in Partitions(2) )
....:       for u in Partitions(3) )
True
```

Some examples from Briand, Orellana, Rosas, "The stability of the Kronecker products of Schur functions." Arxiv 0907.4652:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: s[2,2].itensor(s[2,2])
s[1, 1, 1, 1] + s[2, 2] + s[4]
sage: s[3,2].itensor(s[3,2])
s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2] + s[4, 1] + s[5]
sage: s[4,2].itensor(s[4,2])
s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[4, 1, 1] + 2*s[4, 2] + s[5, 1] + s[6]
```

An example from p. 220 of Thibon, "Hopf algebras of symmetric functions and tensor products of symmetric group representations", International Journal of Algebra and Computation, 1991:

```
sage: s = SymmetricFunctions(QQbar).s()
sage: s[2,1].itensor(s[2,1])
s[1, 1, 1] + s[2, 1] + s[3]
```

TESTS:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([8,8])
sage: a.itensor(a) # long time
s[4, 4, 4, 4] + s[5, 5, 3, 3] + s[5, 5, 5, 1] + s[6, 4, 4, 2]
 + s[6, 6, 2, 2] + s[6, 6, 4] + s[7, 3, 3, 3] + s[7, 5, 3, 1]
 + s[7, 7, 1, 1] + s[8, 4, 2, 2] + s[8, 4, 4] + s[8, 6, 2]
 + s[8, 8] + s[9, 3, 3, 1] + s[9, 5, 1, 1] + s[10, 2, 2, 2]
 + s[10, 4, 2] + s[10, 6] + s[11, 3, 1, 1] + s[12, 2, 2]
 + s[12, 4] + s[13, 1, 1, 1] + s[14, 2] + s[16]
sage: s[8].itensor(s[7])
0
```

```
sage: s(0).itensor(s(0))
0
sage: s(1).itensor(s(0))
0
sage: s(0).itensor(s(1))
0
sage: s(1).itensor(s(1))
s[]
```

Same over the ring of integers:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: a = s([8,8])
sage: a.itensor(a) # long time
s[4, 4, 4, 4] + s[5, 5, 3, 3] + s[5, 5, 5, 1] + s[6, 4, 4, 2]
 + s[6, 6, 2, 2] + s[6, 6, 4] + s[7, 3, 3, 3] + s[7, 5, 3, 1]
 + s[7, 7, 1, 1] + s[8, 4, 2, 2] + s[8, 4, 4] + s[8, 6, 2]
 + s[8, 8] + s[9, 3, 3, 1] + s[9, 5, 1, 1] + s[10, 2, 2, 2]
 + s[10, 4, 2] + s[10, 6] + s[11, 3, 1, 1] + s[12, 2, 2]
 + s[12, 4] + s[13, 1, 1, 1] + s[14, 2] + s[16]
sage: s[8].itensor(s[7])
0
sage: s(0).itensor(s(0))
0
sage: s(1).itensor(s(0))
0
sage: s(0).itensor(s(1))
0
sage: s(1).itensor(s(1))
s[]
```

Theorem 2.1 in Bessenrodt, van Willigenburg, Arxiv 1105.3170v2:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: all( all( max( r[0] for r in s(p).itensor(s(q)).monomial_coefficients().keys() )
....:           == sum( min(p[i], q.get_part(i)) for i in range(len(p)) )
....:           for p in Partitions(4) )
....:       for q in Partitions(4) )
True
sage: all( all( max( len(r) for r in s(p).itensor(s(q)).monomial_coefficients().keys() )
....:           == sum( min(p[i], q.conjugate().get_part(i)) for i in range(len(p)) )
....:           for p in Partitions(4) )
....:       for q in Partitions(4) )
True
```

Check that the basis and ground ring of `self` are preserved:

```
sage: F = CyclotomicField(12)
sage: s = SymmetricFunctions(F).s()
sage: e = SymmetricFunctions(F).e()
sage: e[3].itensor(s[3])
e[3]
sage: s[3].itensor(e[3])
s[1, 1, 1]
sage: parent(e[3].itensor(s[3]))
Symmetric Functions over Cyclotomic Field of order 12 and degree 4 in the elementary basis
sage: parent(s[3].itensor(e[3]))
Symmetric Functions over Cyclotomic Field of order 12 and degree 4 in the Schur basis
```

Note: The currently existing implementation of this function is technically unsatisfactory. It distinguishes the case when the base ring is a $\mathbf{Q}$-algebra (in which case the Kronecker product can be easily computed using the power sum basis) from the case where it isn't. In the latter, it does a computation using universal coefficients, again distinguishing the case when it is able to compute the "corresponding" basis of the symmetric function algebra over $\mathbf{Q}$ (using the `corresponding_basis_over` hack) from the case when it isn't (in which case it transforms everything into the Schur basis, which is slow).

---

kronecker_coproduct()

Return the inner coproduct of `self` in the basis of `self`.

The inner coproduct (also known as the Kronecker coproduct, as the internal coproduct, or as the second comultiplication on the ring of symmetric functions) is a ring homomorphism $\Delta^{\times}$ from the ring of symmetric functions to the tensor product (over the base ring) of this ring with itself. It is uniquely characterized by the formula

$$\Delta^{\times}(h_n) = \sum_{\lambda \vdash n} s_\lambda \otimes s_\lambda = \sum_{\lambda \vdash n} h_\lambda \otimes m_\lambda = \sum_{\lambda \vdash n} m_\lambda \otimes h_\lambda,$$

where $\lambda \vdash n$ means λ is a partition of n, and n is any nonnegative integer. It also satisfies

$$\Delta^{\times}(p_n) = p_n \otimes p_n$$

for any positive integer n. If the base ring is a $\mathbf{Q}$-algebra, it also satisfies

$$\Delta^{\times}(h_n) = \sum_{\lambda \vdash n} z_\lambda^{-1} p_\lambda \otimes p_\lambda,$$

where

$$z_\lambda = \prod_{i=1}^{\infty} i^{m_i(\lambda)} m_i(\lambda)!$$

with $m_i(\lambda)$ meaning the number of appearances of i in λ (see `zee()`).

The method `kronecker_coproduct()` is a synonym of `internal_coproduct()`.

EXAMPLES:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: a = s([2,1])
sage: a.internal_coproduct()
s[1, 1, 1] # s[2, 1] + s[2, 1] # s[1, 1, 1] + s[2, 1] # s[2, 1] + s[2, 1] # s[3] + s[3] # s[

sage: e = SymmetricFunctions(QQ).e()
sage: b = e([2])
sage: b.internal_coproduct()
e[1, 1] # e[2] + e[2] # e[1, 1] - 2*e[2] # e[2]
```

The internal coproduct is adjoint to the internal product with respect to the Hall inner product: Any three symmetric functions f, g and h satisfy $\langle f*g, h\rangle = \sum_i \langle f, h'_i\rangle \langle g, h''_i\rangle$, where we write $\Delta^{\times}(h)$ as $\sum_i h'_i \otimes h''_i$. Let us check this in degree 4:

```
sage: e = SymmetricFunctions(FiniteField(29)).e()
sage: s = SymmetricFunctions(FiniteField(29)).s()
sage: m = SymmetricFunctions(FiniteField(29)).m()
sage: def tensor_incopr(f, g, h):  # computes \sum_i \left< f, h'_i \right> \left< g, h''_i
....:     result = h.base_ring().zero()
....:     for partition_pair, coeff in h.internal_coproduct():
....:         result += coeff * h.parent()(f).scalar(partition_pair[0]) * h.parent()(g).scal
....:     return result
```

```
sage: all( all( all( tensor_incopr(e[u], s[v], m[w]) == (e[u].itensor(s[v])).scalar(m[w])  #
....:                for w in Partitions(5) )
....:           for v in Partitions(2) )
....:      for u in Partitions(3) )
True
```

Let us check the formulas for $\Delta^{\times}(h_n)$ and $\Delta^{\times}(p_n)$ given in the description of this method:

```
sage: e = SymmetricFunctions(QQ).e()
sage: p = SymmetricFunctions(QQ).p()
sage: h = SymmetricFunctions(QQ).h()
sage: s = SymmetricFunctions(QQ).s()
sage: all( s(h([n])).internal_coproduct() == sum([tensor([s(lam), s(lam)]) for lam in Partit
....:      for n in range(6) )
True
sage: all( h([n]).internal_coproduct() == sum([tensor([h(lam), h(m(lam))]) for lam in Partit
....:      for n in range(6) )
True
sage: all( factorial(n) * h([n]).internal_coproduct()
....:      == sum([lam.conjugacy_class_size() * tensor([h(p(lam)), h(p(lam))])
....:              for lam in Partitions(n)])
....:      for n in range(6) )
True
```

TESTS:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([]).internal_coproduct()
s[] # s[]
```

kronecker_product (*x*)

Return the internal (tensor) product of `self` and `x` in the basis of `self`.

The internal tensor product can be defined as the linear extension of the definition on power sums $p_\lambda * p_\mu = \delta_{\lambda,\mu} z_\lambda p_\lambda$, where $z_\lambda = (1^{r_1} r_1!)(2^{r_2} r_2!) \cdots$ for $\lambda = (1^{r_1} 2^{r_2} \cdots)$ and where $*$ denotes the internal tensor product. The internal tensor product is also known as the Kronecker product, or as the second multiplication on the ring of symmetric functions.

Note that the internal product of any two homogeneous symmetric functions of equal degrees is a homogeneous symmetric function of the same degree. On the other hand, the internal product of two homogeneous symmetric functions of distinct degrees is 0.

Note: The internal product is sometimes referred to as "inner product" in the literature, but unfortunately this name is shared by a different operation, namely the Hall inner product (see `scalar()`).

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

OUTPUT:

- the internal product of `self` with `x` (an element of the ring of symmetric functions in the same basis as `self`)

The methods `itensor()`, `internal_product()`, `kronecker_product()`, `inner_tensor()` are all synonyms.

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: b = s([3])
sage: a.itensor(b)
s[2, 1]
sage: c = s([3,2,1])
sage: c.itensor(c)
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2]
 + 4*s[3, 1, 1, 1] + 5*s[3, 2, 1] + 2*s[3, 3] + 4*s[4, 1, 1]
 + 3*s[4, 2] + 2*s[5, 1] + s[6]
```

There are few quantitative results pertaining to Kronecker products in general, which makes their computation so difficult. Let us test a few of them in different bases.

The Kronecker product of any homogeneous symmetric function f of degree n with the n-th complete homogeneous symmetric function `h[n]` (a.k.a. `s[n]`) is f:

```
sage: h = SymmetricFunctions(ZZ).h()
sage: all( h([5]).itensor(h(p)) == h(p) for p in Partitions(5) )
True
```

The Kronecker product of a Schur function s_λ with the n-th elementary symmetric function `e[n]`, where $n = |\lambda|$, is $s_{\lambda'}$ (where λ' is the conjugate partition of λ):

```
sage: F = CyclotomicField(12)
sage: s = SymmetricFunctions(F).s()
sage: e = SymmetricFunctions(F).e()
sage: all( e([5]).itensor(s(p)) == s(p.conjugate()) for p in Partitions(5) )
True
```

The Kronecker product is commutative:

```
sage: e = SymmetricFunctions(FiniteField(19)).e()
sage: m = SymmetricFunctions(FiniteField(19)).m()
sage: all( all( e(p).itensor(m(q)) == m(q).itensor(e(p)) for q in Partitions(4) )
....:      for p in Partitions(4) )
True

sage: F = FractionField(QQ['q','t'])
sage: mq = SymmetricFunctions(F).macdonald().Q()
sage: mh = SymmetricFunctions(F).macdonald().H()
sage: all( all( mq(p).itensor(mh(r)) == mh(r).itensor(mq(p))    # long time
....:           for r in Partitions(4) )
....:      for p in Partitions(3) )
True
```

Let us check (on examples) Proposition 5.2 of Gelfand, Krob, Lascoux, Leclerc, Retakh, Thibon, "Noncommutative symmetric functions", Arxiv hep-th/9407124, for $r = 2$:

```
sage: e = SymmetricFunctions(FiniteField(29)).e()
sage: s = SymmetricFunctions(FiniteField(29)).s()
sage: m = SymmetricFunctions(FiniteField(29)).m()
sage: def tensor_copr(u, v, w):  # computes \mu ((u \otimes v) * \Delta(w)) with
....:                            # * meaning Kronecker product and \mu meaning the
....:                            # usual multiplication.
....:     result = w.parent().zero()
....:     for partition_pair, coeff in w.coproduct():
....:         result += coeff * w.parent()(u).itensor(partition_pair[0]) * w.parent()(v).ite
....:     return result
sage: all( all( all( tensor_copr(e[u], s[v], m[w])    # long time
```

```
....:                 == (e[u] * s[v]).itensor(m[w])
....:                 for w in Partitions(5) )
....:             for v in Partitions(2) )
....:         for u in Partitions(3) )
True
```

Some examples from Briand, Orellana, Rosas, "The stability of the Kronecker products of Schur functions." Arxiv 0907.4652:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: s[2,2].itensor(s[2,2])
s[1, 1, 1, 1] + s[2, 2] + s[4]
sage: s[3,2].itensor(s[3,2])
s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 1, 1] + s[3, 2] + s[4, 1] + s[5]
sage: s[4,2].itensor(s[4,2])
s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[4, 1, 1] + 2*s[4, 2] + s[5, 1] + s[6]
```

An example from p. 220 of Thibon, "Hopf algebras of symmetric functions and tensor products of symmetric group representations", International Journal of Algebra and Computation, 1991:

```
sage: s = SymmetricFunctions(QQbar).s()
sage: s[2,1].itensor(s[2,1])
s[1, 1, 1] + s[2, 1] + s[3]
```

TESTS:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([8,8])
sage: a.itensor(a) # long time
s[4, 4, 4, 4] + s[5, 5, 3, 3] + s[5, 5, 5, 1] + s[6, 4, 4, 2]
 + s[6, 6, 2, 2] + s[6, 6, 4] + s[7, 3, 3, 3] + s[7, 5, 3, 1]
 + s[7, 7, 1, 1] + s[8, 4, 2, 2] + s[8, 4, 4] + s[8, 6, 2]
 + s[8, 8] + s[9, 3, 3, 1] + s[9, 5, 1, 1] + s[10, 2, 2, 2]
 + s[10, 4, 2] + s[10, 6] + s[11, 3, 1, 1] + s[12, 2, 2]
 + s[12, 4] + s[13, 1, 1, 1] + s[14, 2] + s[16]
sage: s[8].itensor(s[7])
0
sage: s(0).itensor(s(0))
0
sage: s(1).itensor(s(0))
0
sage: s(0).itensor(s(1))
0
sage: s(1).itensor(s(1))
s[]
```

Same over the ring of integers:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: a = s([8,8])
sage: a.itensor(a) # long time
s[4, 4, 4, 4] + s[5, 5, 3, 3] + s[5, 5, 5, 1] + s[6, 4, 4, 2]
 + s[6, 6, 2, 2] + s[6, 6, 4] + s[7, 3, 3, 3] + s[7, 5, 3, 1]
 + s[7, 7, 1, 1] + s[8, 4, 2, 2] + s[8, 4, 4] + s[8, 6, 2]
 + s[8, 8] + s[9, 3, 3, 1] + s[9, 5, 1, 1] + s[10, 2, 2, 2]
 + s[10, 4, 2] + s[10, 6] + s[11, 3, 1, 1] + s[12, 2, 2]
 + s[12, 4] + s[13, 1, 1, 1] + s[14, 2] + s[16]
sage: s[8].itensor(s[7])
0
sage: s(0).itensor(s(0))
```

```
0
sage: s(1).itensor(s(0))
0
sage: s(0).itensor(s(1))
0
sage: s(1).itensor(s(1))
s[]
```

Theorem 2.1 in Bessenrodt, van Willigenburg, Arxiv 1105.3170v2:

```
sage: s = SymmetricFunctions(ZZ).s()
sage: all( all( max( r[0] for r in s(p).itensor(s(q)).monomial_coefficients().keys() )
....:           == sum( min(p[i], q.get_part(i)) for i in range(len(p)) )
....:           for p in Partitions(4) )
....:       for q in Partitions(4) )
True
sage: all( all( max( len(r) for r in s(p).itensor(s(q)).monomial_coefficients().keys() )
....:           == sum( min(p[i], q.conjugate().get_part(i)) for i in range(len(p)) )
....:           for p in Partitions(4) )
....:       for q in Partitions(4) )
True
```

Check that the basis and ground ring of `self` are preserved:

```
sage: F = CyclotomicField(12)
sage: s = SymmetricFunctions(F).s()
sage: e = SymmetricFunctions(F).e()
sage: e[3].itensor(s[3])
e[3]
sage: s[3].itensor(e[3])
s[1, 1, 1]
sage: parent(e[3].itensor(s[3]))
Symmetric Functions over Cyclotomic Field of order 12 and degree 4 in the elementary basis
sage: parent(s[3].itensor(e[3]))
Symmetric Functions over Cyclotomic Field of order 12 and degree 4 in the Schur basis
```

Note: The currently existing implementation of this function is technically unsatisfactory. It distinguishes the case when the base ring is a **Q**-algebra (in which case the Kronecker product can be easily computed using the power sum basis) from the case where it isn't. In the latter, it does a computation using universal coefficients, again distinguishing the case when it is able to compute the "corresponding" basis of the symmetric function algebra over **Q** (using the `corresponding_basis_over` hack) from the case when it isn't (in which case it transforms everything into the Schur basis, which is slow).

---

left_padded_kronecker_product(x)

Return the left-padded Kronecker product of `self` and `x` in the basis of `self`.

The left-padded Kronecker product is a bilinear map mapping two symmetric functions to another, not necessarily preserving degree. It can be defined as follows: Let $*$ denote the Kronecker product (`itensor()`) on the space of symmetric functions. For any partitions α, β, γ, let $g_{\alpha,\beta}^{\gamma}$ denote the coefficient of the complete homogeneous symmetric function h_γ in the Kronecker product $h_\alpha * h_\beta$. For every partition $\lambda = (\lambda_1, \lambda_2, \lambda_3, \ldots)$ and every integer $n > |\lambda| + \lambda_1$, let $\lambda[n]$ denote the n-completion of λ (this is the partition $(n - |\lambda|, \lambda_1, \lambda_2, \lambda_3, \ldots)$; see `t_completion()`). Then, for any partitions α and β and every integer $n \geq |\alpha| + |\beta| + \alpha_1 + \beta_1$, we can write the Kronecker product $h_{\alpha[n]} * h_{\beta[n]}$ in the form

$$h_{\alpha[n]} * h_{\beta[n]} = \sum_{\gamma} g_{\alpha[n],\beta[n]}^{\gamma[n]} h_{\gamma[n]}$$

with γ ranging over all partitions. The coefficients $g^{\gamma[n]}_{\alpha[n],\beta[n]}$ are independent on n. These coefficients $g^{\gamma[n]}_{\alpha[n],\beta[n]}$ are denoted by $\overline{g}^{\gamma}_{\alpha,\beta}$, and the symmetric function

$$\sum_{\gamma} \overline{g}^{\gamma}_{\alpha,\beta} h_{\gamma}$$

is said to be the *left-padded Kronecker product* of h_α and h_β. By bilinearity, this extends to a definition of a left-padded Kronecker product of any two symmetric functions.

This notion of left-padded Kronecker product can be lifted to the non-commutative symmetric functions (`left_padded_kronecker_product()`).

Warning: Don't mistake this product for the reduced Kronecker product (`reduced_kronecker_product()`), which uses the Schur functions instead of the complete homogeneous functions in its definition.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

OUTPUT:

- the left-padded Kronecker product of `self` with `x` (an element of the ring of symmetric functions in the same basis as `self`)

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: h = Sym.h()
sage: h[2,1].left_padded_kronecker_product(h[3])
h[1, 1, 1, 1] + h[2, 1] + h[2, 1, 1] + h[2, 1, 1, 1] + h[2, 2, 1] + h[3, 2, 1]
sage: h[2,1].left_padded_kronecker_product(h[1])
h[1, 1, 1] + h[2, 1] + h[2, 1, 1]
sage: h[1].left_padded_kronecker_product(h[2,1])
h[1, 1, 1] + h[2, 1] + h[2, 1, 1]
sage: h[1,1].left_padded_kronecker_product(h[2])
h[1, 1] + 2*h[1, 1, 1] + h[2, 1, 1]
sage: h[1].left_padded_kronecker_product(h[2,1,1])
h[1, 1, 1, 1] + 2*h[2, 1, 1] + h[2, 1, 1, 1]
sage: h[2].left_padded_kronecker_product(h[3])
h[2, 1] + h[2, 1, 1] + h[3, 2]
```

Taking the left-padded Kronecker product with $1 = h$ is the identity map on the ring of symmetric functions:

```
sage: all( h[Partition([])].left_padded_kronecker_product(h[lam])
....:      == h[lam] for i in range(4)
....:      for lam in Partitions(i) )
True
```

Here is a rule for the left-padded Kronecker product of h_1 (this is the same as $h_{(1)}$) with any complete homogeneous function: Let λ be a partition. Then, the left-padded Kronecker product of h_1 and h_λ is $\sum_\mu a_\mu h_\mu$, where the sum runs over all partitions μ, and the coefficient a_μ is defined as the number of ways to obtain μ from λ by one of the following two operations:

- Insert a 1 into λ.
- Subtract 1 from one of the entries of λ (and remove the entry if it thus becomes 0), and insert a 1 into λ.

We check this for partitions of size ≤ 4:

```
sage: def mults1(I):
....:     # Left-padded Kronecker multiplication by h[1].
....:     res = h[I[:] + [1]]
....:     for k in range(len(I)):
....:         I2 = I[:]
....:         if I2[k] == 1:
....:             I2 = I2[:k] + I2[k+1:]
....:         else:
....:             I2[k] -= 1
....:         res += h[sorted(I2 + [1], reverse=True)]
....:     return res
sage: all( mults1(I) == h[1].left_padded_kronecker_product(h[I])
....:                == h[I].left_padded_kronecker_product(h[1])
....:      for i in range(5) for I in Partitions(i) )
True
```

The left-padded Kronecker product is commutative:

```
sage: all( h[lam].left_padded_kronecker_product(h[mu])
....:      == h[mu].left_padded_kronecker_product(h[lam])
....:      for lam in Partitions(3) for mu in Partitions(3) )
True
```

TESTS:

```
sage: h = SymmetricFunctions(QQ).h()
sage: (2*h([])).left_padded_kronecker_product(3*h([]))
6*h[]
```

Different bases and base rings:

```
sage: h = SymmetricFunctions(ZZ).h()
sage: e = SymmetricFunctions(ZZ).e()
sage: h(e[2].left_padded_kronecker_product(h[2]))
h[1, 1] + h[1, 1, 1] - h[2] + h[2, 1, 1] - h[2, 2]

sage: F = CyclotomicField(12)
sage: s = SymmetricFunctions(F).s()
sage: e = SymmetricFunctions(F).e()
sage: v = e[2].left_padded_kronecker_product(e[2]); v
e[1, 1] + e[1, 1, 1] + (-1)*e[2] + e[2, 2]
sage: parent(v)
Symmetric Functions over Cyclotomic Field of order 12 and degree 4 in the elementary basis

sage: s = SymmetricFunctions(ZZ).s()
sage: v = s[1].left_padded_kronecker_product(s[1]); parent(v)
Symmetric Functions over Integer Ring in the Schur basis
```

nabla(*q=None*, *t=None*, *power=1*)

Return the value of the nabla operator applied to `self`.

The eigenvectors of the nabla operator are the Macdonald polynomials in the Ht basis.

If the parameter `power` is an integer then it calculates nabla to that integer. The default value of `power` is 1.

INPUT:

- `q`, `t` – optional parameters (default: `None`, in which case `q` and `t` are used)

- `power` – (default: `1`) an integer indicating how many times to apply the operator ∇. Negative values of `power` indicate powers of ∇^{-1}.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: p = Sym.power()
sage: p([1,1]).nabla()
(-1/2*q*t+1/2*q+1/2*t+1/2)*p[1, 1] + (1/2*q*t-1/2*q-1/2*t+1/2)*p[2]
sage: p([2,1]).nabla(q=1)
(-t-1)*p[1, 1, 1] + t*p[2, 1]
sage: p([2]).nabla(q=1)*p([1]).nabla(q=1)
(-t-1)*p[1, 1, 1] + t*p[2, 1]
sage: s = Sym.schur()
sage: s([2,1]).nabla()
(-q^3*t-q^2*t^2-q*t^3)*s[1, 1, 1] + (-q^2*t-q*t^2)*s[2, 1]
sage: s([1,1,1]).nabla()
(q^3+q^2*t+q*t^2+t^3+q*t)*s[1, 1, 1] + (q^2+q*t+t^2+q+t)*s[2, 1] + s[3]
sage: s([1,1,1]).nabla(t=1)
(q^3+q^2+2*q+1)*s[1, 1, 1] + (q^2+2*q+2)*s[2, 1] + s[3]
sage: s(0).nabla()
0
sage: s(1).nabla()
s[]
sage: s([2,1]).nabla(power=-1)
((-q-t)/(q^2*t^2))*s[2, 1] + ((q^2+q*t+t^2)/(-q^3*t^3))*s[3]
sage: (s([2])+s([3])).nabla()
(-q*t)*s[1, 1] + (q^3*t^2+q^2*t^3)*s[1, 1, 1] + q^2*t^2*s[2, 1]
```

omega()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1}p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda|-\ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

The default implementation converts to the Schur basis, then performs the automorphism and changes back.

`omega_involution()` is a synonym for the `omega()` method.

EXAMPLES:

```
sage: J = SymmetricFunctions(QQ).jack(t=1).P()
sage: a = J([2,1]) + J([1,1,1])
sage: a.omega()
JackP[2, 1] + JackP[3]
sage: J(0).omega()
0
sage: J(1).omega()
JackP[]
```

The forgotten symmetric functions are the images of the monomial symmetric functions under omega:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: m = Sym.m()
sage: f = Sym.f()
sage: all( f(lam) == m(lam).omega() for lam in Partitions(3) )
True
sage: all( m(lam) == f(lam).omega() for lam in Partitions(3) )
True
```

omega_involution()

Return the image of `self` under the omega automorphism.

The *omega automorphism* is defined to be the unique algebra endomorphism ω of the ring of symmetric functions that satisfies $\omega(e_k) = h_k$ for all positive integers k (where e_k stands for the k-th elementary symmetric function, and h_k stands for the k-th complete homogeneous symmetric function). It furthermore is a Hopf algebra endomorphism and an involution, and it is also known as the *omega involution*. It sends the power-sum symmetric function p_k to $(-1)^{k-1}p_k$ for every positive integer k.

The images of some bases under the omega automorphism are given by

$$\omega(e_\lambda) = h_\lambda, \qquad \omega(h_\lambda) = e_\lambda, \qquad \omega(p_\lambda) = (-1)^{|\lambda|-\ell(\lambda)} p_\lambda, \qquad \omega(s_\lambda) = s_{\lambda'},$$

where λ is any partition, where $\ell(\lambda)$ denotes the length (`length()`) of the partition λ, where λ' denotes the conjugate partition (`conjugate()`) of λ, and where the usual notations for bases are used (e = elementary, h = complete homogeneous, p = powersum, s = Schur).

The default implementation converts to the Schur basis, then performs the automorphism and changes back.

`omega_involution()` is a synonym for the `omega()` method.

EXAMPLES:

```
sage: J = SymmetricFunctions(QQ).jack(t=1).P()
sage: a = J([2,1]) + J([1,1,1])
sage: a.omega()
JackP[2, 1] + JackP[3]
sage: J(0).omega()
0
sage: J(1).omega()
JackP[]
```

The forgotten symmetric functions are the images of the monomial symmetric functions under omega:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: m = Sym.m()
sage: f = Sym.f()
sage: all( f(lam) == m(lam).omega() for lam in Partitions(3) )
True
sage: all( m(lam) == f(lam).omega() for lam in Partitions(3) )
True
```

omega_qt(*q=None*, *t=None*)

Return the image of `self` under the q,t-deformed omega automorphism which sends p_k to $(-1)^{k-1} \cdot \frac{1-q^k}{1-t^k} \cdot p_k$ for all positive integers k.

In general, this is well-defined outside of the powersum basis only if the base ring is a $\mathbf{Q}$-algebra.

If $q = t$, then this is the omega automorphism (`omega()`).

INPUT:

- q, t – parameters (default: None, in which case 'q' and 't' are used)

EXAMPLES:

```
sage: QQqt = QQ['q,t'].fraction_field()
sage: q,t = QQqt.gens()
sage: p = SymmetricFunctions(QQqt).p()
sage: p[5].omega_qt()
((-q^5+1)/(-t^5+1))*p[5]
sage: p[5].omega_qt(q,t)
((-q^5+1)/(-t^5+1))*p[5]
sage: p([2]).omega_qt(q,t)
((q^2-1)/(-t^2+1))*p[2]
sage: p([2,1]).omega_qt(q,t)
((-q^3+q^2+q-1)/(t^3-t^2-t+1))*p[2, 1]
sage: p([3,2]).omega_qt(5,q)
-(2976/(q^5-q^3-q^2+1))*p[3, 2]
sage: p(0).omega_qt()
0
sage: p(1).omega_qt()
p[]
sage: H = SymmetricFunctions(QQqt).macdonald().H()
sage: H([1,1]).omega_qt()
((2*q^2-2*q*t-2*q+2*t)/(t^3-t^2-t+1))*McdH[1, 1] + ((q-1)/(t-1))*McdH[2]
sage: H([1,1]).omega_qt(q,t)
((2*q^2-2*q*t-2*q+2*t)/(t^3-t^2-t+1))*McdH[1, 1] + ((q-1)/(t-1))*McdH[2]
sage: H([1,1]).omega_qt(t,q)
((-t^3+t^2+t-1)/(-q^3+q^2+q-1))*McdH[2]
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: S = Sym.macdonald().S()
sage: S([1,1]).omega_qt()
((q^2-q*t-q+t)/(t^3-t^2-t+1))*McdS[1, 1] + ((-q^2*t+q*t+q-1)/(-t^3+t^2+t-1))*McdS[2]
sage: s = Sym.schur()
sage: s(S([1,1]).omega_qt())
s[2]
```

plethysm(*x*, *include=None*, *exclude=None*)

Return the outer plethysm of self with x. This is implemented only over base rings which are $\mathbf{Q}$-algebras. (To compute outer plethysms over general binomial rings, change bases to the fraction field.)

The outer plethysm of f with g is commonly denoted by $f[g]$ or by $f \circ g$. It is an algebra map in f, but not (generally) in g.

By default, the degree one elements are taken to be the generators for the self's base ring. This setting can be modified by specifying the include and exclude keywords.

INPUT:

- x – a symmetric function over the same base ring as self
- include – a list of variables to be treated as degree one elements instead of the default degree one elements
- exclude – a list of variables to be excluded from the default degree one elements

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: h = Sym.h()
sage: s ( h([3])( h([2]) ) )
s[2, 2, 2] + s[4, 2] + s[6]
```

```
sage: p = Sym.p()
sage: p([3])( s([2,1]) )
1/3*p[3, 3, 3] - 1/3*p[9]
sage: e = Sym.e()
sage: e([3])( e([2]) )
e[3, 3] + e[4, 1, 1] - 2*e[4, 2] - e[5, 1] + e[6]

sage: R.<t> = QQ[]
sage: s = SymmetricFunctions(R).s()
sage: a = s([3])
sage: f = t*s([2])
sage: a(f)
t^3*s[2, 2, 2] + t^3*s[4, 2] + t^3*s[6]
sage: f(a)
t*s[4, 2] + t*s[6]
sage: s(0).plethysm(s[1])
0
sage: s(1).plethysm(s[1])
s[]
sage: s(1).plethysm(s(0))
s[]
```

See also:

`frobenius()`

reduced_kronecker_product (*x*)

Return the reduced Kronecker product of `self` and `x` in the basis of `self`.

The reduced Kronecker product is a bilinear map mapping two symmetric functions to another, not necessarily preserving degree. It can be defined as follows: Let $*$ denote the Kronecker product (`itensor()`) on the space of symmetric functions. For any partitions α, β, γ, let $g_{\alpha,\beta}^{\gamma}$ denote the coefficient of the Schur function s_γ in the Kronecker product $s_\alpha * s_\beta$ (this is called a Kronecker coefficient). For every partition $\lambda = (\lambda_1, \lambda_2, \lambda_3, \ldots)$ and every integer $n > |\lambda| + \lambda_1$, let $\lambda[n]$ denote the n-completion of λ (this is the partition $(n - |\lambda|, \lambda_1, \lambda_2, \lambda_3, \ldots)$; see `t_completion()`). Then, Theorem 1.2 of [BOR09] shows that for any partitions α and β and every integer $n \geq |\alpha| + |\beta| + \alpha_1 + \beta_1$, we can write the Kronecker product $s_{\alpha[n]} * s_{\beta[n]}$ in the form

$$s_{\alpha[n]} * s_{\beta[n]} = \sum_{\gamma} g_{\alpha[n],\beta[n]}^{\gamma[n]} s_{\gamma[n]}$$

with γ ranging over all partitions. The coefficients $g_{\alpha[n],\beta[n]}^{\gamma[n]}$ are independent on n. These coefficients $g_{\alpha[n],\beta[n]}^{\gamma[n]}$ are denoted by $\overline{g}_{\alpha,\beta}^{\gamma}$, and the symmetric function

$$\sum_{\gamma} \overline{g}_{\alpha,\beta}^{\gamma} s_\gamma$$

is said to be the *reduced Kronecker product* of s_α and s_β. By bilinearity, this extends to a definition of a reduced Kronecker product of any two symmetric functions.

The definition of the reduced Kronecker product goes back to Murnaghan, and has recently been studied in [BOR09], [BdVO12] and other places (our notation $\overline{g}_{\alpha,\beta}^{\gamma}$ appears in these two sources).

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

OUTPUT:

- the reduced Kronecker product of `self` with `x` (an element of the ring of symmetric functions in the same basis as `self`)

EXAMPLES:

The example from page 2 of [BOR09]:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: s[2].reduced_kronecker_product(s[2])
s[] + s[1] + s[1, 1] + s[1, 1, 1] + 2*s[2] + 2*s[2, 1] + s[2, 2] + s[3] + s[3, 1] + s[4]
```

Taking the reduced Kronecker product with $1 = s$ is the identity map on the ring of symmetric functions:

```
sage: all( s[Partition([])].reduced_kronecker_product(s[lam])
....:      == s[lam] for i in range(4)
....:      for lam in Partitions(i) )
True
```

While reduced Kronecker products are hard to compute in general, there is a rule for taking reduced Kronecker products with s_1. Namely, for every partition λ, the reduced Kronecker product of s_λ with s_1 is $\sum_\mu a_\mu s_\mu$, where the sum runs over all partitions μ, and the coefficient a_μ is defined as the number of ways to obtain μ from λ by one of the following three operations:

- Add an addable cell (`addable_cells()`) to λ.
- Remove a removable cell (`removable_cells()`) from λ.
- First remove a removable cell from λ, then add an addable cell to the resulting Young diagram.

This is, in fact, Proposition 5.15 of [CO10] in an elementary wording. We check this for partitions of size ≤ 4:

```
sage: def mults1(lam):
....:     # Reduced Kronecker multiplication by s[1], according
....:     # to [CO10]_.
....:     res = s.zero()
....:     for mu in lam.up_list():
....:         res += s(mu)
....:     for mu in lam.down_list():
....:         res += s(mu)
....:         for nu in mu.up_list():
....:             res += s(nu)
....:     return res
sage: all( mults1(lam) == s[1].reduced_kronecker_product(s[lam])
....:      for i in range(5) for lam in Partitions(i) )
True
```

Here is the example on page 3 of Christian Gutschwager's Arxiv 0912.4411v3:

```
sage: s[1,1].reduced_kronecker_product(s[2])
s[1] + 2*s[1, 1] + s[1, 1, 1] + s[2] + 2*s[2, 1] + s[2, 1, 1] + s[3] + s[3, 1]
```

Example 39 from F. D. Murnaghan, "The analysis of the Kronecker product of irreducible representations of the symmetric group", American Journal of Mathematics, Vol. 60, No. 3, Jul. 1938:

```
sage: s[3].reduced_kronecker_product(s[2,1])
s[1] + 2*s[1, 1] + 2*s[1, 1, 1] + s[1, 1, 1, 1] + 2*s[2] + 5*s[2, 1] + 4*s[2, 1, 1]
+ s[2, 1, 1, 1] + 3*s[2, 2] + 2*s[2, 2, 1] + 2*s[3] + 5*s[3, 1] + 3*s[3, 1, 1]
+ 3*s[3, 2] + s[3, 2, 1] + 2*s[4] + 3*s[4, 1] + s[4, 1, 1] + s[4, 2] + s[5]
+ s[5, 1]
```

TESTS:

```
sage: h = SymmetricFunctions(QQ).h()
sage: (2*h([])).reduced_kronecker_product(3*h([]))
6*h[]
```

Different bases and base rings:

```
sage: h = SymmetricFunctions(ZZ).h()
sage: e = SymmetricFunctions(ZZ).e()
sage: h(e[2].reduced_kronecker_product(h[2]))
h[1] + 2*h[1, 1] + h[1, 1, 1] - h[2] + h[2, 1, 1] - h[2, 2]

sage: F = CyclotomicField(12)
sage: s = SymmetricFunctions(F).s()
sage: e = SymmetricFunctions(F).e()
sage: v = e[2].reduced_kronecker_product(e[2]); v
e[] + e[1] + 2*e[1, 1] + e[1, 1, 1] + (-1)*e[2] + e[2, 2]
sage: parent(v)
Symmetric Functions over Cyclotomic Field of order 12 and degree 4 in the elementary basis

sage: s = SymmetricFunctions(ZZ).s()
sage: v = s[1].reduced_kronecker_product(s[1]); parent(v)
Symmetric Functions over Integer Ring in the Schur basis
```

Todo

This implementation of the reduced Kronecker product is painfully slow.

---

restrict_degree(*d*, *exact=True*)

Return the degree `d` component of `self`.

INPUT:

- `d` – positive integer, degree of the terms to be returned
- `exact` – boolean, if `True`, returns the terms of degree exactly `d`, otherwise returns all terms of degree less than or equal to `d`

OUTPUT:

- the homogeneous component of `self` of degree `d`

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])
sage: z.restrict_degree(2)
0
sage: z.restrict_degree(1)
s[1]
sage: z.restrict_degree(3)
s[1, 1, 1] + s[2, 1]
sage: z.restrict_degree(3, exact=False)
s[1] + s[1, 1, 1] + s[2, 1]
sage: z.restrict_degree(0)
0
```

restrict_partition_lengths(*l*, *exact=True*)

Return the terms of `self` labelled by partitions of length `l`.

INPUT:

- `l` – nonnegative integer
- `exact` – boolean, defaulting to `True`

OUTPUT:

- if `True`, returns the terms labelled by partitions of length precisely `l`; otherwise returns all terms labelled by partitions of length less than or equal to `l`

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])
sage: z.restrict_partition_lengths(2)
s[2, 1]
sage: z.restrict_partition_lengths(0)
0
sage: z.restrict_partition_lengths(2, exact = False)
s[1] + s[2, 1] + s[4]
```

restrict_parts(*n*)

Return the terms of `self` labelled by partitions λ with $\lambda_1 \leq n$.

INPUT:

- `n` – positive integer, to restrict the parts of the partitions of the terms to be returned

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])
sage: z.restrict_parts(2)
s[1] + s[1, 1, 1] + s[2, 1]
sage: z.restrict_parts(1)
s[1] + s[1, 1, 1]
```

scalar(*x*, *zee=None*)

Return the standard scalar product between `self` and `x`.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`
- `zee` – an optional function on partitions giving the value for the scalar product between p_μ and p_μ (default is to use the standard `zee()` function)

This is the default implementation that converts both `self` and `x` into either Schur functions (if `zee` is not specified) or power-sum functions (if `zee` is specified) and performs the scalar product in that basis.

EXAMPLES:

```
sage: e = SymmetricFunctions(QQ).e()
sage: h = SymmetricFunctions(QQ).h()
sage: m = SymmetricFunctions(QQ).m()
sage: p4 = Partitions(4)
sage: matrix([ [e(a).scalar(h(b)) for a in p4] for b in p4])
[ 0  0  0  0  1]
[ 0  0  0  1  4]
[ 0  0  1  2  6]
[ 0  1  2  5 12]
[ 1  4  6 12 24]
sage: matrix([ [h(a).scalar(e(b)) for a in p4] for b in p4])
[ 0  0  0  0  1]
[ 0  0  0  1  4]
```

```
[ 0  0  1  2  6]
[ 0  1  2  5 12]
[ 1  4  6 12 24]
sage: matrix([ [m(a).scalar(e(b)) for a in p4] for b in p4])
[-1  2  1 -3  1]
[ 0  1  0 -2  1]
[ 0  0  1 -2  1]
[ 0  0  0 -1  1]
[ 0  0  0  0  1]
sage: matrix([ [m(a).scalar(h(b)) for a in p4] for b in p4])
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]

sage: p = SymmetricFunctions(QQ).p()
sage: m(p[3,2]).scalar(p[3,2], zee=lambda mu: 2**mu.length())
4
sage: m(p[3,2]).scalar(p[2,2,1], lambda mu: 1)
0
sage: m[3,2].scalar(h[3,2], zee=lambda mu: 2**mu.length())
2/3
```

TESTS:

```
sage: m(1).scalar(h(1))
1
sage: m(0).scalar(h(1))
0
sage: m(1).scalar(h(0))
0
sage: m(0).scalar(h(0))
0
```

Over the integers, too (as long as `zee` is not set):

```
sage: Sym = SymmetricFunctions(ZZ)
sage: m = Sym.m()
sage: m([2]).scalar(m([2]))
2
```

scalar_hl(*x*, *t=None*)

Return the t-deformed standard Hall-Littlewood scalar product of `self` and `x`.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`
- `t` – parameter (default: `None`, in which case `t` is used)

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: sp = a.scalar_t(a); sp
(-t^2 - 1)/(t^5 - 2*t^4 + t^3 - t^2 + 2*t - 1)
sage: sp.parent()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

scalar_jack(*x*, *t=None*)

Return the Jack-scalar product beween `self` and `x`.

This scalar product is defined so that the power sum elements p_μ are orthogonal and $\langle p_\mu, p_\mu \rangle = z_\mu t^{\ell(\mu)}$, where $\ell(\mu)$ denotes the length of μ.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`
- `t` – an optional parameter (default: `None` in which case `t` is used)

EXAMPLES:

```
sage: p = SymmetricFunctions(QQ['t']).power()
sage: matrix([[p(mu).scalar_jack(p(nu)) for nu in Partitions(4)] for mu in Partitions(4)])
[   4*t      0      0      0      0]
[     0  3*t^2      0      0      0]
[     0      0  8*t^2      0      0]
[     0      0      0  4*t^3      0]
[     0      0      0      0 24*t^4]
sage: matrix([[p(mu).scalar_jack(p(nu),2) for nu in Partitions(4)] for mu in Partitions(4)])
[  8   0   0   0   0]
[  0  12   0   0   0]
[  0   0  32   0   0]
[  0   0   0  32   0]
[  0   0   0   0 384]
sage: JQ = SymmetricFunctions(QQ['t'].fraction_field()).jack().Q()
sage: matrix([[JQ(mu).scalar_jack(JQ(nu)) for nu in Partitions(3)] for mu in Partitions(3)])
[(2*t^2 + 3*t + 1)/(6*t^3)                          0                          0]
[                        0     (t + 2)/(2*t^3 + t^2)                          0]
[                        0                          0     6/(t^3 + 3*t^2 + 2*t)]
```

scalar_qt(*x*, *q=None*, *t=None*)

Returns the q,t-deformed standard Hall-Littlewood scalar product of `self` and `x`.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`
- `q`, `t` – parameters (default: `None` in which case `q` and `t` are used)

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: sp = a.scalar_qt(a); factor(sp)
(t - 1)^-3 * (q - 1) * (t^2 + t + 1)^-1 * (q^2*t^2 - q*t^2 + q^2 - 2*q*t + t^2 - q + 1)
sage: sp.parent()
Fraction Field of Multivariate Polynomial Ring in q, t over Rational Field
sage: a.scalar_qt(a,q=0)
(-t^2 - 1)/(t^5 - 2*t^4 + t^3 - t^2 + 2*t - 1)
sage: a.scalar_qt(a,t=0)
-q^3 + 2*q^2 - 2*q + 1
sage: a.scalar_qt(a,5,7) # q=5 and t=7
490/1539
sage: (x,y) = var('x,y')
sage: a.scalar_qt(a,q=x,t=y)
1/3*(x^3 - 1)/(y^3 - 1) + 2/3*(x - 1)^3/(y - 1)^3
sage: Rn = QQ['q','t','y','z'].fraction_field()
sage: (q,t,y,z) = Rn.gens()
sage: Mac = SymmetricFunctions(Rn).macdonald(q=y,t=z)
sage: a = Mac._sym.schur()([2,1])
```

```
sage: factor(Mac.P()(a).scalar_qt(Mac.Q()(a),q,t))
(t - 1)^-3 * (q - 1) * (t^2 + t + 1)^-1 * (q^2*t^2 - q*t^2 + q^2 - 2*q*t + t^2 - q + 1)
sage: factor(Mac.P()(a).scalar_qt(Mac.Q()(a)))
(z - 1)^-3 * (y - 1) * (z^2 + z + 1)^-1 * (y^2*z^2 - y*z^2 + y^2 - 2*y*z + z^2 - y + 1)
```

scalar_t (*x*, *t=None*)

Return the t-deformed standard Hall-Littlewood scalar product of `self` and `x`.

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`
- `t` – parameter (default: `None`, in which case `t` is used)

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: a = s([2,1])
sage: sp = a.scalar_t(a); sp
(-t^2 - 1)/(t^5 - 2*t^4 + t^3 - t^2 + 2*t - 1)
sage: sp.parent()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

skew_by (*x*)

Return the result of skewing `self` by `x`. (Skewing by `x` is the endomorphism (as additive group) of the ring of symmetric functions adjoint to multiplication by `x` with respect to the Hall inner product.)

INPUT:

- `x` – element of the ring of symmetric functions over the same base ring as `self`

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([3,2]).skew_by(s([2]))
s[2, 1] + s[3]
sage: s([3,2]).skew_by(s([1,1,1]))
0
sage: s([3,2,1]).skew_by(s([2,1]))
s[1, 1, 1] + 2*s[2, 1] + s[3]

sage: p = SymmetricFunctions(QQ).powersum()
sage: p([4,3,3,2,2,1]).skew_by(p([2,1]))
4*p[4, 3, 3, 2]
sage: zee = sage.combinat.sf.sfa.zee
sage: zee([4,3,3,2,2,1])/zee([4,3,3,2])
4
sage: s(0).skew_by(s([1]))
0
sage: s(1).skew_by(s([1]))
0
sage: s([]).skew_by(s([]))
s[]
sage: s([]).skew_by(s[1])
0
```

TESTS:

```
sage: f=s[3,2]
sage: f.skew_by([1])
Traceback (most recent call last):
```

```
...
ValueError: x needs to be a symmetric function
```

theta(*a*)

Return the image of `self` under the theta endomorphism which sends p_k to $a \cdot p_k$ for every positive integer k.

In general, this is well-defined outside of the powersum basis only if the base ring is a $\mathbf{Q}$-algebra.

INPUT:

- `a` – an element of the base ring

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s([2,1]).theta(2)
2*s[1, 1, 1] + 6*s[2, 1] + 2*s[3]
sage: p = SymmetricFunctions(QQ).p()
sage: p([2]).theta(2)
2*p[2]
sage: p(0).theta(2)
0
sage: p(1).theta(2)
p[]
```

theta_qt(*q=None*, *t=None*)

Return the image of `self` under the q,t-deformed theta endomorphism which sends p_k to $\frac{1-q^k}{1-t^k} \cdot p_k$ for all positive integers k.

In general, this is well-defined outside of the powersum basis only if the base ring is a $\mathbf{Q}$-algebra.

INPUT:

- `q`, `t` – parameters (default: `None`, in which case 'q' and 't' are used)

EXAMPLES:

```
sage: QQqt = QQ['q,t'].fraction_field()
sage: q,t = QQqt.gens()
sage: p = SymmetricFunctions(QQqt).p()
sage: p([2]).theta_qt(q,t)
((-q^2+1)/(-t^2+1))*p[2]
sage: p([2,1]).theta_qt(q,t)
((q^3-q^2-q+1)/(t^3-t^2-t+1))*p[2, 1]
sage: p(0).theta_qt(q=1,t=3)
0
sage: p([2,1]).theta_qt(q=2,t=3)
3/16*p[2, 1]
sage: s = p.realization_of().schur()
sage: s([3]).theta_qt(q=0)*(1-t)*(1-t^2)*(1-t^3)
t^3*s[1, 1, 1] + (t^2+t)*s[2, 1] + s[3]
sage: p(1).theta_qt()
p[]
```

verschiebung(*n*)

Return the image of the symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the unique algebra endomorphism V of the ring of symmetric functions that satisfies $V(h_r) = h_{r/n}$ for every positive integer r divisible by n, and satisfies

$V(h_r) = 0$ for every positive integer r not divisible by n. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every nonnegative integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(h_r) = h_{r/n}, \quad \mathbf{V}_n(p_r) = np_{r/n}, \quad \mathbf{V}_n(e_r) = (-1)^{r-r/n} e_{r/n}$$

(where h is the complete homogeneous basis, p is the powersum basis, and e is the elementary basis). For every nonnegative integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(h_r) = \mathbf{V}_n(p_r) = \mathbf{V}_n(e_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism. Its name derives from the Verschiebung (German for "shift") endomorphism of the Witt vectors.

The n-th Verschiebung operator is adjoint to the n-th Frobenius operator (see `frobenius()` for its definition) with respect to the Hall scalar product (`scalar()`).

The action of the n-th Verschiebung operator on the Schur basis can also be computed explicitly. The following (probably clumsier than necessary) description can be obtained by solving exercise 7.61 in Stanley's [STA].

Let λ be a partition. Let n be a positive integer. If the n-core of λ is nonempty, then $\mathbf{V}_n(s_\lambda) = 0$. Otherwise, the following method computes $\mathbf{V}_n(s_\lambda)$: Write the partition λ in the form $(\lambda_1, \lambda_2, \ldots, \lambda_{ns})$ for some nonnegative integer s. (If n does not divide the length of λ, then this is achieved by adding trailing zeroes to λ.) Set $\beta_i = \lambda_i + ns - i$ for every $s \in \{1, 2, \ldots, ns\}$. Then, $(\beta_1, \beta_2, \ldots, \beta_{ns})$ is a strictly decreasing sequence of nonnegative integers. Stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $-1 - \beta_i$ modulo n. Let ξ be the sign of the permutation that is used for this sorting. Let ψ be the sign of the permutation that is used to stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $i - 1$ modulo n. (Notice that $\psi = (-1)^{n(n-1)s(s-1)/4}$.) Then, $\mathbf{V}_n(s_\lambda) = \xi\psi \prod_{i=0}^{n-1} s_{\lambda^{(i)}}$, where $(\lambda^{(0)}, \lambda^{(1)}, \ldots, \lambda^{(n-1)})$ is the n-quotient of λ.

INPUT:

- `n` – a positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of symmetric functions) to `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: p = Sym.p()
sage: h = Sym.h()
sage: s = Sym.s()
sage: m = Sym.m()
sage: s[3].verschiebung(2)
0
sage: s[3].verschiebung(3)
s[1]
sage: p[3].verschiebung(3)
3*p[1]
sage: m[3,2,1].verschiebung(3)
-18*m[1, 1] - 3*m[2]
sage: p[3,2,1].verschiebung(3)
0
sage: h[4].verschiebung(2)
h[2]
sage: p[2].verschiebung(2)
2*p[1]
sage: m[3,2,1].verschiebung(6)
12*m[1]
```

The Verschiebung endomorphisms are multiplicative:

```
sage: all( all( s(lam).verschiebung(2) * s(mu).verschiebung(2)
....:           == (s(lam) * s(mu)).verschiebung(2)
....:           for mu in Partitions(4) )
....:       for lam in Partitions(4) )
True
```

Being Hopf algebra endomorphisms, the Verschiebung operators commute with the antipode:

```
sage: all( p(lam).verschiebung(3).antipode()
....:       == p(lam).antipode().verschiebung(3)
....:       for lam in Partitions(6) )
True
```

Testing the adjointness between the Frobenius operators $\mathbf{f}_n$ and the Verschiebung operators $\mathbf{V}_n$:

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.s()
sage: p = Sym.p()
sage: all( all( s(lam).verschiebung(2).scalar(p(mu))
....:           == s(lam).scalar(p(mu).frobenius(2))
....:           for mu in Partitions(3) )
....:       for lam in Partitions(6) )
True
```

class sage.combinat.sf.sfa.**SymmetricFunctionsBases**(*parent_with_realization*)

Bases: sage.categories.realizations.Category_realization_of_parent

The category of bases of the ring of symmetric functions.

INPUT:

- self – a category of bases for the symmetric functions
- base – ring of symmetric functions

TESTS:

```
sage: from sage.combinat.sf.sfa import SymmetricFunctionsBases
sage: Sym = SymmetricFunctions(QQ)
sage: bases = SymmetricFunctionsBases(Sym); bases
Category of bases of Symmetric Functions over Rational Field
sage: Sym.schur() in bases
True
```

class ParentMethods

Eulerian(*n*, *j*, *k=None*)

Return the Eulerian symmetric function $Q_{n,j}$ (with n either an integer or a partition) or $Q_{n,j,k}$ (if the optional argument k is specified) in terms of the basis self.

It is known that the Eulerian quasisymmetric functions are in fact symmetric functions [SW2010]. For more information, see QuasiSymmetricFunctions.Fundamental.Eulerian(), which accepts the same syntax as this method.

INPUT:

- n – the nonnegative integer n or a partition
- j – the number of excedances
- k – (optional) if specified, determines the number of fixed points of the permutations which are being summed over

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.m()
sage: m.Eulerian(3, 1)
4*m[1, 1, 1] + 3*m[2, 1] + 2*m[3]
sage: h = Sym.h()
sage: h.Eulerian(4, 2)
h[2, 2] + h[3, 1] + h[4]
sage: s = Sym.s()
sage: s.Eulerian(5, 2)
s[2, 2, 1] + s[3, 1, 1] + 5*s[3, 2] + 6*s[4, 1] + 6*s[5]
sage: s.Eulerian([2,2,1], 2)
s[2, 2, 1] + s[3, 2] + s[4, 1] + s[5]
sage: s.Eulerian(5, 2, 2)
s[3, 2] + s[4, 1] + s[5]
```

We check Equation (5.4) in [SW2010]:

```
sage: h.Eulerian([6], 3)
h[3, 2, 1] - h[4, 1, 1] + 2*h[4, 2] + h[5, 1]
sage: s.Eulerian([6], 3)
s[3, 2, 1] + s[3, 3] + 3*s[4, 2] + 3*s[5, 1] + 3*s[6]
```

carlitz_shareshian_wachs(*n*, *d*, *s*, *comparison=None*)

Return the Carlitz-Shareshian-Wachs symmetric function $X_{n,d,s}$ (if `comparison` is `None`), or $U_{n,d,s}$ (if `comparison` is `-1`), or $V_{n,d,s}$ (if `comparison` is `0`), or $W_{n,d,s}$ (if `comparison` is `1`) written in the basis `self`. These functions are defined below.

The Carlitz-Shareshian-Wachs symmetric functions have been introduced in [GriRei2014], Exercise 2.84, as refinements of a certain particular case of chromatic quasisymmetric functions defined by Shareshian and Wachs. Their definitions are as follows:

Let n, d and s be three nonnegative integers. Let $W(n,d,s)$ denote the set of all n-tuples $(w_1, w_2, \ldots, w_n)$ of positive integers having the property that there exist precisely d elements i of $\{1, 2, \ldots, n-1\}$ satisfying $w_i > w_{i+1}$, and precisely s elements i of $\{1, 2, \ldots, n-1\}$ satisfying $w_i = w_{i+1}$. For every $w = (w_1, w_2, \ldots, w_n) \in W(n,d,s)$, let x_w be the monomial $x_{w_1} x_{w_2} \cdots x_{w_n}$. We then define the power series $X_{n,d,s}$ by

$$X_{n,d,s} = \sum_{w \in W(n,d,s)} x_w.$$

This is a symmetric function (according to [GriRei2014], Exercise 2.84(b)), and for $s = 0$ equals the t^d-coefficient of the descent enumerator of Smirnov words of length n (an example of a chromatic quasisymmetric function which happens to be symmetric – see [ShaWach2014], Example 2.5).

Assume that $n > 0$. Then, we can define three further power series as follows:

$$U_{n,d,s} = \sum_{w_1 < w_n} x_w; \qquad V_{n,d,s} = \sum_{w_1 = w_n} x_w; \qquad W_{n,d,s} = \sum_{w_1 > w_n} x_w,$$

where all three sums range over $w = (w_1, w_2, \ldots, w_n) \in W(n,d,s)$. These three power series $U_{n,d,s}$, $V_{n,d,s}$ and $W_{n,d,s}$ are symmetric functions as well ([GriRei2014], Exercise 2.84(c)). Their sum is $X_{n,d,s}$.

REFERENCES:

INPUT:

- `n` – a nonnegative integer
- `d` – a nonnegative integer
- `s` – a nonnegative integer

•comparison (default: None) – a variable which can take the forms None, -1, 0 and 1

OUTPUT:

The Carlitz-Shareshian-Wachs symmetric function $X_{n,d,s}$ (if comparison is None), or $U_{n,d,s}$ (if comparison is -1), or $V_{n,d,s}$ (if comparison is 0), or $W_{n,d,s}$ (if comparison is 1) written in the basis self.

EXAMPLES:

The power series $X_{n,d,s}$:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: m = Sym.m()
sage: m.carlitz_shareshian_wachs(3, 2, 1)
0
sage: m.carlitz_shareshian_wachs(3, 1, 1)
m[2, 1]
sage: m.carlitz_shareshian_wachs(3, 2, 0)
m[1, 1, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 2)
m[3]
sage: m.carlitz_shareshian_wachs(3, 1, 0)
4*m[1, 1, 1] + m[2, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 1)
m[2, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 0)
m[1, 1, 1]
sage: m.carlitz_shareshian_wachs(5, 2, 2)
m[2, 2, 1] + m[3, 1, 1]
sage: m.carlitz_shareshian_wachs(1, 0, 0)
m[1]
sage: m.carlitz_shareshian_wachs(0, 0, 0)
m[]
```

The power series $U_{n,d,s}$:

```
sage: m.carlitz_shareshian_wachs(3, 2, 1, comparison=-1)
0
sage: m.carlitz_shareshian_wachs(3, 1, 1, comparison=-1)
0
sage: m.carlitz_shareshian_wachs(3, 2, 0, comparison=-1)
0
sage: m.carlitz_shareshian_wachs(3, 0, 2, comparison=-1)
0
sage: m.carlitz_shareshian_wachs(3, 1, 0, comparison=-1)
2*m[1, 1, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 1, comparison=-1)
m[2, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 0, comparison=-1)
m[1, 1, 1]
sage: m.carlitz_shareshian_wachs(5, 2, 2, comparison=-1)
0
sage: m.carlitz_shareshian_wachs(4, 2, 0, comparison=-1)
3*m[1, 1, 1, 1]
sage: m.carlitz_shareshian_wachs(1, 0, 0, comparison=-1)
0
```

The power series $V_{n,d,s}$:

```
sage: m.carlitz_shareshian_wachs(3, 2, 1, comparison=0)
0
sage: m.carlitz_shareshian_wachs(3, 1, 1, comparison=0)
```

```
0
sage: m.carlitz_shareshian_wachs(3, 2, 0, comparison=0)
0
sage: m.carlitz_shareshian_wachs(3, 0, 2, comparison=0)
m[3]
sage: m.carlitz_shareshian_wachs(3, 1, 0, comparison=0)
m[2, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 1, comparison=0)
0
sage: m.carlitz_shareshian_wachs(3, 0, 0, comparison=0)
0
sage: m.carlitz_shareshian_wachs(5, 2, 2, comparison=0)
0
sage: m.carlitz_shareshian_wachs(4, 2, 0, comparison=0)
m[2, 1, 1]
sage: m.carlitz_shareshian_wachs(1, 0, 0, comparison=0)
m[1]
```

The power series $W_{n,d,s}$:

```
sage: m.carlitz_shareshian_wachs(3, 2, 1, comparison=1)
0
sage: m.carlitz_shareshian_wachs(3, 1, 1, comparison=1)
m[2, 1]
sage: m.carlitz_shareshian_wachs(3, 2, 0, comparison=1)
m[1, 1, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 2, comparison=1)
0
sage: m.carlitz_shareshian_wachs(3, 1, 0, comparison=1)
2*m[1, 1, 1]
sage: m.carlitz_shareshian_wachs(3, 0, 1, comparison=1)
0
sage: m.carlitz_shareshian_wachs(3, 0, 0, comparison=1)
0
sage: m.carlitz_shareshian_wachs(5, 2, 2, comparison=1)
m[2, 2, 1] + m[3, 1, 1]
sage: m.carlitz_shareshian_wachs(4, 2, 0, comparison=1)
8*m[1, 1, 1, 1] + 2*m[2, 1, 1] + m[2, 2]
sage: m.carlitz_shareshian_wachs(1, 0, 0, comparison=1)
0
```

TESTS:

This works fine over other base rings:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P()
sage: m = Sym.m()
sage: m.carlitz_shareshian_wachs(4, 1, 1)
4*m[2, 1, 1] + 2*m[2, 2] + 2*m[3, 1]
sage: P.carlitz_shareshian_wachs(4, 1, 1) == P(m.carlitz_shareshian_wachs(4, 1, 1))
True
```

corresponding_basis_over(*R*)

Return the realization of symmetric functions corresponding to `self` but over the base ring `R`. Only works when `self` is one of the classical bases, not one of the q, t-dependent ones. In the latter case, `None` is returned instead.

INPUT:

- `R` – a commutative ring

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.monomial()
sage: m.corresponding_basis_over(ZZ)
Symmetric Functions over Integer Ring in the monomial basis

sage: Sym = SymmetricFunctions(CyclotomicField())
sage: s = Sym.schur()
sage: s.corresponding_basis_over(Integers(13))
Symmetric Functions over Ring of integers modulo 13 in the Schur basis

sage: P = ZZ['q','t']
sage: Sym = SymmetricFunctions(P)
sage: mj = Sym.macdonald().J()
sage: mj.corresponding_basis_over(Integers(13))
```

TESTS:

Let's check that this handles each of the bases properly:

```
sage: P = QQ['q','t']
sage: Sym = SymmetricFunctions(P)
sage: Q = CyclotomicField()['q','t']
sage: Sym.s().corresponding_basis_over(CyclotomicField())
Symmetric Functions over Universal Cyclotomic Field in the Schur basis
sage: Sym.p().corresponding_basis_over(CyclotomicField())
Symmetric Functions over Universal Cyclotomic Field in the powersum basis
sage: Sym.m().corresponding_basis_over(CyclotomicField())
Symmetric Functions over Universal Cyclotomic Field in the monomial basis
sage: Sym.e().corresponding_basis_over(CyclotomicField())
Symmetric Functions over Universal Cyclotomic Field in the elementary basis
sage: Sym.h().corresponding_basis_over(CyclotomicField())
Symmetric Functions over Universal Cyclotomic Field in the homogeneous basis
sage: Sym.f().corresponding_basis_over(CyclotomicField())
Symmetric Functions over Universal Cyclotomic Field in the forgotten basis
sage: Sym.w().corresponding_basis_over(CyclotomicField())
Symmetric Functions over Universal Cyclotomic Field in the Witt basis
sage: Sym.macdonald().P().corresponding_basis_over(CyclotomicField())
sage: Sym.macdonald().Q().corresponding_basis_over(CyclotomicField())
sage: Sym.macdonald().J().corresponding_basis_over(CyclotomicField())
sage: Sym.macdonald().H().corresponding_basis_over(CyclotomicField())
sage: Sym.macdonald().Ht().corresponding_basis_over(CyclotomicField())
sage: Sym.macdonald().S().corresponding_basis_over(CyclotomicField())
sage: Sym.macdonald(q=1).S().corresponding_basis_over(CyclotomicField())
sage: Sym.macdonald(q=1,t=3).P().corresponding_basis_over(CyclotomicField())
sage: Sym.hall_littlewood().P().corresponding_basis_over(CyclotomicField())
sage: Sym.hall_littlewood().Q().corresponding_basis_over(CyclotomicField())
sage: Sym.hall_littlewood().Qp().corresponding_basis_over(CyclotomicField())
sage: Sym.hall_littlewood(t=1).P().corresponding_basis_over(CyclotomicField())
sage: Sym.jack().J().corresponding_basis_over(CyclotomicField())
sage: Sym.jack().P().corresponding_basis_over(CyclotomicField())
sage: Sym.jack().Q().corresponding_basis_over(CyclotomicField())
sage: Sym.jack().Qp().corresponding_basis_over(CyclotomicField())
sage: Sym.jack(t=1).J().corresponding_basis_over(CyclotomicField())
sage: Sym.zonal().corresponding_basis_over(CyclotomicField())
Symmetric Functions over Universal Cyclotomic Field in the zonal basis
sage: Sym.llt(3).hspin().corresponding_basis_over(CyclotomicField())
sage: Sym.llt(3).hcospin().corresponding_basis_over(CyclotomicField())
sage: Sym.llt(3, t=1).hspin().corresponding_basis_over(CyclotomicField())
sage: Sym.llt(3, t=1).hcospin().corresponding_basis_over(CyclotomicField())
```

Todo

This function is an ugly hack using strings. It should be rewritten as soon as the bases of SymmetricFunctions are put on a more robust and systematic footing.

degree_on_basis(*b*)

Return the degree of the basis element indexed by b.

INPUT:

- self – a basis of the symmetric functions
- b – a partition

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['q,t'].fraction_field())
sage: m = Sym.monomial()
sage: m.degree_on_basis(Partition([3,2]))
5
sage: P = Sym.macdonald().P()
sage: P.degree_on_basis(Partition([]))
0
```

gessel_reutenauer(*lam*)

Return the Gessel-Reutenauer symmetric function corresponding to the partition lam written in the basis self.

Let λ be a partition. The *Gessel-Reutenauer symmetric function* $\mathbf{GR}_\lambda$ corresponding to λ is the symmetric function denoted L_λ in [GR1993] and in Exercise 7.89 of [STA]. It can be defined in several ways:

- It is the sum of the monomials $\mathbf{x}_w$ over all words w over the alphabet $\{1, 2, 3, \ldots\}$ which have CFL type λ. Here, the monomial $\mathbf{x}_w$ for a word $w = (w_1, w_2, \ldots, w_k)$ is defined as $x_{w_1} x_{w_2} \cdots x_{w_k}$, and the *CFL type* of a word w is defined as the partition obtained by sorting (in decreasing order) the lengths of the factors in the Lyndon factorization (lyndon_factorization()) of w. The fact that this power series $\mathbf{GR}_\lambda$ is symmetric is not obvious.
- It is the sum of the fundamental quasisymmetric functions $F_{\mathrm{Des}\,\sigma}$ over all permutations σ which have cycle type λ. See sage.combinat.ncsf_qsym.qsym.QuasiSymmetricFunctions.Fundamental for the definition of fundamental quasisymmetric functions, and cycle_type() for that of cycle type. For a permutation σ, we use $\mathrm{Des}\,\sigma$ to denote the descent composition (descents_composition()) of σ. Again, this definition makes the symmetry of $\mathbf{GR}_\lambda$ far from obvious.
- For every positive integer n, we have

$$\mathbf{GR}_{(n)} = \frac{1}{n} \sum_{d|n} \mu(d) p_d^{n/d},$$

where p_d denotes the d-th power-sum symmetric function. This $\mathbf{GR}_{(n)}$ is also denoted by L_n. Now, $\mathbf{GR}_\lambda$ is defined as the product:

$$h_{m_1}[L_1] \cdot h_{m_2}[L_2] \cdot h_{m_3}[L_3] \cdots,$$

where m_i denotes the multiplicity of the part i in λ, and where the square brackets stand for plethysm (plethysm()). This definition makes the symmetry (but not the integrality!) of $\mathbf{GR}_\lambda$ obvious.

The equivalences of these three definitions are proven in [GR1993] Sections 2-3.

INPUT:

- `lam` – a partition or a positive integer (in the latter case, it is understood to mean the partition `[lam]`)

OUTPUT:

The Gessel-Reutenauer symmetric function $\mathbf{GR}_\lambda$, where λ is `lam`, expanded in the basis `self`.

REFERENCES:

EXAMPLES:

The first few values of $\mathbf{GR}_{(n)} = L_n$:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: h = Sym.h()
sage: h.gessel_reutenauer(1)
h[1]
sage: h.gessel_reutenauer(2)
h[1, 1] - h[2]
sage: h.gessel_reutenauer(3)
h[2, 1] - h[3]
sage: h.gessel_reutenauer(4)
h[2, 1, 1] - h[2, 2]
sage: h.gessel_reutenauer(5)
h[2, 1, 1, 1] - h[2, 2, 1] - h[3, 1, 1] + h[3, 2] + h[4, 1] - h[5]
sage: h.gessel_reutenauer(6)
h[2, 1, 1, 1, 1] - h[2, 2, 1, 1] - h[2, 2, 2]
 - 2*h[3, 1, 1, 1] + 5*h[3, 2, 1] - 2*h[3, 3] + h[4, 1, 1]
 - h[4, 2] - h[5, 1] + h[6]
```

Gessel-Reutenauer functions indexed by partitions:

```
sage: h.gessel_reutenauer([2, 1])
h[1, 1, 1] - h[2, 1]
sage: h.gessel_reutenauer([2, 2])
h[1, 1, 1, 1] - 3*h[2, 1, 1] + 2*h[2, 2] + h[3, 1] - h[4]
```

The Gessel-Reutenauer functions are Schur-positive:

```
sage: s = Sym.s()
sage: s.gessel_reutenauer([2, 1])
s[1, 1, 1] + s[2, 1]
sage: s.gessel_reutenauer([2, 2, 1])
s[1, 1, 1, 1, 1] + s[2, 1, 1, 1] + s[2, 2, 1] + s[3, 2]
```

They do not form a basis, as the following example (from [GR1993] p. 201) shows:

```
sage: s.gessel_reutenauer([4]) == s.gessel_reutenauer([2, 1, 1])
True
```

Of the above three equivalent definitions of $\mathbf{GR}_\lambda$, we use the third one for computations. Let us check that the second one gives the same results:

```
sage: QSym = QuasiSymmetricFunctions(ZZ)
sage: F = QSym.F() # fundamental basis
sage: def GR_def2(lam): # '\mathbf{GR}_\lambda'
....:     n = lam.size()
....:     r = F.sum_of_monomials([sigma.descents_composition()
....:                             for sigma in Permutations(n)
....:                             if sigma.cycle_type() == lam])
....:     return r.to_symmetric_function()
sage: all( GR_def2(lam) == h.gessel_reutenauer(lam)
....:      for n in range(5) for lam in Partitions(n) )
True
```

And the first one, too (assuming symmetry):

```
sage: m = Sym.m()
sage: def GR_def1(lam): # '\mathbf{GR}_\lambda'
....:     n = lam.size()
....:     Permus_mset = sage.combinat.permutation.Permutations_mset
....:     def coeff_of_m_mu_in_result(mu):
....:         words_to_check = Permus_mset([i for (i, l) in enumerate(mu)
....:                                       for _ in range(l)])
....:         return sum((1 for w in words_to_check if
....:                     Partition(list(reversed(sorted([len(v) for v in Word(w).lyndon_
....:                     == lam))
....:     r = m.sum_of_terms([(mu, coeff_of_m_mu_in_result(mu))
....:                         for mu in Partitions(n)],
....:                        distinct=True)
....:     return r
sage: all( GR_def1(lam) == h.gessel_reutenauer(lam)
....:      for n in range(5) for lam in Partitions(n) )
True
```

TESTS:

This works fine over other base rings:

```
sage: Sym = SymmetricFunctions(FractionField(QQ['q','t']))
sage: P = Sym.macdonald().P()
sage: h = Sym.h()
sage: P.gessel_reutenauer(3) == P(h.gessel_reutenauer(3))
True
```

Note: The currently existing implementation of this function is technically unsatisfactory. It distinguishes the case when the base ring is a $\mathbf{Q}$-algebra from the case where it isn't. In the latter, it does a computation using universal coefficients, again distinguishing the case when it is able to compute the "corresponding" basis of the symmetric function algebra over $\mathbf{Q}$ (using the `corresponding_basis_over` hack) from the case when it isn't (in which case it transforms everything into the Schur basis, which is slow).

---

is_commutative()

Returns whether this symmetric function algebra is commutative.

INPUT:

- `self` – a basis of the symmetric functions

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s.is_commutative()
True
```

is_field(*proof=True*)

Return whether `self` is a field. (It is not.)

INPUT:

- `self` – a basis of the symmetric functions
- `proof` – an optional argument (default value: `True`)

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s.is_field()
False
```

is_integral_domain(*proof=True*)

Return whether `self` is an integral domain. (It is if and only if the base ring is an integral domain.)

INPUT:

- self – a basis of the symmetric functions
- proof – an optional argument (default value: True)

EXAMPLES:

```
sage: s = SymmetricFunctions(QQ).s()
sage: s.is_integral_domain()
True
```

The following doctest is disabled pending trac ticket #15475:

```
sage: s = SymmetricFunctions(Zmod(14)).s() # not tested
sage: s.is_integral_domain() # not tested
False
```

one_basis()

Returns the empty partition, as per AlgebrasWithBasis.ParentMethods.one_basis

INPUT:

- self – a basis of the ring of symmetric functions

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ['t'].fraction_field())
sage: s = Sym.s()
sage: s.one_basis()
[]
sage: Q = Sym.hall_littlewood().Q()
sage: Q.one_basis()
[]
```

Todo

generalize to Modules.Graded.Connected.ParentMethods

---

skew_schur(*x*)

Return the skew Schur function indexed by x in self.

INPUT:

- x – a skew partition

EXAMPLES:

```
sage: sp = SkewPartition([[5,3,3,1], [3,2,1]])
sage: s = SymmetricFunctions(QQ).s()
sage: s.skew_schur(sp)
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 3*s[3, 2, 1]
 + s[3, 3] + 2*s[4, 1, 1] + 2*s[4, 2] + s[5, 1]

sage: e = SymmetricFunctions(QQ).e()
sage: ess = e.skew_schur(sp); ess
e[2, 1, 1, 1, 1] - e[2, 2, 1, 1] - e[3, 1, 1, 1] + e[3, 2, 1]
sage: ess == e(s.skew_schur(sp))
True
```

TESTS:

```
sage: s.skew_schur([[2,1], [1]])
s[1, 1] + s[2]

sage: s.skew_schur([[2,1], [3]])
Traceback (most recent call last):
...
ValueError: not a valid skew partition
```

SymmetricFunctionsBases.**super_categories**()
The super categories of self.

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import SymmetricFunctionsBases
sage: Sym = SymmetricFunctions(QQ)
sage: bases = SymmetricFunctionsBases(Sym)
sage: bases.super_categories()
[Category of realizations of Symmetric Functions over Rational Field,
 Category of commutative hopf algebras with basis over Rational Field,
 Join of Category of realizations of hopf algebras over Rational Field
     and Category of graded algebras over Rational Field]
```

sage.combinat.sf.sfa.**is_SymmetricFunction**(*x*)
Checks whether x is a symmetric function.

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import is_SymmetricFunction
sage: s = SymmetricFunctions(QQ).s()
sage: is_SymmetricFunction(2)
False
sage: is_SymmetricFunction(s(2))
True
sage: is_SymmetricFunction(s([2,1]))
True
```

sage.combinat.sf.sfa.**is_SymmetricFunctionAlgebra**(*x*)
Checks whether x is a symmetric function algebra.

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import is_SymmetricFunctionAlgebra
sage: is_SymmetricFunctionAlgebra(5)
False
sage: is_SymmetricFunctionAlgebra(ZZ)
False
sage: is_SymmetricFunctionAlgebra(SymmetricFunctions(ZZ).schur())
True
sage: is_SymmetricFunctionAlgebra(SymmetricFunctions(QQ).e())
True
sage: is_SymmetricFunctionAlgebra(SymmetricFunctions(QQ).macdonald(q=1,t=1).P())
True
sage: is_SymmetricFunctionAlgebra(SymmetricFunctions(FractionField(QQ['q','t'])).macdonald().P()
True
```

sage.combinat.sf.sfa.**zee**(*part*)
Return the size of the centralizer of any permutation of cycle type part.

Note that the size of the centralizer is the inner product between p(part) and itself, where p is the power-sum symmetric functions.

INPUT:

- part – an integer partition (for example, [2,1,1])

OUTPUT:

- the integer $\prod_i i^{m_i(part)} m_i(part)!$ where $m_i(part)$ is the number of parts in the partition part equal to i

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import zee
sage: zee([2,1,1])
4
```

5.1.269 Witt symmetric functions

class sage.combinat.sf.witt.**SymmetricFunctionAlgebra_witt**(*Sym*, *coerce_h=True*, *coerce_e=False*, *coerce_p=False*)

Bases: sage.combinat.sf.multiplicative.SymmetricFunctionAlgebra_multiplicative

The Witt symmetric function basis (or Witt basis, to be short).

The Witt basis of the ring of symmetric functions is denoted by (x_λ) in [HazWitt1], section 9.63, and by (q_λ) in [DoranIV1996]. We will denote this basis by (w_λ) (which is precisely how it is denoted in [GriRei2014], Exercise 2.76(d)). It is a multiplicative basis (meaning that $w_\emptyset = 1$ and that every partition λ satisfies $w_\lambda = w_{\lambda_1} w_{\lambda_2} w_{\lambda_3} \cdots$, where w_i means $w_{(i)}$ for every nonnegative integer i).

This basis can be defined in various ways. Probably the most well-known one is using the equation

$$\prod_{d=1}^{\infty} (1 - w_d t^d)^{-1} = \sum_{n=0}^{\infty} h_n t^n$$

where t is a formal variable and h_n are the complete homogeneous symmetric functions, extended to 0 by $h_0 = 1$. This equation allows one to uniquely determine the functions $w_1, w_2, w_3, \ldots$ by recursion; one consequently extends the definition to all w_λ by requiring multiplicativity.

A way to rewrite the above equation without power series is:

$$h_n = \sum_{\lambda \vdash n} w_\lambda$$

for all nonnegative integers n, where $\lambda \vdash n$ means that λ is a partition of n.

A similar equation (which is easily seen to be equivalent to the former) is

$$e_n = \sum_{\lambda} (-1)^{n-\ell(\lambda)} w_\lambda,$$

with the sum running only over *strict* partitions λ of n this time. This equation can also be used to recursively define the w_n. Furthermore, every positive integer n satisfies

$$p_n = \sum_{d \mid n} d w_d^{n/d},$$

and this can be used to define the w_n recursively over any ring which is torsion-free as a $\mathbf{Z}$-module. While these equations all yield easy formulas for classical bases of the ring of symmetric functions in terms of the Witt symmetric functions, it seems difficult to obtain explicit formulas in the other direction.

The Witt symmetric functions owe their name to the fact that the ring of symmetric functions can be viewed as the coordinate ring of the group scheme of Witt vectors, and the Witt symmetric functions are the functions that send a Witt vector to its components (whereas the powersum symmetric functions send a Witt vector to its ghost components). Details can be found in [HazWitt1] or section 3.2 of [BorWi2004].

INPUT:

- Sym – an instance of the ring of the symmetric functions.

- `coerce_h` – (default: `True`) a boolean that determines whether the transition maps between the Witt basis and the complete homogeneous basis will be cached and registered as coercions.
- `coerce_e` – (default: `False`) a boolean that determines whether the transition maps between the Witt basis and the elementary symmetric basis will be cached and registered as coercions.
- `coerce_p` – (default: `False`) a boolean that determines whether the transition maps between the Witt basis and the powersum basis will be cached and registered as coercions (or conversions, if the base ring is not a $\mathbf{Q}$-algebra).

REFERENCES:

EXAMPLES:

Here are the first few Witt symmetric functions, in various bases:

```
sage: Sym = SymmetricFunctions(QQ)
sage: w = Sym.w()
sage: e = Sym.e()
sage: h = Sym.h()
sage: p = Sym.p()
sage: s = Sym.s()
sage: m = Sym.m()

sage: p(w([1]))
p[1]
sage: m(w([1]))
m[1]
sage: e(w([1]))
e[1]
sage: h(w([1]))
h[1]
sage: s(w([1]))
s[1]

sage: p(w([2]))
-1/2*p[1, 1] + 1/2*p[2]
sage: m(w([2]))
-m[1, 1]
sage: e(w([2]))
-e[2]
sage: h(w([2]))
-h[1, 1] + h[2]
sage: s(w([2]))
-s[1, 1]

sage: p(w([3]))
-1/3*p[1, 1, 1] + 1/3*p[3]
sage: m(w([3]))
-2*m[1, 1, 1] - m[2, 1]
sage: e(w([3]))
-e[2, 1] + e[3]
sage: h(w([3]))
-h[2, 1] + h[3]
sage: s(w([3]))
-s[2, 1]

sage: Sym = SymmetricFunctions(ZZ)
sage: w = Sym.w()
sage: e = Sym.e()
sage: h = Sym.h()
```

```
sage: s = Sym.s()
sage: m = Sym.m()
sage: p = Sym.p()
sage: m(w([4]))
-9*m[1, 1, 1, 1] - 4*m[2, 1, 1] - 2*m[2, 2] - m[3, 1]
sage: e(w([4]))
-e[2, 1, 1] + e[3, 1] - e[4]
sage: h(w([4]))
-h[1, 1, 1, 1] + 2*h[2, 1, 1] - h[2, 2] - h[3, 1] + h[4]
sage: s(w([4]))
-s[1, 1, 1, 1] - s[2, 1, 1] - s[2, 2] - s[3, 1]
```

Some examples of conversions the other way:

```
sage: w(h[3])
w[1, 1, 1] + w[2, 1] + w[3]
sage: w(e[3])
-w[2, 1] + w[3]
sage: w(m[2,1])
2*w[2, 1] - 3*w[3]
sage: w(p[3])
w[1, 1, 1] + 3*w[3]
```

Antipodes:

```
sage: w([1]).antipode()
-w[1]
sage: w([2]).antipode()
-w[1, 1] - w[2]
```

The following holds for all odd i and is easily proven by induction:

```
sage: all( w([i]).antipode() == -w([i]) for i in range(1, 10, 2) )
True
```

The Witt basis does not allow for simple expressions for comultiplication and antipode in general (this is related to the fact that the sum of two Witt vectors isn't easily described in terms of the components). Therefore, most computations with Witt symmetric functions, as well as conversions and coercions, pass through the complete homogeneous symmetric functions by default. However, one can also use the elementary symmetric functions instead, or (if the base ring is a $\mathbf{Q}$-algebra) the powersum symmetric functions. This is what the optional keyword variables `coerce_e`, `coerce_h` and `coerce_p` are for. These variables do not affect the results of the (non-underscored) methods of `self`, but they affect the speed of the computations (the more of these variables are set to `True`, the faster these are) and the size of the cache (the more of these variables are set to `True`, the bigger the cache). Let us check that the results are the same no matter to what the variables are set:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.p()
sage: wh = Sym.w()
sage: we = Sym.w(coerce_h=False, coerce_e=True)
sage: wp = Sym.w(coerce_h=False, coerce_p=True)
sage: all( p(wh(lam)) == p(we(lam)) == p(wp(lam)) for lam in Partitions(4) )
True
sage: all ( wh(p(lam)).monomial_coefficients()
....:       == we(p(lam)).monomial_coefficients()
....:       == wp(p(lam)).monomial_coefficients() for lam in Partitions(4) )
True
```

TESTS:

Let us check that all the above computations work with a non-default setting as well:

```
sage: Sym = SymmetricFunctions(QQ)
sage: w = Sym.w(coerce_h=False, coerce_p=True)
sage: e = Sym.e()
sage: h = Sym.h()
sage: p = Sym.p()
sage: s = Sym.s()
sage: m = Sym.m()

sage: p(w([1]))
p[1]
sage: m(w([1]))
m[1]
sage: e(w([1]))
e[1]
sage: h(w([1]))
h[1]
sage: s(w([1]))
s[1]

sage: p(w([2]))
-1/2*p[1, 1] + 1/2*p[2]
sage: m(w([2]))
-m[1, 1]
sage: e(w([2]))
-e[2]
sage: h(w([2]))
-h[1, 1] + h[2]
sage: s(w([2]))
-s[1, 1]

sage: p(w([3]))
-1/3*p[1, 1, 1] + 1/3*p[3]
sage: m(w([3]))
-2*m[1, 1, 1] - m[2, 1]
sage: e(w([3]))
-e[2, 1] + e[3]
sage: h(w([3]))
-h[2, 1] + h[3]
sage: s(w([3]))
-s[2, 1]

sage: Sym = SymmetricFunctions(ZZ)
sage: w = Sym.w()
sage: e = Sym.e()
sage: h = Sym.h()
sage: s = Sym.s()
sage: m = Sym.m()
sage: p = Sym.p()
sage: m(w([4]))
-9*m[1, 1, 1, 1] - 4*m[2, 1, 1] - 2*m[2, 2] - m[3, 1]
sage: e(w([4]))
-e[2, 1, 1] + e[3, 1] - e[4]
sage: h(w([4]))
-h[1, 1, 1, 1] + 2*h[2, 1, 1] - h[2, 2] - h[3, 1] + h[4]
sage: s(w([4]))
-s[1, 1, 1, 1] - s[2, 1, 1] - s[2, 2] - s[3, 1]
```

```
sage: w(h[3])
w[1, 1, 1] + w[2, 1] + w[3]
sage: w(e[3])
-w[2, 1] + w[3]
sage: w(m[2,1])
2*w[2, 1] - 3*w[3]
sage: w(p[3])
w[1, 1, 1] + 3*w[3]

sage: w([1]).antipode()
-w[1]
sage: w([2]).antipode()
-w[1, 1] - w[2]
sage: all( w([i]).antipode() == -w([i]) for i in range(1, 10, 2) )
True
```

Another non-default setting:

```
sage: Sym = SymmetricFunctions(QQ)
sage: w = Sym.w(coerce_h=False, coerce_e=True)
sage: e = Sym.e()
sage: h = Sym.h()
sage: p = Sym.p()
sage: s = Sym.s()
sage: m = Sym.m()

sage: p(w([1]))
p[1]
sage: m(w([1]))
m[1]
sage: e(w([1]))
e[1]
sage: h(w([1]))
h[1]
sage: s(w([1]))
s[1]

sage: p(w([2]))
-1/2*p[1, 1] + 1/2*p[2]
sage: m(w([2]))
-m[1, 1]
sage: e(w([2]))
-e[2]
sage: h(w([2]))
-h[1, 1] + h[2]
sage: s(w([2]))
-s[1, 1]

sage: p(w([3]))
-1/3*p[1, 1, 1] + 1/3*p[3]
sage: m(w([3]))
-2*m[1, 1, 1] - m[2, 1]
sage: e(w([3]))
-e[2, 1] + e[3]
sage: h(w([3]))
-h[2, 1] + h[3]
sage: s(w([3]))
-s[2, 1]
```

```
sage: Sym = SymmetricFunctions(ZZ)
sage: w = Sym.w()
sage: e = Sym.e()
sage: h = Sym.h()
sage: s = Sym.s()
sage: m = Sym.m()
sage: p = Sym.p()
sage: m(w([4]))
-9*m[1, 1, 1, 1] - 4*m[2, 1, 1] - 2*m[2, 2] - m[3, 1]
sage: e(w([4]))
-e[2, 1, 1] + e[3, 1] - e[4]
sage: h(w([4]))
-h[1, 1, 1, 1] + 2*h[2, 1, 1] - h[2, 2] - h[3, 1] + h[4]
sage: s(w([4]))
-s[1, 1, 1, 1] - s[2, 1, 1] - s[2, 2] - s[3, 1]
sage: [type(coeff) for a, coeff in h(w([4]))]
[<type 'sage.rings.integer.Integer'>,
 <type 'sage.rings.integer.Integer'>,
 <type 'sage.rings.integer.Integer'>,
 <type 'sage.rings.integer.Integer'>,
 <type 'sage.rings.integer.Integer'>]

sage: w(h[3])
w[1, 1, 1] + w[2, 1] + w[3]
sage: w(e[3])
-w[2, 1] + w[3]
sage: w(m[2,1])
2*w[2, 1] - 3*w[3]
sage: w(p[3])
w[1, 1, 1] + 3*w[3]

sage: w([1]).antipode()
-w[1]
sage: w([2]).antipode()
-w[1, 1] - w[2]
sage: all( w([i]).antipode() == -w([i]) for i in range(1, 10, 2) )
....:      #this holds for all odd i and is easily proven by induction
True
```

coproduct(*elt*)

Return the coproduct of the element `elt`.

INPUT:

- `elt` – a symmetric function written in this basis

OUTPUT:

- The coproduct acting on `elt`; the result is an element of the tensor squared of the basis `self`

EXAMPLES:

```
sage: w = SymmetricFunctions(QQ).w()
sage: w[2].coproduct()
w[] # w[2] - w[1] # w[1] + w[2] # w[]
sage: w.coproduct(w[2])
w[] # w[2] - w[1] # w[1] + w[2] # w[]
sage: w[2,1].coproduct()
w[] # w[2, 1] - w[1] # w[1, 1] + w[1] # w[2] - w[1, 1] # w[1] + w[2] # w[1] + w[2, 1] # w[]
sage: w.coproduct(w[2,1])
```

```
w[] # w[2, 1] - w[1] # w[1, 1] + w[1] # w[2] - w[1, 1] # w[1] + w[2] # w[1] + w[2, 1] # w[]
```

TESTS:

The same, but with other settings:

```
sage: w = SymmetricFunctions(QQ).w(coerce_h=False, coerce_e=True)
sage: w[2].coproduct()
w[] # w[2] - w[1] # w[1] + w[2] # w[]
sage: w.coproduct(w[2])
w[] # w[2] - w[1] # w[1] + w[2] # w[]
sage: w[2,1].coproduct()
w[] # w[2, 1] - w[1] # w[1, 1] + w[1] # w[2] - w[1, 1] # w[1] + w[2] # w[1] + w[2, 1] # w[]
sage: w.coproduct(w[2,1])
w[] # w[2, 1] - w[1] # w[1, 1] + w[1] # w[2] - w[1, 1] # w[1] + w[2] # w[1] + w[2, 1] # w[]

sage: w = SymmetricFunctions(QQ).w(coerce_h=False, coerce_p=True)
sage: w[2].coproduct()
w[] # w[2] - w[1] # w[1] + w[2] # w[]
sage: w.coproduct(w[2])
w[] # w[2] - w[1] # w[1] + w[2] # w[]
sage: w[2,1].coproduct()
w[] # w[2, 1] - w[1] # w[1, 1] + w[1] # w[2] - w[1, 1] # w[1] + w[2] # w[1] + w[2, 1] # w[]
sage: w.coproduct(w[2,1])
w[] # w[2, 1] - w[1] # w[1, 1] + w[1] # w[2] - w[1, 1] # w[1] + w[2] # w[1] + w[2, 1] # w[]
```

from_other_uncached(*u*)

Return an element u of another basis of the ring of symmetric functions, expanded in the Witt basis self. The result is the same as self(u), but the from_other_uncached method does not precompute a cache with transition matrices. Thus, from_other_uncached is faster when u is sparse.

INPUT:

- u – an element of self.realization_of()

OUTPUT:

- the expansion of u in the Witt basis self

EXAMPLES:

```
sage: Sym = SymmetricFunctions(QQ)
sage: p = Sym.p()
sage: w = Sym.w()
sage: a = p([3,2]) - p([4,1]) + 27 * p([3])
sage: w.from_other_uncached(a) == w(a)
True
```

Here's a verification of an obvious fact that would take long with regular coercion:

```
sage: fouc = w.from_other_uncached
sage: fouc(p([15]))
w[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] + 3*w[3, 3, 3, 3, 3] + 5*w[5, 5, 5] + 15*w[15
sage: fouc(p([15])) * fouc(p([14])) == fouc(p([15, 14]))
True
```

Other bases:

```
sage: e = Sym.e()
sage: h = Sym.h()
sage: s = Sym.s()
sage: all( fouc(e(lam)) == w(e(lam)) for lam in Partitions(5) )
```

```
True
sage: all( fouc(h(lam)) == w(h(lam)) for lam in Partitions(5) )
True
sage: all( fouc(p(lam)) == w(p(lam)) for lam in Partitions(5) )
True
sage: all( fouc(s(lam)) == w(s(lam)) for lam in Partitions(5) )
True
```

verschiebung(*n*)

Return the image of the symmetric function `self` under the n-th Verschiebung operator.

The n-th Verschiebung operator $\mathbf{V}_n$ is defined to be the unique algebra endomorphism V of the ring of symmetric functions that satisfies $V(h_r) = h_{r/n}$ for every positive integer r divisible by n, and satisfies $V(h_r) = 0$ for every positive integer r not divisible by n. This operator $\mathbf{V}_n$ is a Hopf algebra endomorphism. For every nonnegative integer r with $n \mid r$, it satisfies

$$\mathbf{V}_n(h_r) = h_{r/n}, \quad \mathbf{V}_n(p_r) = np_{r/n}, \quad \mathbf{V}_n(e_r) = (-1)^{r-r/n} e_{r/n}, \quad \mathbf{V}_n(w_r) = w_{r/n},$$

(where h is the complete homogeneous basis, p is the powersum basis, e is the elementary basis, and w is the Witt basis). For every nonnegative integer r with $n \nmid r$, it satisfes

$$\mathbf{V}_n(h_r) = \mathbf{V}_n(p_r) = \mathbf{V}_n(e_r) = \mathbf{V}_n(w_r) = 0.$$

The n-th Verschiebung operator is also called the n-th Verschiebung endomorphism. Its name derives from the Verschiebung (German for "shift") endomorphism of the Witt vectors.

The n-th Verschiebung operator is adjoint to the n-th Frobenius operator (see `frobenius()` for its definition) with respect to the Hall scalar product (`scalar()`).

The action of the n-th Verschiebung operator on the Schur basis can also be computed explicitly. The following (probably clumsier than necessary) description can be obtained by solving exercise 7.61 in Stanley's [STA].

Let λ be a partition. Let n be a positive integer. If the n-core of λ is nonempty, then $\mathbf{V}_n(s_\lambda) = 0$. Otherwise, the following method computes $\mathbf{V}_n(s_\lambda)$: Write the partition λ in the form $(\lambda_1, \lambda_2, \ldots, \lambda_{ns})$ for some nonnegative integer s. (If n does not divide the length of λ, then this is achieved by adding trailing zeroes to λ.) Set $\beta_i = \lambda_i + ns - i$ for every $s \in \{1, 2, \ldots, ns\}$. Then, $(\beta_1, \beta_2, \ldots, \beta_{ns})$ is a strictly decreasing sequence of nonnegative integers. Stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $-1 - \beta_i$ modulo n. Let ξ be the sign of the permutation that is used for this sorting. Let ψ be the sign of the permutation that is used to stably sort the list $(1, 2, \ldots, ns)$ in order of (weakly) increasing remainder of $i - 1$ modulo n. (Notice that $\psi = (-1)^{n(n-1)s(s-1)/4}$.) Then, $\mathbf{V}_n(s_\lambda) = \xi\psi \prod_{i=0}^{n-1} s_{\lambda^{(i)}}$, where $(\lambda^{(0)}, \lambda^{(1)}, \ldots, \lambda^{(n-1)})$ is the n-quotient of λ.

INPUT:

- `n` – a positive integer

OUTPUT:

The result of applying the n-th Verschiebung operator (on the ring of symmetric functions) to `self`.

EXAMPLES:

```
sage: Sym = SymmetricFunctions(ZZ)
sage: w = Sym.w()
sage: w[3].verschiebung(2)
0
sage: w[4].verschiebung(4)
w[1]
```

TESTS:

Let us check that this method on the Witt basis gives the same result as the implementation in sfa.py on the complete homogeneous basis:

```
sage: Sym = SymmetricFunctions(QQ)
sage: w = Sym.w(); h = Sym.h()
sage: all( w(h(lam)).verschiebung(3) == w(h(lam).verschiebung(3))
....:      for lam in Partitions(6) )
True
sage: all( h(w(lam)).verschiebung(2) == h(w(lam).verschiebung(2))
....:      for lam in Partitions(4) )
True
```

5.1.270 Shard intersection order

This file builds a combinatorial version of the shard intersection order of type A (in the classification of finite Coxeter groups). This is a lattice on the set of permutations, closely related to noncrossing partitions and the weak order.

For technical reasons, the elements of the posets are not permutations, but can be easily converted from and to permutations:

```
sage: from sage.combinat.shard_order import ShardPosetElement
sage: p0 = Permutation([1,3,4,2])
sage: e0 = ShardPosetElement(p0); e0
(1, 3, 4, 2)
sage: Permutation(list(e0)) == p0
True
```

REFERENCES:

class sage.combinat.shard_order.**ShardPosetElement**(*p*)

Bases: tuple

An element of the shard poset.

This is basically a permutation with extra stored arguments:

- p – the permutation itself as a tuple
- runs – the decreasing runs as a tuple of tuples
- run_indices – a list integer -> index of the run
- dpg – the transitive closure of the shard preorder graph
- spg – the transitive reduction of the shard preorder graph

These elements can easily be converted from and to permutations:

```
sage: from sage.combinat.shard_order import ShardPosetElement
sage: p0 = Permutation([1,3,4,2])
sage: e0 = ShardPosetElement(p0); e0
(1, 3, 4, 2)
sage: Permutation(list(e0)) == p0
True
```

sage.combinat.shard_order.**shard_poset**(*n*)

Return the shard intersection order on permutations of size n.

This is defined on the set of permutations. To every permutation, one can attach a pre-order, using the descending runs and their relative positions.

The shard intersection order is given by the implication (or refinement) order on the set of pre-orders defined from all permutations.

This can also be seen in a geometrical way. Every pre-order defines a cone in a vector space of dimension n. The shard poset is given by the inclusion of these cones.

See also:

`shard_preorder_graph()`

EXAMPLES:

```
sage: P = posets.ShardPoset(4); P  # indirect doctest
Finite poset containing 24 elements
sage: P.chain_polynomial()
34*q^4 + 90*q^3 + 79*q^2 + 24*q + 1
sage: P.characteristic_polynomial()
q^3 - 11*q^2 + 23*q - 13
sage: P.zeta_polynomial()
17/3*q^3 - 6*q^2 + 4/3*q
sage: P.is_selfdual()
False
```

sage.combinat.shard_order.**shard_preorder_graph**(*runs*)

Return the preorder attached to a tuple of decreasing runs.

This is a directed graph, whose vertices correspond to the runs.

There is an edge from a run R to a run S if R is before S in the list of runs and the two intervals defined by the initial and final indices of R and S overlap.

This only depends on the initial and final indices of the runs. For this reason, this input can also be given in that shorten way.

INPUT:

- a tuple of tuples, the runs of a permutation, or
- a tuple of pairs (i, j), each one standing for a run from i to j.

OUTPUT:

a directed graph, with vertices labelled by integers

EXAMPLES:

```
sage: from sage.combinat.shard_order import shard_preorder_graph
sage: s = Permutation([2,8,3,9,6,4,5,1,7])
sage: def cut(lr):
....:     return tuple((r[0], r[-1]) for r in lr)
sage: shard_preorder_graph(cut(s.decreasing_runs()))
Digraph on 5 vertices
sage: s = Permutation([9,4,3,2,8,6,5,1,7])
sage: P = shard_preorder_graph(s.decreasing_runs())
sage: P.is_isomorphic(digraphs.TransitiveTournament(3))
True
```

5.1.271 Shuffle product of iterables

The shuffle product of two sequences of lengths m and n is a sum over the $\binom{m+n}{n}$ ways of interleaving the two sequences.

That could be defined inductively by:

$$(a_n)_{n\geqslant 0} \Cup (b_m)_{m\geqslant 0} = a_0 \cdot ((a_n)_{n\geqslant 1} \Cup (b_m)_{m\geqslant 0}) + b_0 \cdot ((a_n)_{n\geqslant 0} \Cup (b_m)_{m\geqslant 1})$$

with (a_n) and (b_m) two non-empty sequences and if one of them is empty then the product is equals to the other.

The shuffle product has been introduced by S. Eilenberg and S. Mac Lane in 1953 [EilLan53].

EXAMPLE:

```
sage: from sage.combinat.shuffle import ShuffleProduct
sage: list(ShuffleProduct([1,2], ["a", "b", "c"]))
[[1, 2, 'a', 'b', 'c'],
 ['a', 1, 2, 'b', 'c'],
 [1, 'a', 2, 'b', 'c'],
 ['a', 'b', 1, 2, 'c'],
 ['a', 1, 'b', 2, 'c'],
 [1, 'a', 'b', 2, 'c'],
 ['a', 'b', 'c', 1, 2],
 ['a', 'b', 1, 'c', 2],
 ['a', 1, 'b', 'c', 2],
 [1, 'a', 'b', 'c', 2]]
```

References:

Author:

- Jean-Baptiste Priez

class sage.combinat.shuffle.**SetShuffleProduct**(*l1*, *l2*, *element_constructor=None*)

Bases: sage.structure.sage_object.SageObject

The union of all possible shuffle products of two sets of iterables.

TESTS:

```
sage: from sage.combinat.shuffle import SetShuffleProduct
sage: TestSuite(SetShuffleProduct).run()
```

EXAMPLE:

```
sage: from sage.combinat.shuffle import SetShuffleProduct
sage: sorted(SetShuffleProduct({(1,), (2,3)}, {(4,5), (6,)}))
[[1, 4, 5],
 [1, 6],
 [2, 3, 4, 5],
 [2, 3, 6],
 [2, 4, 3, 5],
 [2, 4, 5, 3],
 [2, 6, 3],
 [4, 1, 5],
 [4, 2, 3, 5],
 [4, 2, 5, 3],
 [4, 5, 1],
 [4, 5, 2, 3],
 [6, 1],
 [6, 2, 3]]
```

cardinality()

The cardinality is defined by the sum of the cardinality of all shuffles. That means by a sum of binomials.

TESTS:

```
sage: from sage.combinat.shuffle import SetShuffleProduct
sage: SetShuffleProduct([[1,2],[3,4]], [[1,4]], element_constructor=set).cardinality()
12
```

class sage.combinat.shuffle.**ShuffleProduct**(*l1*, *l2*, *element_constructor=None*)

Bases: sage.structure.sage_object.SageObject

Shuffle product of two iterable.

EXAMPLES:

```
sage: from sage.combinat.shuffle import ShuffleProduct
sage: list(ShuffleProduct("abc", "de", element_constructor="".join))
['abcde',
 'adbce',
 'dabce',
 'abdce',
 'adebc',
 'daebc',
 'deabc',
 'adbec',
 'dabec',
 'abdec']
sage: list(ShuffleProduct("", "de", element_constructor="".join))
['de']
```

cardinality()

Return the number of shuffles of l_1 and l_2, respectively of lengths m and n, which is $\binom{m+n}{n}$.

TESTS:

```
sage: from sage.combinat.shuffle import ShuffleProduct
sage: ShuffleProduct([3,1,2], [4,2,1,3]).cardinality()
35
sage: ShuffleProduct([3,1,2,5,6,4], [4,2,1,3]).cardinality() == binomial(10,4)
True
```

5.1.272 Sidon sets and their generalizations, Sidon g-sets

AUTHORS:

- Martin Raum (07-25-2011)

sage.combinat.sidon_sets.**sidon_sets**(*N*, *g=1*)

Return the set of all Sidon-g sets that have elements less than or equal to N.

A Sidon-g set is a set of positive integers $A \subset [1, N]$ such that any integer M can be obtain at most g times as sums of unordered pairs of elements of A (the two elements are not necessary distinct):

$$\#\{(a_i, a_j) | a_i, a_j \in A, a_i + a_j = M, a_i \leq a_j\} \leq g$$

INPUT:

- N – A positive integer.
- g – A positive integer (default: 1).

OUTPUT:

- A Sage set with categories whose element are also set of integers.

EXAMPLES:

```
sage: S = sidon_sets(3, 2)
sage: S
{{2}, {3}, {1, 2}, {}, {2, 3}, {1}, {1, 3}, {1, 2, 3}}
sage: S.cardinality()
8
sage: S.category()
Category of finite sets
sage: sid = S.an_element()
sage: sid
{2}
sage: sid.category()
Category of finite sets
```

TESTS:

```
sage: S = sidon_sets(10)
sage: TestSuite(S).run()
sage: Set([1,2,4,8,13]) in sidon_sets(13)
True
```

The following piece of code computes the first values of the Sloane sequence entitled 'Length of shortest (or optimal) Golomb ruler with n marks' with a very dumb algorithm. (sequence identifier A003022):

```
sage: n = 1
sage: L = []
sage: for i in range(1,19):
...       nb = max([S.cardinality() for S in sidon_sets(i)])
...       if nb > n:
...           L.append(i-1)
...           n = nb
sage: L
[1, 3, 6, 11, 17]
```

The following tests check that some generalized Sidon sets satisfy the right conditions, using a dumb but exhaustive algorithm:

```
sage: from itertools import groupby
sage: all(all(l <= 3 for l in map(lambda s: len(list(s[1])), groupby(sorted(a + ap for a in sid
True
sage: all(all(l <= 5 for l in map(lambda s: len(list(s[1])), groupby(sorted(a + ap for a in sid
True
```

Checking of arguments:

```
sage: sidon_sets(1,1)
{{}, {1}}
sage: sidon_sets(-1,3)
Traceback (most recent call last):
...
ValueError: N must be a positive integer
sage: sidon_sets(1, -3)
Traceback (most recent call last):
...
ValueError: g must be a positive integer
```

sage.combinat.sidon_sets.**sidon_sets_rec**(*N*, *g=1*)

Return the set of all Sidon-g sets that have elements less than or equal to N without checking the arguments. This internal function should not be call directly by user.

TESTS:

```
sage: from sage.combinat.sidon_sets import sidon_sets_rec
sage: sidon_sets_rec(3,2)
{{2}, {3}, {1, 2}, {}, {2, 3}, {1}, {1, 3}, {1, 2, 3}}
```

5.1.273 Similarity class types of matrices with entries in a finite field

The notion of a matrix conjugacy class type was introduced by J. A. Green in [Green55], in the context of computing the irreducible charcaters of finite general linear groups. The class types are equivalence classes of similarity classes of square matrices with entries in a finite field which, roughly speaking, have the same qualitative properties.

For example, all similarity classes of the same class type have centralizers of the same cardinality and the same degrees of elementary divisors. Qualitative properties of similarity classes such as semisimplicity and regularity descend to class types.

The most important feature of similarity class types is that, for any n, the number of similarity class types of $n \times n$ matrices is independent of q. This makes it possible to perform many combinatorial calculations treating q as a formal variable.

In order to define similarity class types, recall that similarity classes of $n \times n$ matrices with entries in $\mathbf{F}_q$ correspond to functions

$$c : \mathrm{Irr}\mathbf{F}_{q[t]} \to \Lambda$$

such that

$$\sum_{f \in \mathrm{Irr}\mathbf{F}_{q[t]}} |c(f)| \deg f = n,$$

where we denote the set of irreducible monic polynomials in $\mathbf{F}_{q[t]}$ by $\mathrm{Irr}\mathbf{F}_{q[t]}$, the set of all partitions by Λ, and the size of $\lambda \in \Lambda$ by $|\lambda|$.

Similarity classes indexed by functions c_1 and c_2 as above are said to be of the same type if there exists a degree-preserving self-bijection σ of $\mathrm{Irr}\mathbf{F}_{q[t]}$ such that $c_2 = c_1 \circ \sigma$. Thus, the type of c remembers only the degrees of the polynomials (and not the polynomials themselves) for which c takes a certain value λ. Replacing each irreducible polynomial of degree d for which c takes a non-trivial value λ by the pair (d, λ), we obtain a multiset of such pairs. Clearly, c_1 and c_2 have the same type if and only if these multisets are equal. Thus a similarity class type may be viewed as a multiset of pairs of the form (d, λ).

For 2×2 matrices there are four types:

```
sage: for tau in SimilarityClassTypes(2):
....:     print tau
[[1, [1]], [1, [1]]]
[[1, [2]]]
[[1, [1, 1]]]
[[2, [1]]]
```

These four types correspond to the regular split semisimple matrices, the non-semisimple matrices, the central matrices and the irreducble matrices respectively.

For any matrix A in a given similarity class type, it is possible to calculate the number elements in the similarity class of A, the dimension of the algebra of matrices in $M_n(A)$ that commite with A, and the cardinality of the subgroup of $GL_n(\mathbf{F}_q)$ that commute with A. For each similarity class type, it is also possible to compute the number of classes of that type (and hence, the total number of matrices of that type). All these calculations treat the cardinality q of the finite field as a formal variable:

```
sage: M = SimilarityClassType([[1, [1]], [1, [1]]])
sage: M.class_card()
q^2 + q
sage: M.centralizer_algebra_dim()
2
sage: M.centralizer_group_card()
q^2 - 2*q + 1
sage: M.number_of_classes()
1/2*q^2 - 1/2*q
sage: M.number_of_matrices()
1/2*q^4 - 1/2*q^2
```

We now describe two applications of similarity class types.

We say that an $n \times n$ matrix has rational canonical form type λ for some partition λ of n if the diagonal blocks in the rational canonical form have sizes given by the parts of λ. Thus the matrices with rational canonical type (n) are the regular ones, while the matrices with rational canonical type (1^n) are the central ones.

Using similarity class types, it becomes easy to get a formula for the number of matrices with a given rational canonical type:

```
sage: def matrices_with_rcf(la):
....:     return sum([tau.number_of_matrices() for tau in filter(lambda tau:tau.rcf()==la, SimilarityC
sage: matrices_with_rcf(Partition([2,1]))
q^6 + q^5 + q^4 - q^3 - q^2 - q
```

Similarity class types can also be used to calculate the number of simultaneous similarity classes of k-tuples of $n \times n$ matrices with entries in $\mathbf{F}_q$ by using Burnside's lemma:

```
sage: from sage.combinat.similarity_class_type import order_of_general_linear_group, centralizer_alge
sage: q = ZZ['q'].gen()
sage: def simultaneous_similarity_classes(n,k):
....:     return SimilarityClassTypes(n).sum(lambda la: q**(k*centralizer_algebra_dim(la)), invertibl
sage: simultaneous_similarity_classes(3, 2)
q^10 + q^8 + 2*q^7 + 2*q^6 + 2*q^5 + q^4
```

Similarity class types can be used to calculate the coefficients of generating functions coming from the cycle index type techniques of Kung and Stong (see Morrison [Morrison06]).

Along with the results of [PSS13], similarity class types can be used to calculate the number of similarity classes of matrices of order n with entries in a principal ideal local ring of length two with residue field of cardinality q with centralizer of any given cardinality up to $n = 4$. Among these, the classes which are selftranspose can also be counted:

```
sage: from sage.combinat.similarity_class_type import matrix_centralizer_cardinalities_length_two
sage: list(matrix_centralizer_cardinalities_length_two(3))
[(q^6 - 3*q^5 + 3*q^4 - q^3, 1/6*q^6 - 1/2*q^5 + 1/3*q^4),
(q^6 - 2*q^5 + q^4, q^5 - q^4),
(q^8 - 3*q^7 + 3*q^6 - q^5, 1/2*q^5 - q^4 + 1/2*q^3),
(q^8 - 2*q^7 + q^6, q^4 - q^3),
(q^10 - 2*q^9 + 2*q^7 - q^6, q^4 - q^3),
(q^8 - q^7 - q^6 + q^5, 1/2*q^5 - q^4 + 1/2*q^3),
(q^6 - q^5 - q^4 + q^3, 1/2*q^6 - 1/2*q^5),
(q^6 - q^5, q^4),
(q^10 - 2*q^9 + q^8, q^3),
(q^8 - 2*q^7 + q^6, q^4 - q^3),
(q^8 - q^7, q^3 + q^2),
(q^12 - 3*q^11 + 3*q^10 - q^9, 1/6*q^4 - 1/2*q^3 + 1/3*q^2),
(q^12 - 2*q^11 + q^10, q^3 - q^2),
```

```
(q^14 - 2*q^13 + 2*q^11 - q^10, q^3 - q^2),
(q^12 - q^11 - q^10 + q^9, 1/2*q^4 - 1/2*q^3),
(q^12 - q^11, q^2),
(q^14 - 2*q^13 + q^12, q^2),
(q^18 - q^17 - q^16 + q^14 + q^13 - q^12, q^2),
(q^12 - q^9, 1/3*q^4 - 1/3*q^2),
(q^6 - q^3, 1/3*q^6 - 1/3*q^4)]
```

REFERENCES:

AUTHOR:

- Amritanshu Prasad (2013-07-18): initial implementation
- Amritanshu Prasad (2013-09-09): added functions for similarity classes over rings of length two

class sage.combinat.similarity_class_type.**PrimarySimilarityClassType**(*parent*, *deg*, *par*)

Bases: sage.structure.element.Element

A primary similarity class type is a pair consisting of a partition and a positive integer.

For a partition λ and a positive integer d, the primary similarity class type (d, λ) represents similarity classes of square matrices of order $|\lambda| \cdot d$ with entries in a finite field of order q which correspond to the $\mathbf{F}_{q[t]}$-module

$$\frac{\mathbf{F}_{q[t]}}{p(t)^{\lambda_1}} \oplus \frac{\mathbf{F}_{q[t]}}{p(t)^{\lambda_2}} \oplus \cdots$$

for some irreducible polynomial $p(t)$ of degree d.

centralizer_algebra_dim()

Return the dimension of the algebra of matrices which commute with a matrix of type self.

For a partition (d, λ) this dimension is given by $d(\lambda_1 + 3\lambda_2 + 5\lambda_3 + \cdots)$.

EXAMPLES:

```
sage: PT = PrimarySimilarityClassType(2, [3, 2, 1])
sage: PT.centralizer_algebra_dim()
28
```

centralizer_group_card(*q=None*)

Return the cardinality of the centralizer group of a matrix of type self in a field of order q.

INPUT:

- q – an integer or an indeterminate

EXAMPLES:

```
sage: PT = PrimarySimilarityClassType(1, [])
sage: PT.centralizer_group_card()
1
sage: PT = PrimarySimilarityClassType(2, [1, 1])
sage: PT.centralizer_group_card()
q^8 - q^6 - q^4 + q^2
```

degree()

Return degree of self.

EXAMPLES:

```
sage: PT = PrimarySimilarityClassType(2, [3, 2, 1])
sage: PT.degree()
2
```

partition()
Return partition corresponding to `self`.

EXAMPLES:

```
sage: PT = PrimarySimilarityClassType(2, [3, 2, 1])
sage: PT.partition()
[3, 2, 1]
```

size()
Return the size of `self`.

EXAMPLES:

```
sage: PT = PrimarySimilarityClassType(2, [3, 2, 1])
sage: PT.size()
12
```

statistic(*func*, *q=None*)
Return $n_\lambda(q^d)$ where n_λ is the value returned by `func` upon input λ, if `self` is (d, λ).

EXAMPLES:

```
sage: PT = PrimarySimilarityClassType(2, [3, 1])
sage: q = ZZ['q'].gen()
sage: PT.statistic(lambda la:q**la.size(), q = q)
q^8
```

class sage.combinat.similarity_class_type.**PrimarySimilarityClassTypes**(*n*, *min*)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

All primary similarity class types of size `n` whose degree is greater than that of `min` or whose degree is that of `min` and whose partition is less than of `min` in lexicographic order.

A primary similarity class type of size n is a pair (λ, d) consisting of a partition λ and a positive integer d such that $|\lambda|d = n$.

INPUT:

- `n` – a positive integer
- `min` – a primary matrix type of size `n`

EXAMPLES:

If `min` is not specified, then the class of all primary similarity class types of size `n` is created:

```
sage: PTC = PrimarySimilarityClassTypes(2)
sage: for PT in PTC:
....:     print PT
[1, [2]]
[1, [1, 1]]
[2, [1]]
```

If `min` is specified, then the class consists of only those primary similarity class types whose degree is greater than that of `min` or whose degree is that of `min` and whose partition is less than of `min` in lexicographic order:

```
sage: PTC = PrimarySimilarityClassTypes(2, min = PrimarySimilarityClassType(1, [1, 1]))
sage: for PT in PTC:
....:     print PT
[1, [1, 1]]
[2, [1]]
```

Element
alias of PrimarySimilarityClassType

size()
Return size of elements of self.

The size of a primary similarity class type (d, λ) is $d|\lambda|$.

EXAMPLES:

```
sage: PTC = PrimarySimilarityClassTypes(2)
sage: PTC.size()
2
```

class sage.combinat.similarity_class_type.**SimilarityClassType**(*parent*, *tau*)
Bases: sage.combinat.combinat.CombinatorialElement

A similarity class type.

A matrix type is a multiset of primary similairty class types.

INPUT:

- tau – A list of primary similarity class types

EXAMPLES:

```
sage: tau1 = SimilarityClassType([[3, [3, 2, 1]], [2, [2, 1]]]); tau1
[[2, [2, 1]], [3, [3, 2, 1]]]
```

as_partition_dictionary()
Return a dictionary whose keys are the partitions of types occuring in self and the value at the key λ is the partition formed by sorting the degrees of primary types with partition λ.

EXAMPLES:

```
sage: tau = SimilarityClassType([[1, [1]], [1, [1]]])
sage: tau.as_partition_dictionary()
{[1]: [1, 1]}
```

centralizer_algebra_dim()
Return the dimension of the algebra of matrices which commute with a matrix of type self.

EXAMPLES:

```
sage: tau = SimilarityClassType([[1, [1]], [1, [1]]])
sage: tau.centralizer_algebra_dim()
2
```

centralizer_group_card(*q=None*)
Return the cardinality of the group of matrices in $GL_n(\mathbf{F}_q)$ which commute with a matrix of type self.

INPUT:

- q – an integer or an indeterminate

EXAMPLES:

```
sage: tau = SimilarityClassType([[1, [1]], [1, [1]]])
sage: tau.centralizer_group_card()
q^2 - 2*q + 1
```

class_card(*q=None*)

Return the number of matrices in each similarity class of type `self`.

INPUT:

- `q` – an integer or an indeterminate

EXAMPLES:

```
sage: tau = SimilarityClassType([[1, [1, 1, 1, 1]]])
sage: tau.class_card()
1
sage: tau = SimilarityClassType([[1, [1]], [1, [1]]])
sage: tau.class_card()
q^2 + q
```

is_regular()

Return `True` if every primary type in `self` has partition with one part.

EXAMPLES:

```
sage: tau = SimilarityClassType([[2, [1]], [1, [3]]])
sage: tau.is_regular()
True
sage: tau = SimilarityClassType([[2, [1, 1]], [1, [3]]])
sage: tau.is_regular()
False
```

is_semisimple()

Return `True` if every primary similarity class type in `self` has all parts equal to `1`.

EXAMPLES:

```
sage: tau = SimilarityClassType([[2, [1, 1]], [1, [1]]])
sage: tau.is_semisimple()
True
sage: tau = SimilarityClassType([[2, [1, 1]], [1, [2]]])
sage: tau.is_semisimple()
False
```

number_of_classes(*invertible=False*, *q=None*)

Return the number of similarity classes of matrices of type `self`.

INPUT:

- `invertible` – Boolean; return number of invertible classes if set to `True`
- `q` – An integer or an indeterminate

EXAMPLES:

```
sage: tau = SimilarityClassType([[1, [1]], [1, [1]]])
sage: tau.number_of_classes()
1/2*q^2 - 1/2*q
```

number_of_matrices(*invertible=False*, *q=None*)

Return the number of matrices of type `self`.

INPUT:

- `invertible` – A boolean; return the number of invertible matrices if set

EXAMPLES:

```
sage: tau = SimilarityClassType([[1, [1]]])
sage: tau.number_of_matrices()
q
sage: tau.number_of_matrices(invertible = True)
q - 1
sage: tau = SimilarityClassType([[1, [1]], [1, [1]]])
sage: tau.number_of_matrices()
1/2*q^4 - 1/2*q^2
```

rcf()

Return the partition corresponding to the rational canonical form of a matrix of type `self`.

EXAMPLES:

```
sage: tau = SimilarityClassType([[2, [1, 1, 1]], [1, [3, 2]]])
sage: tau.rcf()
[5, 4, 2]
```

size()

Return the sum of the sizes of the primary parts of `self`.

EXAMPLES:

```
sage: tau = SimilarityClassType([[3, [3, 2, 1]], [2, [2, 1]]])
sage: tau.size()
24
```

statistic(*func*, *q=None*)

Return

$$\prod_{(d,\lambda)\in\tau} n_\lambda(q^d)$$

where $n_\lambda(q)$ is the value returned by `func` on the input λ.

INPUT:

- `func` – a function that takes a partition to a polynomial in `q`
- `q` – an integer or an indeterminate

EXAMPLES:

```
sage: tau = SimilarityClassType([[1, [1]], [1, [2, 1]], [2, [1, 1]]])
sage: from sage.combinat.similarity_class_type import fq
sage: tau.statistic(lambda la: prod([fq(m) for m in la.to_exp()]))
(q^9 - 3*q^8 + 2*q^7 + 2*q^6 - 4*q^5 + 4*q^4 - 2*q^3 - 2*q^2 + 3*q - 1)/q^9
sage: q = ZZ['q'].gen()
sage: tau.statistic(lambda la: q**la.size(), q = q)
q^8
```

class sage.combinat.similarity_class_type.**SimilarityClassTypes**(*n*, *min*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Class of all similarity class types of size `n` with all primary matrix types greater than or equal to the primary matrix type `min`.

A similarity class type is a multiset of primary matrix types.

INPUT:

- n – a non-negative integer
- min – a primary similarity class type

EXAMPLES:

If min is not specified, then the class of all matrix types of size n is constructed:

```
sage: M = SimilarityClassTypes(2)
sage: for tau in M:
....:     print tau
[[1, [1]], [1, [1]]]
[[1, [2]]]
[[1, [1, 1]]]
[[2, [1]]]
```

If min is specified, then the class consists of only those similarity class types which are multisets of primary matrix types which either have size greater than that of min, or if they have size equal to that of min, then they occur after min in the iterator for PrimarySimilarityClassTypes(n), where n is the size of min:

```
sage: M = SimilarityClassTypes(2, min = [1, [1, 1]])
sage: for tau in M:
....:     print tau
[[1, [1, 1]]]
[[2, [1]]]
```

Element

alias of SimilarityClassType

size()

Return size of self.

EXAMPLES:

```
sage: tau = SimilarityClassType([[3, [3, 2, 1]], [2, [2, 1]]])
sage: tau.parent().size()
24
```

sum(*stat*, *sumover='matrices'*, *invertible=False*, *q=None*)

Return the sum of a local statistic over all types.

Given a set of functions $n_\lambda(q)$ (these could be polynomials or rational functions in q, for each similarity class type τ define

$$n_\tau(q) = \prod_{(d,\lambda)\in\tau} n_\lambda(q^d).$$

This function returns

$$\sum n_{\tau(g)}(q)$$

where $\tau(g)$ denotes the type of a matrix g, and the sum is over all $n \times n$ matrices if sumover is set to "matrices", is over all $n \times n$ similarity classes if sumover is set to "classes", and over all $n \times n$ types if sumover is set to "types". If invertible is set to True, then the sum is only over invertible matrices or classes.

INPUT:

- stat – a function which takes partitions and returns a function of q
- sumover – can be one of the following:

- "matrices"
- "classes"
- "types"

• q – an integer or an indeterminate

OUTPUT:

A function of q.

EXAMPLES:

```
sage: M = SimilarityClassTypes(2)
sage: M.sum(lambda la:1)
q^4
sage: M.sum(lambda la:1, invertible = True)
q^4 - q^3 - q^2 + q
sage: M.sum(lambda la:1, sumover = "classes")
q^2 + q
sage: M.sum(lambda la:1, sumover = "classes", invertible = True)
q^2 - 1
```

Burside's lemma can be used to calculate the number of similarity classes of matrices:

```
sage: from sage.combinat.similarity_class_type import centralizer_algebra_dim, order_of_gene
sage: q = ZZ['q'].gen()
sage: M.sum(lambda la:q**centralizer_algebra_dim(la), invertible = True)/order_of_general_li
q^2 + q
```

sage.combinat.similarity_class_type.**centralizer_algebra_dim**(*la*)

Return the dimension of the centralizer algebra in $M_n(\mathbf{F}_q)$ of a nilpotent matrix whose Jordan blocks are given by la.

EXAMPLES:

```
sage: from sage.combinat.similarity_class_type import centralizer_algebra_dim
sage: centralizer_algebra_dim(Partition([2, 1]))
5
```

Note: If it is a list, la is expected to be sorted in decreasing order.

---

sage.combinat.similarity_class_type.**centralizer_group_cardinality**(*la*, *q=None*)

Return the cardinality of the centralizer group in $GL_n(\mathbf{F}_q)$ of a nilpotent matrix whose Jordan blocks are given by la.

INPUT:

• lambda – a partition

• q – an integer or an indeterminate

OUTPUT:

A polynomial function of q.

EXAMPLES:

```
sage: from sage.combinat.similarity_class_type import centralizer_group_cardinality
sage: q = ZZ['q'].gen()
sage: centralizer_group_cardinality(Partition([2, 1]))
q^5 - 2*q^4 + q^3
```

sage.combinat.similarity_class_type.**dictionary_from_generator**(*gen*)

Given a generator for a list of pairs (c, f), construct a dictionary whose keys are the distinct values for c and whose value at c is the sum of f over all pairs of the form (c', f) such that $c = c'$.

EXAMPLES:

```
sage: from sage.combinat.similarity_class_type import dictionary_from_generator
sage: dictionary_from_generator(((floor(x/2), x) for x in xrange(10)))
{0: 1, 1: 5, 2: 9, 3: 13, 4: 17}
```

It also works with lists:

```
sage: dictionary_from_generator([(floor(x/2),x) for x in range(10)])
{0: 1, 1: 5, 2: 9, 3: 13, 4: 17}
```

Note: Since the generator is first converted to a list, memory usage could be high.

---

sage.combinat.similarity_class_type.**ext_orbit_centralizers**(*input_data*, *q=None*, *selftranspose=False*)

Generate pairs consisting of centralizer cardinalities of orbits in $\mathrm{Ext}^1(M, M)$ for the action of $\mathrm{Aut}(M, M)$, where M is the $\mathbf{F}_{q[t]}$-module constructed from `input` and their frequencies.

INPUT:

- `input_data` – input for `input_parsing()`
- `q` – (default: q) an integer or an indeterminate
- `selftranspose` – (default: `False`) boolean stating if we only want selftranspose type

TESTS:

```
sage: from sage.combinat.similarity_class_type import ext_orbit_centralizers
sage: list(ext_orbit_centralizers([6, 1]))
[(q^9 - 2*q^8 + q^7, q^6),
 (q^7 - 2*q^6 + q^5, q^7 - q^6),
 (q^7 - q^6, q^6 + q^5)]
sage: list(ext_orbit_centralizers([6, 1], selftranspose = True))
[(q^9 - 2*q^8 + q^7, q^6),
 (q^7 - 2*q^6 + q^5, q^7 - q^6),
 (q^7 - q^6, q^6 - q^5)]
sage: list(ext_orbit_centralizers([6, 1, 1]))
[(q^12 - 3*q^11 + 3*q^10 - q^9, 1/2*q^7 - 1/2*q^6),
 (q^8 - 3*q^7 + 3*q^6 - q^5, 1/2*q^8 - q^7 + 1/2*q^6),
 (q^12 - 2*q^11 + q^10, q^6),
 (q^8 - 2*q^7 + q^6, q^7 - q^6),
 (q^14 - 2*q^13 + 2*q^11 - q^10, q^6),
 (q^10 - 2*q^9 + 2*q^7 - q^6, q^7 - q^6),
 (q^12 - q^11 - q^10 + q^9, 1/2*q^7 - 1/2*q^6),
 (q^8 - q^7 - q^6 + q^5, 1/2*q^8 - q^7 + 1/2*q^6),
 (q^8 - 2*q^7 + q^6, q^7 - q^6),
 (q^8 - q^7, q^6 + 2*q^5),
 (q^10 - 2*q^9 + q^8, 2*q^6)]
sage: list(ext_orbit_centralizers([6, 1, 1], selftranspose = True))
[(q^12 - 3*q^11 + 3*q^10 - q^9, 1/2*q^7 - 1/2*q^6),
 (q^8 - 3*q^7 + 3*q^6 - q^5, 1/2*q^8 - q^7 + 1/2*q^6),
 (q^12 - 2*q^11 + q^10, q^6),
 (q^8 - 2*q^7 + q^6, q^7 - q^6),
 (q^14 - 2*q^13 + 2*q^11 - q^10, q^6),
```

```
 (q^10 - 2*q^9 + 2*q^7 - q^6, q^7 - q^6),
 (q^12 - q^11 - q^10 + q^9, 1/2*q^7 - 1/2*q^6),
 (q^8 - q^7 - q^6 + q^5, 1/2*q^8 - q^7 + 1/2*q^6),
 (q^8 - 2*q^7 + q^6, q^7 - q^6),
 (q^8 - q^7, q^6)]
sage: list(ext_orbit_centralizers([2, [6, 1, 1]], selftranspose = True))
[(q^24 - 3*q^22 + 3*q^20 - q^18, 1/2*q^14 - 1/2*q^12),
 (q^16 - 3*q^14 + 3*q^12 - q^10, 1/2*q^16 - q^14 + 1/2*q^12),
 (q^24 - 2*q^22 + q^20, q^12),
 (q^16 - 2*q^14 + q^12, q^14 - q^12),
 (q^28 - 2*q^26 + 2*q^22 - q^20, q^12),
 (q^20 - 2*q^18 + 2*q^14 - q^12, q^14 - q^12),
 (q^24 - q^22 - q^20 + q^18, 1/2*q^14 - 1/2*q^12),
 (q^16 - q^14 - q^12 + q^10, 1/2*q^16 - q^14 + 1/2*q^12),
 (q^16 - 2*q^14 + q^12, q^14 - q^12),
 (q^16 - q^14, q^12)]
sage: list(ext_orbit_centralizers([[2, [6, 1, 1]]], selftranspose = True))
[(q^24 - 3*q^22 + 3*q^20 - q^18, 1/2*q^14 - 1/2*q^12),
 (q^16 - 3*q^14 + 3*q^12 - q^10, 1/2*q^16 - q^14 + 1/2*q^12),
 (q^24 - 2*q^22 + q^20, q^12),
 (q^16 - 2*q^14 + q^12, q^14 - q^12),
 (q^28 - 2*q^26 + 2*q^22 - q^20, q^12),
 (q^20 - 2*q^18 + 2*q^14 - q^12, q^14 - q^12),
 (q^24 - q^22 - q^20 + q^18, 1/2*q^14 - 1/2*q^12),
 (q^16 - q^14 - q^12 + q^10, 1/2*q^16 - q^14 + 1/2*q^12),
 (q^16 - 2*q^14 + q^12, q^14 - q^12),
 (q^16 - q^14, q^12)]
```

sage.combinat.similarity_class_type.**ext_orbits**(*input_data*, *q=None*, *selftranspose=False*)

Return the number of orbits in $\mathrm{Ext}^1(M, M)$ for the action of $\mathrm{Aut}(M, M)$, where M is the $\mathbf{F}_{q[t]}$-module constructed from `input_data`.

INPUT:

- `input_data` – input for `input_parsing()`
- `q` – (default: q) an integer or an indeterminate
- `selftranspose` – (default: `False`) boolean stating if we only want selftranspose type

TESTS:

```
sage: from sage.combinat.similarity_class_type import ext_orbits
sage: ext_orbits([6, 1])
q^7 + q^6 + q^5
sage: ext_orbits([6, 1], selftranspose = True)
q^7 + q^6 - q^5
sage: ext_orbits([6, 1, 1])
q^8 + 2*q^7 + 2*q^6 + 2*q^5
sage: ext_orbits ([6, 1, 1], selftranspose = True)
q^8 + 2*q^7
sage: ext_orbits([2, 2])
q^4 + q^3 + q^2
sage: ext_orbits([2, 2], selftranspose = True)
q^4 + q^3 + q^2
sage: ext_orbits([2, 2, 2])
q^6 + q^5 + 2*q^4 + q^3 + 2*q^2
sage: ext_orbits([2, 2, 2], selftranspose = True)
q^6 + q^5 + 2*q^4 + q^3
```

```
sage: ext_orbits([2, 2, 2, 2])
q^8 + q^7 + 3*q^6 + 3*q^5 + 5*q^4 + 3*q^3 + 3*q^2
sage: ext_orbits([2, 2, 2, 2], selftranspose = True)
q^8 + q^7 + 3*q^6 + 3*q^5 + 3*q^4 + q^3 + q^2
sage: ext_orbits([2, [6, 1]])
q^14 + q^12 + q^10
sage: ext_orbits([[2, [6, 1]]])
q^14 + q^12 + q^10
```

sage.combinat.similarity_class_type.**fq**(*n*, *q=None*)

Return $(1-q^{-1})(1-q^{-2})\cdots(1-q^{-n})$.

INPUT:

- n – A non-negative integer
- q – an integer or an indeterminate

OUTPUT:

A rational function in q.

EXAMPLES:

```
sage: from sage.combinat.similarity_class_type import fq
sage: fq(0)
1
sage: fq(3)
(q^6 - q^5 - q^4 + q^2 + q - 1)/q^6
```

sage.combinat.similarity_class_type.**input_parsing**(*data*)

Recognize and return the intended type of input.

TESTS:

```
sage: from sage.combinat.similarity_class_type import input_parsing
sage: input_parsing(Partition([2, 1]))
('par', [2, 1])
sage: input_parsing(PrimarySimilarityClassType(2, [2, 1]))
('pri', [2, [2, 1]])
sage: input_parsing(SimilarityClassType([[2, [2, 1]]]))
('sim', [[2, [2, 1]]])
sage: input_parsing([2, 1])
('par', [2, 1])
sage: input_parsing([2, [2, 1]])
('pri', [2, [2, 1]])
sage: input_parsing([[2, [2, 1]]])
('sim', [[2, [2, 1]]])
```

sage.combinat.similarity_class_type.**matrix_centralizer_cardinalities**(*n*, *q=None*, *invertible=False*)

Generate pairs consisting of centralizer cardinalities of matrices over a finite field and their frequencies.

TESTS:

```
sage: from sage.combinat.similarity_class_type import matrix_centralizer_cardinalities
sage: list(matrix_centralizer_cardinalities(1))
[(q - 1, q)]
sage: list(matrix_centralizer_cardinalities(2))
[(q^2 - 2*q + 1, 1/2*q^2 - 1/2*q),
```

```
(q^2 - q, q),
(q^4 - q^3 - q^2 + q, q),
(q^2 - 1, 1/2*q^2 - 1/2*q)]
sage: list(matrix_centralizer_cardinalities(2, invertible = True))
[(q^2 - 2*q + 1, 1/2*q^2 - 3/2*q + 1),
(q^2 - q, q - 1),
(q^4 - q^3 - q^2 + q, q - 1),
(q^2 - 1, 1/2*q^2 - 1/2*q)]
```

sage.combinat.similarity_class_type.**matrix_centralizer_cardinalities_length_two**(*n*, *q=None*, *selftranspose=False*, *invertible=False*)

Generate pairs consisting of centralizer cardinalities of matrices over a principal ideal local ring of length two with residue field of order `q` and their frequencies.

INPUT:

- `n` – the order
- `q` – (default: q) an integer or an indeterminate
- `selftranspose` – (default: `False`) boolean stating if we only want selftranspose type
- `invertible` – (default: `False`) boolean stating if we only want invertible type

TESTS:

```
sage: from sage.combinat.similarity_class_type import matrix_centralizer_cardinalities_length_tw
sage: list(matrix_centralizer_cardinalities_length_two(1))
[(q^2 - q, q^2)]
sage: list(matrix_centralizer_cardinalities_length_two(2))
[(q^4 - 2*q^3 + q^2, 1/2*q^4 - 1/2*q^3),
(q^4 - q^3, q^3),
(q^6 - 2*q^5 + q^4, 1/2*q^3 - 1/2*q^2),
(q^6 - q^5, q^2),
(q^8 - q^7 - q^6 + q^5, q^2),
(q^6 - q^4, 1/2*q^3 - 1/2*q^2),
(q^4 - q^2, 1/2*q^4 - 1/2*q^3)]
sage: from sage.combinat.similarity_class_type import dictionary_from_generator
sage: dictionary_from_generator(matrix_centralizer_cardinalities_length_two(2, q = 2))
{4: 4, 8: 8, 12: 4, 16: 2, 32: 4, 48: 2, 96: 4}
```

sage.combinat.similarity_class_type.**matrix_similarity_classes**(*n*, *q=None*, *invertible=False*)

Return the number of matrix similarity classes over a finite field of order `q`.

TESTS:

```
sage: from sage.combinat.similarity_class_type import matrix_similarity_classes
sage: matrix_similarity_classes(2)
q^2 + q
sage: matrix_similarity_classes(2, invertible = True)
q^2 - 1
sage: matrix_similarity_classes(2, invertible = True, q = 4)
15
```

sage.combinat.similarity_class_type.**matrix_similarity_classes_length_two**(*n*, *q=None*, *self-transpose=False*, *invertible=False*)

Return the number of similarity classes of matrices of order `n` with entries in a principal ideal local ring of length two.

INPUT:

- `n` – the order
- `q` – (default: q) an integer or an indeterminate
- `selftranspose` – (default: `False`) boolean stating if we only want selftranspose type
- `invertible` – (default: `False`) boolean stating if we only want invertible type

EXAMPLES:

We can generate Table 6 of [PSS13]:

```
sage: from sage.combinat.similarity_class_type import matrix_similarity_classes_length_two
sage: matrix_similarity_classes_length_two(2)
q^4 + q^3 + q^2
sage: matrix_similarity_classes_length_two(2, invertible = True)
q^4 - q
sage: matrix_similarity_classes_length_two(3)
q^6 + q^5 + 2*q^4 + q^3 + 2*q^2
sage: matrix_similarity_classes_length_two(3, invertible = true)
q^6 - q^3 + 2*q^2 - 2*q
sage: matrix_similarity_classes_length_two(4)
q^8 + q^7 + 3*q^6 + 3*q^5 + 5*q^4 + 3*q^3 + 3*q^2
sage: matrix_similarity_classes_length_two(4, invertible = True)
q^8 + q^6 - q^5 + 2*q^4 - 2*q^3 + 2*q^2 - 3*q
```

And also Table 7:

```
sage: matrix_similarity_classes_length_two(2, selftranspose = True)
q^4 + q^3 + q^2
sage: matrix_similarity_classes_length_two(2, selftranspose = True, invertible = True)
q^4 - q
sage: matrix_similarity_classes_length_two(3, selftranspose = True)
q^6 + q^5 + 2*q^4 + q^3
sage: matrix_similarity_classes_length_two(3, selftranspose = True, invertible = True)
q^6 - q^3
sage: matrix_similarity_classes_length_two(4, selftranspose = True)
q^8 + q^7 + 3*q^6 + 3*q^5 + 3*q^4 + q^3 + q^2
sage: matrix_similarity_classes_length_two(4, selftranspose = True, invertible = True)
q^8 + q^6 - q^5 - q
```

sage.combinat.similarity_class_type.**order_of_general_linear_group**(*n*, *q=None*)

Return the cardinality of the group of $n \times n$ invertible matrices with entries in a field of order `q`.

INPUT:

- `n` – a non-negative integer
- `q` – an integer or an indeterminate

EXAMPLES:

```
sage: from sage.combinat.similarity_class_type import order_of_general_linear_group
sage: order_of_general_linear_group(0)
1
sage: order_of_general_linear_group(2)
q^4 - q^3 - q^2 + q
```

sage.combinat.similarity_class_type.**primitives**(*n*, *invertible=False*, *q=None*)

Return the number of similarity classes of simple matrices of order `n` with entries in a finite field of order `q`. This is the same as the number of irreducible polynomials of degree *d*.

If `invertible` is `True`, then only the number of similarity classes of invertible matrices is returned.

Note: All primitive classes are invertible unless `n` is 1.

INPUT:

- `n` – a positive integer
- `invertible` – boolean; if set, only number of non-zero classes is returned
- `q` – an integer or an indeterminate

OUTPUT:

- a rational function of the variable `q`

EXAMPLES:

```
sage: from sage.combinat.similarity_class_type import primitives
sage: primitives(1)
q
sage: primitives(1, invertible = True)
q - 1
sage: primitives(4)
1/4*q^4 - 1/4*q^2
sage: primitives(4, invertible = True)
1/4*q^4 - 1/4*q^2
```

5.1.274 sine-Gordon Y-system plotter

This class builds the triangulations associated to sine-Gordon and reduced sine-Gordon Y-systems as constructed in [NS].

AUTHORS:

- Salvatore Stella (2014-07-18): initial version

EXAMPLES:

A reduced sine-Gordon example with 3 generations:

```
sage: Y = SineGordonYsystem('A',(6,4,3)); Y
A sine-Gordon Y-system of type A with defining integer tuple (6, 4, 3)
sage: Y.plot()      #not tested
```

The same integer tuple but for the non-reduced case:

```
sage: Y = SineGordonYsystem('D',(6,4,3)); Y
A sine-Gordon Y-system of type D with defining integer tuple (6, 4, 3)
sage: Y.plot()      #not tested
```

Todo

The code for plotting is extremely slow.

REFERENCES:

class sage.combinat.sine_gordon.**SineGordonYsystem**(*X*, *na*)

Bases: sage.structure.sage_object.SageObject

A class to model a (reduced) sine-Gordon Y-system

Note that the generations, together with all integer tuples, in this implementation are numbered from 0 while in [NS] they are numbered from 1

INPUT:

- X – the type of the Y-system to construct (either 'A' or 'D')
- na – the tuple of positive integers defining the Y-system with na[0] > 2

See [NS]

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3)); Y
A sine-Gordon Y-system of type A with defining integer tuple (6, 4, 3)
sage: Y.intervals()
(((0, 0, 'R'),),
 ((0, 17, 'L'),
  (17, 34, 'L'),
...
  (104, 105, 'R'),
  (105, 0, 'R')))
sage: Y.triangulation()
((17, 89),
 (17, 72),
 (34, 72),
...
 (102, 105),
 (103, 105))
sage: Y.plot()      #not tested
```

F()

Return the number of generations in self.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3));
sage: Y.F()
3
```

intervals()

Return, divided by generation, the list of intervals used to construct the initial triangulation.

Each such interval is a triple (p, q, X) where p and q are the two extremal vertices of the interval and X is the type of the interval (one of 'L', 'R', 'NL', 'NR').

ALGORITHM:

The algorithm used here is the one described in section 5.1 of [NS]. The only difference is that we get rid of the special case of the first generation by treating the whole disk as a type 'R' interval.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3));
sage: Y.intervals()
(((0, 0, 'R'),),
 ((0, 17, 'L'),
  (17, 34, 'L'),
...
  (104, 105, 'R'),
  (105, 0, 'R')))
```

na()

Return the sequence of the integers n_a defining `self`.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3));
sage: Y.na()
(6, 4, 3)
```

pa()

Return the sequence of integers `p_a`, i.e. the total number of intervals of types 'NL' and 'NR' in the `(a+1)`-th generation.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3));
sage: Y.pa()
(1, 6, 25)
```

plot(***kwds*)

Plot the initial triangulation associated to `self`.

INPUT:

- `radius` - the radius of the disk; by default the length of the circle is the number of vertices
- `points_color` - the color of the vertices; default 'black'
- `points_size` - the size of the vertices; default 7
- `triangulation_color` - the color of the arcs; default 'black'
- `triangulation_thickness` - the thickness of the arcs; default 0.5
- `shading_color` - the color of the shading used on neuter intervals; default 'lightgray'
- `reflections_color` - the color of the reflection axes; default 'blue'
- `reflections_thickness` - the thickness of the reflection axes; default 1

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3));
sage: Y.plot()      # not tested
```

qa()

Return the sequence of integers `q_a`, i.e. the total number of intervals of types 'L' and 'R' in the `(a+1)`-th generation.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3));
sage: Y.qa()
(6, 25, 81)
```

r()

Return the number of vertices in the polygon realizing `self`.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3));
sage: Y.r()
106
```

rk()

Return the sequence of integers `r^{(k)}`, i.e. the width of an interval of type 'L' or 'R' in the `k`-th generation.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3));
sage: Y.rk()
(106, 17, 4)
```

triangulation()

Return the initial triangulation of the polygon realizing `self` as a tuple of pairs of vertices.

Warning: In type 'D' the returned triangulation does NOT contain the two radii.

ALGORITHM:

We implement the four cases described by Figure 14 in [NS].

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3));
sage: Y.triangulation()
((17, 89),
 (17, 72),
...
 (102, 105),
 (103, 105))
```

type()

Return the type of `self`.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3));
sage: Y.type()
'A'
```

vertices()

Return the vertices of the polygon realizing `self` as the ring of integers modulo `self.r()`.

EXAMPLES:

```
sage: Y = SineGordonYsystem('A',(6,4,3));
sage: Y.vertices()
Ring of integers modulo 106
```

5.1.275 Six Vertex Model

class sage.combinat.six_vertex_model.**SixVertexConfiguration**
Bases: sage.structure.list_clone.ClonableArray

A configuration in the six vertex model.

check()
Check if self is a valid 6 vertex configuration.

EXAMPLES:

```
sage: M = SixVertexModel(3, boundary_conditions='ice')
sage: M[0].check()
```

energy(*epsilon*)
Return the energy of the configuration.

The energy of a configuration ν is defined as

$$E(\nu) = n_0\epsilon_0 + n_1\epsilon_1 + \cdots + n_5\epsilon_5$$

where n_i is the number of vertices of type i and ϵ_i is the i-th energy constant.

Note: We number our configurations as:

0.LR

1.LU

2.LD

3.UD

4.UR

5.RD

which differs from Wikipedia article Ice-type_model.

EXAMPLES:

```
sage: M = SixVertexModel(3, boundary_conditions='ice')
sage: nu = M[2]; nu
    ^   ^   ^
    |   |   |
--> # -> # <- # <--
    ^   |   ^
    |   V   |
--> # <- # -> # <--
    |   ^   |
    V   |   V
--> # -> # <- # <--
    |   |   |
    V   V   V
sage: nu.energy([1,2,1,2,1,2])
15
```

A KDP energy:

```
sage: nu.energy([1,1,0,1,0,1])
7
```

A Rys F energy:

```
sage: nu.energy([0,1,1,0,1,1])
4
```

The zero field assumption:

```
sage: nu.energy([1,2,3,1,3,2])
15
```

plot(*color='sign'*)

Return a plot of `self`.

INPUT:

- `color` – can be any of the following:
  - `4` - use 4 colors: black, red, blue, and green with each corresponding to up, right, down, and left respectively
  - `2` - use 2 colors: red for horizontal, blue for vertical arrows
  - `'sign'` - use red for right and down arrows, blue for left and up arrows
  - a list of 4 colors for each direction
  - a function which takes a direction and a boolean corresponding to the sign

EXAMPLES:

```
sage: M = SixVertexModel(2, boundary_conditions='ice')
sage: print M[0].plot().description()
Arrow from (-1.0,0.0) to (0.0,0.0)
Arrow from (-1.0,1.0) to (0.0,1.0)
Arrow from (0.0,0.0) to (0.0,-1.0)
Arrow from (0.0,0.0) to (1.0,0.0)
Arrow from (0.0,1.0) to (0.0,0.0)
Arrow from (0.0,1.0) to (0.0,2.0)
Arrow from (1.0,0.0) to (1.0,-1.0)
Arrow from (1.0,0.0) to (1.0,1.0)
Arrow from (1.0,1.0) to (0.0,1.0)
Arrow from (1.0,1.0) to (1.0,2.0)
Arrow from (2.0,0.0) to (1.0,0.0)
Arrow from (2.0,1.0) to (1.0,1.0)
```

to_signed_matrix()

Return the signed matrix of `self`.

The signed matrix corresponding to a six vertex configuration is given by 0 if there is a cross flow, a 1 if the outward arrows are vertical and -1 if the outward arrows are horizonal.

EXAMPLES:

```
sage: M = SixVertexModel(3, boundary_conditions='ice')
sage: map(lambda x: x.to_signed_matrix(), M)
[
[1 0 0]  [1 0 0]  [ 0  1  0]  [0 1 0]  [0 1 0]  [0 0 1]  [0 0 1]
[0 1 0]  [0 0 1]  [ 1 -1  1]  [1 0 0]  [0 0 1]  [1 0 0]  [0 1 0]
[0 0 1], [0 1 0], [ 0  1  0], [0 0 1], [1 0 0], [0 1 0], [1 0 0]
]
```

class sage.combinat.six_vertex_model.**SixVertexModel**(*n*, *m*, *boundary_conditions*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The six vertex model.

We model a configuration by indicating which configuration by the following six configurations which are determined by the two outgoing arrows in the Up, Right, Down, Left directions:

1.LR:

```
     |
     V
<-- # -->
     ^
     |
```

2.LU:

```
     ^
     |
<-- # <--
     ^
     |
```

3.LD:

```
     |
     V
<-- # <--
     |
     V
```

4.UD:

```
     ^
     |
--> # <--
     |
     V
```

5.UR:

```
     ^
     |
--> # -->
     ^
     |
```

6.RD:

```
     |
     V
--> # -->
     |
     V
```

INPUT:

- `n` – the number of rows
- `m` – (optional) the number of columns, if not specified, then the number of columns is the number of rows
- `boundary_conditions` – (optional) a quadruple of tuples whose entries are either:
  - `True` for an inward arrow,
  - `False` for an outward arrow, or
  - `None` for no boundary condition.

  There are also the following predefined boundary conditions:
  - `'ice'` - The top and bottom boundary conditions are outward and the left and right boundary conditions are inward; this gives the square ice model. Also called domain wall boundary conditions.
  - `'domain wall'` - Same as `'ice'`.
  - `'alternating'` - The boundary conditions alternate between inward and outward.
  - `'free'` - There are no boundary conditions.

EXAMPLES:

Here are the six types of vertices that can be created:

```
sage: M = SixVertexModel(1)
sage: list(M)
[
    |           ^           |           ^           ^           |
    V           |           V           |           |           V
<-- # -->   <-- # <--   <-- # <--   --> # <--   --> # -->   --> # -->
    ^           ^           |           |           ^           |
    |     ,     |     ,     V     ,     V     ,     |     ,     V
]
```

When using the square ice model, it is known that the number of configurations is equal to the number of alternating sign matrices:

```
sage: M = SixVertexModel(1, boundary_conditions='ice')
sage: len(M)
1
sage: M = SixVertexModel(4, boundary_conditions='ice')
sage: len(M)
42
sage: all(len(SixVertexModel(n, boundary_conditions='ice'))
....:     == AlternatingSignMatrices(n).cardinality() for n in range(1, 7))
True
```

An example with a specified non-standard boundary condition and non-rectangular shape:

```
sage: M = SixVertexModel(2, 1, [[None], [True,True], [None], [None,None]])
sage: list(M)
[
    ^           ^           |           ^
    |           |           V           |
<-- # <--   <-- # <--   <-- # <--   --> # <--
    ^           ^           |           |
    |           |           V           V
<-- # <--   --> # <--   <-- # <--   <-- # <--
    ^           |           |           |
```

```
    |    ,    V    ,    V    ,    V
]
```

REFERENCES:

- Wikipedia article Vertex_model
- Wikipedia article Ice-type_model

Element
alias of SixVertexConfiguration

boundary_conditions()
Return the boundary conditions of self.

EXAMPLES:

```
sage: M = SixVertexModel(2, boundary_conditions='ice')
sage: M.boundary_conditions()
((False, False), (True, True), (False, False), (True, True))
```

partition_function(*beta*, *epsilon*)
Return the partition function of self.

The partition function of a 6 vertex model is defined by:

$$Z = \sum_{\nu} e^{-\beta E(\nu)}$$

where we sum over all configurations and E is the energy function. The constant β is known as the *inverse temperature* and is equal to $1/k_B T$ where k_B is Boltzmann's constant and T is the system's temperature.

INPUT:

- beta – the inverse temperature constant β
- epsilon – the energy constants, see energy()

EXAMPLES:

```
sage: M = SixVertexModel(3, boundary_conditions='ice')
sage: M.partition_function(2, [1,2,1,2,1,2])
e^(-24) + 2*e^(-28) + e^(-30) + 2*e^(-32) + e^(-36)
```

REFERENCES:

Wikipedia article Partition_function_(statistical_mechanics)

class sage.combinat.six_vertex_model.**SquareIceModel**(*n*)
Bases: sage.combinat.six_vertex_model.SixVertexModel

The square ice model.

The square ice model is a 6 vertex model on an $n \times n$ grid with the boundary conditions that the top and bottom boundaries are pointing outward and the left and right boundaries are pointing inward. These boundary conditions are also called domain wall boundary conditions.

Configurations of the 6 vertex model with domain wall boundary conditions are in bijection with alternating sign matrices.

class Element
Bases: sage.combinat.six_vertex_model.SixVertexConfiguration

An element in the square ice model.

to_alternating_sign_matrix()
Return an alternating sign matrix of self.

See also:

to_signed_matrix()

EXAMPLES:

```
sage: M = SixVertexModel(4, boundary_conditions='ice')
sage: M[6].to_alternating_sign_matrix()
[1 0 0 0]
[0 0 0 1]
[0 0 1 0]
[0 1 0 0]
sage: M[7].to_alternating_sign_matrix()
[ 0  1  0  0]
[ 1 -1  1  0]
[ 0  1 -1  1]
[ 0  0  1  0]
```

SquareIceModel.**from_alternating_sign_matrix**(*asm*)
Return a configuration from the alternating sign matrix asm.

EXAMPLES:

```
sage: M = SixVertexModel(3, boundary_conditions='ice')
sage: asm = AlternatingSignMatrix([[0,1,0],[1,-1,1],[0,1,0]])
sage: M.from_alternating_sign_matrix(asm)
    ^    ^    ^
    |    |    |
--> # -> # <- # <--
    ^    |    ^
    |    V    |
--> # <- # -> # <--
    |    ^    |
    V    |    V
--> # -> # <- # <--
    |    |    |
    V    V    V
```

TESTS:

```
sage: M = SixVertexModel(5, boundary_conditions='ice')
sage: ASM = AlternatingSignMatrices(5)
sage: all(M.from_alternating_sign_matrix(x.to_alternating_sign_matrix()) == x
....:     for x in M)
True
sage: all(M.from_alternating_sign_matrix(x).to_alternating_sign_matrix() == x
....:     for x in ASM)
True
```

5.1.276 Skew Partitions

A skew partition skp of size n is a pair of partitions $[p_1, p_2]$ where p_1 is a partition of the integer n_1, p_2 is a partition of the integer n_2, p_2 is an inner partition of p_1, and $n = n_1 - n_2$. We say that p_1 and p_2 are respectively the *inner* and *outer* partitions of skp.

A skew partition can be depicted by a diagram made of rows of cells, in the same way as a partition. Only the cells of the outer partition p_1 which are not in the inner partition p_2 appear in the picture. For example, this is the diagram of

the skew partition [[5,4,3,1],[3,3,1]].

```
sage: print SkewPartition([[5,4,3,1],[3,3,1]]).diagram()
   **
   *
 **
*
```

A skew partition can be *connected*, which can easily be described in graphic terms: for each pair of consecutive rows, there are at least two cells (one in each row) which have a common edge. This is the diagram of the connected skew partition `[[5,4,3,1],[3,1]]`:

```
sage: print SkewPartition([[5,4,3,1],[3,1]]).diagram()
   **
 ***
***
*
sage: SkewPartition([[5,4,3,1],[3,1]]).is_connected()
True
```

The first example of a skew partition is not a connected one.

Applying a reflection with respect to the main diagonal yields the diagram of the *conjugate skew partition*, here `[[4,3,3,2,1],[3,3,2]]`:

```
sage: SkewPartition([[5,4,3,1],[3,3,1]]).conjugate()
[4, 3, 3, 2, 1] / [3, 2, 2]
sage: print SkewPartition([[5,4,3,1],[3,3,1]]).conjugate().diagram()
   *
  *
  *
**
*
```

The *outer corners* of a skew partition are the corners of its outer partition. The *inner corners* are the internal corners of the outer partition when the inner partition is taken off. Shown below are the coordinates of the inner and outer corners.

```
sage: SkewPartition([[5,4,3,1],[3,3,1]]).outer_corners()
[(0, 4), (1, 3), (2, 2), (3, 0)]
sage: SkewPartition([[5,4,3,1],[3,3,1]]).inner_corners()
[(0, 3), (2, 1), (3, 0)]
```

EXAMPLES:

There are 9 skew partitions of size 3, with no empty row nor empty column:

```
sage: SkewPartitions(3).cardinality()
9
sage: SkewPartitions(3).list()
[[3] / [],
 [2, 1] / [],
 [3, 1] / [1],
 [2, 2] / [1],
 [3, 2] / [2],
 [1, 1, 1] / [],
 [2, 2, 1] / [1, 1],
 [2, 1, 1] / [1],
 [3, 2, 1] / [2, 1]]
```

There are 4 connected skew partitions of size 3:

```
sage: SkewPartitions(3, overlap=1).cardinality()
4
sage: SkewPartitions(3, overlap=1).list()
[[3] / [], [2, 1] / [], [2, 2] / [1], [1, 1, 1] / []]
```

This is the conjugate of the skew partition `[[4,3,1], [2]]`

```
sage: SkewPartition([[4,3,1], [2]]).conjugate()
[3, 2, 2, 1] / [1, 1]
```

Geometrically, we just applied a reflection with respect to the main diagonal on the diagram of the partition. Of course, this operation is an involution:

```
sage: SkewPartition([[4,3,1],[2]]).conjugate().conjugate()
[4, 3, 1] / [2]
```

The `jacobi_trudi()` method computes the Jacobi-Trudi matrix. See [Mac95] for a definition and discussion.

```
sage: SkewPartition([[4,3,1],[2]]).jacobi_trudi()
[h[2]  h[]    0]
[h[5] h[3]  h[]]
[h[6] h[4] h[1]]
```

This example shows how to compute the corners of a skew partition.

```
sage: SkewPartition([[4,3,1],[2]]).inner_corners()
[(0, 2), (1, 0)]
sage: SkewPartition([[4,3,1],[2]]).outer_corners()
[(0, 3), (1, 2), (2, 0)]
```

REFERENCES:

AUTHORS:

- Mike Hansen: Initial version
- Travis Scrimshaw (2013-02-11): Factored out `CombinatorialClass`

class sage.combinat.skew_partition.**SkewPartition**(*parent*, *skp*)
Bases: sage.combinat.combinat.CombinatorialElement

A skew partition.

A skew partition of shape λ/μ is the Young diagram from the partition λ and removing the partition μ from the upper-left corner in English convention.

cell_poset(*orientation='SE'*)
Return the Young diagram of `self` as a poset. The optional keyword variable `orientation` determines the order relation of the poset.

The poset always uses the set of cells of the Young diagram of `self` as its ground set. The order relation of the poset depends on the `orientation` variable (which defaults to `"SE"`). Concretely, `orientation` has to be specified to one of the strings `"NW"`, `"NE"`, `"SW"`, and `"SE"`, standing for "northwest", "northeast", "southwest" and "southeast", respectively. If `orientation` is `"SE"`, then the order relation of the poset is such that a cell u is greater or equal to a cell v in the poset if and only if u lies weakly southeast of v (this means that u can be reached from v by a sequence of south and east steps; the sequence is allowed to consist of south steps only, or of east steps only, or even be empty). Similarly the order relation is defined for the other three orientations. The Young diagram is supposed to be drawn in English notation.

The elements of the poset are the cells of the Young diagram of `self`, written as tuples of zero-based coordinates (so that $(3, 7)$ stands for the 8-th cell of the 4-th row, etc.).

EXAMPLES:

```
sage: p = SkewPartition([[3,3,1], [2,1]])
sage: Q = p.cell_poset(); Q
Finite poset containing 4 elements
sage: sorted(Q)
[(0, 2), (1, 1), (1, 2), (2, 0)]
sage: sorted(Q.maximal_elements())
[(1, 2), (2, 0)]
sage: sorted(Q.minimal_elements())
[(0, 2), (1, 1), (2, 0)]
sage: sorted(Q.upper_covers((1, 1)))
[(1, 2)]
sage: sorted(Q.upper_covers((0, 2)))
[(1, 2)]

sage: P = p.cell_poset(orientation="NW"); P
Finite poset containing 4 elements
sage: sorted(P)
[(0, 2), (1, 1), (1, 2), (2, 0)]
sage: sorted(P.minimal_elements())
[(1, 2), (2, 0)]
sage: sorted(P.maximal_elements())
[(0, 2), (1, 1), (2, 0)]
sage: sorted(P.upper_covers((1, 2)))
[(0, 2), (1, 1)]

sage: R = p.cell_poset(orientation="NE"); R
Finite poset containing 4 elements
sage: sorted(R)
[(0, 2), (1, 1), (1, 2), (2, 0)]
sage: R.maximal_elements()
[(0, 2)]
sage: R.minimal_elements()
[(2, 0)]
sage: R.upper_covers((2, 0))
[(1, 1)]
sage: sorted([len(R.upper_covers(v)) for v in R])
[0, 1, 1, 1]
```

TESTS:

We check that the posets are really what they should be for size up to 6:

```
sage: def check_NW(n):
....:     for p in SkewPartitions(n):
....:         P = p.cell_poset(orientation="NW")
....:         for c in p.cells():
....:             for d in p.cells():
....:                 if P.le(c, d) != (c[0] >= d[0]
....:                                   and c[1] >= d[1]):
....:                     return False
....:     return True
sage: all( check_NW(n) for n in range(7) )
True

sage: def check_NE(n):
```

```
....:     for p in SkewPartitions(n):
....:         P = p.cell_poset(orientation="NE")
....:         for c in p.cells():
....:             for d in p.cells():
....:                 if P.le(c, d) != (c[0] >= d[0]
....:                                   and c[1] <= d[1]):
....:                     return False
....:     return True
sage: all( check_NE(n) for n in range(7) )
True

sage: def test_duality(n, ori1, ori2):
....:     for p in SkewPartitions(n):
....:         P = p.cell_poset(orientation=ori1)
....:         Q = p.cell_poset(orientation=ori2)
....:         for c in p.cells():
....:             for d in p.cells():
....:                 if P.lt(c, d) != Q.lt(d, c):
....:                     return False
....:     return True
sage: all( test_duality(n, "NW", "SE") for n in range(7) )
True
sage: all( test_duality(n, "NE", "SW") for n in range(7) )
True
sage: all( test_duality(n, "NE", "SE") for n in range(4) )
False
```

cells()

Return the coordinates of the cells of `self`. Coordinates are given as `(row-index, column-index)` and are 0 based.

EXAMPLES:

```
sage: SkewPartition([[4, 3, 1], [2]]).cells()
[(0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (2, 0)]
sage: SkewPartition([[4, 3, 1], []]).cells()
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (2, 0)]
sage: SkewPartition([[2], []]).cells()
[(0, 0), (0, 1)]
```

column_lengths()

Return the column lengths of `self`.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[1,1]]).column_lengths()
[1, 2, 1]
sage: SkewPartition([[5,2,2,2],[2,1]]).column_lengths()
[2, 3, 1, 1, 1]
```

columns_intersection_set()

Return the set of cells in the columns of the outer shape of `self` which columns intersect the skew diagram of `self`.

EXAMPLES:

```
sage: skp = SkewPartition([[3,2,1],[2,1]])
sage: cells = Set([ (0,0), (0, 1), (0,2), (1, 0), (1, 1), (2, 0)])
sage: skp.columns_intersection_set() == cells
True
```

conjugate()
Return the conjugate of the skew partition skp.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[2]]).conjugate()
[3, 2, 1] / [1, 1]
```

diagram()
Return the Ferrers diagram of `self`.

EXAMPLES:

```
sage: print SkewPartition([[5,4,3,1],[3,3,1]]).ferrers_diagram()
   **
    *
 **
*
sage: print SkewPartition([[5,4,3,1],[3,1]]).diagram()
   **
 ***
***
*
sage: SkewPartitions.global_options(diagram_str='#', convention="French")
sage: print SkewPartition([[5,4,3,1],[3,1]]).diagram()
#
###
 ###
   ##
sage: SkewPartitions.global_options.reset()
```

ferrers_diagram()
Return the Ferrers diagram of `self`.

EXAMPLES:

```
sage: print SkewPartition([[5,4,3,1],[3,3,1]]).ferrers_diagram()
   **
    *
 **
*
sage: print SkewPartition([[5,4,3,1],[3,1]]).diagram()
   **
 ***
***
*
sage: SkewPartitions.global_options(diagram_str='#', convention="French")
sage: print SkewPartition([[5,4,3,1],[3,1]]).diagram()
#
###
 ###
   ##
sage: SkewPartitions.global_options.reset()
```

frobenius_rank()
Return the Frobenius rank of the skew partition `self`.

The Frobenius rank of a skew partition λ/μ can be defined in various ways. The quickest one is probably the following: Writing λ as $(\lambda_1, \lambda_2, \cdots, \lambda_N)$, and writing μ as $(\mu_1, \mu_2, \cdots, \mu_N)$, we define the Frobenius

rank of λ/μ to be the number of all $1 \leq i \leq N$ such that

$$\lambda_i - i \notin \{\mu_1 - 1, \mu_2 - 2, \cdots, \mu_N - N\}.$$

In other words, the Frobenius rank of λ/μ is the number of rows in the Jacobi-Trudi matrix of λ/μ which don't contain h_0. Further definitions have been considered in [Stan2002] (where Frobenius rank is just being called rank).

If μ is the empty shape, then the Frobenius rank of λ/μ is just the usual Frobenius rank of the partition λ (see `frobenius_rank()`).

REFERENCES:

EXAMPLES:

```
sage: SkewPartition([[8,8,7,4], [4,1,1]]).frobenius_rank()
4
sage: SkewPartition([[2,1], [1]]).frobenius_rank()
2
sage: SkewPartition([[2,1,1], [1]]).frobenius_rank()
2
sage: SkewPartition([[2,1,1], [1,1]]).frobenius_rank()
2
sage: SkewPartition([[5,4,3,2], [2,1,1]]).frobenius_rank()
3
sage: SkewPartition([[4,2,1], [3,1,1]]).frobenius_rank()
2
sage: SkewPartition([[4,2,1], [3,2,1]]).frobenius_rank()
1
```

If the inner shape is empty, then the Frobenius rank of the skew partition is just the standard Frobenius rank of the partition:

```
sage: all( SkewPartition([lam, Partition([])]).frobenius_rank()
....:      == lam.frobenius_rank() for i in range(6)
....:      for lam in Partitions(i) )
True
```

If the inner and outer shapes are equal, then the Frobenius rank is zero:

```
sage: all( SkewPartition([lam, lam]).frobenius_rank() == 0
....:      for i in range(6) for lam in Partitions(i) )
True
```

inner()

Return the inner partition of `self`.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[1,1]]).inner()
[1, 1]
```

inner_corners()

Return a list of the inner corners of `self`.

EXAMPLES:

```
sage: SkewPartition([[4, 3, 1], [2]]).inner_corners()
[(0, 2), (1, 0)]
sage: SkewPartition([[4, 3, 1], []]).inner_corners()
[(0, 0)]
```

is_connected()
Return `True` if `self` is a connected skew partition.

A skew partition is said to be *connected* if for each pair of consecutive rows, there are at least two cells (one in each row) which have a common edge.

EXAMPLES:

```
sage: SkewPartition([[5,4,3,1],[3,3,1]]).is_connected()
False
sage: SkewPartition([[5,4,3,1],[3,1]]).is_connected()
True
```

is_overlap(*n*)
Return `True` if the overlap of `self` is at most `n`.

See also:

`overlap()`

EXAMPLES:

```
sage: SkewPartition([[5,4,3,1],[3,1]]).is_overlap(1)
True
```

is_ribbon()
Return `True` if and only if `self` is a ribbon, that is, if it has exactly one cell in each of q consecutive diagonals for some nonnegative integer q.

EXAMPLES:

```
sage: P=SkewPartition([[4,4,3,3],[3,2,2]])
sage: P.pp()
   *
  **
  *
***
sage: P.is_ribbon()
True

sage: P=SkewPartition([[4,3,3],[1,1]])
sage: P.pp()
 ***
 **
***
sage: P.is_ribbon()
False

sage: P=SkewPartition([[4,4,3,2],[3,2,2]])
sage: P.pp()
   *
  **
  *
**
sage: P.is_ribbon()
False

sage: P=SkewPartition([[4,4,3,3],[4,2,2,1]])
sage: P.pp()

  **
  *
```

```
 **
sage: P.is_ribbon()
True

sage: P=SkewPartition([[4,4,3,3],[4,2,2]])
sage: P.pp()

  **
  *
***
sage: P.is_ribbon()
True

sage: SkewPartition([[2,2,1],[2,2,1]]).is_ribbon()
True
```

jacobi_trudi()

Return the Jacobi-Trudi matrix of `self`.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[2,1]]).jacobi_trudi()
[h[1]    0    0]
[h[3] h[1]    0]
[h[5] h[3] h[1]]
sage: SkewPartition([[4,3,2],[2,1]]).jacobi_trudi()
[h[2]  h[]    0]
[h[4] h[2]  h[]]
[h[6] h[4] h[2]]
```

k_conjugate(k)

Return the k-conjugate of the skew partition.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[2,1]]).k_conjugate(3)
[2, 1, 1, 1, 1] / [2, 1]
sage: SkewPartition([[3,2,1],[2,1]]).k_conjugate(4)
[2, 2, 1, 1] / [2, 1]
sage: SkewPartition([[3,2,1],[2,1]]).k_conjugate(5)
[3, 2, 1] / [2, 1]
```

outer()

Return the outer partition of `self`.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[1,1]]).outer()
[3, 2, 1]
```

outer_corners()

Return a list of the outer corners of `self`.

EXAMPLES:

```
sage: SkewPartition([[4, 3, 1], [2]]).outer_corners()
[(0, 3), (1, 2), (2, 0)]
```

overlap()

Return the overlap of `self`.

The overlap of two consecutive rows in a skew partition is the number of pairs of cells (one in each row) that share a common edge. This number can be positive, zero, or negative.

The overlap of a skew partition is the minimum of the overlap of the consecutive rows, or infinity in the case of at most one row. If the overlap is positive, then the skew partition is called *connected*.

EXAMPLES:

```
sage: SkewPartition([[],[]]).overlap()
+Infinity
sage: SkewPartition([[1],[]]).overlap()
+Infinity
sage: SkewPartition([[10],[]]).overlap()
+Infinity
sage: SkewPartition([[10],[2]]).overlap()
+Infinity
sage: SkewPartition([[10,1],[2]]).overlap()
-1
sage: SkewPartition([[10,10],[1]]).overlap()
9
```

pieri_macdonald_coeffs()

Computation of the coefficients which appear in the Pieri formula for Macdonald polynomials given in his book (Chapter 6.6 formula 6.24(ii))

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[2,1]]).pieri_macdonald_coeffs()
1
sage: SkewPartition([[3,2,1],[2,2]]).pieri_macdonald_coeffs()
(q^2*t^3 - q^2*t - t^2 + 1)/(q^2*t^3 - q*t^2 - q*t + 1)
sage: SkewPartition([[3,2,1],[2,2,1]]).pieri_macdonald_coeffs()
(q^6*t^8 - q^6*t^6 - q^4*t^7 - q^5*t^5 + q^4*t^5 - q^3*t^6 + q^5*t^3 + 2*q^3*t^4 + q*t^5 - q
sage: SkewPartition([[3,3,2,2],[3,2,2,1]]).pieri_macdonald_coeffs()
(q^5*t^6 - q^5*t^5 + q^4*t^6 - q^4*t^5 - q^4*t^3 + q^4*t^2 - q^3*t^3 - q^2*t^4 + q^3*t^2 + q
```

pp()

Pretty-print `self`.

EXAMPLES:

```
sage: SkewPartition([[5,4,3,1],[3,3,1]]).pp()
   **
   *
 **
*
```

quotient(*k*)

The quotient map extended to skew partitions.

EXAMPLES:

```
sage: SkewPartition([[3, 3, 2, 1], [2, 1]]).quotient(2)
[[3] / [], [] / []]
```

row_lengths()

Return the row lengths of `self`.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[1,1]]).row_lengths()
[2, 1, 1]
```

rows_intersection_set()
Return the set of cells in the rows of the outer shape of `self` which rows intersect the skew diagram of `self`.

EXAMPLES:

```
sage: skp = SkewPartition([[3,2,1],[2,1]])
sage: cells = Set([ (0,0), (0, 1), (0,2), (1, 0), (1, 1), (2, 0)])
sage: skp.rows_intersection_set() == cells
True
```

size()
Return the size of `self`.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[1,1]]).size()
4
```

to_dag(*format='string'*)
Return a directed acyclic graph corresponding to the skew partition `self`.

The directed acyclic graph corresponding to a skew partition p is the digraph whose vertices are the cells of p, and whose edges go from each cell to its lower and right neighbors (in English notation).

INPUT:

- `format` – either `'string'` or `'tuple'` (default: `'string'`); determines whether the vertices of the resulting dag will be strings or 2-tuples of coordinates

EXAMPLES:

```
sage: dag = SkewPartition([[3, 3, 1], [1, 1]]).to_dag()
sage: dag.edges()
[('0,1', '0,2', None),
('0,1', '1,1', None),
('0,2', '1,2', None),
('1,1', '1,2', None)]
sage: dag.vertices()
['0,1', '0,2', '1,1', '1,2', '2,0']
sage: dag = SkewPartition([[3, 2, 1], [1, 1]]).to_dag(format="tuple")
sage: dag.edges()
[((0, 1), (0, 2), None), ((0, 1), (1, 1), None)]
sage: dag.vertices()
[(0, 1), (0, 2), (1, 1), (2, 0)]
```

to_list()
Return `self` as a list of lists.

EXAMPLES:

```
sage: s = SkewPartition([[4,3,1],[2]])
sage: s.to_list()
[[4, 3, 1], [2]]
sage: type(s.to_list())
<type 'list'>
```

class sage.combinat.skew_partition.**SkewPartitions**(*is_infinite=False*)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Skew partitions.

Warning: The iterator of this class only yields skew partitions which are reduced, in the sense that there are no empty rows before the last nonempty row, and there are no empty columns before the last nonempty column.

EXAMPLES:

```
sage: SkewPartitions(4)
Skew partitions of 4
sage: SkewPartitions(4).cardinality()
28
sage: SkewPartitions(row_lengths=[2,1,2])
Skew partitions with row lengths [2, 1, 2]
sage: SkewPartitions(4, overlap=2)
Skew partitions of 4 with a minimum overlap of 2
sage: SkewPartitions(4, overlap=2).list()
[[4] / [], [2, 2] / []]
```

Element
alias of SkewPartition

from_row_and_column_length(*rowL*, *colL*)
Construct a partition from its row lengths and column lengths.

INPUT:

- rowL – A composition or a list of positive integers
- colL – A composition or a list of positive integers

OUTPUT:

- If it exists the unique skew-partitions with row lengths rowL and column lengths colL.
- Raise a ValueError if rowL and colL are not compatible.

EXAMPLES:

```
sage: S = SkewPartitions()
sage: print S.from_row_and_column_length([3,1,2,2],[2,3,1,1,1]).diagram()
  ***
 *
**
**
sage: S.from_row_and_column_length([],[])
[] / []
sage: S.from_row_and_column_length([1],[1])
[1] / []
sage: S.from_row_and_column_length([2,1],[2,1])
[2, 1] / []
sage: S.from_row_and_column_length([1,2],[1,2])
[2, 2] / [1]
sage: S.from_row_and_column_length([1,2],[1,3])
Traceback (most recent call last):
...
ValueError: Sum mismatch : [1, 2] and [1, 3]
sage: S.from_row_and_column_length([3,2,1,2],[2,3,1,1,1])
Traceback (most recent call last):
...
ValueError: Incompatible row and column length : [3, 2, 1, 2] and [2, 3, 1, 1, 1]
```

Warning: If some rows and columns have length zero, there is no way to retrieve unambiguously the skew partition. We therefore raise a `ValueError`. For examples here are two skew partitions with the same row and column lengths:

```
sage: skp1 = SkewPartition([[2,2],[2,2]])
sage: skp2 = SkewPartition([[2,1],[2,1]])
sage: skp1.row_lengths(), skp1.column_lengths()
([0, 0], [0, 0])
sage: skp2.row_lengths(), skp2.column_lengths()
([0, 0], [0, 0])
sage: SkewPartitions().from_row_and_column_length([0,0], [0,0])
Traceback (most recent call last):
...
ValueError: row and column length must be positive
```

TESTS:

```
sage: all(SkewPartitions().from_row_and_column_length(p.row_lengths(), p.column_lengths()) =
....:       for i in range(8) for p in SkewPartitions(i))
True
```

global_options(**get_value*, ***set_value*)

Sets and displays the global options for elements of the skew partition classes. If no parameters are set, then the function returns a copy of the options dictionary.

The `options` to skew partitions can be accessed as the method `SkewPartitions.global_options` of `SkewPartitions` and related parent classes.

OPTIONS:

- `convention` – (default: `English`) Sets the convention used for displaying tableaux and partitions
  - `English` – use the English convention
  - `French` – use the French convention
- `diagram_str` – (default: `*`) The character used for the cells when printing Ferrers diagrams
- `display` – (default: `quotient`) Specifies how skew partitions should be printed
  - `array` – alias for `diagram`
  - `diagram` – as a skew Ferrers diagram
  - `ferrers_diagram` – alias for `diagram`
  - `lists` – displayed as a pair of lists
  - `pair` – alias for `lists`
  - `quotient` – displayed as a quotient of partitions
  - `young_diagram` – alias for `diagram`
- `latex` – (default: `young_diagram`) Specifies how skew partitions should be latexed
  - `array` – alias for `diagram`
  - `diagram` – latex as a skew Ferrers diagram
  - `ferrers_diagram` – alias for `diagram`
  - `marked` – latex as a partition where the skew shape is marked

– young_diagram – latex as a skew Young diagram

- latex_diagram_str – (default: \ast) The character used for the cells when latexing Ferrers diagrams
- latex_marking_str – (default: X) The character used to marked the deleted cells when latexing marked partitions
- notation – alternative name for convention

EXAMPLES:

```
sage: SP = SkewPartition([[4,2,2,1], [3, 1, 1]])
sage: SP
[4, 2, 2, 1] / [3, 1, 1]
sage: SkewPartitions.global_options(display="lists")
sage: SP
[[4, 2, 2, 1], [3, 1, 1]]
```

Changing the convention for skew partitions also changes the convention option for partitions and tableaux and vice versa:

```
sage: SkewPartitions.global_options(display="diagram", convention='French')
sage: SP
*
 *
 *
   *
sage: T = Tableau([[1,2,3],[4,5]])
sage: T.pp()
  4  5
  1  2  3
sage: P = Partition([4, 2, 2, 1])
sage: P.pp()
*
**
**
****
sage: Tableaux.global_options(convention="english")
sage: SP
   *
 *
 *
*
sage: T.pp()
  1  2  3
  4  5
sage: SkewPartitions.global_options.reset()
```

See GlobalOptions for more features of these options.

class sage.combinat.skew_partition.**SkewPartitions_all**

Bases: sage.combinat.skew_partition.SkewPartitions

Class of all skew partitions.

class sage.combinat.skew_partition.**SkewPartitions_n**(*n*, *overlap*)

Bases: sage.combinat.skew_partition.SkewPartitions

The set of skew partitions of n with overlap at least overlap and no empty row.

INPUT:

- n – a non-negative integer
- overlap – an integer (default: 0)

Caveat: this set is stable under conjugation only for overlap equal to 0 or 1. What exactly happens for negative overlaps is not yet well specified and subject to change (we may want to introduce vertical overlap constraints as well).

Todo

As is, this set is essentially the composition of Compositions(n) (which give the row lengths) and SkewPartition(n, row_lengths=...), and one would want to "inherit" list and cardinality from this composition.

---

cardinality()

Return the number of skew partitions of the integer n (with given overlap, if specified; and with no empty rows before the last row).

EXAMPLES:

```
sage: SkewPartitions(0).cardinality()
1
sage: SkewPartitions(4).cardinality()
28
sage: SkewPartitions(5).cardinality()
87
sage: SkewPartitions(4, overlap=1).cardinality()
9
sage: SkewPartitions(5, overlap=1).cardinality()
20
sage: s = SkewPartitions(5, overlap=-1)
sage: s.cardinality() == len(s.list())
True
```

class sage.combinat.skew_partition.**SkewPartitions_rowlengths**(*co*, *overlap*)

Bases: sage.combinat.skew_partition.SkewPartitions

All skew partitions with given row lengths.

sage.combinat.skew_partition.**row_lengths_aux**(*skp*)

EXAMPLES:

```
sage: from sage.combinat.skew_partition import row_lengths_aux
sage: row_lengths_aux([[5,4,3,1],[3,3,1]])
[2, 1, 2]
sage: row_lengths_aux([[5,4,3,1],[3,1]])
[2, 3]
```

5.1.277 Skew Tableaux

AUTHORS:

- Mike Hansen: Initial version
- Travis Scrimshaw, Arthur Lubovsky (2013-02-11): Factored out CombinatorialClass

class sage.combinat.skew_tableau.**SemistandardSkewTableaux**(*category=None*)

Bases: sage.combinat.skew_tableau.SkewTableaux

Semistandard skew tableaux.

This class can be initialized with several optional variables: the size of the skew tableaux (as a nameless integer variable), their shape (as a nameless skew partition variable), their weight (`weight()`, as a nameless second variable after either the size or the shape) and their maximum entry (as an optional keyword variable called `max_entry`, unless the weight has been specified). If neither the weight nor the maximum entry is specified, the maximum entry defaults to the size of the tableau.

Note that "maximum entry" does not literally mean the highest entry; instead it is just an upper bound that no entry is allowed to surpass.

EXAMPLES:

The (infinite) class of all semistandard skew tableaux:

```
sage: SemistandardSkewTableaux()
Semistandard skew tableaux
```

The (still infinite) class of all semistandard skew tableaux with maximum entry 2:

```
sage: SemistandardSkewTableaux(max_entry=2)
Semistandard skew tableaux with maximum entry 2
```

The class of all semistandard skew tableaux of given size 3 and maximum entry 3:

```
sage: SemistandardSkewTableaux(3)
Semistandard skew tableaux of size 3 and maximum entry 3
```

To set a different maximum entry:

```
sage: SemistandardSkewTableaux(3, max_entry = 7)
Semistandard skew tableaux of size 3 and maximum entry 7
```

Specifying a shape:

```
sage: SemistandardSkewTableaux([[2,1],[]])
Semistandard skew tableaux of shape [2, 1] / [] and maximum entry 3
```

Specifying both a shape and a maximum entry:

```
sage: S = SemistandardSkewTableaux([[2,1],[1]], max_entry = 3); S
Semistandard skew tableaux of shape [2, 1] / [1] and maximum entry 3
sage: S.list()
[[[None, 1], [1]],
 [[None, 2], [1]],
 [[None, 1], [2]],
 [[None, 3], [1]],
 [[None, 1], [3]],
 [[None, 2], [2]],
 [[None, 3], [2]],
 [[None, 2], [3]],
 [[None, 3], [3]]]

sage: for n in range(5):
....:     print n, len(SemistandardSkewTableaux([[2,2,1],[1]], max_entry = n))
0 0
1 0
2 1
3 9
4 35
```

Specifying a shape and a weight:

```
sage: SemistandardSkewTableaux([[2,1],[]],[2,1])
Semistandard skew tableaux of shape [2, 1] / [] and weight [2, 1]
```

(the maximum entry is redundant in this case and thus is ignored).

Specifying a size and a weight:

```
sage: SemistandardSkewTableaux(3, [2,1])
Semistandard skew tableaux of size 3 and weight [2, 1]
```

Warning: If the shape is not specified, the iterator of this class yields only skew tableaux whose shape is reduced, in the sense that there are no empty rows before the last nonempty row, and there are no empty columns before the last nonempty column. (Otherwise it would go on indefinitely.)

Warning: This class acts as a factory. The resulting classes are mainly useful for iteration. Do not rely on their containment tests, as they are not correct, e. g.:

```
sage: SkewTableau([[None]]) in SemistandardSkewTableaux(2)
True
```

class sage.combinat.skew_tableau.**SemistandardSkewTableaux_all**(*max_entry*)
Bases: sage.combinat.skew_tableau.SemistandardSkewTableaux

Class of all semistandard skew tableaux, possibly with a given maximum entry.

class sage.combinat.skew_tableau.**SemistandardSkewTableaux_shape**(*p*, *max_entry*)
Bases: sage.combinat.skew_tableau.SemistandardSkewTableaux

Class of semistandard skew tableaux of a fixed skew shape λ/μ with a given max entry.

A semistandard skew tableau with max entry i is required to have all its entries less or equal to i. It is not required to actually contain an entry i.

INPUT:

- p – A skew partition
- max_entry – The max entry; defaults to the size of p.

Warning: Input is not checked; please use SemistandardSkewTableaux to ensure the options are properly parsed.

cardinality()
EXAMPLES:

```
sage: SemistandardSkewTableaux([[2,1],[]]).cardinality()
8
sage: SemistandardSkewTableaux([[2,1],[]], max_entry=2).cardinality()
2
```

class sage.combinat.skew_tableau.**SemistandardSkewTableaux_shape_weight**(*p*, *mu*)
Bases: sage.combinat.skew_tableau.SemistandardSkewTableaux

Class of semistandard skew tableaux of a fixed skew shape λ/ν and weight μ.

class sage.combinat.skew_tableau.**SemistandardSkewTableaux_size**(*n*, *max_entry*)
Bases: sage.combinat.skew_tableau.SemistandardSkewTableaux

Class of all semistandard skew tableaux of a fixed size n, possibly with a given maximum entry.

cardinality()
EXAMPLES:
```
sage: SemistandardSkewTableaux(2).cardinality()
8
```

class sage.combinat.skew_tableau.**SemistandardSkewTableaux_size_weight**(*n*, *mu*)
Bases: sage.combinat.skew_tableau.SemistandardSkewTableaux

Class of semistandard tableaux of a fixed size n and weight μ.

cardinality()
EXAMPLES:
```
sage: SemistandardSkewTableaux(2,[1,1]).cardinality()
4
```

class sage.combinat.skew_tableau.**SkewTableau**(*parent*, *st*)
Bases: sage.structure.list_clone.ClonableList

A skew tableau.

Note that Sage by default uses the English convention for partitions and tableaux. To change this, see TableauOptions.

EXAMPLES:
```
sage: st = SkewTableau([[None, 1],[2,3]]); st
[[None, 1], [2, 3]]
sage: st.inner_shape()
[1]
sage: st.outer_shape()
[2, 2]
```

The expr form of a skew tableau consists of the inner partition followed by a list of the entries in each row from bottom to top:
```
sage: SkewTableau(expr=[[1,1],[[5],[3,4],[1,2]]])
[[None, 1, 2], [None, 3, 4], [5]]
```

bender_knuth_involution(*k*, *rows=None*, *check=True*)
Return the image of self under the k-th Bender–Knuth involution, assuming self is a skew semistandard tableau.

Let T be a tableau, then a *lower free 'k' in 'T'* means a cell of T which is filled with the integer k and whose direct lower neighbor is not filled with the integer $k+1$ (in particular, this lower neighbor might not exist at all). Let an *upper free 'k + 1' in 'T'* mean a cell of T which is filled with the integer $k+1$ and whose direct upper neighbor is not filled with the integer k (in particular, this neighbor might not exist at all). It is clear that for any row r of T, the lower free k's and the upper free $k+1$'s in r together form a contiguous interval or r.

The *'k'-th Bender–Knuth switch at row 'i'* changes the entries of the cells in this interval in such a way that if it used to have a entries of k and b entries of $k+1$, it will now have b entries of k and a entries of $k+1$. For fixed k, the k-th Bender–Knuth switches for different i commute. The composition of the k-th Bender–Knuth switches for all rows is called the *'k'-th Bender–Knuth involution*. This is used to show that the Schur functions defined by semistandard (skew) tableaux are symmetric functions.

INPUT:

- k – an integer
- rows – (Default None) When set to None, the method computes the k-th Bender–Knuth involution as defined above. When an iterable, this computes the composition of the k-th Bender–Knuth switches

at row i over all i in `rows`. When set to an integer i, the method computes the k-th Bender–Knuth switch at row i. Note the indexing of the rows starts with 1.

- `check` – (Default: `True`) Check to make sure `self` is semistandard. Set to `False` to avoid this check.

OUTPUT:

The image of `self` under either the k-th Bender–Knuth involution, the k-th Bender–Knuth switch at a certain row, or the composition of such switches, as detailed in the INPUT section.

EXAMPLES:

```
sage: t = SkewTableau([[None,None,None,4,4,5,6,7],[None,2,4,6,7,7,7],[None,4,5,8,8,9],[None,
sage: t
[[None, None, None, 4, 4, 5, 6, 7], [None, 2, 4, 6, 7, 7, 7], [None, 4, 5, 8, 8, 9], [None,
sage: t.bender_knuth_involution(1)
[[None, None, None, 4, 4, 5, 6, 7], [None, 1, 4, 6, 7, 7, 7], [None, 4, 5, 8, 8, 9], [None,
sage: t.bender_knuth_involution(4)
[[None, None, None, 4, 5, 5, 6, 7], [None, 2, 4, 6, 7, 7, 7], [None, 5, 5, 8, 8, 9], [None,
sage: t.bender_knuth_involution(5)
[[None, None, None, 4, 4, 5, 6, 7], [None, 2, 4, 5, 7, 7, 7], [None, 4, 6, 8, 8, 9], [None,
sage: t.bender_knuth_involution(6)
[[None, None, None, 4, 4, 5, 6, 6], [None, 2, 4, 6, 6, 7, 7], [None, 4, 5, 8, 8, 9], [None,
sage: t.bender_knuth_involution(666) == t
True
sage: t.bender_knuth_involution(4, 2) == t
True
sage: t.bender_knuth_involution(4, 3)
[[None, None, None, 4, 4, 5, 6, 7], [None, 2, 4, 6, 7, 7, 7], [None, 5, 5, 8, 8, 9], [None,
```

The Bender–Knuth involution is an involution:

```
sage: t = SkewTableau([[None,3,4,4],[None,6,10],[7,7,11],[18]])
sage: all(t.bender_knuth_involution(k).bender_knuth_involution(k) == t for k in range(1,4))
True
```

The same for the single switches:

```
sage: all(t.bender_knuth_involution(k, j).bender_knuth_involution(k, j) == t for k in range(
True
```

Locality of the Bender–Knuth involutions:

```
sage: all(t.bender_knuth_involution(k).bender_knuth_involution(l) == t.bender_knuth_involuti
True
```

TESTS:

```
sage: t = SkewTableau([])
sage: t.bender_knuth_involution(3)
[]
sage: t = SkewTableau([[None,None],[None]])
sage: t.bender_knuth_involution(3)
[[None, None], [None]]
```

The $(s_1 s_2)^6 = id$ identity that holds for Bender–Knuth involutions on straight shapes does not generally hold for skew shapes:

```
sage: p = lambda t, k: t.bender_knuth_involution(k).bender_knuth_involution(k + 1)
sage: t = SkewTableau([[None,1,2],[2,3]])
sage: x = t
```

```
sage: for i in range(6): x = p(x, 1)
sage: x
[[None, 2, 2], [1, 3]]
sage: x == t
False
```

AUTHORS:

- Darij Grinberg (2013-05-14)

cells()

Return the cells in `self`.

EXAMPLES:

```
sage: s = SkewTableau([[None,1,2],[3],[6]])
sage: s.cells()
[(0, 1), (0, 2), (1, 0), (2, 0)]
```

cells_by_content(*c*)

Return the coordinates of the cells in `self` with content `c`.

EXAMPLES:

```
sage: s = SkewTableau([[None,1,2],[3,4,5],[6]])
sage: s.cells_by_content(0)
[(1, 1)]
sage: s.cells_by_content(1)
[(0, 1), (1, 2)]
sage: s.cells_by_content(2)
[(0, 2)]
sage: s.cells_by_content(-1)
[(1, 0)]
sage: s.cells_by_content(-2)
[(2, 0)]
```

cells_containing(*i*)

Return the list of cells in which the letter `i` appears in the tableau `self`. The list is ordered with cells appearing from left to right.

Cells are given as pairs of coordinates (a, b), where both rows and columns are counted from 0 (so $a = 0$ means the cell lies in the leftmost column of the tableau, etc.).

EXAMPLES:

```
sage: t = SkewTableau([[None,None,3],[None,3,5],[4,5]])
sage: t.cells_containing(5)
[(2, 1), (1, 2)]
sage: t.cells_containing(4)
[(2, 0)]
sage: t.cells_containing(2)
[]

sage: t = SkewTableau([[None,None,None,None],[None,4,5],[None,5,6],[None,9],[None]])
sage: t.cells_containing(2)
[]
sage: t.cells_containing(4)
[(1, 1)]
sage: t.cells_containing(5)
[(2, 1), (1, 2)]
```

```
sage: SkewTableau([]).cells_containing(3)
[]

sage: SkewTableau([[None,None],[None]]).cells_containing(3)
[]
```

check()

Check that `self` is a valid skew tableau. This is currently far too liberal, and only checks some trivial things.

EXAMPLES:

```
sage: t = SkewTableau([[None,1,1],[2]])
sage: t.check()

sage: t = SkewTableau([[None, None, 1], [2, 4], [], [3, 4, 5]])
Traceback (most recent call last):
...
TypeError: a skew tableau cannot have an empty list for a row
```

conjugate()

Return the conjugate of `self`.

EXAMPLES:

```
sage: SkewTableau([[None,1],[2,3]]).conjugate()
[[None, 2], [1, 3]]
```

entries_by_content(c)

Return the entries in `self` with content `c`.

EXAMPLES:

```
sage: s = SkewTableau([[None,1,2],[3,4,5],[6]])
sage: s.entries_by_content(0)
[4]
sage: s.entries_by_content(1)
[1, 5]
sage: s.entries_by_content(2)
[2]
sage: s.entries_by_content(-1)
[3]
sage: s.entries_by_content(-2)
[6]
```

evaluation()

Return the weight (aka evaluation) of the tableau `self`. Trailing zeroes are omitted when returning the weight.

The weight of a skew tableau T is the sequence $(a_1, a_2, a_3, \ldots)$, where a_k is the number of entries of T equal to k. This sequence contains only finitely many nonzero entries.

The weight of a skew tableau T is the same as the weight of the reading word of T, for any reading order.

`evaluation()` is a synonym for this method.

EXAMPLES:

```
sage: SkewTableau([[1,2],[3,4]]).weight()
[1, 1, 1, 1]
```

```
sage: SkewTableau([[None,2],[None,4],[None,5],[None]]).weight()
[0, 1, 0, 1, 1]

sage: SkewTableau([]).weight()
[]

sage: SkewTableau([[None,None,None],[None]]).weight()
[]

sage: SkewTableau([[None,3,4],[None,6,7],[4,8],[5,13],[6],[7]]).weight()
[0, 0, 1, 2, 1, 2, 2, 1, 0, 0, 0, 0, 1]
```

TESTS:

We check that this agrees with going to the word:

```
sage: t = SkewTableau([[None,None,4,7,15],[6,2,16],[2,3,19],[4,5],[7]])
sage: def by_word(T):
....:     ed = T.to_word().evaluation_dict()
....:     m = max(ed.keys()) + 1
....:     return [ed.get(k,0) for k in range(1,m)]
sage: by_word(t) == t.weight()
True
sage: SST = SemistandardTableaux(shape=[3,1,1])
sage: all(by_word(t) == SkewTableau(t).weight() for t in SST)
True
```

filling()

Return a list of the non-empty entries in `self`.

EXAMPLES:

```
sage: t = SkewTableau([[None,1],[2,3]])
sage: t.filling()
[[1], [2, 3]]
```

inner_shape()

Return the inner shape of `self`.

EXAMPLES:

```
sage: SkewTableau([[None,1,2],[None,3],[4]]).inner_shape()
[1, 1]
sage: SkewTableau([[1,2],[3,4],[7]]).inner_shape()
[]
sage: SkewTableau([[None,None,None,2,3],[None,1],[None],[2]]).inner_shape()
[3, 1, 1]
```

inner_size()

Return the size of the inner shape of `self`.

EXAMPLES:

```
sage: SkewTableau([[None, 2, 4], [None, 3], [1]]).inner_size()
2
sage: SkewTableau([[None, 2], [1, 3]]).inner_size()
1
```

is_k_tableau(k)

Checks whether `self` is a valid skew weak k-tableau.

EXAMPLES:

```
sage: t = SkewTableau([[None,2,3],[2,3],[3]])
sage: t.is_k_tableau(3)
True
sage: t = SkewTableau([[None,1,3],[2,2],[3]])
sage: t.is_k_tableau(3)
False
```

is_ribbon()

Return `True` if and only if the shape of `self` is a ribbon, that is, if it has exactly one cell in each of q consecutive diagonals for some nonnegative integer q.

EXAMPLES:

```
sage: S=SkewTableau([[None, None, 1, 2],[None, None, 3],[1, 3, 4]])
sage: S.pp()
  .  .  1  2
  .  .  3
  1  3  4
sage: S.is_ribbon()
True

sage: S=SkewTableau([[None, 1, 1, 2],[None, 2, 3],[1, 3, 4]])
sage: S.pp()
  .  1  1  2
  .  2  3
  1  3  4
sage: S.is_ribbon()
False

sage: S=SkewTableau([[None, None, 1, 2],[None, None, 3],[1]])
sage: S.pp()
  .  .  1  2
  .  .  3
  1
sage: S.is_ribbon()
False

sage: S=SkewTableau([[None, None, None, None],[None, None, 3],[1, 2, 4]])
sage: S.pp()
  .  .  .  .
  .  .  3
  1  2  4
sage: S.is_ribbon()
True

sage: S=SkewTableau([[None, None, None, None],[None, None, 3],[None, 2, 4]])
sage: S.pp()
  .  .  .  .
  .  .  3
  .  2  4
sage: S.is_ribbon()
True

sage: S=SkewTableau([[None, None],[None]])
sage: S.pp()
  .  .
  .
sage: S.is_ribbon()
```

```
True
```

is_semistandard()

Return `True` if `self` is a semistandard skew tableau and `False` otherwise.

EXAMPLES:

```
sage: SkewTableau([[None, 2, 2], [1, 3]]).is_semistandard()
True
sage: SkewTableau([[None, 2], [2, 4]]).is_semistandard()
True
sage: SkewTableau([[None, 3], [2, 4]]).is_semistandard()
True
sage: SkewTableau([[None, 2], [1, 2]]).is_semistandard()
False
sage: SkewTableau([[None, 2, 3]]).is_semistandard()
True
sage: SkewTableau([[None, 3, 2]]).is_semistandard()
False
sage: SkewTableau([[None, 2, 3], [1, 4]]).is_semistandard()
True
sage: SkewTableau([[None, 2, 3], [1, 2]]).is_semistandard()
False
sage: SkewTableau([[None, 2, 3], [None, None, 4]]).is_semistandard()
False
```

is_standard()

Return `True` if `self` is a standard skew tableau and `False` otherwise.

EXAMPLES:

```
sage: SkewTableau([[None, 2], [1, 3]]).is_standard()
True
sage: SkewTableau([[None, 2], [2, 4]]).is_standard()
False
sage: SkewTableau([[None, 3], [2, 4]]).is_standard()
False
sage: SkewTableau([[None, 2], [2, 4]]).is_standard()
False
```

outer_shape()

Return the outer shape of `self`.

EXAMPLES:

```
sage: SkewTableau([[None,1,2],[None,3],[4]]).outer_shape()
[3, 2, 1]
```

outer_size()

Return the size of the outer shape of `self`.

EXAMPLES:

```
sage: SkewTableau([[None, 2, 4], [None, 3], [1]]).outer_size()
6
sage: SkewTableau([[None, 2], [1, 3]]).outer_size()
4
```

pp()

Return a pretty print string of the tableau.

EXAMPLES:

```
sage: SkewTableau([[None,2,3],[None,4],[5]]).pp()
  .  2  3
  .  4
  5
```

rectify(*algorithm=None*)

Return a StandardTableau, SemistandardTableau, or just Tableau formed by applying the jeu de taquin process to self.

See page 15 of [Fulton97].

INPUT:

- algorithm – optional: if set to 'jdt', rectifies by jeu de taquin; if set to 'schensted', rectifies by Schensted insertion of the reading word; otherwise, guesses which will be faster.

EXAMPLES:

```
sage: S = SkewTableau([[None,1],[2,3]])
sage: S.rectify()
[[1, 3], [2]]
sage: T = SkewTableau([[None, None, None, 4],[None,None,1,6],[None,None,5],[2,3]])
sage: T.rectify()
[[1, 3, 4, 6], [2, 5]]
sage: T.rectify(algorithm='jdt')
[[1, 3, 4, 6], [2, 5]]
sage: T.rectify(algorithm='schensted')
[[1, 3, 4, 6], [2, 5]]
sage: T.rectify(algorithm='spaghetti')
Traceback (most recent call last):
...
ValueError: algorithm must be 'jdt', 'schensted', or None
```

TESTS:

```
sage: S
[[None, 1], [2, 3]]
sage: T
[[None, None, None, 4], [None, None, 1, 6], [None, None, 5], [2, 3]]
```

REFERENCES:

restrict(*n*)

Return the restriction of the (semi)standard skew tableau to all the numbers less than or equal to n.

Note: If only the outer shape of the restriction, rather than the whole restriction, is needed, then the faster method restriction_outer_shape() is preferred. Similarly if only the skew shape is needed, use restriction_shape().

EXAMPLES:

```
sage: SkewTableau([[None,1],[2],[3]]).restrict(2)
[[None, 1], [2]]
sage: SkewTableau([[None,1],[2],[3]]).restrict(1)
[[None, 1]]
sage: SkewTableau([[None,1],[1],[2]]).restrict(1)
[[None, 1], [1]]
```

restriction_outer_shape(*n*)

Return the outer shape of the restriction of the semistandard skew tableau `self` to n.

If T is a semistandard skew tableau and n is a nonnegative integer, then the restriction of T to n is defined as the (semistandard) skew tableau obtained by removing all cells filled with entries greater than n from T.

This method computes merely the outer shape of the restriction. For the restriction itself, use `restrict()`.

EXAMPLES:

```
sage: SkewTableau([[None,None],[2,3],[3,4]]).restriction_outer_shape(3)
[2, 2, 1]
sage: SkewTableau([[None,2],[None],[4],[5]]).restriction_outer_shape(2)
[2, 1]
sage: T = SkewTableau([[None,None,3,5],[None,4,4],[17]])
sage: T.restriction_outer_shape(0)
[2, 1]
sage: T.restriction_outer_shape(2)
[2, 1]
sage: T.restriction_outer_shape(3)
[3, 1]
sage: T.restriction_outer_shape(4)
[3, 3]
sage: T.restriction_outer_shape(19)
[4, 3, 1]
```

restriction_shape(n)

Return the skew shape of the restriction of the semistandard skew tableau `self` to `n`.

If T is a semistandard skew tableau and n is a nonnegative integer, then the restriction of T to n is defined as the (semistandard) skew tableau obtained by removing all cells filled with entries greater than n from T.

This method computes merely the skew shape of the restriction. For the restriction itself, use `restrict()`.

EXAMPLES:

```
sage: SkewTableau([[None,None],[2,3],[3,4]]).restriction_shape(3)
[2, 2, 1] / [2]
sage: SkewTableau([[None,2],[None],[4],[5]]).restriction_shape(2)
[2, 1] / [1, 1]
sage: T = SkewTableau([[None,None,3,5],[None,4,4],[17]])
sage: T.restriction_shape(0)
[2, 1] / [2, 1]
sage: T.restriction_shape(2)
[2, 1] / [2, 1]
sage: T.restriction_shape(3)
[3, 1] / [2, 1]
sage: T.restriction_shape(4)
[3, 3] / [2, 1]
```

shape()

Return the shape of `self`.

EXAMPLES:

```
sage: SkewTableau([[None,1,2],[None,3],[4]]).shape()
[3, 2, 1] / [1, 1]
```

shuffle(*t2*)

Shuffle the standard tableaux `self` and `t2`.

Let `t1 = self`. The shape of `t2` must extend the shape of `t1`, that is, `self.outer_shape() == t2.inner_shape()`. Then this function computes the pair of tableaux `(t2_new, t1_new)` obtained by using jeu de taquin slides to move the boxes of `t2` behind the boxes of `self`.

The entries of `t2_new` are obtained by performing successive inwards jeu de taquin slides on `t2` in the order indicated by the entries of `t1`, from largest to smallest. The entries of `t1` then slide outwards one by one and land in the squares vacated successively by `t2`, forming `t1_new`.

Note: Equivalently, the entries of `t1_new` are obtained by performing outer jeu de taquin slides on `t1` in the order indicated by the entries of `t2`, from smallest to largest. In this case the entries of `t2` slide backwards and fill the squares successively vacated by `t1` and so form `t2_new`. (This is not how the algorithm is implemented.)

INPUT:

- `self, t2` – a pair of standard SkewTableaux with `self.outer_shape() == t2.inner_shape()`

OUTPUT:

- `t2_new, t1_new` – a pair of standard `SkewTableaux` with `t2_new.outer_shape() == t1_new.inner_shape()`

EXAMPLES:

```
sage: t1 = SkewTableau([[None, 1, 2], [3, 4]])
sage: t2 = SkewTableau([[None, None, None, 3], [None, None, 4], [1, 2, 5]])
sage: (t2_new, t1_new) = t1.shuffle(t2)
sage: t1_new
[[None, None, None, 2], [None, None, 1], [None, 3, 4]]
sage: t2_new
[[None, 2, 3], [1, 4], [5]]
sage: t1_new.outer_shape() == t2.outer_shape()
True
sage: t2_new.inner_shape() == t1.inner_shape()
True
```

Shuffling is an involution:

```
sage: t1 = SkewTableau([[None, 1, 2], [3, 4]])
sage: t2 = SkewTableau([[None, None, None, 3], [None, None, 4], [1, 2, 5]])
sage: sh = lambda x,y : x.shuffle(y)
sage: (t1, t2) == sh(*sh(t1, t2))
True
```

Both tableaux must be standard:

```
sage: t1 = SkewTableau([[None, 1, 2], [2, 4]])
sage: t2 = SkewTableau([[None, None, None, 3], [None, None, 4], [1, 2, 5]])
sage: t1.shuffle(t2)
Traceback (most recent call last):
...
ValueError: the tableaux must be standard
sage: t1 = SkewTableau([[None, 1, 2], [3, 4]])
sage: t2 = SkewTableau([[None, None, None, 3], [None, None, 4], [1, 2, 6]])
sage: t1.shuffle(t2)
Traceback (most recent call last):
```

```
...
ValueError: the tableaux must be standard
```

The shapes (not just the nonempty cells) must be adjacent:

```
sage: t1 = SkewTableau([[None, None, None], [1]])
sage: t2 = SkewTableau([[None], [None], [1]])
sage: t1.shuffle(t2)
Traceback (most recent call last):
...
ValueError: the shapes must be adjacent
```

TESTS:

A corner case, where one tableau has no cells:

```
sage: t1 = SkewTableau([[None]])
sage: t2 = SkewTableau([[None, 1, 2], [3, 4]])
sage: (t2_new, t1_new) = t1.shuffle(t2)
sage: t1_new
[[None, None, None], [None, None]]
sage: t2_new == t2
True
sage: t2_new.shuffle(t1_new) == (t1, t2)
True
```

size()

Return the number of cells in `self`.

EXAMPLES:

```
sage: SkewTableau([[None, 2, 4], [None, 3], [1]]).size()
4
sage: SkewTableau([[None, 2], [1, 3]]).size()
3
```

slide(*corner=None*, *return_vacated=False*)

Apply a jeu-de-taquin slide to `self` on the specified inner corner and return the resulting tableau.

If no corner is given, the topmost inner corner is chosen.

The optional parameter `return_vacated=True` causes the output to be the pair `(t, (i, j))` where `t` is the new tableau and `(i, j)` are the coordinates of the vacated square.

See [Fulton97] p12-13.

EXAMPLES:

```
sage: st = SkewTableau([[None, None, None, None, 2], [None, None, None, None, 6], [None, 2,
sage: st.slide((2, 0))
[[None, None, None, None, 2], [None, None, None, None, 6], [2, 2, 4, 4], [3, 5, 6], [5]]
sage: st2 = SkewTableau([[None, None, 3], [None, 2, 4], [1, 5]])
sage: st2.slide((1, 0), True)
([[None, None, 3], [1, 2, 4], [5]], (2, 1))
```

TESTS:

```
sage: st
[[None, None, None, None, 2], [None, None, None, None, 6],
 [None, 2, 4, 4], [2, 3, 6], [5, 5]]
```

standardization (*check=True*)

Return the standardization of `self`, assuming `self` is a semistandard skew tableau.

The standardization of a semistandard skew tableau T is the standard skew tableau $\mathrm{st}(T)$ of the same shape as T whose reversed reading word is the standardization of the reversed reading word of T.

The standardization of a word w can be formed by replacing all 1's in w by $1, 2, \ldots, k_1$ from left to right, all 2's in w by $k_1 + 1, k_1 + 2, \ldots, k_2$, and repeating for all letters that appear in w. See also `Word.standard_permutation()`.

INPUT:

- `check` – (Default: `True`) Check to make sure `self` is semistandard. Set to `False` to avoid this check.

EXAMPLES:

```
sage: t = SkewTableau([[None,None,3,4,7,19],[None,4,4,8],[None,5,16,17],[None],[2],[3]])
sage: t.standardization()
[[None, None, 3, 6, 8, 12], [None, 4, 5, 9], [None, 7, 10, 11], [None], [1], [2]]
```

Standard skew tableaux are fixed under standardization:

```
sage: p = Partition([4,3,3,2])
sage: q = Partitions(3).random_element()
sage: all((t == t.standardization() for t in StandardSkewTableaux([p, q])))
True
```

The reading word of the standardization is the standardization of the reading word:

```
sage: t = SkewTableau([[None,3,4,4],[None,6,10],[7,7,11],[18]])
sage: t.to_word().standard_permutation() == t.standardization().to_permutation()
True
```

TESTS:

Some corner cases:

```
sage: t = SkewTableau([[None,None],[None]])
sage: t.standardization()
[[None, None], [None]]
sage: t = SkewTableau([])
sage: t.standardization()
[]
```

to_chain (*max_entry=None*)

Return the chain of partitions corresponding to the (semi)standard skew tableau `self`.

The optional keyword parameter `max_entry` can be used to customize the length of the chain. Specifically, if this parameter is set to a nonnegative integer `n`, then the chain is constructed from the positions of the letters $1, 2, \ldots, n$ in the tableau.

EXAMPLES:

```
sage: SkewTableau([[None,1],[2],[3]]).to_chain()
[[1], [2], [2, 1], [2, 1, 1]]
sage: SkewTableau([[None,1],[1],[2]]).to_chain()
[[1], [2, 1], [2, 1, 1]]
sage: SkewTableau([[None,1],[1],[2]]).to_chain(max_entry=2)
[[1], [2, 1], [2, 1, 1]]
sage: SkewTableau([[None,1],[1],[2]]).to_chain(max_entry=3)
[[1], [2, 1], [2, 1, 1], [2, 1, 1]]
sage: SkewTableau([[None,1],[1],[2]]).to_chain(max_entry=1)
```

```
[[1], [2, 1]]
sage: SkewTableau([[None,None,2],[None,3],[None,5]]).to_chain(max_entry=6)
[[2, 1, 1], [2, 1, 1], [3, 1, 1], [3, 2, 1], [3, 2, 1], [3, 2, 2], [3, 2, 2]]
sage: SkewTableau([]).to_chain()
[[]]
sage: SkewTableau([]).to_chain(max_entry=1)
[[], []]
```

TESTS:

Check that `to_chain()` does not skip letters:

```
sage: t = SkewTableau([[None, 2, 3], [3]])
sage: t.to_chain()
[[1], [1], [2], [3, 1]]

sage: T = SkewTableau([[None]])
sage: T.to_chain()
[[1]]
```

to_expr()

The first list in a result corresponds to the inner partition of the skew shape. The second list is a list of the rows in the skew tableau read from the bottom up.

Provided for compatibility with MuPAD-Combinat. In MuPAD-Combinat, if `t` is a skew tableau, then to_expr gives the same result as `expr(t)` would give in MuPAD-Combinat.

EXAMPLES:

```
sage: SkewTableau([[None,1,1,3],[None,2,2],[1]]).to_expr()
[[1, 1], [[1], [2, 2], [1, 1, 3]]]
sage: SkewTableau([]).to_expr()
[[], []]
```

to_list()

Return a (mutable) list representation of `self`.

EXAMPLES:

```
sage: stlist = [[None, None, 3], [None, 1, 3], [2, 2]]
sage: st = SkewTableau(stlist)
sage: st.to_list()
[[None, None, 3], [None, 1, 3], [2, 2]]
sage: st.to_list() == stlist
True
```

to_permutation()

Return a permutation with the entries of `self` obtained by reading `self` row by row, from the bottommost to the topmost row, with each row being read from left to right, in English convention. See `to_word_by_row()`.

EXAMPLES:

```
sage: SkewTableau([[None,2],[3,4],[None],[1]]).to_permutation()
[1, 3, 4, 2]
sage: SkewTableau([[None,2],[None,4],[1],[3]]).to_permutation()
[3, 1, 4, 2]
sage: SkewTableau([[None]]).to_permutation()
[]
```

to_ribbon(*check_input=True*)

Return `self` as a ribbon-shaped tableau (`RibbonShapedTableau`), provided that the shape of `self` is a ribbon.

INPUT:

- `check_input` – (default: `True`) whether or not to check that `self` indeed has ribbon shape

EXAMPLES:

```
sage: SkewTableau([[None,1],[2,3]]).to_ribbon()
[[None, 1], [2, 3]]
```

to_tableau()

Returns a tableau with the same filling. This only works if the inner shape of the skew tableau has size zero.

EXAMPLES:

```
sage: SkewTableau([[1,2],[3,4]]).to_tableau()
[[1, 2], [3, 4]]
```

to_word()

Return a word obtained from a row reading of `self`.

This is the word obtained by concatenating the rows from the bottommost one (in English notation) to the topmost one.

EXAMPLES:

```
sage: s = SkewTableau([[None,1],[2,3]])
sage: s.pp()
  .  1
  2  3
sage: s.to_word_by_row()
word: 231
sage: s = SkewTableau([[None, 2, 4], [None, 3], [1]])
sage: s.pp()
  .  2  4
  .  3
  1
sage: s.to_word_by_row()
word: 1324
```

TESTS:

```
sage: SkewTableau([[None, None, None], [None]]).to_word_by_row()
word:
sage: SkewTableau([]).to_word_by_row()
word:
```

to_word_by_column()

Return the word obtained from a column reading of the skew tableau.

This is the word obtained by concatenating the columns from the rightmost one (in English notation) to the leftmost one.

EXAMPLES:

```
sage: s = SkewTableau([[None,1],[2,3]])
sage: s.pp()
  .  1
  2  3
```

```
sage: s.to_word_by_column()
word: 132

sage: s = SkewTableau([[None, 2, 4], [None, 3], [1]])
sage: s.pp()
.  2  4
.  3
1
sage: s.to_word_by_column()
word: 4231
```

to_word_by_row()

Return a word obtained from a row reading of `self`.

This is the word obtained by concatenating the rows from the bottommost one (in English notation) to the topmost one.

EXAMPLES:

```
sage: s = SkewTableau([[None,1],[2,3]])
sage: s.pp()
  .  1
  2  3
sage: s.to_word_by_row()
word: 231
sage: s = SkewTableau([[None, 2, 4], [None, 3], [1]])
sage: s.pp()
  .  2  4
  .  3
  1
sage: s.to_word_by_row()
word: 1324
```

TESTS:

```
sage: SkewTableau([[None, None, None], [None]]).to_word_by_row()
word:
sage: SkewTableau([]).to_word_by_row()
word:
```

weight()

Return the weight (aka evaluation) of the tableau `self`. Trailing zeroes are omitted when returning the weight.

The weight of a skew tableau T is the sequence $(a_1, a_2, a_3, \ldots)$, where a_k is the number of entries of T equal to k. This sequence contains only finitely many nonzero entries.

The weight of a skew tableau T is the same as the weight of the reading word of T, for any reading order.

`evaluation()` is a synonym for this method.

EXAMPLES:

```
sage: SkewTableau([[1,2],[3,4]]).weight()
[1, 1, 1, 1]

sage: SkewTableau([[None,2],[None,4],[None,5],[None]]).weight()
[0, 1, 0, 1, 1]

sage: SkewTableau([]).weight()
[]
```

```
sage: SkewTableau([[None,None,None],[None]]).weight()
[]

sage: SkewTableau([[None,3,4],[None,6,7],[4,8],[5,13],[6],[7]]).weight()
[0, 0, 1, 2, 1, 2, 2, 1, 0, 0, 0, 0, 1]
```

TESTS:

We check that this agrees with going to the word:

```
sage: t = SkewTableau([[None,None,4,7,15],[6,2,16],[2,3,19],[4,5],[7]])
sage: def by_word(T):
....:     ed = T.to_word().evaluation_dict()
....:     m = max(ed.keys()) + 1
....:     return [ed.get(k,0) for k in range(1,m)]
sage: by_word(t) == t.weight()
True
sage: SST = SemistandardTableaux(shape=[3,1,1])
sage: all(by_word(t) == SkewTableau(t).weight() for t in SST)
True
```

class sage.combinat.skew_tableau.**SkewTableau_class**(*parent*, *st*)

Bases: sage.combinat.skew_tableau.SkewTableau

This exists solely for unpickling SkewTableau_class objects.

class sage.combinat.skew_tableau.**SkewTableaux**(*category=None*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Class of all skew tableaux.

Element

alias of SkewTableau

from_expr(*expr*)

Return a SkewTableau from a MuPAD-Combinat expr for a skew tableau. The first list in expr is the inner shape of the skew tableau. The second list are the entries in the rows of the skew tableau from bottom to top.

Provided primarily for compatibility with MuPAD-Combinat.

EXAMPLES:

```
sage: SkewTableaux().from_expr([[1,1],[[5],[3,4],[1,2]]])
[[None, 1, 2], [None, 3, 4], [5]]
```

from_shape_and_word(*shape*, *word*)

Return the skew tableau corresponding to the skew partition shape and the word word obtained from the row reading.

EXAMPLES:

```
sage: t = SkewTableau([[None, 1, 3], [None, 2], [4]])
sage: shape = t.shape()
sage: word  = t.to_word()
sage: SkewTableaux().from_shape_and_word(shape, word)
[[None, 1, 3], [None, 2], [4]]
```

global_options(**get_value*, ***set_value*)

Sets the global options for elements of the tableau, skew_tableau, and tableau tuple classes. The defaults

are for tableau to be displayed as a list, latexed as a Young diagram using the English convention.

OPTIONS:

- `ascii_art` – (default: `repr`) Controls the ascii art output for tableaux
  - `compact` – minimal length ascii art
  - `repr` – display using the diagram string representation
  - `table` – display as a table
- `convention` – (default: `English`) Sets the convention used for displaying tableaux and partitions
  - `English` – use the English convention
  - `French` – use the French convention
- `display` – (default: `list`) Controls the way in which tableaux are printed
  - `array` – alias for `diagram`
  - `compact` – minimal length string representation
  - `diagram` – display as Young diagram (similar to `pp()`
  - `ferrers_diagram` – alias for `diagram`
  - `list` – print tableaux as lists
  - `young_diagram` – alias for `diagram`
- `latex` – (default: `diagram`) Controls the way in which tableaux are latexed
  - `array` – alias for `diagram`
  - `diagram` – as a Young diagram
  - `ferrers_diagram` – alias for `diagram`
  - `list` – as a list
  - `young_diagram` – alias for `diagram`
- `notation` – alternative name for `convention`

Note: Changing the `convention` for tableaux also changes the `convention` for partitions.

If no parameters are set, then the function returns a copy of the options dictionary.

EXAMPLES:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T
[[1, 2, 3], [4, 5]]
sage: Tableaux.global_options(display="array")
sage: T
  1  2  3
  4  5
sage: Tableaux.global_options(convention="french")
sage: T
  4  5
  1  2  3
```

Changing the `convention` for tableaux also changes the `convention` for partitions and vice versa:

```
sage: P = Partition([3,3,1])
sage: print P.ferrers_diagram()
*
***
***
sage: Partitions.global_options(convention="english")
sage: print P.ferrers_diagram()
***
***
*
sage: T
  1  2  3
  4  5
```

The ASCII art can also be changed:

```
sage: t = Tableau([[1,2,3],[4,5]])
sage: ascii_art(t)
  1  2  3
  4  5
sage: Tableaux.global_options(ascii_art="table")
sage: ascii_art(t)
+---+---+
| 4 | 5 |
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
sage: Tableaux.global_options(ascii_art="compact")
sage: ascii_art(t)
|4|5|
|1|2|3|
sage: Tableaux.global_options.reset()
```

See `GlobalOptions` for more features of these options.

class sage.combinat.skew_tableau.**StandardSkewTableaux**(*category=None*)

Bases: sage.combinat.skew_tableau.SkewTableaux

Standard skew tableaux.

EXAMPLES:

```
sage: S = StandardSkewTableaux(); S
Standard skew tableaux
sage: S.cardinality()
+Infinity

sage: S = StandardSkewTableaux(2); S
Standard skew tableaux of size 2
sage: S.cardinality()
4

sage: StandardSkewTableaux([[3, 2, 1], [1, 1]]).list()
[[[None, 1, 2], [None, 3], [4]],
 [[None, 1, 2], [None, 4], [3]],
 [[None, 1, 3], [None, 2], [4]],
 [[None, 1, 4], [None, 2], [3]],
 [[None, 1, 3], [None, 4], [2]],
 [[None, 1, 4], [None, 3], [2]],
 [[None, 2, 3], [None, 4], [1]],
```

```
[[None, 2, 4], [None, 3], [1]]]
```

class sage.combinat.skew_tableau.**StandardSkewTableaux_all**
Bases: sage.combinat.skew_tableau.StandardSkewTableaux

Class of all standard skew tableaux.

class sage.combinat.skew_tableau.**StandardSkewTableaux_shape**(*skp*)
Bases: sage.combinat.skew_tableau.StandardSkewTableaux

Standard skew tableaux of a fixed skew shape λ/μ.

cardinality()
Return the number of standard skew tableaux with shape of the skew partition `skp`. This uses a formula due to Aitken (see Cor. 7.16.3 of [Sta-EC2]).

EXAMPLES:

```
sage: StandardSkewTableaux([[3, 2, 1], [1, 1]]).cardinality()
8
```

class sage.combinat.skew_tableau.**StandardSkewTableaux_size**(*n*)
Bases: sage.combinat.skew_tableau.StandardSkewTableaux

Standard skew tableaux of a fixed size n.

cardinality()
EXAMPLES:

```
sage: StandardSkewTableaux(1).cardinality()
1
sage: StandardSkewTableaux(2).cardinality()
4
sage: StandardSkewTableaux(3).cardinality()
24
sage: StandardSkewTableaux(4).cardinality()
194
```

5.1.278 Functions that compute some of the sequences in Sloane's tables

EXAMPLES:

Type sloane.[tab] to see a list of the sequences that are defined.

```
sage: a = sloane.A000005; a
 The integer sequence tau(n), which is the number of divisors of n.
 sage: a(1)
 1
 sage: a(6)
 4
 sage: a(100)
 9
```

Type `d._eval??` to see how the function that computes an individual term of the sequence is implemented.

The input must be a positive integer:

```
sage: a(0)
Traceback (most recent call last):
...
```

```
ValueError: input n (=0) must be a positive integer
sage: a(1/3)
Traceback (most recent call last):
...
TypeError: input must be an int, long, or Integer
```

You can also change how a sequence prints:

```
sage: a = sloane.A000005; a
The integer sequence tau(n), which is the number of divisors of n.
sage: a.rename('(..., tau(n), ...)')
sage: a
(..., tau(n), ...)
sage: a.reset_name()
sage: a
The integer sequence tau(n), which is the number of divisors of n.
```

TESTS:

```
sage: a = sloane.A000001
sage: a == loads(dumps(a))
True
```

We agree with the online database:

```
sage: for t in sloane.trait_names():    # long time; optional -- internet; known bug
....:     online_list = list(oeis(t).first_terms())
....:     L = max(2, len(online_list) // 2)
....:     sage_list = sloane.__getattribute__(t).list(L)
....:     if online_list[:L] != sage_list:
....:         print t, 'seems wrong'
```

See also:

- If you want to get more informations relative to a sequence (references, links, examples, programs, ...), you can use the On-Line Encyclopedia of Integer Sequences provided by the `OEIS` module.
- If you plan to do a lot of automatic searches for subsequences, you should consider installing `SloaneEncyclopedia`, a local partial copy of the OEIS.

AUTHORS:

- William Stein: framework
- Jaap Spies: most sequences
- Nick Alexander: updated framework

class `sage.combinat.sloane_functions.`**A000001**
Bases: `sage.combinat.sloane_functions.SloaneSequence`

Number of groups of order n.

Note: The package database_gap must be installed for $n > 50$: run `sage -i database_gap` first.

INPUT:

- `n` – positive integer

OUTPUT: integer

EXAMPLES:

```
sage: a = sloane.A000001;a
Number of groups of order n.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
1
sage: a(9)
2
sage: a.list(16)
[1, 1, 1, 2, 1, 2, 1, 5, 2, 2, 1, 5, 1, 2, 1, 14]
sage: a(60)  # optional - database_gap
13
```

AUTHORS:

- Jaap Spies (2007-02-04)

class sage.combinat.sloane_functions.**A000004**

Bases: sage.combinat.sloane_functions.SloaneSequence

The zero sequence.

INPUT:

- n - non negative integer

OUTPUT:

EXAMPLES:

```
sage: a = sloane.A000004; a
The zero sequence.
sage: a(1)
0
sage: a(2007)
0
sage: a.list(12)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

AUTHORS:

- Jaap Spies (2006-12-10)

class sage.combinat.sloane_functions.**A000005**

Bases: sage.combinat.sloane_functions.SloaneSequence

The sequence $tau(n)$, which is the number of divisors of n.

This sequence is also denoted $d(n)$ (also called $\tau(n)$ or $\sigma_0(n)$), the number of divisors of n.

INPUT:

- n - positive integer

OUTPUT:

EXAMPLES:

```
sage: d = sloane.A000005; d
The integer sequence tau(n), which is the number of divisors of n.
```

```
sage: d(1)
1
sage: d(6)
4
sage: d(51)
4
sage: d(100)
9
sage: d(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: d.list(10)
[1, 2, 2, 3, 2, 4, 2, 4, 3, 4]
```

AUTHORS:

- Jaap Spies (2006-12-10)
- William Stein (2007-01-08)

class sage.combinat.sloane_functions.**A000007**

Bases: sage.combinat.sloane_functions.SloaneSequence

The characteristic function of 0: $a(n) = 0^n$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000007;a
The characteristic function of 0: a(n) = 0^n.
sage: a(0)
1
sage: a(2)
0
sage: a(12)
0
sage: a.list(12)
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

AUTHORS:

- Jaap Spies (2007-01-12)

class sage.combinat.sloane_functions.**A000008**

Bases: sage.combinat.sloane_functions.SloaneSequence

Number of ways of making change for n cents using coins of 1, 2, 5, 10 cents.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000008;a
Number of ways of making change for n cents using coins of 1, 2, 5, 10 cents.
sage: a(0)
1
sage: a(1)
1
sage: a(13)
16
sage: a.list(14)
[1, 1, 2, 2, 3, 4, 5, 6, 7, 8, 11, 12, 15, 16]
```

AUTHOR:

- 10.Gaski (2009-05-29)

class sage.combinat.sloane_functions.**A000009**

Bases: sage.combinat.sloane_functions.SloaneSequence

Number of partitions of n into odd parts.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000009;a
Number of partitions of n into odd parts.
sage: a(0)
1
sage: a(1)
1
sage: a(13)
18
sage: a.list(14)
[1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15, 18]
```

AUTHOR:

- Jaap Spies (2007-01-30)

cf()

EXAMPLES:

```
sage: it = sloane.A000009.cf()
sage: [next(it) for i in range(14)]
[1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15, 18]
```

list(*n*)

EXAMPLES:

```
sage: sloane.A000009.list(14)
[1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15, 18]
```

class sage.combinat.sloane_functions.**A000010**

Bases: sage.combinat.sloane_functions.SloaneSequence

The integer sequence A000010 is Euler's totient function.

Number of positive integers $i < n$ that are relative prime to n. Number of totatives of n.

Euler totient function $\phi(n)$: count numbers n and prime to n. euler_phi is a standard Sage function implemented in PARI

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000010; a
Euler's totient function
sage: a(1)
1
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(11)
10
sage: a.list(12)
[1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4]
sage: a(1/3)
Traceback (most recent call last):
...
TypeError: input must be an int, long, or Integer
```

AUTHORS:

- Jaap Spies (2007-01-12)

class sage.combinat.sloane_functions.**A000012**

Bases: sage.combinat.sloane_functions.SloaneSequence

The all 1's sequence.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000012; a
The all 1's sequence.
sage: a(1)
1
sage: a(2007)
1
sage: a.list(12)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

AUTHORS:

- Jaap Spies (2007-01-12)

class sage.combinat.sloane_functions.**A000015**
Bases: sage.combinat.sloane_functions.SloaneSequence

Smallest prime power $\geq n$ (where 1 is considered a prime power).

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000015; a
Smallest prime power >= n.
sage: a(1)
1
sage: a(8)
8
sage: a(305)
307
sage: a(-4)
Traceback (most recent call last):
...
ValueError: input n (=-4) must be a positive integer
sage: a.list(12)
[1, 2, 3, 4, 5, 7, 7, 8, 9, 11, 11, 13]
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-18)

class sage.combinat.sloane_functions.**A000016**
Bases: sage.combinat.sloane_functions.SloaneSequence

Sloane's A000016

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000016; a
Sloane's A000016.
sage: a(1)
1
sage: a(0)
1
sage: a(8)
16
sage: a(75)
251859545753048193000
sage: a(-4)
Traceback (most recent call last):
```

```
...
ValueError: input n (=-4) must be an integer >= 0
sage: a.list(12)
[1, 1, 1, 2, 2, 4, 6, 10, 16, 30, 52, 94]
```

AUTHORS:

- Jaap Spies (2007-01-18)

class sage.combinat.sloane_functions.**A000027**

Bases: sage.combinat.sloane_functions.SloaneSequence

The natural numbers. Also called the whole numbers, the counting numbers or the positive integers.

The following examples are tests of SloaneSequence more than A000027.

EXAMPLES:

```
sage: s = sloane.A000027; s
The natural numbers.
sage: s(10)
10
```

Index n is interpreted as _eval(n):

```
sage: s[10]
10
```

Slices are interpreted with absolute offsets, so the following returns the terms of the sequence up to but not including the third term:

```
sage: s[:3]
[1, 2]
sage: s[3:6]
[3, 4, 5]
sage: s.list(5)
[1, 2, 3, 4, 5]
```

class sage.combinat.sloane_functions.**A000030**

Bases: sage.combinat.sloane_functions.SloaneSequence

Initial digit of n.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000030; a
Initial digit of n
sage: a(0)
0
sage: a(1)
1
sage: a(8)
8
sage: a(454)
4
sage: a(-4)
```

```
Traceback (most recent call last):
...
ValueError: input n (=-4) must be an integer >= 0
sage: a.list(12)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1]
```

AUTHORS:

- Jaap Spies (2007-01-18)

class sage.combinat.sloane_functions.**A000032**

Bases: sage.combinat.sloane_functions.SloaneSequence

Lucas numbers (beginning at 2): $L(n) = L(n-1) + L(n-2)$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000032; a
Lucas numbers (beginning at 2): L(n) = L(n-1) + L(n-2).
sage: a(0)
2
sage: a(1)
1
sage: a(8)
47
sage: a(200)
627376215338105766356982006981782561278127
sage: a(-4)
Traceback (most recent call last):
...
ValueError: input n (=-4) must be an integer >= 0
sage: a.list(12)
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199]
```

AUTHORS:

- Jaap Spies (2007-01-18)

class sage.combinat.sloane_functions.**A000035**

Bases: sage.combinat.sloane_functions.SloaneSequence

A simple periodic sequence.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000035;a
A simple periodic sequence.
sage: a(0.0)
```

```
Traceback (most recent call last):
...
TypeError: input must be an int, long, or Integer
sage: a(1)
1
sage: a(2)
0
sage: a(9)
1
sage: a.list(10)
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
```

AUTHORS:

- Jaap Spies (2007-02-02)

class sage.combinat.sloane_functions.**A000040**

Bases: sage.combinat.sloane_functions.SloaneSequence

The prime numbers.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000040; a
The prime numbers.
sage: a(1)
2
sage: a(8)
19
sage: a(305)
2011
sage: a.list(12)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-17)

class sage.combinat.sloane_functions.**A000041**

Bases: sage.combinat.sloane_functions.SloaneSequence

$a(n)$ = number of partitions of n (the partition numbers).

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000041;a
a(n) = number of partitions of n (the partition numbers).
sage: a(0)
1
sage: a(2)
2
sage: a(8)
22
sage: a(200)
3972999029388
sage: a.list(9)
[1, 1, 2, 3, 5, 7, 11, 15, 22]
```

AUTHORS:

- Jaap Spies (2007-01-18)

class sage.combinat.sloane_functions.**A000043**

Bases: sage.combinat.sloane_functions.SloaneSequence

Primes p such that $2^p - 1$ is prime. $2^p - 1$ is then called a Mersenne prime.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000043;a
Primes p such that 2^p - 1 is prime. 2^p - 1 is then called a Mersenne prime.
sage: a(1)
2
sage: a(2)
3
sage: a(39)
13466917
sage: a(40)
Traceback (most recent call last):
...
IndexError: list index out of range
sage: a.list(12)
[2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000045**

Bases: sage.combinat.sloane_functions.SloaneSequence

Sequence of Fibonacci numbers, offset 0,4.

REFERENCES:

- S. Plouffe, Project Gutenberg, The First 1001 Fibonacci Numbers, http://ibiblio.org/pub/docs/books/gutenberg/etext01/fbncc10.txt

We have one more. Our first Fibonacci number is 0.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000045; a
Fibonacci numbers with index n >= 0
sage: a(0)
0
sage: a(1)
1
sage: a.list(12)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
sage: a(1/3)
Traceback (most recent call last):
...
TypeError: input must be an int, long, or Integer
```

AUTHORS:

- Jaap Spies (2007-01-13)

fib()

Returns a generator over all Fibonacci numbers, starting with 0.

EXAMPLES:

```
sage: it = sloane.A000045.fib()
sage: [next(it) for i in range(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

list(*n*)

EXAMPLES:

```
sage: sloane.A000045.list(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

class sage.combinat.sloane_functions.**A000069**

Bases: sage.combinat.sloane_functions.SloaneSequence

Odious numbers: odd number of 1's in binary expansion.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000069; a
Odious numbers: odd number of 1's in binary expansion.
sage: a(0)
1
sage: a(2)
4
sage: a.list(9)
[1, 2, 4, 7, 8, 11, 13, 14, 16]
```

AUTHORS:

- Jaap Spies (2007-02-02)

class sage.combinat.sloane_functions.**A000073**
Bases: sage.combinat.sloane_functions.SloaneSequence

Tribonacci numbers: a(n) = a(n-1) + a(n-2) + a(n-3). Starting with 0, 0, 1, ...

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000073;a
Tribonacci numbers: a(n) = a(n-1) + a(n-2) + a(n-3).
sage: a(0)
0
sage: a(1)
0
sage: a(2)
1
sage: a(11)
149
sage: a.list(12)
[0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149]
```

AUTHORS:

- Jaap Spies (2007-01-19)

list(*n*)
EXAMPLES:

```
sage: sloane.A000073.list(10)
[0, 0, 1, 1, 2, 4, 7, 13, 24, 44]
```

class sage.combinat.sloane_functions.**A000079**
Bases: sage.combinat.sloane_functions.SloaneSequence

Powers of 2: $a(n) = 2^n$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000079;a
Powers of 2: a(n) = 2^n.
sage: a(0)
1
sage: a(2)
4
sage: a(8)
```

```
256
sage: a(100)
1267650600228229401496703205376
sage: a.list(9)
[1, 2, 4, 8, 16, 32, 64, 128, 256]
```

AUTHORS:

- Jaap Spies (2007-01-18)

class sage.combinat.sloane_functions.**A000085**
Bases: sage.combinat.sloane_functions.SloaneSequence

Number of self-inverse permutations on n letters, also known as involutions; number of Young tableaux with n cells.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000085;a
Number of self-inverse permutations on n letters.
sage: a(0)
1
sage: a(1)
1
sage: a(2)
2
sage: a(12)
140152
sage: a.list(13)
[1, 1, 2, 4, 10, 26, 76, 232, 764, 2620, 9496, 35696, 140152]
```

AUTHORS:

- Jaap Spies (2007-02-03)

class sage.combinat.sloane_functions.**A000100**
Bases: sage.combinat.sloane_functions.SloaneSequence

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000100;a
Number of compositions of n in which the maximum part size is 3.
sage: a(0)
0
sage: a(1)
0
sage: a(2)
0
```

```
sage: a(3)
1
sage: a(11)
360
sage: a.list(12)
[0, 0, 0, 1, 2, 5, 11, 23, 47, 94, 185, 360]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000108**

Bases: sage.combinat.sloane_functions.SloaneSequence

Catalan numbers: $C_n = \frac{\binom{2n}{n}}{n+1} = \frac{(2n)!}{n!(n+1)!}$. Also called Segner numbers.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000108;a
Catalan numbers: C(n) = binomial(2n,n)/(n+1) = (2n)!/(n!(n+1)!). Also called Segner numbers.
sage: a(0)
1
sage: a.offset
0
sage: a(8)
1430
sage: a(40)
2622127042276492108820
sage: a.list(9)
[1, 1, 2, 5, 14, 42, 132, 429, 1430]
```

AUTHORS:

- Jaap Spies (2007-01-12)

class sage.combinat.sloane_functions.**A000110**

Bases: sage.combinat.sloane_functions.ExponentialNumbers

The sequence of Bell numbers.

The Bell number B_n counts the number of ways to put n distinguishable things into indistinguishable boxes such that no box is empty.

Let $S(n, k)$ denote the Stirling number of the second kind. Then

$$B_n = \sum k = 0^n S(n, k).$$

INPUT:

- n - integer = 0

OUTPUT:

- integer - B_n

EXAMPLES:

```
sage: a = sloane.A000110; a
Sequence of Bell numbers
sage: a.offset
0
sage: a(0)
1
sage: a(100)
4758539127676483365879076884138720782636366968682561146661633463755911449789244262267272404421777
sage: a.list(10)
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147]
```

AUTHORS:

- Nick Alexander

class sage.combinat.sloane_functions.**A000120**

Bases: sage.combinat.sloane_functions.SloaneSequence

1's-counting sequence: number of 1's in binary expansion of n.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000120;a
1's-counting sequence: number of 1's in binary expansion of n.
sage: a(0)
0
sage: a(2)
1
sage: a(12)
2
sage: a.list(12)
[0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3]
```

AUTHORS:

- Jaap Spies (2007-01-26)

f(n)

EXAMPLES:

```
sage: [sloane.A000120.f(n) for n in range(10)]
[0, 1, 1, 2, 1, 2, 2, 3, 1, 2]
```

class sage.combinat.sloane_functions.**A000124**

Bases: sage.combinat.sloane_functions.SloaneSequence

Central polygonal numbers (the Lazy Caterer's sequence): $n(n+1)/2+1$.

Or, maximal number of pieces formed when slicing a pancake with n cuts.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000124;a
Central polygonal numbers (the Lazy Caterer's sequence): n(n+1)/2 + 1.
sage: a(0)
1
sage: a(1)
2
sage: a(2)
4
sage: a(9)
46
sage: a.list(10)
[1, 2, 4, 7, 11, 16, 22, 29, 37, 46]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000129**

Bases: sage.combinat.sloane_functions.RecurrenceSequence2

Pell numbers: $a(0) = 0$, $a(1) = 1$; for $n > 1$, $a(n) = 2a(n-1) + a(n-2)$.

Denominators of continued fraction convergents to $\sqrt{2}$.

See also A001333

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000129;a
Pell numbers: a(0) = 0, a(1) = 1; for n > 1, a(n) = 2*a(n-1) + a(n-2).
sage: a(0)
0
sage: a(2)
2
sage: a(12)
13860
sage: a.list(12)
[0, 1, 2, 5, 12, 29, 70, 169, 408, 985, 2378, 5741]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000142**

Bases: sage.combinat.sloane_functions.SloaneSequence

Factorial numbers: $n! = 1 \cdot 2 \cdot 3 \cdots n$

Order of symmetric group S_n, number of permutations of n letters.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000142;a
Factorial numbers: n! = 1*2*3*4*...*n (order of symmetric group S_n, number of permutations of n
sage: a(0)
1
sage: a(8)
40320
sage: a(40)
815915283247897734345611269596115894272000000000
sage: a.list(9)
[1, 1, 2, 6, 24, 120, 720, 5040, 40320]
```

AUTHORS:

- Jaap Spies (2007-01-12)

class sage.combinat.sloane_functions.**A000153**

Bases: sage.combinat.sloane_functions.ExtremesOfPermanentsSequence

$a(n) = n * a(n-1) + (n-2) * a(n-2)$, with $a(0) = 0$, $a(1) = 1$.

With offset 1, permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 2$ and n zeros not on a line. This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000153; a
a(n) = n*a(n-1) + (n-2)*a(n-2), with a(0) = 0, a(1) = 1.
sage: a(0)
0
sage: a(1)
1
sage: a(8)
82508
sage: a(20)
10315043624498196944
sage: a.list(8)
[0, 1, 2, 7, 32, 181, 1214, 9403]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A000165**

Bases: sage.combinat.sloane_functions.SloaneSequence

Double factorial numbers: $(2n)!! = 2^n * n!$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000165;a
Double factorial numbers: (2n)!! = 2^n*n!.
sage: a(0)
1
sage: a.offset
0
sage: a(8)
10321920
sage: a(20)
2551082656125828464640000
sage: a.list(9)
[1, 2, 8, 48, 384, 3840, 46080, 645120, 10321920]
```

AUTHORS:

- Jaap Spies (2007-01-24)

class sage.combinat.sloane_functions.**A000166**

Bases: sage.combinat.sloane_functions.SloaneSequence

Subfactorial or rencontres numbers, or derangements: number of permutations of n elements with no fixed points.

With offset 1 also the permanent of a (0,1)-matrix of order n with n 0's not on a line.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000166;a
Subfactorial or rencontres numbers, or derangements: number of permutations of $n$ elements with
sage: a(0)
1
sage: a(1)
0
sage: a(2)
1
sage: a.offset
0
sage: a(8)
14833
sage: a(20)
895014631192902121
sage: a.list(9)
[1, 0, 1, 2, 9, 44, 265, 1854, 14833]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A000169**

Bases: sage.combinat.sloane_functions.SloaneSequence

Number of labeled rooted trees with n nodes: $n^{(n-1)}$.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000169;a
Number of labeled rooted trees with n nodes: n^(n-1).
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
2
sage: a(10)
1000000000
sage: a.list(11)
[1, 2, 9, 64, 625, 7776, 117649, 2097152, 43046721, 1000000000, 25937424601]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000203**

Bases: sage.combinat.sloane_functions.SloaneSequence

The sequence $\sigma(n)$, where $\sigma(n)$ is the sum of the divisors of n. Also called $\sigma_1(n)$.

The function sigma(n, k) implements $\sigma_k(n)$ in Sage.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000203; a
sigma(n) = sum of divisors of n. Also called sigma_1(n).
sage: a(1)
1
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(256)
511
sage: a.list(12)
[1, 3, 4, 7, 6, 12, 8, 15, 13, 18, 12, 28]
sage: a(1/3)
Traceback (most recent call last):
...
TypeError: input must be an int, long, or Integer
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A000204**
Bases: sage.combinat.sloane_functions.SloaneSequence

Lucas numbers (beginning with 1): $L(n) = L(n-1) + L(n-2)$ with $L(1) = 1$, $L(2) = 3$.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000204; a
Lucas numbers (beginning at 1): L(n) = L(n-1) + L(n-2), L(2) = 3.
sage: a(1)
1
sage: a(8)
47
sage: a(200)
627376215338105766356982006981782561278127
sage: a(-4)
Traceback (most recent call last):
...
ValueError: input n (=-4) must be a positive integer
sage: a.list(12)
[1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322]
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-18)

class sage.combinat.sloane_functions.**A000213**
Bases: sage.combinat.sloane_functions.SloaneSequence

Tribonacci numbers: a(n) = a(n-1) + a(n-2) + a(n-3). Starting with 1, 1, 1, ...

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000213;a
Tribonacci numbers: a(n) = a(n-1) + a(n-2) + a(n-3).
sage: a(0)
1
sage: a(1)
1
sage: a(2)
1
```

```
sage: a(11)
355
sage: a.list(12)
[1, 1, 1, 3, 5, 9, 17, 31, 57, 105, 193, 355]
```

AUTHORS:

- Jaap Spies (2007-01-19)

list(*n*)

EXAMPLES:

```
sage: sloane.A000213.list(10)
[1, 1, 1, 3, 5, 9, 17, 31, 57, 105]
```

class sage.combinat.sloane_functions.**A000217**

Bases: sage.combinat.sloane_functions.SloaneSequence

Triangular numbers: $\binom{a(n)=n+1}{2)=n(n+1)/2}$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000217;a
Triangular numbers: a(n) = C(n+1,2) = n(n+1)/2 = 0+1+2+...+n.
sage: a(0)
0
sage: a(2)
3
sage: a(8)
36
sage: a(2000)
2001000
sage: a.list(9)
[0, 1, 3, 6, 10, 15, 21, 28, 36]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000225**

Bases: sage.combinat.sloane_functions.SloaneSequence

$2^n - 1$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000225;a
2^n - 1.
sage: a(0)
```

```
0
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(12)
4095
sage: a.list(12)
[0, 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000244**

Bases: sage.combinat.sloane_functions.SloaneSequence

Powers of 3: $a(n) = 3^n$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000244;a
Powers of 3: a(n) = 3^n.
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(0)
1
sage: a(3)
27
sage: a(11)
177147
sage: a.list(12)
[1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049, 177147]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000255**

Bases: sage.combinat.sloane_functions.ExtremesOfPermanentsSequence

$a(n) = n * a(n-1) + (n-1) * a(n-2)$, with $a(0) = 1$, $a(1) = 1$.

With offset 1, permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 1$ and n zeros not on a line. This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000255;a
a(n) = n*a(n-1) + (n-1)*a(n-2), a(0) = 1, a(1) = 1.
sage: a(0)
1
sage: a(1)
1
sage: a.offset
0
sage: a(8)
148329
sage: a(22)
9923922230666898717143
sage: a.list(9)
[1, 1, 3, 11, 53, 309, 2119, 16687, 148329]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A000261**

Bases: sage.combinat.sloane_functions.ExtremesOfPermanentsSequence

$a(n) = n * a(n-1) + (n-3) * a(n-2)$, with $a(1) = 1$, $a(2) = 1$.

With offset 1, permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 3$ and n zeros not on a line. This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000261;a
a(n) = n*a(n-1) + (n-3)*a(n-2), a(1) = 0, a(2) = 1.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
0
sage: a.offset
1
sage: a(8)
30637
sage: a(22)
1801366114380914335441
sage: a.list(9)
[0, 1, 3, 13, 71, 465, 3539, 30637, 296967]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A000272**
Bases: sage.combinat.sloane_functions.SloaneSequence

Number of labeled rooted trees on n nodes: $n^{(n-2)}$.

INPUT:

- n - integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000272;a
Number of labeled rooted trees with n nodes: n^(n-2).
sage: a(0)
1
sage: a(1)
1
sage: a(2)
1
sage: a(10)
100000000
sage: a.list(12)
[1, 1, 1, 3, 16, 125, 1296, 16807, 262144, 4782969, 100000000, 2357947691]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000290**
Bases: sage.combinat.sloane_functions.SloaneSequence

The squares: $a(n) = n^2$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000290;a
The squares: a(n) = n^2.
sage: a(0)
0
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(16)
256
sage: a.list(17)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000292**
Bases: sage.combinat.sloane_functions.SloaneSequence

Tetrahedral (or pyramidal) numbers: $\binom{n+2}{3}_{=n(n+1)(n+2)/6}$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000292;a
Tetrahedral (or pyramidal) numbers: C(n+2,3) = n(n+1)(n+2)/6.
sage: a(0)
0
sage: a(2)
4
sage: a(11)
286
sage: a.list(12)
[0, 1, 4, 10, 20, 35, 56, 84, 120, 165, 220, 286]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000302**
Bases: sage.combinat.sloane_functions.SloaneSequence

Powers of 4: $a(n) = 4^n$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000302;a
Powers of 4: a(n) = 4^n.
sage: a(0)
1
sage: a(1)
4
sage: a(2)
16
sage: a(10)
1048576
sage: a.list(12)
[1, 4, 16, 64, 256, 1024, 4096, 16384, 65536, 262144, 1048576, 4194304]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000312**
Bases: sage.combinat.sloane_functions.SloaneSequence

Number of labeled mappings from n points to themselves (endofunctions): n^n.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000312;a
Number of labeled mappings from n points to themselves (endofunctions): n^n.
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(0)
1
sage: a(1)
1
sage: a(9)
387420489
sage: a.list(11)
[1, 1, 4, 27, 256, 3125, 46656, 823543, 16777216, 387420489, 10000000000]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000326**

Bases: sage.combinat.sloane_functions.SloaneSequence

Pentagonal numbers: $n(3n-1)/2$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000326;a
Pentagonal numbers: n(3n-1)/2.
sage: a(0)
0
sage: a(1)
1
sage: a(2)
5
sage: a(10)
145
sage: a.list(12)
[0, 1, 5, 12, 22, 35, 51, 70, 92, 117, 145, 176]
sage: a(1/3)
Traceback (most recent call last):
...
TypeError: input must be an int, long, or Integer
```

AUTHORS:

•Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000330**
Bases: sage.combinat.sloane_functions.SloaneSequence

Square pyramidal numbers" $0^2 + 1^2 \cdots n^2 = n(n+1)(2n+1)/6$.

INPUT:

•n - non negative integer

OUTPUT:

•integer - function value

EXAMPLES:

```
sage: a = sloane.A000330;a
Square pyramidal numbers: 0^2+1^2+2^2+...+n^2 = n(n+1)(2n+1)/6.
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(0)
0
sage: a(3)
14
sage: a(11)
506
sage: a.list(12)
[0, 1, 5, 14, 30, 55, 91, 140, 204, 285, 385, 506]
```

AUTHORS:

•Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000396**
Bases: sage.combinat.sloane_functions.SloaneSequence

Perfect numbers: equal to sum of proper divisors.

INPUT:

•n - positive integer

OUTPUT:

•integer - function value

EXAMPLES:

```
sage: a = sloane.A000396;a
Perfect numbers: equal to sum of proper divisors.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
6
sage: a(2)
28
sage: a(7)
137438691328
sage: a.list(7)
[6, 28, 496, 8128, 33550336, 8589869056, 137438691328]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000578**

Bases: sage.combinat.sloane_functions.SloaneSequence

The cubes: $a(n) = n^3$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000578;a
The cubes: n^3
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(0)
0
sage: a(3)
27
sage: a(11)
1331
sage: a.list(12)
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A000583**

Bases: sage.combinat.sloane_functions.SloaneSequence

Fourth powers: $a(n) = n^4$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000583;a
Fourth powers: n^4.
sage: a(0.0)
Traceback (most recent call last):
...
TypeError: input must be an int, long, or Integer
sage: a(1)
1
sage: a(2)
16
```

```
sage: a(9)
6561
sage: a.list(10)
[0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561]
```

AUTHORS:

- Jaap Spies (2007-02-04)

class sage.combinat.sloane_functions.**A000587**
Bases: sage.combinat.sloane_functions.ExponentialNumbers

The sequence of Uppuluri-Carpenter numbers.

The Uppuluri-Carpenter number C_n counts the imbalance in the number of ways to put n distinguishable things into an even number of indistinguishable boxes versus into an odd number of indistinguishable boxes, such that no box is empty.

Let $S(n, k)$ denote the Stirling number of the second kind. Then

$$C_n = \sum k = 0^n (-1)^k S(n, k).$$

INPUT:

- n - integer = 0

OUTPUT:

- integer - C_n

EXAMPLES:

```
sage: a = sloane.A000587; a
Sequence of Uppuluri-Carpenter numbers
sage: a.offset
0
sage: a(0)
1
sage: a(100)
39757702645651850796976238225418704884562035523854513087506991294423510520443446609586237103212
sage: a.list(10)
[1, -1, 0, 1, 1, -2, -9, -9, 50, 267]
```

AUTHORS:

- Nick Alexander

class sage.combinat.sloane_functions.**A000668**
Bases: sage.combinat.sloane_functions.SloaneSequence

Mersenne primes (of form $2^p - 1$ where p is a prime).

(See A000043 for the values of p.)

Warning: a(39) has 4,053,946 digits!

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000668;a
Mersenne primes (of form 2^p - 1 where p is a prime). (See A000043 for the values of p.)
sage: a(1)
3
sage: a(2)
7
sage: a(12)
170141183460469231731687303715884105727
```

Warning: a(39) has 4,053,946 digits!

```
sage: a(40)
Traceback (most recent call last):
...
IndexError: list index out of range
sage: a.list(8)
[3, 7, 31, 127, 8191, 131071, 524287, 2147483647]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000670**

Bases: sage.combinat.sloane_functions.SloaneSequence

Number of preferential arrangements of n labeled elements; or number of weak orders on n labeled elements.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000670;a
Number of preferential arrangements of n labeled elements.
sage: a(0)
1
sage: a(1)
1
sage: a(2)
3
sage: a(9)
7087261
sage: a.list(10)
[1, 1, 3, 13, 75, 541, 4683, 47293, 545835, 7087261]
```

AUTHORS:

- Jaap Spies (2007-02-03)

class sage.combinat.sloane_functions.**A000720**

Bases: sage.combinat.sloane_functions.SloaneSequence

$pi(n)$, the number of primes $\leq n$. Sometimes called $PrimePi(n)$.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000720;a
pi(n), the number of primes <= n. Sometimes called PrimePi(n)
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
1
sage: a(8)
4
sage: a(1000)
168
sage: a.list(12)
[0, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A000796**

Bases: sage.combinat.sloane_functions.SloaneSequence

Decimal expansion of π.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000796;a
Decimal expansion of Pi.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
3
sage: a(13)
9
sage: a.list(14)
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7]
sage: a(100)
7
```

AUTHOR:

- Jaap Spies (2007-01-30)

list(*n*)

EXAMPLES:

```
sage: sloane.A000796.list(10)
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
```

pi()

Based on an algorithm of Lambert Meertens The ABC-programming language!!!

EXAMPLES:

```
sage: it = sloane.A000796.pi()
sage: [next(it) for i in range(10)]
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
```

class sage.combinat.sloane_functions.**A000961**

Bases: sage.combinat.sloane_functions.SloaneSequence

Prime powers

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000961;a
Prime powers.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
2
sage: a(12)
17
sage: a.list(12)
[1, 2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17]
```

AUTHORS:

- Jaap Spies (2007-01-25)

list(*n*)

EXAMPLES:

```
sage: sloane.A000961.list(10)
[1, 2, 3, 4, 5, 7, 8, 9, 11, 13]
```

class sage.combinat.sloane_functions.**A000984**

Bases: sage.combinat.sloane_functions.SloaneSequence

Central binomial coefficients: $\left(_{n=\frac{(2n)!}{(n!)^2}}^{2n}\right)$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A000984;a
Central binomial coefficients: C(2n,n) = (2n)!/(n!)^2
sage: a(0)
```

```
1
sage: a(2)
6
sage: a(8)
12870
sage: a.list(9)
[1, 2, 6, 20, 70, 252, 924, 3432, 12870]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A001006**

Bases: sage.combinat.sloane_functions.SloaneSequence

Motzkin numbers: number of ways of drawing any number of nonintersecting chords among n points on a circle.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001006;a
Motzkin numbers: number of ways of drawing any number of nonintersecting chords among n points c
sage: a(0)
1
sage: a(1)
1
sage: a(2)
2
sage: a(12)
15511
sage: a.list(13)
[1, 1, 2, 4, 9, 21, 51, 127, 323, 835, 2188, 5798, 15511]
```

AUTHORS:

- Jaap Spies (2007-02-02)

class sage.combinat.sloane_functions.**A001045**

Bases: sage.combinat.sloane_functions.RecurrenceSequence2

Jacobsthal sequence: $a(n) = a(n-1) + 2a(n-2)$, $a(0) = 0$ and $a(1) = 1$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001045;a
Jacobsthal sequence: a(n) = a(n-1) + 2a(n-2).
sage: a(0)
0
```

```
sage: a(1)
1
sage: a(2)
1
sage: a(11)
683
sage: a.list(12)
[0, 1, 1, 3, 5, 11, 21, 43, 85, 171, 341, 683]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A001055**
Bases: sage.combinat.sloane_functions.SloaneSequence

Number of ways of factoring n with all factors 1.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001055;a
Number of ways of factoring n with all factors >1.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
1
sage: a(9)
2
sage: a.list(16)
[1, 1, 1, 2, 1, 2, 1, 3, 2, 2, 1, 4, 1, 2, 2, 5]
```

AUTHORS:

- Jaap Spies (2007-02-04)

nwf(n, m)

EXAMPLES:

```
sage: sloane.A001055.nwf(4,1)
0
sage: sloane.A001055.nwf(4,2)
1
sage: sloane.A001055.nwf(4,3)
1
sage: sloane.A001055.nwf(4,4)
2
```

class sage.combinat.sloane_functions.**A001109**
Bases: sage.combinat.sloane_functions.RecurrenceSequence2

$a(n)^2$ is a triangular number: $a(n) = 6 * a(n-1) - a(n-2)$ with $a(0) = 0$, $a(1) = 1$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001109;a
a(n)^2 is a triangular number: a(n) = 6*a(n-1) - a(n-2) with a(0)=0, a(1)=1
sage: a(0)
0
sage: a(1)
1
sage: a(2)
6
sage: a.offset
0
sage: a(8)
235416
sage: a(60)
1515330104844857898115857393785728383101709300
sage: a.list(9)
[0, 1, 6, 35, 204, 1189, 6930, 40391, 235416]
```

AUTHORS:

- Jaap Spies (2007-01-24)

class sage.combinat.sloane_functions.**A001110**

Bases: sage.combinat.sloane_functions.RecurrenceSequence

Numbers that are both triangular and square: $a(n) = 34a(n-1) - a(n-2) + 2$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001110; a
Numbers that are both triangular and square: a(n) = 34a(n-1) - a(n-2) + 2.
sage: a(0)
0
sage: a(1)
1
sage: a(8)
55420693056
sage: a(21)
4446390382511295358038307980025
sage: a.list(8)
[0, 1, 36, 1225, 41616, 1413721, 48024900, 1631432881]
```

AUTHORS:

- Jaap Spies (2007-01-19)

g(*k*)

EXAMPLES:

```
sage: sloane.A001110.g(2)
2
sage: sloane.A001110.g(1)
0
```

class sage.combinat.sloane_functions.**A001147**

Bases: sage.combinat.sloane_functions.SloaneSequence

Double factorial numbers: $(2n-1)!! = 1 \cdot 3 \cdot 5 \cdots (2n-1)$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001147;a
Double factorial numbers: (2n-1)!! = 1.3.5....(2n-1).
sage: a(0)
1
sage: a.offset
0
sage: a(8)
2027025
sage: a(20)
319830986772877770815625
sage: a.list(9)
[1, 1, 3, 15, 105, 945, 10395, 135135, 2027025]
```

AUTHORS:

- Jaap Spies (2007-01-24)

class sage.combinat.sloane_functions.**A001157**

Bases: sage.combinat.sloane_functions.SloaneSequence

The sequence $\sigma_2(n)$, sum of squares of divisors of n.

The function sigma(n, k) implements σ_k* in Sage.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001157;a
sigma_2(n): sum of squares of divisors of n
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
5
```

```
sage: a(8)
85
sage: a.list(9)
[1, 5, 10, 21, 26, 50, 50, 85, 91]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A001189**
Bases: sage.combinat.sloane_functions.SloaneSequence

Number of degree-n permutations of order exactly 2.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001189;a
Number of degree-n permutations of order exactly 2.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
0
sage: a(2)
1
sage: a(12)
140151
sage: a.list(13)
[0, 1, 3, 9, 25, 75, 231, 763, 2619, 9495, 35695, 140151, 568503]
```

AUTHORS:

- Jaap Spies (2007-02-03)

class sage.combinat.sloane_functions.**A001221**
Bases: sage.combinat.sloane_functions.SloaneSequence

Number of different prime divisors of n

Also called omega(n) or $\omega(n)$. Maximal number of terms in any factorization of n. Number of prime powers that divide n.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001221; a
Number of distinct primes dividing n (also called omega(n)).
sage: a(0)
Traceback (most recent call last):
```

```
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
0
sage: a(8)
1
sage: a(41)
1
sage: a(84792)
3
sage: a.list(12)
[0, 1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 2]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A001222**
Bases: sage.combinat.sloane_functions.SloaneSequence

Number of prime divisors of n (counted with multiplicity).

Also called bigomega(n) or $\Omega(n)$. Maximal number of terms in any factorization of n. Number of prime powers that divide n.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001222; a
Number of prime divisors of n (counted with multiplicity).
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
0
sage: a(8)
3
sage: a(41)
1
sage: a(84792)
5
sage: a.list(12)
[0, 1, 1, 2, 1, 2, 1, 3, 2, 2, 1, 3]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A001227**
Bases: sage.combinat.sloane_functions.SloaneSequence

Number of odd divisors of n.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001227; a
Number of odd divisors of n
sage: a.offset
1
sage: a(1)
1
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(100)
3
sage: a(256)
1
sage: a(29)
2
sage: a.list(20)
[1, 1, 2, 1, 2, 2, 2, 1, 3, 2, 2, 2, 2, 2, 4, 1, 2, 3, 2, 2]
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-14)

class sage.combinat.sloane_functions.**A001333**

Bases: sage.combinat.sloane_functions.RecurrenceSequence2

Numerators of continued fraction convergents to $\sqrt{2}$.

See also A000129

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001333;a
Numerators of continued fraction convergents to sqrt(2).
sage: a(0)
1
sage: a(1)
1
sage: a(2)
3
sage: a(3)
7
sage: a(11)
8119
```

```
sage: a.list(12)
[1, 1, 3, 7, 17, 41, 99, 239, 577, 1393, 3363, 8119]
```

AUTHORS:

- Jaap Spies (2007-02-01)

class sage.combinat.sloane_functions.**A001358**
Bases: sage.combinat.sloane_functions.SloaneSequence

Products of two primes.

These numbers have been called semiprimes (or semi-primes), biprimes or 2-almost primes.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001358;a
Products of two primes.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
6
sage: a(8)
22
sage: a(200)
669
sage: a.list(9)
[4, 6, 9, 10, 14, 15, 21, 22, 25]
```

AUTHORS:

- Jaap Spies (2007-01-25)

list(*n*)
EXAMPLES:

```
sage: sloane.A001358.list(9)
[4, 6, 9, 10, 14, 15, 21, 22, 25]
```

class sage.combinat.sloane_functions.**A001405**
Bases: sage.combinat.sloane_functions.SloaneSequence

Central binomial coefficients: $\binom{n}{\lfloor \frac{n}{2} \rfloor}$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001405;a
Central binomial coefficients: C(n,floor(n/2)).
sage: a(0)
1
sage: a(2)
2
sage: a(12)
924
sage: a.list(12)
[1, 1, 2, 3, 6, 10, 20, 35, 70, 126, 252, 462]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A001477**

Bases: sage.combinat.sloane_functions.SloaneSequence

The nonnegative integers.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001477;a
The nonnegative integers.
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(0)
0
sage: a(3382789)
3382789
sage: a(11)
11
sage: a.list(12)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A001694**

Bases: sage.combinat.sloane_functions.SloaneSequence

This function returns the n-th Powerful Number:

A positive integer n is powerful if for every prime p dividing n, p^2 also divides n.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001694; a
Powerful Numbers (also called squarefull, square-full or 2-full numbers).
sage: a.offset
1
sage: a(1)
1
sage: a(4)
9
sage: a(100)
3136
sage: a(156)
7225
sage: a.list(19)
[1, 4, 8, 9, 16, 25, 27, 32, 36, 49, 64, 72, 81, 100, 108, 121, 125, 128, 144]
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-14)

is_powerful(*n*)

This function returns True if and only if n is a Powerful Number:

A positive integer n is powerful if for every prime p dividing n, p^2 also divides n. See Sloane's OEIS A001694.

INPUT:

- n - integer

OUTPUT:

- True - if n is a Powerful number, else False

EXAMPLES:

```
sage: a = sloane.A001694
sage: a.is_powerful(2500)
True
sage: a.is_powerful(20)
False
```

AUTHORS:

- Jaap Spies (2006-12-07)

list(*n*)

EXAMPLES:

```
sage: sloane.A001694.list(9)
[1, 4, 8, 9, 16, 25, 27, 32, 36]
```

class sage.combinat.sloane_functions.**A001836**

Bases: sage.combinat.sloane_functions.SloaneSequence

Numbers n such that $\phi(2n-1) < \phi(2n)$, where ϕ is Euler's totient function.

Euler's totient function is also known as euler_phi, euler_phi is a standard Sage function.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001836; a
Numbers n such that phi(2n-1) < phi(2n), where phi is Euler's totient function A000010.
sage: a.offset
1
sage: a(1)
53
sage: a(8)
683
sage: a(300)
17798
sage: a.list(12)
[53, 83, 158, 263, 293, 368, 578, 683, 743, 788, 878, 893]
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

Compare: Searching Sloane's online database... Numbers n such that phi(2n-1) phi(2n), where phi is Euler's totient function A000010. [53, 83, 158, 263, 293, 368, 578, 683, 743, 788, 878, 893]

AUTHORS:

- Jaap Spies (2007-01-17)

list(*n*)

EXAMPLES:

```
sage: sloane.A001836.list(9)
[53, 83, 158, 263, 293, 368, 578, 683, 743]
```

class sage.combinat.sloane_functions.**A001906**

Bases: sage.combinat.sloane_functions.RecurrenceSequence2

$F(2n)$ = bisection of Fibonacci sequence: $a(n) = 3a(n-1) - a(n-2)$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001906; a
F(2n) = bisection of Fibonacci sequence: a(n)=3a(n-1)-a(n-2).
sage: a(0)
0
sage: a(1)
1
sage: a(8)
987
sage: a(22)
```

```
701408733
sage: a.list(12)
[0, 1, 3, 8, 21, 55, 144, 377, 987, 2584, 6765, 17711]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A001909**

Bases: sage.combinat.sloane_functions.ExtremesOfPermanentsSequence

$a(n) = n * a(n-1) + (n-4) * a(n-2)$, with $a(2) = 0$, $a(3) = 1$.

With offset 1, permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 4$ and n zeros not on a line. This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n - positive integer = 2

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001909;a
a(n) = n*a(n-1) + (n-4)*a(n-2), a(2) = 0, a(3) = 1.
sage: a(1)
Traceback (most recent call last):
...
ValueError: input n (=1) must be an integer >= 2
sage: a.offset
2
sage: a(2)
0
sage: a(8)
8544
sage: a(22)
470033715095287415734
sage: a.list(9)
[0, 1, 4, 21, 134, 1001, 8544, 81901, 870274]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A001910**

Bases: sage.combinat.sloane_functions.ExtremesOfPermanentsSequence

$a(n) = n * a(n-1) + (n-5) * a(n-2)$, with $a(3) = 0$, $a(4) = 1$.

With offset 1, permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 5$ and n zeros not on a line. This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n - positive integer = 3

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001910;a
a(n) = n*a(n-1) + (n-5)*a(n-2), a(3) = 0, a(4) = 1.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be an integer >= 3
sage: a(3)
0
sage: a.offset
3
sage: a(8)
1909
sage: a(22)
98125321641110663023
sage: a.list(9)
[0, 1, 5, 31, 227, 1909, 18089, 190435, 2203319]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A001969**

Bases: sage.combinat.sloane_functions.SloaneSequence

Evil numbers: even number of 1's in binary expansion.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A001969;a
Evil numbers: even number of 1's in binary expansion.
sage: a(0)
0
sage: a(1)
3
sage: a(2)
5
sage: a(12)
24
sage: a.list(13)
[0, 3, 5, 6, 9, 10, 12, 15, 17, 18, 20, 23, 24]
```

AUTHORS:

- Jaap Spies (2007-02-02)

class sage.combinat.sloane_functions.**A002110**

Bases: sage.combinat.sloane_functions.SloaneSequence

Primorial numbers (first definition): product of first n primes. Sometimes written $p\#$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A002110;a
Primorial numbers (first definition): product of first n primes. Sometimes written p#.
sage: a(0)
1
sage: a(2)
6
sage: a(8)
9699690
sage: a(17)
1922760350154212639070
sage: a.list(9)
[1, 2, 6, 30, 210, 2310, 30030, 510510, 9699690]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A002113**

Bases: sage.combinat.sloane_functions.SloaneSequence

Palindromes in base 10.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A002113;a
Palindromes in base 10.
sage: a(0)
0
sage: a(1)
1
sage: a(2)
2
sage: a(12)
33
sage: a.list(13)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 22, 33]
```

AUTHORS:

- Jaap Spies (2007-02-02)

list(*n*)

EXAMPLES:

```
sage: sloane.A002113.list(15)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 22, 33, 44, 55]
```

class sage.combinat.sloane_functions.**A002275**

Bases: sage.combinat.sloane_functions.SloaneSequence

Repunits: $\frac{(10^n-1)}{9}$. Often denoted by R_n.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A002275;a
Repunits: (10^n - 1)/9. Often denoted by R_n.
sage: a(0)
0
sage: a(2)
11
sage: a(8)
11111111
sage: a(20)
11111111111111111111
sage: a.list(9)
[0, 1, 11, 111, 1111, 11111, 111111, 1111111, 11111111]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A002378**

Bases: sage.combinat.sloane_functions.SloaneSequence

Oblong (or pronic, or heteromecic) numbers: $n(n+1)$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A002378;a
Oblong (or pronic, or heteromecic) numbers: n(n+1).
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(0)
0
sage: a(1)
2
sage: a(11)
132
sage: a.list(12)
[0, 2, 6, 12, 20, 30, 42, 56, 72, 90, 110, 132]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A002620**
Bases: sage.combinat.sloane_functions.SloaneSequence

Quarter-squares: floor(n/2)*ceiling(n/2). Equivalently, $\lfloor n^2/4 \rfloor$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A002620;a
Quarter-squares: floor(n/2)*ceiling(n/2). Equivalently, floor(n^2/4).
sage: a(0)
0
sage: a(1)
0
sage: a(2)
1
sage: a(10)
25
sage: a.list(12)
[0, 0, 1, 2, 4, 6, 9, 12, 16, 20, 25, 30]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A002808**
Bases: sage.combinat.sloane_functions.SloaneSequence

The composite numbers: numbers n of the form xy for $x > 1$ and $y > 1$.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A002808;a
The composite numbers: numbers n of the form x*y for x > 1 and y > 1.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
6
sage: a(11)
20
sage: a.list(12)
[4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21]
```

AUTHORS:

- Jaap Spies (2007-01-26)

list(*n*)

EXAMPLES:

```
sage: sloane.A002808.list(10)
[4, 6, 8, 9, 10, 12, 14, 15, 16, 18]
```

class sage.combinat.sloane_functions.**A003418**

Bases: sage.combinat.sloane_functions.SloaneSequence

Least common multiple (or lcm) of $\{1, 2, \cdots, n\}$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A003418;a
Least common multiple (or lcm) of {1, 2, ..., n}.
sage: a(0)
1
sage: a(1)
1
sage: a(13)
360360
sage: a.list(14)
[1, 1, 2, 6, 12, 60, 60, 420, 840, 2520, 2520, 27720, 27720, 360360]
sage: a(20.0)
Traceback (most recent call last):
...
TypeError: input must be an int, long, or Integer
```

AUTHOR:

- Jaap Spies (2007-01-31)

class sage.combinat.sloane_functions.**A004086**

Bases: sage.combinat.sloane_functions.SloaneSequence

Read n backwards (referred to as $R(n)$ in many sequences).

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A004086;a
Read n backwards (referred to as R(n) in many sequences).
sage: a(0)
0
sage: a(1)
1
sage: a(2)
2
sage: a(3333)
3333
```

```
sage: a(12345)
54321
sage: a.list(13)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 11, 21]
```

AUTHORS:

- Jaap Spies (2007-02-02)

class sage.combinat.sloane_functions.**A004526**
Bases: sage.combinat.sloane_functions.SloaneSequence

The nonnegative integers repeated

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A004526;a
The nonnegative integers repeated.
sage: a(0)
0
sage: a(1)
0
sage: a(2)
1
sage: a(10)
5
sage: a.list(12)
[0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A005100**
Bases: sage.combinat.sloane_functions.SloaneSequence

Deficient numbers: $\sigma(n) < 2n$.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A005100;a
Deficient numbers: sigma(n) < 2n
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
```

```
sage: a(2)
2
sage: a(12)
14
sage: a.list(12)
[1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 13, 14]
```

AUTHORS:

- Jaap Spies (2007-01-26)

list(*n*)

EXAMPLES:

```
sage: sloane.A005100.list(10)
[1, 2, 3, 4, 5, 7, 8, 9, 10, 11]
```

class sage.combinat.sloane_functions.**A005101**

Bases: sage.combinat.sloane_functions.SloaneSequence

Abundant numbers (sum of divisors of n exceeds $2n$).

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A005101;a
Abundant numbers (sum of divisors of n exceeds 2n).
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
12
sage: a(2)
18
sage: a(12)
60
sage: a.list(12)
[12, 18, 20, 24, 30, 36, 40, 42, 48, 54, 56, 60]
```

AUTHORS:

- Jaap Spies (2007-01-26)

list(*n*)

EXAMPLES:

```
sage: sloane.A005101.list(10)
[12, 18, 20, 24, 30, 36, 40, 42, 48, 54]
```

class sage.combinat.sloane_functions.**A005117**

Bases: sage.combinat.sloane_functions.SloaneSequence

Square-free numbers

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A005117;a
Square-free numbers.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
2
sage: a(12)
17
sage: a.list(12)
[1, 2, 3, 5, 6, 7, 10, 11, 13, 14, 15, 17]
```

AUTHORS:

- Jaap Spies (2007-01-25)

list(*n*)

EXAMPLES:

```
sage: sloane.A005117.list(10)
[1, 2, 3, 5, 6, 7, 10, 11, 13, 14]
```

class sage.combinat.sloane_functions.**A005408**

Bases: sage.combinat.sloane_functions.SloaneSequence

The odd numbers a(n) = 2n + 1.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A005408;a
The odd numbers a(n) = 2n + 1.
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be an integer >= 0
sage: a(0)
1
sage: a(4)
9
sage: a(11)
23
sage: a.list(12)
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
```

AUTHORS:

- Jaap Spies (2007-01-26)

class sage.combinat.sloane_functions.**A005843**
Bases: sage.combinat.sloane_functions.SloaneSequence

The even numbers: $a(n) = 2n$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A005843;a
The even numbers: a(n) = 2n.
sage: a(0.0)
Traceback (most recent call last):
...
TypeError: input must be an int, long, or Integer
sage: a(1)
2
sage: a(2)
4
sage: a(9)
18
sage: a.list(10)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

AUTHORS:

- Jaap Spies (2007-02-03)

class sage.combinat.sloane_functions.**A006318**
Bases: sage.combinat.sloane_functions.SloaneSequence

Large Schroeder numbers.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A006318;a
Large Schroeder numbers.
sage: a(0)
1
sage: a(1)
2
sage: a(2)
6
sage: a(9)
206098
sage: a.list(10)
[1, 2, 6, 22, 90, 394, 1806, 8558, 41586, 206098]
```

AUTHORS:

- Jaap Spies (2007-02-03)

class sage.combinat.sloane_functions.**A006530**
Bases: sage.combinat.sloane_functions.SloaneSequence

Largest prime dividing n (with $a(1) = 1$).

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A006530;a
Largest prime dividing n (with a(1)=1).
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
2
sage: a(8)
2
sage: a(11)
11
sage: a.list(15)
[1, 2, 3, 2, 5, 3, 7, 2, 3, 5, 11, 3, 13, 7, 5]
```

AUTHORS:

- Jaap Spies (2007-01-25)

class sage.combinat.sloane_functions.**A006882**
Bases: sage.combinat.sloane_functions.SloaneSequence

Double factorials $n!!$: $a(n) = n \cdot a(n-2)$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A006882;a
Double factorials n!!: a(n)=n*a(n-2).
sage: a(0)
1
sage: a(2)
2
sage: a(8)
384
sage: a(20)
3715891200
sage: a.list(9)
[1, 1, 2, 3, 8, 15, 48, 105, 384]
```

AUTHORS:

- Jaap Spies (2007-01-24)

df()

Double factorials n!!: a(n)=n*a(n-2).

EXAMPLES:

```
sage: it = sloane.A006882.df()
sage: [next(it) for i in range(10)]
[1, 1, 2, 3, 8, 15, 48, 105, 384, 945]
```

list(*n*)

EXAMPLES:

```
sage: sloane.A006882.list(10)
[1, 1, 2, 3, 8, 15, 48, 105, 384, 945]
```

class sage.combinat.sloane_functions.**A007318**

Bases: sage.combinat.sloane_functions.SloaneSequence

Pascal's triangle read by rows: $C(n,k) = \binom{n}{k} = \frac{n!}{(k!(n-k)!)}, 0 \le k \le n$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A007318
sage: a(0)
1
sage: a(1)
1
sage: a(13)
4
sage: a.list(15)
[1, 1, 1, 1, 2, 1, 1, 3, 3, 1, 1, 4, 6, 4, 1]
sage: a(100)
715
```

AUTHORS:

- Jaap Spies (2007-01-31)

class sage.combinat.sloane_functions.**A008275**

Bases: sage.combinat.sloane_functions.SloaneSequence

Triangle of Stirling numbers of first kind, $s(n,k)$, $n \ge 1$, $1 \le k \le n$.

The unsigned numbers are also called Stirling cycle numbers:

$|s(n,k)|$ = number of permutations of n objects with exactly k cycles.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A008275;a
Triangle of Stirling numbers of first kind, s(n,k), n >= 1, 1<=k<=n.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
-1
sage: a(3)
1
sage: a(11)
24
sage: a.list(12)
[1, -1, 1, 2, -3, 1, -6, 11, -6, 1, 24, -50]
```

AUTHORS:

- Jaap Spies (2007-02-02)

s (n, k)

EXAMPLES:

```
sage: sloane.A008275.s(4,2)
11
sage: sloane.A008275.s(5,2)
-50
sage: sloane.A008275.s(5,3)
35
```

class sage.combinat.sloane_functions.**A008277**

Bases: sage.combinat.sloane_functions.SloaneSequence

Triangle of Stirling numbers of 2nd kind, $S2(n, k)$, $n \geq 1$, $1 \leq k \leq n$.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A008277;a
Triangle of Stirling numbers of 2nd kind, S2(n,k), n >= 1, 1<=k<=n.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
1
sage: a(3)
1
sage: a(4.0)
Traceback (most recent call last):
...
```

```
TypeError: input must be an int, long, or Integer
sage: a.list(15)
[1, 1, 1, 1, 3, 1, 1, 7, 6, 1, 1, 15, 25, 10, 1]
```

AUTHORS:

- Jaap Spies (2007-01-31)

s2(n, k)

Returns the Stirling number S2(n,k) of the 2nd kind.

EXAMPLES:

```
sage: sloane.A008277.s2(4,2)
7
```

class sage.combinat.sloane_functions.**A008683**

Bases: sage.combinat.sloane_functions.SloaneSequence

Möbius function $\mu(n)$.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A008683;a
Moebius function mu(n).
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
-1
sage: a(12)
0
sage: a.list(12)
[1, -1, -1, 0, -1, 1, -1, 0, 0, 1, -1, 0]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A010060**

Bases: sage.combinat.sloane_functions.SloaneSequence

Thue-Morse sequence.

Let A_k denote the first 2^k terms; then $A_0 = 0$, and for $k \geq 0$, $A_{k+1} = A_k B_k$, where B_k is obtained from A_k by interchanging 0's and 1's.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A010060;a
Thue-Morse sequence.
sage: a(0)
0
sage: a(1)
1
sage: a(2)
1
sage: a(12)
0
sage: a.list(13)
[0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0]
```

AUTHORS:

- Jaap Spies (2007-02-02)

class sage.combinat.sloane_functions.**A015521**

Bases: sage.combinat.sloane_functions.RecurrenceSequence2

Linear 2nd order recurrence, $a(0) = 0$, $a(1) = 1$ and $a(n) = 3a(n-1) + 4a(n-2)$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A015521; a
Linear 2nd order recurrence, a(n) = 3 a(n-1) + 4 a(n-2).
sage: a(0)
0
sage: a(1)
1
sage: a(8)
13107
sage: a(41)
967140655691703339764941
sage: a.list(12)
[0, 1, 3, 13, 51, 205, 819, 3277, 13107, 52429, 209715, 838861]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A015523**

Bases: sage.combinat.sloane_functions.RecurrenceSequence2

Linear 2nd order recurrence, $a(0) = 0$, $a(1) = 1$ and $a(n) = 3a(n-1) + 5a(n-2)$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A015523; a
Linear 2nd order recurrence, a(n) = 3 a(n-1) + 5 a(n-2).
sage: a(0)
0
sage: a(1)
1
sage: a(8)
17727
sage: a(41)
617371956647452973909148 1
sage: a.list(12)
[0, 1, 3, 14, 57, 241, 1008, 4229, 17727, 74326, 311613, 1306469]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A015530**

Bases: sage.combinat.sloane_functions.RecurrenceSequence2

Linear 2nd order recurrence, $a(0) = 0$, $a(1) = 1$ and $a(n) = 4a(n-1) + 3a(n-2)$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A015530;a
Linear 2nd order recurrence, a(n) = 4 a(n-1) + 3 a(n-2).
sage: a(0)
0
sage: a(1)
1
sage: a(2)
4
sage: a.offset
0
sage: a(8)
41008
sage: a.list(9)
[0, 1, 4, 19, 88, 409, 1900, 8827, 41008]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A015531**

Bases: sage.combinat.sloane_functions.RecurrenceSequence2

Linear 2nd order recurrence, $a(0) = 0$, $a(1) = 1$ and $a(n) = 4a(n-1) + 5a(n-2)$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A015531;a
Linear 2nd order recurrence, a(n) = 4 a(n-1) + 5 a(n-2).
sage: a(0)
0
sage: a(1)
1
sage: a(2)
4
sage: a.offset
0
sage: a(8)
65104
sage: a(60)
1445602896647339245343270401159922281901 04
sage: a.list(9)
[0, 1, 4, 21, 104, 521, 2604, 13021, 65104]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A015551**

Bases: sage.combinat.sloane_functions.RecurrenceSequence2

Linear 2nd order recurrence, $a(0) = 0$, $a(1) = 1$ and $a(n) = 6a(n-1) + 5a(n-2)$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A015551;a
Linear 2nd order recurrence, a(n) = 6 a(n-1) + 5 a(n-2).
sage: a(0)
0
sage: a(1)
1
sage: a(2)
6
sage: a.offset
0
sage: a(8)
570216
sage: a(60)
711060660653005973676148455715586382253197057 3036
sage: a.list(9)
[0, 1, 6, 41, 276, 1861, 12546, 84581, 570216]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A018252**

Bases: sage.combinat.sloane_functions.SloaneSequence

The nonprime numbers, starting with 1.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A018252;a
The nonprime numbers.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
4
sage: a(9)
15
sage: a.list(10)
[1, 4, 6, 8, 9, 10, 12, 14, 15, 16]
```

AUTHORS:

- Jaap Spies (2007-02-04)

class sage.combinat.sloane_functions.**A020639**

Bases: sage.combinat.sloane_functions.SloaneSequence

Least prime dividing n with $a(1) = 1$.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A020639;a
Least prime dividing n (a(1)=1).
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(13)
13
sage: a.list(14)
[1, 2, 3, 2, 5, 2, 7, 2, 3, 2, 11, 2, 13, 2]
```

AUTHORS:

- Jaap Spies (2007-01-25)

list(*n*)

EXAMPLES:

```
sage: sloane.A020639.list(10)
[1, 2, 3, 2, 5, 2, 7, 2, 3, 2]
```

class sage.combinat.sloane_functions.**A046660**(*offset=1*)

Bases: sage.combinat.sloane_functions.SloaneSequence

Excess of n = number of prime divisors (with multiplicity) - number of prime divisors (without multiplicity).

$\Omega(n) - \omega(n)$.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A046660; a
Excess of n = Bigomega (with multiplicity) - omega (without multiplicity).
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
0
sage: a(8)
2
sage: a(41)
0
sage: a(84792)
2
sage: a.list(12)
[0, 0, 0, 1, 0, 0, 0, 2, 1, 0, 0, 1]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A049310**

Bases: sage.combinat.sloane_functions.SloaneSequence

Triangle of coefficients of Chebyshev's $S(n,x)$: $U(n, \frac{x}{2})$ polynomials (exponents in increasing order).

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A049310;a
Triangle of coefficients of Chebyshev's S(n,x) := U(n,x/2) polynomials (exponents in increasing
sage: a(0)
1
sage: a(1)
0
sage: a(13)
0
```

```
sage: a.list(15)
[1, 0, 1, -1, 0, 1, 0, -2, 0, 1, 1, 0, -3, 0, 1]
sage: a(200)
0
sage: a.keyword
['sign', 'tabl', 'nice', 'easy', 'core', 'triangle']
```

AUTHORS:

- Jaap Spies (2007-01-31)

class sage.combinat.sloane_functions.**A051959**

Bases: sage.combinat.sloane_functions.RecurrenceSequence

Linear second order recurrence. A051959.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A051959; a
Linear second order recurrence. A051959.
sage: a(0)
1
sage: a(1)
10
sage: a(8)
9969
sage: a(41)
42834431872413650
sage: a.list(12)
[1, 10, 36, 104, 273, 686, 1688, 4112, 9969, 24114, 58268, 140728]
```

AUTHORS:

- Jaap Spies (2007-01-19)

g(*k*)

EXAMPLES:

```
sage: sloane.A051959.g(2)
15
sage: sloane.A051959.g(1)
0
```

class sage.combinat.sloane_functions.**A055790**

Bases: sage.combinat.sloane_functions.ExtremesOfPermanentsSequence2

$a(n) = n * a(n-1) + (n-2) * a(n-2)[a(0) = 0, a(1) = 2]$.

With offset 1, permanent of (0,1)-matrix of size n X (n+d) with d=1 and n-1 zeros not on a line. This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

REFERENCES:

- Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A055790;a
a(n) = n*a(n-1) + (n-2)*a(n-2) [a(0) = 0, a(1) = 2].
sage: a(0)
0
sage: a(1)
2
sage: a(2)
4
sage: a.offset
0
sage: a(8)
165016
sage: a(22)
10356214297533070441564
sage: a.list(9)
[0, 2, 4, 14, 64, 362, 2428, 18806, 165016]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A061084**

Bases: sage.combinat.sloane_functions.SloaneSequence

Fibonacci-type sequence based on subtraction: $a(0) = 1$, $a(1) = 2$ and $a(n) = a(n-2) - a(n-1)$.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A061084; a
Fibonacci-type sequence based on subtraction: a(0) = 1, a(1) = 2 and a(n) = a(n-2)-a(n-1).
sage: a(0)
1
sage: a(1)
2
sage: a(8)
-29
sage: a(22)
-24476
sage: a.list(12)
[1, 2, -1, 3, -4, 7, -11, 18, -29, 47, -76, 123]
sage: a.keyword
['sign', 'easy', 'nice']
```

AUTHORS:

- Jaap Spies (2007-01-18)

class sage.combinat.sloane_functions.**A064553**
Bases: sage.combinat.sloane_functions.SloaneSequence

$a(1) = 1$, $a(prime(i)) = i + 1$ for $i > 0$ and $a(u \cdot v) = a(u) \cdot a(v)$ for $u, v > 0$.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A064553;a
a(1) = 1, a(prime(i)) = i+1 for i > 0 and a(u*v) = a(u)*a(v) for u,v > 0
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
1
sage: a(2)
2
sage: a(9)
9
sage: a.list(16)
[1, 2, 3, 4, 4, 6, 5, 8, 9, 8, 6, 12, 7, 10, 12, 16]
```

AUTHORS:

- Jaap Spies (2007-02-04)

class sage.combinat.sloane_functions.**A079922**(*offset=1*)
Bases: sage.combinat.sloane_functions.SloaneSequence

function returns solutions to the Dancing School problem with n girls and $n + 3$ boys.

The value is $per(B)$, the permanent of the (0,1)-matrix B of size $n \times n + 3$ with $b(i, j) = 1$ if and only if $i \leq j \leq i + n$.

REFERENCES:

- Jaap Spies, Nieuw Archief voor Wiskunde, 5/7 nr 4, December 2006

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A079922; a
Solutions to the Dancing School problem with n girls and n+3 boys
sage: a.offset
1
sage: a(1)
4
sage: a(8)
2227
sage: a.list(8)
[4, 13, 36, 90, 212, 478, 1044, 2227]
```

Compare: Searching Sloane's online database... Solution to the Dancing School Problem with n girls and n+3 boys: f(n,3). [4, 13, 36, 90, 212, 478, 1044, 2227]

```
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-14)

class sage.combinat.sloane_functions.**A079923**(*offset=1*)
Bases: sage.combinat.sloane_functions.SloaneSequence

function returns solutions to the Dancing School problem with n girls and $n + 4$ boys.

The value is $per(B)$, the permanent of the (0,1)-matrix B of size $n \times n + 3$ with $b(i, j) = 1$ if and only if $i \leq j \leq i + n$.

REFERENCES:

- Jaap Spies, Nieuw Archief voor Wiskunde, 5/7 nr 4, December 2006

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A079923; a
Solutions to the Dancing School problem with n girls and n+4 boys
sage: a.offset
1
sage: a(1)
5
sage: a(8)
15458
sage: a.list(8)
[5, 21, 76, 246, 738, 2108, 5794, 15458]
```

Compare: Searching Sloane's online database... Solution to the Dancing School Problem with n girls and n+4 boys: f(n,4). [5, 21, 76, 246, 738, 2108, 5794, 15458]

```
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-17)

class sage.combinat.sloane_functions.**A082411**
Bases: sage.combinat.sloane_functions.RecurrenceSequence2

Second-order linear recurrence sequence with $a(n) = a(n - 1) + a(n - 2)$.

$a(0) = 407389224418$, $a(1) = 76343678551$. This is the second-order linear recurrence sequence with $a(0)$ and $a(1)$ co-prime, that R. L. Graham in 1964 stated did not contain any primes.

INPUT:

- `n` - non negative integer

OUTPUT:

- `integer` - function value

EXAMPLES:

```
sage: a = sloane.A082411;a
Second-order linear recurrence sequence with a(n) = a(n-1) + a(n-2).
sage: a(1)
76343678551
sage: a(2)
483732902969
sage: a(3)
560076581520
sage: a(20)
2219759332689173
sage: a.list(4)
[407389224418, 76343678551, 483732902969, 560076581520]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class `sage.combinat.sloane_functions.`**A083103**

Bases: `sage.combinat.sloane_functions.RecurrenceSequence2`

Second-order linear recurrence sequence with $a(n) = a(n-1) + a(n-2)$.

$a(0) = 1786772701928802632268715130455793$, $a(1) = 1059683225053915111058165141686995$. This is the second-order linear recurrence sequence with $a(0)$ and $a(1)$ co- prime, that R. L. Graham in 1964 stated did not contain any primes. It has not been verified. Graham made a mistake in the calculation that was corrected by D. E. Knuth in 1990.

INPUT:

- `n` - non negative integer

OUTPUT:

- `integer` - function value

EXAMPLES:

```
sage: a = sloane.A083103;a
Second-order linear recurrence sequence with a(n) = a(n-1) + a(n-2).
sage: a(1)
1059683225053915111058165141686995
sage: a(2)
2846455926982717743326880272142788
sage: a(3)
3906139152036632854385045413829783
sage: a.offset
0
sage: a(8)
4548139285120665155171476467135220 4
sage: a(20)
1463925368425405953182398514394819 1708
```

```
sage: a.list(4)
[178677270192880263226871513045579 3, 10596832250539151110581651416869 95, 2846455926982717743326 8
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A083104**
Bases: sage.combinat.sloane_functions.RecurrenceSequence2

Second-order linear recurrence sequence with $a(n) = a(n-1) + a(n-2)$.

$a(0) = 331635635998274737472200656430763$, $a(1) = 151002891108840197118959030549878 5$. This is the second-order linear recurrence sequence with $a(0)$ and $a(1)$ co-prime. It was found by Ronald Graham in 1990.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A083104;a
Second-order linear recurrence sequence with a(n) = a(n-1) + a(n-2).
sage: a(3)
335169345817507867985138126742 8333
sage: a.offset
0
sage: a(8)
360218704008340129821200049490 74404
sage: a(20)
116019141776218260124688493612363006 28
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A083105**
Bases: sage.combinat.sloane_functions.RecurrenceSequence2

Second-order linear recurrence sequence with $a(n) = a(n-1) + a(n-2)$.

$a(0) = 62638280004239857$, $a(1) = 49463435743205655$. This is the second-order linear recurrence sequence with $a(0)$ and $a(1)$ co-prime. It was found by Donald Knuth in 1990.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A083105;a
Second-order linear recurrence sequence with a(n) = a(n-1) + a(n-2).
sage: a(1)
49463435743205655
sage: a(2)
112101715747445512
```

```
sage: a(3)
161565151490651167
sage: a.offset
0
sage: a(8)
1853029790662436896
sage: a(20)
596510791500513098192
sage: a.list(4)
[62638280004239857, 4946343574320565, 112101715747445512, 161565151490651167]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A083216**

Bases: sage.combinat.sloane_functions.RecurrenceSequence2

Second-order linear recurrence sequence with $a(n) = a(n-1) + a(n-2)$.

$a(0) = 20615674205555510$, $a(1) = 3794765361567513$. This is a second-order linear recurrence sequence with $a(0)$ and $a(1)$ co-prime that does not contain any primes. It was found by Herbert Wilf in 1990.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A083216; a
Second-order linear recurrence sequence with a(n) = a(n-1) + a(n-2).
sage: a(0)
20615674205555510
sage: a(1)
3794765361567513
sage: a(8)
347693837265139403
sage: a(41)
2738025383211084205003383
sage: a.list(4)
[20615674205555510, 3794765361567513, 24410439567123023, 28205204928690536]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class sage.combinat.sloane_functions.**A090010**

Bases: sage.combinat.sloane_functions.ExtremesOfPermanentsSequence2

Permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 6$ and n zeros not on a line.

‘ a(n) = (n+5)*a(n-1) + (n-1)*a(n-2), a(1)=6, a(2)=43‘.

This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

REFERENCES:

- Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A090010;a
Permanent of (0,1)-matrix of size n X (n+d) with d=6 and n zeros not on a line.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
6
sage: a(2)
43
sage: a.offset
1
sage: a(8)
67741129
sage: a(22)
19241659302915898900327 0143
sage: a.list(9)
[6, 43, 356, 3333, 34754, 398959, 4996032, 67741129, 988344062]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A090012**

Bases: sage.combinat.sloane_functions.SloaneSequence

Permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 2$ and $n - 1$ zeros not on a line.

$a(n) = (n+1) * a(n-1) + (n-2) * a(n-2)$, $a(1) = 3$ and $a(2) = 9$

This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

REFERENCES:

- Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A090012;a
Permanent of (0,1)-matrix of size n X (n+d) with d=2 and n-1 zeros not on a line.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
```

```
3
sage: a(2)
9
sage: a.offset
1
sage: a(8)
890901
sage: a(22)
129020386652297208795129
sage: a.list(9)
[3, 9, 39, 213, 1395, 10617, 91911, 890901, 9552387]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A090013**

Bases: sage.combinat.sloane_functions.SloaneSequence

Permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 3$ and $n - 1$ zeros not on a line.

$a(n) = (n+1) * a(n-1) + (n-2) * a(n-2)[a(1) = 4, a(2) = 16]$

This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

REFERENCES:

- Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A090013;a
Permanent of (0,1)-matrix of size n X (n+d) with d=3 and n-1 zeros not on a line.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
4
sage: a(2)
16
sage: a.offset
1
sage: a(8)
3481096
sage: a(22)
1112998577171142607670336
sage: a.list(9)
[4, 16, 84, 536, 4004, 34176, 327604, 3481096, 40585284]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A090014**
Bases: sage.combinat.sloane_functions.SloaneSequence

Permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 4$ and $n-1$ zeros not on a line.

$a(n) = (n+1) * a(n-1) + (n-2) * a(n-2)[a(1) = 5, a(2) = 25]$

This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

REFERENCES:

- Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A090014;a
Permanent of (0,1)-matrix of size n X (n+d) with d=4 and n-1 zeros not on a line.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
5
sage: a(2)
25
sage: a.offset
1
sage: a(8)
11016595
sage: a(22)
7469733600354446865509725
sage: a.list(9)
[5, 25, 155, 1135, 9545, 90445, 952175, 11016595, 138864365]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A090015**
Bases: sage.combinat.sloane_functions.SloaneSequence

Permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 5$ and $n-1$ zeros not on a line.

$a(n) = (n+1) * a(n-1) + (n-2) * a(n-2)[a(1) = 6, a(2) = 36]$

This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

REFERENCES:

- Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A090015;a
Permanent of (0,1)-matrix of size n X (n+d) with d=3 and n-1 zeros not on a line.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
6
sage: a(2)
36
sage: a.offset
1
sage: a(8)
29976192
sage: a(22)
415522585176921167949368 76
sage: a.list(9)
[6, 36, 258, 2136, 19998, 208524, 2393754, 29976192, 406446774]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A090016**

Bases: sage.combinat.sloane_functions.SloaneSequence

Permanent of (0,1)-matrix of size $n \times (n+d)$ with $d = 6$ and $n-1$ zeros not on a line.

$a(n) = (n+1) * a(n-1) + (n-2) * a(n-2)[a(1) = 7, a(2) = 49]$

$A090016a(n) = A090010(n-1) + A090010(n), a(1) = 7$

This is a special case of Theorem 2.3 of Seok-Zun Song et al. Extremes of permanents of (0,1)-matrices, p. 201-202.

REFERENCES:

- Seok-Zun Song et al., Extremes of permanents of (0,1)-matrices, Lin. Algebra and its Applic. 373 (2003), p. 197-210.

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A090016;a
Permanent of (0,1)-matrix of size n X (n+d) with d=6 and n-1 zeros not on a line.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
```

```
7
sage: a(2)
49
sage: a.offset
1
sage: a(8)
72737161
sage: a(22)
199341969448774341802426289
sage: a.list(9)
[7, 49, 399, 3689, 38087, 433713, 5394991, 72737161, 1056085191]
```

AUTHORS:

- Jaap Spies (2007-01-23)

class sage.combinat.sloane_functions.**A109814**
Bases: sage.combinat.sloane_functions.SloaneSequence

The n th term of the sequence $a(n)$ is the largest k such that n can be written as sum of k consecutive integers.

By definition, n is the sum of at most $a(n)$ consecutive positive integers. Suppose n is to be written as sum of k consecutive integers starting with m, then $2n = k(2m + k - 1)$. Only one of the factors is odd. For each odd divisor d of n there is a unique corresponding $k = min(d, 2n/d)$. $a(n)$ can be alternatively defined as the largest among those k .

See also:

- Wikipedia article on polite numbers.
- An exercise sheet (with answers) about sums of consecutive integers.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A109814; a
a(n) is the largest k such that n can be written as sum of k consecutive positive integers.
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(2)
1
sage: a.list(9)
[1, 1, 2, 1, 2, 3, 2, 1, 3]
```

AUTHORS:

- Jaap Spies (2007-01-13)

class sage.combinat.sloane_functions.**A111774**
Bases: sage.combinat.sloane_functions.SloaneSequence

Sequence of numbers of the third kind, i.e., numbers that can be written as a sum of at least three consecutive positive integers.

Odd primes can only be written as a sum of two consecutive integers. Powers of 2 do not have a representation as a sum of k consecutive integers (other than the trivial $n = n$ for $k = 1$).

See: http://www.jaapspies.nl/mathfiles/problem2005-2C.pdf

INPUT:

- `n` - positive integer

OUTPUT:

- `integer` - function value

EXAMPLES:

```
sage: a = sloane.A111774; a
Numbers that can be written as a sum of at least three consecutive positive integers.
sage: a(1)
6
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(100)
141
sage: a(156)
209
sage: a(302)
386
sage: a.list(12)
[6, 9, 10, 12, 14, 15, 18, 20, 21, 22, 24, 25]
sage: a(1/3)
Traceback (most recent call last):
...
TypeError: input must be an int, long, or Integer
```

AUTHORS:

- Jaap Spies (2007-01-13)

is_number_of_the_third_kind(n)

This function returns True if and only if n is a number of the third kind.

A number is of the third kind if it can be written as a sum of at least three consecutive positive integers. Odd primes can only be written as a sum of two consecutive integers. Powers of 2 do not have a representation as a sum of k consecutive integers (other than the trivial $n = n$ for $k = 1$).

See: http://www.jaapspies.nl/mathfiles/problem2005-2C.pdf

INPUT:

- `n` - positive integer

OUTPUT:

- `True` - if n is not prime and not a power of 2 False -

EXAMPLES:

```
sage: a = sloane.A111774
sage: a.is_number_of_the_third_kind(6)
True
sage: a.is_number_of_the_third_kind(100)
True
sage: a.is_number_of_the_third_kind(16)
```

```
False
sage: a.is_number_of_the_third_kind(97)
False
```

AUTHORS:

- Jaap Spies (2006-12-09)

list(*n*)

EXAMPLES:

```
sage: sloane.A111774.list(12)
[6, 9, 10, 12, 14, 15, 18, 20, 21, 22, 24, 25]
```

class sage.combinat.sloane_functions.**A111775**

Bases: sage.combinat.sloane_functions.SloaneSequence

Number of ways n can be written as a sum of at least three consecutive integers.

Powers of 2 and (odd) primes can not be written as a sum of at least three consecutive integers. $a(n)$ strongly depends on the number of odd divisors of n (A001227): Suppose n is to be written as sum of k consecutive integers starting with m, then $2n = k(2m + k - 1)$. Only one of the factors is odd. For each odd divisor of n there is a unique corresponding k, $k = 1$ and $k = 2$ must be excluded.

See: http://www.jaapspies.nl/mathfiles/problem2005-2C.pdf

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A111775; a
Number of ways n can be written as a sum of at least three consecutive integers.

sage: a(1)
0
sage: a(0)
0
```

We have a(15)=2 because 15 = 4+5+6 and 15 = 1+2+3+4+5. The number of odd divisors of 15 is 4.

```
sage: a(15)
2

sage: a(100)
2
sage: a(256)
0
sage: a(29)
0
sage: a.list(20)
[0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 2, 0, 0, 2, 0]
sage: a(1/3)
Traceback (most recent call last):
...
TypeError: input must be an int, long, or Integer
```

AUTHORS:

- Jaap Spies (2006-12-09)

class sage.combinat.sloane_functions.**A111787**
Bases: sage.combinat.sloane_functions.SloaneSequence

This function returns the n-th number of Sloane's sequence A111787

$a(n) = 0$ if n is an odd prime or a power of 2. For numbers of the third kind (see A111774) we proceed as follows: suppose n is to be written as sum of k consecutive integers starting with m, then $2n = k(2m + k - 1)$. Let p be the smallest odd prime divisor of n then $a(n) = min(p, 2n/p)$.

See: http://www.jaapspies.nl/mathfiles/problem2005-2C.pdf

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A111787; a
a(n) is the least k >= 3 such that n can be written as sum of k consecutive integers. a(n)=0 if
sage: a.offset
1
sage: a(1)
0
sage: a(0)
Traceback (most recent call last):
...
ValueError: input n (=0) must be a positive integer
sage: a(100)
5
sage: a(256)
0
sage: a(29)
0
sage: a.list(20)
[0, 0, 0, 0, 0, 3, 0, 0, 3, 4, 0, 3, 0, 4, 3, 0, 0, 3, 0, 5]
sage: a(-1)
Traceback (most recent call last):
...
ValueError: input n (=-1) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-14)

class sage.combinat.sloane_functions.**ExponentialNumbers**(*a*)
Bases: sage.combinat.sloane_functions.SloaneSequence

A sequence of Exponential numbers.

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import ExponentialNumbers
sage: ExponentialNumbers(0)
Sequence of Exponential numbers around 0
```

class sage.combinat.sloane_functions.**ExtremesOfPermanentsSequence**(*offset=1*)
Bases: sage.combinat.sloane_functions.SloaneSequence

A sequence starting at offset (=1 by default).

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import SloaneSequence
sage: SloaneSequence().offset
1
sage: SloaneSequence(4).offset
4
```

gen(*a0*, *a1*, *d*)
EXAMPLES:

```
sage: it = sloane.A000153.gen(0,1,2)
sage: [next(it) for i in range(5)]
[0, 1, 2, 7, 32]
```

list(*n*)
EXAMPLES:

```
sage: sloane.A000153.list(8)
[0, 1, 2, 7, 32, 181, 1214, 9403]
```

class sage.combinat.sloane_functions.**ExtremesOfPermanentsSequence2**(*offset=1*)
Bases: sage.combinat.sloane_functions.ExtremesOfPermanentsSequence

A sequence starting at offset (=1 by default).

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import SloaneSequence
sage: SloaneSequence().offset
1
sage: SloaneSequence(4).offset
4
```

gen(*a0*, *a1*, *d*)
EXAMPLES:

```
sage: from sage.combinat.sloane_functions import ExtremesOfPermanentsSequence2
sage: e = ExtremesOfPermanentsSequence2()
sage: it = e.gen(6,43,6)
sage: [next(it) for i in range(5)]
[6, 43, 307, 2542, 23799]
```

class sage.combinat.sloane_functions.**RecurrenceSequence**(*offset=1*)
Bases: sage.combinat.sloane_functions.SloaneSequence

A sequence starting at offset (=1 by default).

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import SloaneSequence
sage: SloaneSequence().offset
1
sage: SloaneSequence(4).offset
4
```

list(*n*)
EXAMPLES:

```
sage: sloane.A001110.list(8)
[0, 1, 36, 1225, 41616, 1413721, 48024900, 1631432881]
```

class sage.combinat.sloane_functions.**RecurrenceSequence2**(*offset=1*)
Bases: sage.combinat.sloane_functions.SloaneSequence

A sequence starting at offset (=1 by default).

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import SloaneSequence
sage: SloaneSequence().offset
1
sage: SloaneSequence(4).offset
4
```

list(*n*)
EXAMPLES:

```
sage: sloane.A001906.list(10)
[0, 1, 3, 8, 21, 55, 144, 377, 987, 2584]
```

class sage.combinat.sloane_functions.**Sloane**
Bases: sage.structure.sage_object.SageObject

A collection of Sloane generating functions.

This class inspects sage.combinat.sloane_functions, accumulating all the SloaneSequence classes starting with 'A'. These are listed for tab completion, but not instantiated until requested.

EXAMPLES: Ensure we have lots of entries:

```
sage: len(sloane.trait_names()) > 100
True
```

And ensure none are being incorrectly returned:

```
sage: [ None for n in sloane.trait_names() if not n.startswith('A') ]
[]
```

Ensure we can access dynamic constructions and cache correctly:

```
sage: s = sloane.A000587
sage: s is sloane.A000587
True
```

And that we can access other functions in parent classes:

```
sage: sloane.__class__
<class 'sage.combinat.sloane_functions.Sloane'>
```

AUTHORS:

- Nick Alexander

trait_names()
List Sloane generating functions for tab-completion. The member classes are inspected from module sage.combinat.sloane_functions.

They must be sub classes of SloaneSequence and must start with 'A'. These restrictions are only to prevent typos, incorrect inspecting, etc.

EXAMPLES:

```
sage: type(sloane.trait_names())
<type 'list'>
```

class sage.combinat.sloane_functions.**SloaneSequence**(*offset=1*)
Bases: sage.structure.sage_object.SageObject

Base class for a Sloane integer sequence.

EXAMPLES:

We create a dummy sequence:

list(*n*)
Return n terms of the sequence: sequence[offset], sequence[offset+1], ... , sequence[offset+n-1]. EXAMPLES:

```
sage: sloane.A000012.list(4)
[1, 1, 1, 1]
```

sage.combinat.sloane_functions.**perm_mh**(*m*, *h*)
This functions calculates $f(g, h)$ from Sloane's sequences A079908-A079928

INPUT:

- m - positive integer
- h - non negative integer

OUTPUT: permanent of the m x (m+h) matrix, etc.

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import perm_mh
sage: perm_mh(3,3)
36
sage: perm_mh(3,4)
76
```

AUTHORS:

- Jaap Spies (2006)

sage.combinat.sloane_functions.**recur_gen2**(*a0*, *a1*, *a2*, *a3*)
homogeneous general second-order linear recurrence generator with fixed coefficients

a(0) = a0, a(1) = a1, a(n) = a2*a(n-1) + a3*a(n-2)

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import recur_gen2
sage: it = recur_gen2(1,1,1,1)
sage: [next(it) for i in range(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

sage.combinat.sloane_functions.**recur_gen2b**(*a0*, *a1*, *a2*, *a3*, *b*)
inhomogenous second-order linear recurrence generator with fixed coefficients and $b = f(n)$

$a(0) = a0, a(1) = a1, a(n) = a2 * a(n-1) + a3 * a(n-2) + f(n)$.

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import recur_gen2b
sage: it = recur_gen2b(1,1,1,1, lambda n: 0)
sage: [next(it) for i in range(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

sage.combinat.sloane_functions.**recur_gen3**(*a0*, *a1*, *a2*, *a3*, *a4*, *a5*)
homogeneous general third-order linear recurrence generator with fixed coefficients

a(0) = a0, a(1) = a1, a(2) = a2, a(n) = a3*a(n-1) + a4*a(n-2) + a5*a(n-3)

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import recur_gen3
sage: it = recur_gen3(1,1,1,1,1,1)
sage: [next(it) for i in range(10)]
[1, 1, 1, 3, 5, 9, 17, 31, 57, 105]
```

5.1.279 Combinatorial Species

Todo

Short blurb about species

Todo

Proofread / point to the main classes rather than the modules?

Introductory material

- *Enumeration of trees using generating functions*
- *Species, decomposable combinatorial classes*

Lazy Power Series

- *Streams or Infinite Arrays*
- *Series Order*
- *Lazy Power Series*
- *Generating Series*

Basic Species

- *Combinatorial Species*
- *Empty Species*
- *Recursive Species*
- *Characteristic Species*
- *Cycle Species*
- *Partition Species*
- *Permutation species*

- *Linear-order Species*
- *Set Species*
- *Subset Species*
- *Examples of Combinatorial Species*

Operations on Species

- *Sum species*
- *Sum species*
- *Composition species*
- *Functorial composition species*

Miscellaneous

- *Species structures*
- *Miscellaneous Functions*
- *Combinatorial Logarithm*

5.1.280 Combinatorial species features that are imported by default in the interpreter namespace

5.1.281 Characteristic Species

class sage.combinat.species.characteristic_species.**CharacteristicSpecies**(*n*, *min=None*, *max=None*, *weight=None*)

Bases: sage.combinat.species.species.GenericCombinatorialSpecies, sage.structure.unique_representation.UniqueRepresentation

Returns the characteristic species of order n.

This species has exactly one structure on a set of of size n and no structures of on sets of any other size.

EXAMPLES:

```
sage: X = species.CharacteristicSpecies(1)
sage: X.structures([1]).list()
[1]
sage: X.structures([1,2]).list()
[]
sage: X.generating_series().coefficients(4)
[0, 1, 0, 0]
sage: X.isotype_generating_series().coefficients(4)
[0, 1, 0, 0]
sage: X.cycle_index_series().coefficients(4)
[0, p[1], 0, 0]

sage: F = species.CharacteristicSpecies(3)
sage: c = F.generating_series().coefficients(4)
sage: F._check()
```

```
True
sage: F == loads(dumps(F))
True
```

TESTS:

```
sage: S1 = species.CharacteristicSpecies(1)
sage: S2 = species.CharacteristicSpecies(1)
sage: S3 = species.CharacteristicSpecies(2)
sage: S4 = species.CharacteristicSpecies(2, weight=2)
sage: S1 is S2
True
sage: S1 == S3
False
```

class sage.combinat.species.characteristic_species.**CharacteristicSpeciesStructure**(*parent*, *labels*, *list*)

Bases: sage.combinat.species.structure.GenericSpeciesStructure

This is a base class from which the classes for the structures inherit.

EXAMPLES:

```
sage: from sage.combinat.species.structure import GenericSpeciesStructure
sage: a = GenericSpeciesStructure(None, [2,3,4], [1,2,3])
sage: a
[2, 3, 4]
sage: a.parent() is None
True
sage: a == loads(dumps(a))
True
```

automorphism_group()

Returns the group of permutations whose action on this structure leave it fixed. For the characteristic species, there is only one structure, so every permutation is in its automorphism group.

EXAMPLES:

```
sage: F = species.CharacteristicSpecies(3)
sage: a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
sage: a.automorphism_group()
Symmetric group of order 3! as a permutation group
```

canonical_label()

EXAMPLES:

```
sage: F = species.CharacteristicSpecies(3)
sage: a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
sage: a.canonical_label()
{'a', 'b', 'c'}
```

transport(*perm*)

Returns the transport of this structure along the permutation perm.

EXAMPLES:

```
sage: F = species.CharacteristicSpecies(3)
sage: a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
sage: p = PermutationGroupElement((1,2))
sage: a.transport(p)
{'a', 'b', 'c'}
```

sage.combinat.species.characteristic_species.**CharacteristicSpecies_class**
alias of CharacteristicSpecies

class sage.combinat.species.characteristic_species.**EmptySetSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: sage.combinat.species.characteristic_species.CharacteristicSpecies

Returns the empty set species.

This species has exactly one structure on the empty set. It is the same (and is implemented) as CharacteristicSpecies(0).

EXAMPLES:

```
sage: X = species.EmptySetSpecies()
sage: X.structures([]).list()
[{}]
sage: X.structures([1,2]).list()
[]
sage: X.generating_series().coefficients(4)
[1, 0, 0, 0]
sage: X.isotype_generating_series().coefficients(4)
[1, 0, 0, 0]
sage: X.cycle_index_series().coefficients(4)
[p[], 0, 0, 0]
```

TESTS:

```
sage: E1 = species.EmptySetSpecies()
sage: E2 = species.EmptySetSpecies()
sage: E1 is E2
True

sage: E = species.EmptySetSpecies()
sage: E._check()
True
sage: E == loads(dumps(E))
True
```

sage.combinat.species.characteristic_species.**EmptySetSpecies_class**
alias of EmptySetSpecies

class sage.combinat.species.characteristic_species.**SingletonSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: sage.combinat.species.characteristic_species.CharacteristicSpecies

Returns the species of singletons.

This species has exactly one structure on a set of size 1. It is the same (and is implemented) as CharacteristicSpecies(1).

EXAMPLES:

```
sage: X = species.SingletonSpecies()
sage: X.structures([1]).list()
[1]
sage: X.structures([1,2]).list()
[]
sage: X.generating_series().coefficients(4)
[0, 1, 0, 0]
sage: X.isotype_generating_series().coefficients(4)
[0, 1, 0, 0]
sage: X.cycle_index_series().coefficients(4)
[0, p[1], 0, 0]
```

TESTS:

```
sage: S1 = species.SingletonSpecies()
sage: S2 = species.SingletonSpecies()
sage: S1 is S2
True

sage: S = species.SingletonSpecies()
sage: S._check()
True
sage: S == loads(dumps(S))
True
```

sage.combinat.species.characteristic_species.**SingletonSpecies_class**
alias of SingletonSpecies

5.1.282 Combinatorial Logarithm

This file provides the cycle index series for the virtual species Ω, the 'combinatorial logarithm', defined to be the compositional inverse of the species E^+ of nonempty sets:

$$\Omega \circ E^+ = E^+ \circ \Omega = X.$$

Warning: This module is now deprecated. Please use sage.combinat.species.generating_series.CycleIndexS instead of CombinatorialLogarithmSeries().

AUTHORS:

- Andrew Gainer-Dewar (2013): initial version

sage.combinat.species.combinatorial_logarithm.**CombinatorialLogarithmSeries**(*R=Rational Field*)

Return the cycle index series of the virtual species Ω, the compositional inverse of the species E^+ of nonempty sets.

The notion of virtual species is treated thoroughly in [BLL]. The specific algorithm used here to compute the cycle index of Ω is found in [Labelle].

EXAMPLES:

The virtual species Ω is 'properly virtual', in the sense that its cycle index has negative coefficients:

```
sage: from sage.combinat.species.combinatorial_logarithm import CombinatorialLogarithmSeries
sage: CombinatorialLogarithmSeries().coefficients(4)
doctest:...: DeprecationWarning: CombinatorialLogarithmSeries is deprecated, use CycleIndexSerie
See http://trac.sagemath.org/14846 for details.
[0, p[1], -1/2*p[1, 1] - 1/2*p[2], 1/3*p[1, 1, 1] - 1/3*p[3]]
```

Its defining property is that $\Omega \circ E^+ = E^+ \circ \Omega = X$ (that is, that composition with E^+ in both directions yields the multiplicative identity X):

```
sage: Eplus = sage.combinat.species.set_species.SetSpecies(min=1).cycle_index_series()
sage: CombinatorialLogarithmSeries().compose(Eplus).coefficients(4)
[0, p[1], 0, 0]
```

5.1.283 Composition species

class sage.combinat.species.composition_species.**CompositionSpecies**(*F*, *G*, *min=None*, *max=None*, *weight=None*)

Bases: sage.combinat.species.species.GenericCombinatorialSpecies, sage.structure.unique_representation.UniqueRepresentation

Returns the composition of two species.

EXAMPLES:

```
sage: E = species.SetSpecies()
sage: C = species.CycleSpecies()
sage: S = E(C)
sage: S.generating_series().coefficients(5)
[1, 1, 1, 1, 1]
sage: E(C) is S
True
```

TESTS:

```
sage: E = species.SetSpecies(); C = species.CycleSpecies()
sage: L = E(C)
sage: c = L.generating_series().coefficients(3)
sage: L._check() #False due to isomorphism types not being implemented
False
sage: L == loads(dumps(L))
True
```

weight_ring()

Returns the weight ring for this species. This is determined by asking Sage's coercion model what the result is when you multiply (and add) elements of the weight rings for each of the operands.

EXAMPLES:

```
sage: E = species.SetSpecies(); C = species.CycleSpecies()
sage: L = E(C)
sage: L.weight_ring()
Rational Field
```

class sage.combinat.species.composition_species.**CompositionSpeciesStructure**(*parent*, *labels*, *pi*, *f*, *gs*)

Bases: sage.combinat.species.structure.GenericSpeciesStructure

TESTS:

```
sage: E = species.SetSpecies(); C = species.CycleSpecies()
sage: L = E(C)
sage: a = L.structures(['a','b','c']).random_element()
sage: a == loads(dumps(a))
True
```

change_labels(*labels*)

Return a relabelled structure.

INPUT:

- `labels`, a list of labels.

OUTPUT:

A structure with the i-th label of self replaced with the i-th label of the list.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: E = species.SetSpecies(); C = species.CycleSpecies()
sage: L = E(C)
sage: S = L.structures(['a','b','c']).list()
sage: a = S[2]; a
F-structure: {{'a', 'c'}, {'b'}}; G-structures: (('a', 'c'), ('b'))
sage: a.change_labels([1,2,3])
F-structure: {{1, 3}, {2}}; G-structures: [(1, 3), (2)]
```

transport(*perm*)

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: E = species.SetSpecies(); C = species.CycleSpecies()
sage: L = E(C)
sage: S = L.structures(['a','b','c']).list()
sage: a = S[2]; a
F-structure: {{'a', 'c'}, {'b'}}; G-structures: (('a', 'c'), ('b'))
sage: a.transport(p)
F-structure: {{'a', 'b'}, {'c'}}; G-structures: (('a', 'c'), ('b'))
```

sage.combinat.species.composition_species.**CompositionSpecies_class**

alias of CompositionSpecies

5.1.284 Cycle Species

class sage.combinat.species.cycle_species.**CycleSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: sage.combinat.species.species.GenericCombinatorialSpecies, sage.structure.unique_representation.UniqueRepresentation

Returns the species of cycles.

EXAMPLES:

```
sage: C = species.CycleSpecies(); C
Cyclic permutation species
sage: C.structures([1,2,3,4]).list()
[(1, 2, 3, 4),
 (1, 2, 4, 3),
 (1, 3, 2, 4),
```

```
 (1, 3, 4, 2),
 (1, 4, 2, 3),
 (1, 4, 3, 2)]
```

TESTS:

We check to verify that the caching of species is actually working.

```
sage: species.CycleSpecies() is species.CycleSpecies()
True

sage: P = species.CycleSpecies()
sage: c = P.generating_series().coefficients(3)
sage: P._check()
True
sage: P == loads(dumps(P))
True
```

class sage.combinat.species.cycle_species.**CycleSpeciesStructure**(*parent*, *labels*, *list*)

Bases: sage.combinat.species.structure.GenericSpeciesStructure

This is a base class from which the classes for the structures inherit.

EXAMPLES:

```
sage: from sage.combinat.species.structure import GenericSpeciesStructure
sage: a = GenericSpeciesStructure(None, [2,3,4], [1,2,3])
sage: a
[2, 3, 4]
sage: a.parent() is None
True
sage: a == loads(dumps(a))
True
```

automorphism_group()

Returns the group of permutations whose action on this structure leave it fixed.

EXAMPLES:

```
sage: P = species.CycleSpecies()
sage: a = P.structures([1, 2, 3, 4]).random_element(); a
(1, 2, 3, 4)
sage: a.automorphism_group()
Permutation Group with generators [(1,2,3,4)]

sage: [a.transport(perm) for perm in a.automorphism_group()]
[(1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4)]
```

canonical_label()

EXAMPLES:

```
sage: P = species.CycleSpecies()
sage: P.structures(["a","b","c"]).random_element().canonical_label()
('a', 'b', 'c')
```

permutation_group_element()

Returns this cycle as a permutation group element.

EXAMPLES:

```
sage: F = species.CycleSpecies()
sage: a = F.structures(["a", "b", "c"]).random_element(); a
('a', 'b', 'c')
sage: a.permutation_group_element()
(1,2,3)
```

transport(*perm*)

Returns the transport of this structure along the permutation perm.

EXAMPLES:

```
sage: F = species.CycleSpecies()
sage: a = F.structures(["a", "b", "c"]).random_element(); a
('a', 'b', 'c')
sage: p = PermutationGroupElement((1,2))
sage: a.transport(p)
('a', 'c', 'b')
```

sage.combinat.species.cycle_species.**CycleSpecies_class**

alias of CycleSpecies

5.1.285 Empty Species

class sage.combinat.species.empty_species.**EmptySpecies**(*min=None*, *max=None*, *weight=None*)

Bases: sage.combinat.species.species.GenericCombinatorialSpecies, sage.structure.unique_representation.UniqueRepresentation

Returns the empty species. This species has no structure at all. It is the zero of the semi-ring of species.

EXAMPLES:

```
sage: X = species.EmptySpecies(); X
Empty species
sage: X.structures([]).list()
[]
sage: X.structures([1]).list()
[]
sage: X.structures([1,2]).list()
[]
sage: X.generating_series().coefficients(4)
[0, 0, 0, 0]
sage: X.isotype_generating_series().coefficients(4)
[0, 0, 0, 0]
sage: X.cycle_index_series().coefficients(4)
[0, 0, 0, 0]
```

The empty species is the zero of the semi-ring of species. The following tests that it is neutral with respect to addition:

```
sage: Empt  = species.EmptySpecies()
sage: S = species.CharacteristicSpecies(2)
sage: X = S + Empt
sage: X == S    # TODO: Not Implemented
True
sage: (X.generating_series().coefficients(4) ==
...    S.generating_series().coefficients(4))
True
sage: (X.isotype_generating_series().coefficients(4) ==
```

```
...     S.isotype_generating_series().coefficients(4))
True
sage: (X.cycle_index_series().coefficients(4) ==
...     S.cycle_index_series().coefficients(4))
True
```

The following tests that it is the zero element with respect to multiplication:

```
sage: Y = Empt*S
sage: Y == Empt    # TODO: Not Implemented
True
sage: Y.generating_series().coefficients(4)
[0, 0, 0, 0]
sage: Y.isotype_generating_series().coefficients(4)
[0, 0, 0, 0]
sage: Y.cycle_index_series().coefficients(4)
[0, 0, 0, 0]
```

TESTS:

```
sage: Empt  = species.EmptySpecies()
sage: Empt2 = species.EmptySpecies()
sage: Empt is Empt2
True
```

sage.combinat.species.empty_species.**EmptySpecies_class**
alias of EmptySpecies

5.1.286 Functorial composition species

class sage.combinat.species.functorial_composition_species.**FunctorialCompositionSpecies**(*F*, *G*, *min=N* *max=N* *weight*

Bases: sage.combinat.species.species.GenericCombinatorialSpecies

Returns the functorial composition of two species.

EXAMPLES:

```
sage: E = species.SetSpecies()
sage: E2 = species.SetSpecies(size=2)
sage: WP = species.SubsetSpecies()
sage: P2 = E2*E
sage: G = WP.functorial_composition(P2)
sage: G.isotype_generating_series().coefficients(5)
[1, 1, 2, 4, 11]

sage: G = species.SimpleGraphSpecies()
sage: c = G.generating_series().coefficients(2)
sage: type(G)
<class 'sage.combinat.species.functorial_composition_species.FunctorialCompositionSpecies'>
sage: G == loads(dumps(G))
True
sage: G._check() #False due to isomorphism types not being implemented
False
```

weight_ring()
Returns the weight ring for this species. This is determined by asking Sage's coercion model what the result is when you multiply (and add) elements of the weight rings for each of the operands.

EXAMPLES:

```
sage: G = species.SimpleGraphSpecies()
sage: G.weight_ring()
Rational Field
```

sage.combinat.species.functorial_composition_species.**FunctorialCompositionSpecies_class**
alias of FunctorialCompositionSpecies

class sage.combinat.species.functorial_composition_species.**FunctorialCompositionStructure**(*parent*, *labels*, *list*)

Bases: sage.combinat.species.structure.GenericSpeciesStructure

This is a base class from which the classes for the structures inherit.

EXAMPLES:

```
sage: from sage.combinat.species.structure import GenericSpeciesStructure
sage: a = GenericSpeciesStructure(None, [2,3,4], [1,2,3])
sage: a
[2, 3, 4]
sage: a.parent() is None
True
sage: a == loads(dumps(a))
True
```

5.1.287 Generating Series

This file makes a number of extensions to lazy power series by endowing them with some semantic content for how they're to be interpreted.

This code is based on the work of Ralf Hemmecke and Martin Rubey's Aldor-Combinat, which can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/aldor/combinat/index.html. In particular, the relevant section for this file can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/AldorCombinat/combinatse10.html. One notable difference is that we use power-sum symmetric functions as the coefficients of our cycle index series.

TESTS:

```
sage: from sage.combinat.species.stream import Stream, _integers_from
sage: from sage.combinat.species.generating_series import CycleIndexSeriesRing
sage: p = SymmetricFunctions(QQ).power()
sage: CIS = CycleIndexSeriesRing(QQ)


sage: geo1 = CIS((p([1])^i  for i in _integers_from(0)))
sage: geo2 = CIS((p([2])^i  for i in _integers_from(0)))
sage: s = geo1 * geo2
sage: s[0]
p[]
sage: s[1]
p[1] + p[2]
sage: s[2]
p[1, 1] + p[2, 1] + p[2, 2]
```

```
sage: s[3]
p[1, 1, 1] + p[2, 1, 1] + p[2, 2, 1] + p[2, 2, 2]
```

Whereas the coefficients of the above test are homogeneous with respect to total degree, the following test groups with respect to weighted degree where each variable x_i has weight i.

```
sage: def g():
....:     for i in _integers_from(0):
....:         yield p([2])^i
....:         yield p(0)
sage: geo1 = CIS((p([1])^i  for i in _integers_from(0)))
sage: geo2 = CIS(g())
sage: s = geo1 * geo2
sage: s[0]
p[]
sage: s[1]
p[1]
sage: s[2]
p[1, 1] + p[2]
sage: s[3]
p[1, 1, 1] + p[2, 1]
sage: s[4]
p[1, 1, 1, 1] + p[2, 1, 1] + p[2, 2]
```

REFERENCES:

class sage.combinat.species.generating_series.**CycleIndexSeries**(*A*, *stream=None*, *order=None*, *aorder=None*, *aorder_changed=True*, *is_initialized=False*, *name=None*)

Bases: sage.combinat.species.series.LazyPowerSeries

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L()
sage: loads(dumps(f))
Uninitialized lazy power series
```

arithmetic_product(*g*, *check_input=True*)

Return the arithmetic product of `self` with `g`.

For species M and N such that $M[\varnothing] = N[\varnothing] = \varnothing$, their arithmetic product is the species $M \boxdot N$ of "M-assemblies of cloned N-structures". This operation is defined and several examples are given in [MM].

The cycle index series for $M \boxdot N$ can be computed in terms of the component series Z_M and Z_N, as implemented in this method.

INPUT:

- `g` – a cycle index series having the same parent as `self`.
- `check_input` – (default: `True`) a Boolean which, when set to `False`, will cause input checks to be skipped.

OUTPUT:

The arithmetic product of `self` with `g`. This is a cycle index series defined in terms of `self` and `g` such that if `self` and `g` are the cycle index series of two species M and N, their arithmetic product is the cycle index series of the species $M \boxdot N$.

EXAMPLES:

For C the species of (oriented) cycles and L_+ the species of nonempty linear orders, $C \boxdot L_+$ corresponds to the species of "regular octopuses"; a $(C \boxdot L_+)$-structure is a cycle of some length, each of whose elements is an ordered list of a length which is consistent for all the lists in the structure.

```
sage: C = species.CycleSpecies().cycle_index_series()
sage: Lplus = species.LinearOrderSpecies(min=1).cycle_index_series()
sage: RegularOctopuses = C.arithmetic_product(Lplus)
sage: RegOctSpeciesSeq = RegularOctopuses.generating_series().counts(8)
sage: RegOctSpeciesSeq
[0, 1, 3, 8, 42, 144, 1440, 5760]
```

It is shown in [MM] that the exponential generating function for regular octopuses satisfies $(C \boxdot L_+)(x) = \sum_{n>1} \sigma(n)(n-1)!\frac{x^n}{n!}$ (where $\sigma(n)$ is the sum of the divisors of n).

```
sage: RegOctDirectSeq = [0] + [sum(divisors(i))*factorial(i-1) for i in range(1,8)]
sage: RegOctDirectSeq == RegOctSpeciesSeq
True
```

AUTHORS:

- Andrew Gainer-Dewar (2013)

REFERENCES:

coefficient_cycle_type(*t*)

Returns the coefficient of a cycle type `t` in `self`.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import CycleIndexSeriesRing
sage: p = SymmetricFunctions(QQ).power()
sage: CIS = CycleIndexSeriesRing(QQ)
sage: f = CIS([0, p([1]), 2*p([1,1]),3*p([2,1])])
sage: f.coefficient_cycle_type([1])
1
sage: f.coefficient_cycle_type([1,1])
2
sage: f.coefficient_cycle_type([2,1])
3
```

compositional_inverse()

Return the compositional inverse of `self` if possible.

(Specifically, if `self` is of the form $0 + p_1 + \ldots$.)

The compositional inverse is the inverse with respect to plethystic substitution. This is the operation on cycle index series which corresponds to substitution, a.k.a. partitional composition, on the level of species. See Section 2.2 of [BLL] for a definition of this operation.

EXAMPLES:

```
sage: Eplus = species.SetSpecies(min=1).cycle_index_series()
sage: Eplus(Eplus.compositional_inverse()).coefficients(8)
[0, p[1], 0, 0, 0, 0, 0, 0]
```

TESTS:

```
sage: Eplus = species.SetSpecies(min=2).cycle_index_series()
sage: Eplus.compositional_inverse()
Traceback (most recent call last):
...
ValueError: not an invertible series
```

ALGORITHM:

Let F be a species satisfying $F = 0 + X + F_2 + F_3 + \dots$ for X the species of singletons. (Equivalently, $|F[\varnothing]| = 0$ and $|F[\{1\}]| = 1$.) Then there exists a (virtual) species G satisfying $F \circ G = G \circ F = X$.

It follows that $(F - X) \circ G = F \circ G - X \circ G = X - G$. Rearranging, we obtain the recursive equation $G = X - (F - X) \circ G$, which can be solved using iterative methods.

Warning: This algorithm is functional but can be very slow. Use with caution!

See also:

The compositional inverse Ω of the species E_+ of nonempty sets can be handled much more efficiently using specialized methods. These are implemented in `CombinatorialLogarithmSeries`.

AUTHORS:

- Andrew Gainer-Dewar

count(*t*)

Return the number of structures corresponding to a certain cycle type `t`.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import CycleIndexSeriesRing
sage: p = SymmetricFunctions(QQ).power()
sage: CIS = CycleIndexSeriesRing(QQ)
sage: f = CIS([0, p([1]), 2*p([1,1]), 3*p([2,1])])
sage: f.count([1])
1
sage: f.count([1,1])
4
sage: f.count([2,1])
6
```

derivative(*order=1*)

Return the species-theoretic nth derivative of `self`, where n is `order`.

For a cycle index series $F(p_1, p_2, p_3, \dots)$, its derivative is the cycle index series $F' = D_{p_1} F$ (that is, the formal derivative of F with respect to the variable p_1).

If F is the cycle index series of a species S then F' is the cycle index series of an associated species S' of S-structures with a "hole".

EXAMPLES:

The species E of sets satisfies the relationship $E' = E$:

```
sage: E = species.SetSpecies().cycle_index_series()
sage: E.coefficients(8) == E.derivative().coefficients(8)
True
```

The species C of cyclic orderings and the species L of linear orderings satisfy the relationship $C' = L$:

```
sage: C = species.CycleSpecies().cycle_index_series()
sage: L = species.LinearOrderSpecies().cycle_index_series()
```

```
sage: L.coefficients(8) == C.derivative().coefficients(8)
True
```

expand_as_sf(*n*, *alphabet='x'*)

Returns the expansion of a cycle index series as a symmetric function in `n` variables.

Specifically, this returns a `LazyPowerSeries` whose ith term is obtained by calling `expand()` on the ith term of `self`.

This relies on the (standard) interpretation of a cycle index series as a symmetric function in the power sum basis.

INPUT:

- `self` – a cycle index series
- `n` – a positive integer
- `alphabet` – a variable for the expansion (default: x)

EXAMPLES:

```
sage: from sage.combinat.species.set_species import SetSpecies
sage: SetSpecies().cycle_index_series().expand_as_sf(2).coefficients(4)
[1, x0 + x1, x0^2 + x0*x1 + x1^2, x0^3 + x0^2*x1 + x0*x1^2 + x1^3]
```

exponential()

Return the species-theoretic exponential of `self`.

For a cycle index Z_F of a species F, its exponential is the cycle index series Z_E $circ Z_F$, where Z_E is the `ExponentialCycleIndexSeries()`.

The exponential $Z_E \circ Z_F$ is then the cycle index series of the species E $circ F$ of "sets of F-structures".

EXAMPLES:

Let BT be the species of binary trees, BF the species of binary forests, and E the species of sets. Then we have $BF = E \circ BT$:

```
sage: BT = species.BinaryTreeSpecies().cycle_index_series()
sage: BF = species.BinaryForestSpecies().cycle_index_series()
sage: BT.exponential().isotype_generating_series().coefficients(8) == BF.isotype_generating_
True
```

functorial_composition(*g*)

Returns the functorial composition of `self` and `g`.

If F and G are species, their functorial composition is the species $F \Box G$ obtained by setting $(F \Box G)[A] = F[G[A]]$. In other words, an $(F \Box G)$-structure on a set A of labels is an F-structure whose labels are the set of all G-structures on A.

It can be shown (as in section 2.2 of [BLL]) that there is a corresponding operation on cycle indices:

$$Z_F \Box Z_G = \sum_{n \geq 0} \frac{1}{n!} \sum_{\sigma \in \mathfrak{S}_n} \operatorname{fix} F[(G[\sigma])_1, (G[\sigma])_2, \dots] \, p_1^{\sigma_1} p_2^{\sigma_2} \cdots.$$

This method implements that operation on cycle index series.

EXAMPLES:

The species G of simple graphs can be expressed in terms of a functorial composition: $G = \mathfrak{p} \Box \mathfrak{p}_2$, where $\mathfrak{p}$ is the `SubsetSpecies`. This is how it is implemented in `SimpleGraphSpecies()`:

```
sage: S = species.SimpleGraphSpecies()
sage: S.cycle_index_series().coefficients(5)
[p[],
 p[1],
 p[1, 1] + p[2],
 4/3*p[1, 1, 1] + 2*p[2, 1] + 2/3*p[3],
 8/3*p[1, 1, 1, 1] + 4*p[2, 1, 1] + 2*p[2, 2] + 4/3*p[3, 1] + p[4]]
```

generating_series()

EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: cis = P.cycle_index_series()
sage: f = cis.generating_series()
sage: f.coefficients(5)
[1, 1, 1, 5/6, 5/8]
```

integral(**args*)

Given a cycle index G, it is not in general possible to recover a single cycle index F such that $F' = G$ (even up to addition of a constant term).

More broadly, it may be the case that there are many non-isomorphic species S such that $S' = T$ for a given species T. For example, the species $3C_3$ of 3-cycles from three distinct classes and the species X^3 of 3-sets are not isomorphic, but $(3C_3)' = (X^3)' = 3X^2$.

EXAMPLES:

```
sage: C3 = species.CycleSpecies(size=3).cycle_index_series()
sage: X = species.SingletonSpecies().cycle_index_series()
sage: (3*C3).derivative().coefficients(8) == (3*X^2).coefficients(8)
True
sage: (X^3).derivative().coefficients(8) == (3*X^2).coefficients(8)
True
```

Warning: This method has no implementation and exists only to prevent you from doing something strange. Calling it raises a `NotImplementedError`!

isotype_generating_series()

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: cis = P.cycle_index_series()
sage: f = cis.isotype_generating_series()
sage: f.coefficients(10)
[1, 1, 2, 3, 5, 7, 11, 15, 22, 30]
```

logarithm()

Return the combinatorial logarithm of `self`.

For a cycle index Z_F of a species F, its logarithm is the cycle index series $Z_\Omega \circ Z_F$, where Z_Ω is the `LogarithmCycleIndexSeries()`.

The logarithm $Z_\Omega \circ Z_F$ is then the cycle index series of the (virtual) species $\Omega \circ F$ of "connected F-structures". In particular, if $F = E^+ \circ G$ for E^+ the species of nonempty sets and G some other species, then $\Omega \circ F = G$.

EXAMPLES:

Let G be the species of nonempty graphs and CG be the species of nonempty connected graphs. Then

$G = E^+ \circ CG$, so $CG = \Omega \circ G$:

```
sage: G = species.SimpleGraphSpecies().cycle_index_series() - 1
sage: from sage.combinat.species.generating_series import LogarithmCycleIndexSeries
sage: CG = LogarithmCycleIndexSeries().compose(G)
sage: CG.isotype_generating_series().coefficients(8)
[0, 1, 1, 2, 6, 21, 112, 853]
```

pointing()

Return the species-theoretic pointing of `self`.

For a cycle index F, its pointing is the cycle index series $F^\bullet = p_1 \cdot F'$.

If F is the cycle index series of a species S then $F^\bullet$ is the cycle index series of an associated species $S^\bullet$ of S-structures with a marked "root".

EXAMPLES:

The species $E^\bullet$ of "pointed sets" satisfies $E^\bullet = X \cdot E$:

```
sage: E = species.SetSpecies().cycle_index_series()
sage: X = species.SingletonSpecies().cycle_index_series()
sage: E.pointing().coefficients(8) == (X*E).coefficients(8)
True
```

stretch(k)

Return the stretch of the cycle index series `self` by a positive integer k.

If

$$f = \sum_{n=0}^{\infty} f_n(p_1, p_2, p_3, \ldots),$$

then the stretch g of f by k is

$$g = \sum_{n=0}^{\infty} f_n(p_k, p_{2k}, p_{3k}, \ldots).$$

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import CycleIndexSeriesRing
sage: p = SymmetricFunctions(QQ).power()
sage: CIS = CycleIndexSeriesRing(QQ)
sage: f = CIS([p([]), p([1]), p([2]), p.zero()])
sage: f.stretch(3).coefficients(10)
[p[], 0, 0, p[3], 0, 0, p[6], 0, 0, 0]
```

weighted_composition(*y_species*)

Returns the composition of this cycle index series with the cycle index series of y_species where y_species is a weighted species.

Note that this is basically the same algorithm as composition except we can not use the optimization that the powering of cycle index series commutes with 'stretching'.

EXAMPLES:

```
sage: E = species.SetSpecies(); C = species.CycleSpecies()
sage: E_cis = E.cycle_index_series()
sage: E_cis.weighted_composition(C).coefficients(4)
[p[], p[1], p[1, 1] + p[2], p[1, 1, 1] + p[2, 1] + p[3]]
sage: E(C).cycle_index_series().coefficients(4)
[p[], p[1], p[1, 1] + p[2], p[1, 1, 1] + p[2, 1] + p[3]]
```

sage.combinat.species.generating_series.**CycleIndexSeriesRing**(*R*)

Return the ring of cycle index series over R.

This is the ring of formal power series $\Lambda[x]$, where Λ is the ring of symmetric functions over R in the p-basis. Its purpose is to house the cycle index series of species (in a somewhat nonstandard notation tailored to Sage): If F is a species, then the *cycle index series* of F is defined to be the formal power series

$$\sum_{n\geq 0}\frac{1}{n!}\Big(\sum_{\sigma\in S_n}\operatorname{fix} F[\sigma]\prod_{z \text{ is a cycle of } \sigma} p_{\text{length of } z}\Big)x^n \in \Lambda_{\mathbf{Q}}[x],$$

where $\operatorname{fix} F[\sigma]$ denotes the number of fixed points of the permutation $F[\sigma]$ of $F[n]$. We notice that this power series is "equigraded" (meaning that its x^n-coefficient is homogeneous of degree n). A more standard convention in combinatorics would be to use x_i instead of p_i, and drop the x (that is, evaluate the above power series at $x = 1$); but this would be more difficult to implement in Sage, as it would be an element of a power series ring in infinitely many variables.

Note that is is just a LazyPowerSeriesRing (whose base ring is Λ) whose elements have some extra methods.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import CycleIndexSeriesRing
sage: R = CycleIndexSeriesRing(QQ); R
Cycle Index Series Ring over Symmetric Functions over Rational Field in the powersum basis
sage: R([1]).coefficients(4) # This is not combinatorially
....:                        # meaningful.
[1, 1, 1, 1]
```

TESTS:

We test to make sure that caching works.

```
sage: R is CycleIndexSeriesRing(QQ)
True
```

class sage.combinat.species.generating_series.**CycleIndexSeriesRing_class**(*R*)

Bases: sage.combinat.species.series.LazyPowerSeriesRing

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import CycleIndexSeriesRing
sage: R = CycleIndexSeriesRing(QQ); R
Cycle Index Series Ring over Symmetric Functions over Rational Field in the powersum basis
sage: R == loads(dumps(R))
True
```

sage.combinat.species.generating_series.**ExponentialCycleIndexSeries**(*R=Rational Field*)

Return the cycle index series of the species E of sets.

This cycle index satisfies

$$Z_E = \sum_{n\geq 0}\sum_{\lambda\vdash n}\frac{p_\lambda}{z_\lambda}.$$

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import ExponentialCycleIndexSeries
sage: ExponentialCycleIndexSeries().coefficients(5)
[p[], p[1], 1/2*p[1, 1] + 1/2*p[2], 1/6*p[1, 1, 1] + 1/2*p[2, 1] + 1/3*p[3], 1/24*p[1, 1, 1, 1]
```

class sage.combinat.species.generating_series.**ExponentialGeneratingSeries**(*A*, *stream=None*, *order=None*, *aorder=None*, *aorder_changed=True*, *is_initialized=False*, *name=None*)

Bases: sage.combinat.species.series.LazyPowerSeries

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L()
sage: loads(dumps(f))
Uninitialized lazy power series
```

count(*n*)

Return the number of structures of size n.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import ExponentialGeneratingSeriesRing
sage: R = ExponentialGeneratingSeriesRing(QQ)
sage: f = R([1])
sage: [f.count(i) for i in range(7)]
[1, 1, 2, 6, 24, 120, 720]
```

counts(*n*)

Return the number of structures on a set for size i for each i in range(n).

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import ExponentialGeneratingSeriesRing
sage: R = ExponentialGeneratingSeriesRing(QQ)
sage: f = R(range(20))
sage: f.counts(5)
[0, 1, 4, 18, 96]
```

functorial_composition(*y*)

Return the exponential generating series which is the functorial composition of self with y.

If $f = \sum_{n=0}^{\infty} f_n \frac{x^n}{n!}$ and $g = \sum_{n=0}^{\infty} g_n \frac{x^n}{n!}$, then functorial composition $f \Box g$ is defined as

$$f \Box g = \sum_{n=0}^{\infty} f_{g_n} \frac{x^n}{n!}$$

REFERENCES:

- Section 2.2 of [BLL].

EXAMPLES:

```
sage: G = species.SimpleGraphSpecies()
sage: g = G.generating_series()
sage: g.coefficients(10)
[1, 1, 1, 4/3, 8/3, 128/15, 2048/45, 131072/315, 2097152/315, 536870912/2835]
```

sage.combinat.species.generating_series.**ExponentialGeneratingSeriesRing**(*R*)

Return the ring of exponential generating series over R.

Note that is is just a LazyPowerSeriesRing whose elements have some extra methods.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import ExponentialGeneratingSeriesRing
sage: R = ExponentialGeneratingSeriesRing(QQ); R
Lazy Power Series Ring over Rational Field
sage: R([1]).coefficients(4)
[1, 1, 1, 1]
sage: R([1]).counts(4)
[1, 1, 2, 6]
```

TESTS:

We test to make sure that caching works.

```
sage: R is ExponentialGeneratingSeriesRing(QQ)
True
```

class sage.combinat.species.generating_series.**ExponentialGeneratingSeriesRing_class**(*R*)

Bases: sage.combinat.species.series.LazyPowerSeriesRing

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import ExponentialGeneratingSeriesRing
sage: R = ExponentialGeneratingSeriesRing(QQ)
sage: R == loads(dumps(R))
True
```

sage.combinat.species.generating_series.**LogarithmCycleIndexSeries**(*R=Rational Field*)

Return the cycle index series of the virtual species Ω, the compositional inverse of the species E^+ of nonempty sets.

The notion of virtual species is treated thoroughly in [BLL]. The specific algorithm used here to compute the cycle index of Ω is found in [Labelle].

EXAMPLES:

The virtual species Ω is 'properly virtual', in the sense that its cycle index has negative coefficients:

```
sage: from sage.combinat.species.generating_series import LogarithmCycleIndexSeries
sage: LogarithmCycleIndexSeries().coefficients(4)
[0, p[1], -1/2*p[1, 1] - 1/2*p[2], 1/3*p[1, 1, 1] - 1/3*p[3]]
```

Its defining property is that $\Omega \circ E^+ = E^+ \circ \Omega = X$ (that is, that composition with E^+ in both directions yields the multiplicative identity X):

```
sage: Eplus = sage.combinat.species.set_species.SetSpecies(min=1).cycle_index_series()
sage: LogarithmCycleIndexSeries().compose(Eplus).coefficients(4)
[0, p[1], 0, 0]
```

REFERENCES:

class sage.combinat.species.generating_series.**OrdinaryGeneratingSeries**(*A*, *stream=None*, *order=None*, *aorder=None*, *aorder_changed=True*, *is_initialized=False*, *name=None*)

Bases: sage.combinat.species.series.LazyPowerSeries

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L()
sage: loads(dumps(f))
Uninitialized lazy power series
```

count(*n*)

Return the number of structures on a set of size `n`.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import OrdinaryGeneratingSeriesRing
sage: R = OrdinaryGeneratingSeriesRing(QQ)
sage: f = R(range(20))
sage: f.count(10)
10
```

counts(*n*)

Return the number of structures on a set for size `i` for each `i` in `range(n)`.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import OrdinaryGeneratingSeriesRing
sage: R = OrdinaryGeneratingSeriesRing(QQ)
sage: f = R(range(20))
sage: f.counts(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

sage.combinat.species.generating_series.**OrdinaryGeneratingSeriesRing**(*R*)

Return the ring of ordinary generating series over `R`.

Note that is is just a `LazyPowerSeriesRing` whose elements have some extra methods.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import OrdinaryGeneratingSeriesRing
sage: R = OrdinaryGeneratingSeriesRing(QQ); R
Lazy Power Series Ring over Rational Field
sage: R([1]).coefficients(4)
[1, 1, 1, 1]
sage: R([1]).counts(4)
[1, 1, 1, 1]
```

TESTS:

We test to make sure that caching works.

```
sage: R is OrdinaryGeneratingSeriesRing(QQ)
True
```

class sage.combinat.species.generating_series.**OrdinaryGeneratingSeriesRing_class**(*R*)

Bases: `sage.combinat.species.series.LazyPowerSeriesRing`

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import OrdinaryGeneratingSeriesRing
sage: R = OrdinaryGeneratingSeriesRing(QQ)
sage: R == loads(dumps(R))
True
```

sage.combinat.species.generating_series.**factorial_gen**()

A generator for the factorials starting at 0.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import factorial_gen
sage: g = factorial_gen()
sage: [next(g) for i in range(5)]
[1, 1, 2, 6, 24]
```

5.1.288 Examples of Combinatorial Species

sage.combinat.species.library.**BinaryForestSpecies**()

Returns the species of binary forests. Binary forests are defined as sets of binary trees.

EXAMPLES:

```
sage: F = species.BinaryForestSpecies()
sage: F.generating_series().counts(10)
[1, 1, 3, 19, 193, 2721, 49171, 1084483, 28245729, 848456353]
sage: F.isotype_generating_series().counts(10)
[1, 1, 2, 4, 10, 26, 77, 235, 758, 2504]
sage: F.cycle_index_series().coefficients(7)
[p[],
 p[1],
 3/2*p[1, 1] + 1/2*p[2],
 19/6*p[1, 1, 1] + 1/2*p[2, 1] + 1/3*p[3],
 193/24*p[1, 1, 1, 1] + 3/4*p[2, 1, 1] + 5/8*p[2, 2] + 1/3*p[3, 1] + 1/4*p[4],
 907/40*p[1, 1, 1, 1, 1] + 19/12*p[2, 1, 1, 1] + 5/8*p[2, 2, 1] + 1/2*p[3, 1, 1] + 1/6*p[3, 2] +
 49171/720*p[1, 1, 1, 1, 1, 1] + 193/48*p[2, 1, 1, 1, 1] + 15/16*p[2, 2, 1, 1] + 61/48*p[2, 2, 2
```

TESTS:

```
sage: seq = F.isotype_generating_series().counts(10)[1:]
sage: oeis(seq)[0]                                  # optional -- internet
A052854: Number of forests of ordered trees on n total nodes.
```

sage.combinat.species.library.**BinaryTreeSpecies**()

Returns the species of binary trees on n leaves. The species of binary trees B is defined by B = X + B*B where X is the singleton species.

EXAMPLES:

```
sage: B = species.BinaryTreeSpecies()
sage: B.generating_series().counts(10)
[0, 1, 2, 12, 120, 1680, 30240, 665280, 17297280, 518918400]
sage: B.isotype_generating_series().counts(10)
[0, 1, 1, 2, 5, 14, 42, 132, 429, 1430]
sage: B._check()
True
```

```
sage: B = species.BinaryTreeSpecies()
sage: a = B.structures([1,2,3,4,5]).random_element(); a
2*((5*3)*(4*1))
sage: a.automorphism_group()
Permutation Group with generators [()]
```

TESTS:

```
sage: seq = B.isotype_generating_series().counts(10)[1:]
sage: oeis(seq)[0]                                  # optional -- internet
A000108: Catalan numbers: C(n) = binomial(2n,n)/(n+1) = (2n)!/(n!(n+1)!). Also called Segner num
```

sage.combinat.species.library.**SimpleGraphSpecies**()

Returns the species of simple graphs.

EXAMPLES:

```
sage: S = species.SimpleGraphSpecies()
sage: S.generating_series().counts(10)
[1, 1, 2, 8, 64, 1024, 32768, 2097152, 268435456, 68719476736]
sage: S.cycle_index_series().coefficients(5)
[p[],
 p[1],
 p[1, 1] + p[2],
 4/3*p[1, 1, 1] + 2*p[2, 1] + 2/3*p[3],
 8/3*p[1, 1, 1, 1] + 4*p[2, 1, 1] + 2*p[2, 2] + 4/3*p[3, 1] + p[4]]
sage: S.isotype_generating_series().coefficients(6)
[1, 1, 2, 4, 11, 34]
```

TESTS:

```
sage: seq = S.isotype_generating_series().counts(6)[1:]
sage: oeis(seq)[0]                                  # optional -- internet
A000088: Number of graphs on n unlabeled nodes.

sage: seq = S.generating_series().counts(10)[1:]
sage: oeis(seq)[0]                                  # optional -- internet
A006125: a(n) = 2^(n(n-1)/2).
```

5.1.289 Linear-order Species

class sage.combinat.species.linear_order_species.**LinearOrderSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: sage.combinat.species.species.GenericCombinatorialSpecies, sage.structure.unique_representation.UniqueRepresentation

Returns the species of linear orders.

EXAMPLES:

```
sage: L = species.LinearOrderSpecies()
sage: L.generating_series().coefficients(5)
[1, 1, 1, 1, 1]

sage: L = species.LinearOrderSpecies()
sage: L._check()
True
sage: L == loads(dumps(L))
True
```

class sage.combinat.species.linear_order_species.**LinearOrderSpeciesStructure**(*parent*, *labels*, *list*)

Bases: sage.combinat.species.structure.GenericSpeciesStructure

This is a base class from which the classes for the structures inherit.

EXAMPLES:

```
sage: from sage.combinat.species.structure import GenericSpeciesStructure
sage: a = GenericSpeciesStructure(None, [2,3,4], [1,2,3])
sage: a
[2, 3, 4]
sage: a.parent() is None
True
sage: a == loads(dumps(a))
True
```

automorphism_group()
Returns the group of permutations whose action on this structure leave it fixed. For the species of linear orders, there is no non-trivial automorphism.

EXAMPLES:

```
sage: F = species.LinearOrderSpecies()
sage: a = F.structures(["a", "b", "c"]).random_element(); a
['a', 'b', 'c']
sage: a.automorphism_group()
Symmetric group of order 1! as a permutation group
```

canonical_label()
EXAMPLES:

```
sage: P = species.LinearOrderSpecies()
sage: s = P.structures(["a", "b", "c"]).random_element()
sage: s.canonical_label()
['a', 'b', 'c']
```

transport(*perm*)
Returns the transport of this structure along the permutation perm.

EXAMPLES:

```
sage: F = species.LinearOrderSpecies()
sage: a = F.structures(["a", "b", "c"]).random_element(); a
['a', 'b', 'c']
sage: p = PermutationGroupElement((1,2))
sage: a.transport(p)
['b', 'a', 'c']
```

sage.combinat.species.linear_order_species.**LinearOrderSpecies_class**
alias of LinearOrderSpecies

5.1.290 Miscellaneous Functions

sage.combinat.species.misc.**accept_size**(*f*)
The purpose of this decorator is to change calls like species.SetSpecies(size=1) to species.SetSpecies(min=1, max=2). This is to make caching species easier and to restrict the number of parameters that the lower level code needs to know about.

EXAMPLES:

```
sage: from sage.combinat.species.misc import accept_size
sage: def f(*args, **kwds):
...       print args, list(sorted(kwds.items()))
sage: f = accept_size(f)
sage: f(min=1)
() [('min', 1)]
```

```
sage: f(size=2)
() [('max', 3), ('min', 2)]
```

sage.combinat.species.misc.**change_support**(*perm*, *support*, *change_perm=None*)

Changes the support of a permutation defined on [1, ..., n] to support.

EXAMPLES:

```
sage: from sage.combinat.species.misc import change_support
sage: p = PermutationGroupElement((1,2,3)); p
(1,2,3)
sage: change_support(p, [3,4,5])
(3,4,5)
```

5.1.291 Partition Species

class sage.combinat.species.partition_species.**PartitionSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: sage.combinat.species.species.GenericCombinatorialSpecies

Returns the species of partitions.

EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: P.generating_series().coefficients(5)
[1, 1, 1, 5/6, 5/8]
sage: P.isotype_generating_series().coefficients(5)
[1, 1, 2, 3, 5]

sage: P = species.PartitionSpecies()
sage: P._check()
True
sage: P == loads(dumps(P))
True
```

class sage.combinat.species.partition_species.**PartitionSpeciesStructure**(*parent*, *labels*, *list*)

Bases: sage.combinat.species.structure.GenericSpeciesStructure

EXAMPLES:

```
sage: from sage.combinat.species.partition_species import PartitionSpeciesStructure
sage: P = species.PartitionSpecies()
sage: s = PartitionSpeciesStructure(P, ['a','b','c'], [[1,2],[3]]); s
{{'a', 'b'}, {'c'}}
sage: s == loads(dumps(s))
True
```

automorphism_group()

Returns the group of permutations whose action on this set partition leave it fixed.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: from sage.combinat.species.partition_species import PartitionSpeciesStructure
sage: a = PartitionSpeciesStructure(None, [2,3,4], [[1,2],[3]]); a
```

```
{{2, 3}, {4}}
sage: a.automorphism_group()
Permutation Group with generators [(1,2)]
```

canonical_label()

EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.canonical_label() for s in S]
[{{'a', 'b', 'c'}},
 {{'a', 'b'}, {'c'}},
 {{'a', 'b'}, {'c'}},
 {{'a', 'b'}, {'c'}},
 {{'a'}, {'b'}, {'c'}}]
```

change_labels(*labels*)

Return a relabelled structure.

INPUT:

- `labels`, a list of labels.

OUTPUT:

A structure with the i-th label of self replaced with the i-th label of the list.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: from sage.combinat.species.partition_species import PartitionSpeciesStructure
sage: a = PartitionSpeciesStructure(None, [2,3,4], [[1,2],[3]]); a
{{2, 3}, {4}}
sage: a.change_labels([1,2,3])
{{1, 2}, {3}}
```

transport(*perm*)

Returns the transport of this set partition along the permutation perm. For set partitions, this is the direct product of the automorphism groups for each of the blocks.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: from sage.combinat.species.partition_species import PartitionSpeciesStructure
sage: a = PartitionSpeciesStructure(None, [2,3,4], [[1,2],[3]]); a
{{2, 3}, {4}}
sage: a.transport(p)
{{2, 4}, {3}}
```

sage.combinat.species.partition_species.**PartitionSpecies_class**

alias of PartitionSpecies

5.1.292 Permutation species

class sage.combinat.species.permutation_species.**PermutationSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: sage.combinat.species.species.GenericCombinatorialSpecies, sage.structure.unique_representation.UniqueRepresentation

Returns the species of permutations.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: P.generating_series().coefficients(5)
[1, 1, 1, 1, 1]
sage: P.isotype_generating_series().coefficients(5)
[1, 1, 2, 3, 5]

sage: P = species.PermutationSpecies()
sage: c = P.generating_series().coefficients(3)
sage: P._check()
True
sage: P == loads(dumps(P))
True
```

class sage.combinat.species.permutation_species.**PermutationSpeciesStructure**(*parent*, *labels*, *list*)

Bases: sage.combinat.species.structure.GenericSpeciesStructure

This is a base class from which the classes for the structures inherit.

EXAMPLES:

```
sage: from sage.combinat.species.structure import GenericSpeciesStructure
sage: a = GenericSpeciesStructure(None, [2,3,4], [1,2,3])
sage: a
[2, 3, 4]
sage: a.parent() is None
True
sage: a == loads(dumps(a))
True
```

automorphism_group()

Returns the group of permutations whose action on this structure leave it fixed.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3,4))
sage: P = species.PermutationSpecies()
sage: a = P.structures(["a", "b", "c", "d"]).random_element(); a
['a', 'c', 'b', 'd']
sage: a.automorphism_group()
Permutation Group with generators [(2,3), (1,4)]

sage: [a.transport(perm) for perm in a.automorphism_group()]
[['a', 'c', 'b', 'd'],
 ['a', 'c', 'b', 'd'],
 ['a', 'c', 'b', 'd'],
 ['a', 'c', 'b', 'd']]
```

canonical_label()

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.canonical_label() for s in S]
[['a', 'b', 'c'],
 ['b', 'a', 'c'],
```

```
['b', 'a', 'c'],
['b', 'c', 'a'],
['b', 'c', 'a'],
['b', 'a', 'c']]
```

permutation_group_element()
Returns self as a permutation group element.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3,4))
sage: P = species.PermutationSpecies()
sage: a = P.structures(["a", "b", "c", "d"]).random_element(); a
['a', 'c', 'b', 'd']
sage: a.permutation_group_element()
(2,3)
```

transport(*perm*)
Returns the transport of this structure along the permutation perm.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3,4))
sage: P = species.PermutationSpecies()
sage: a = P.structures(["a", "b", "c", "d"]).random_element(); a
['a', 'c', 'b', 'd']
sage: a.transport(p)
['a', 'd', 'c', 'b']
```

sage.combinat.species.permutation_species.**PermutationSpecies_class**
alias of PermutationSpecies

5.1.293 Sum species

class sage.combinat.species.product_species.**ProductSpecies**(*F*, *G*, *min=None*, *max=None*, *weight=None*)

Bases: sage.combinat.species.species.GenericCombinatorialSpecies, sage.structure.unique_representation.UniqueRepresentation

EXAMPLES:

```
sage: X = species.SingletonSpecies()
sage: A = X*X
sage: A.generating_series().coefficients(4)
[0, 0, 1, 0]

sage: P = species.PermutationSpecies()
sage: F = P * P; F
Product of (Permutation species) and (Permutation species)
sage: F == loads(dumps(F))
True
sage: F._check()
True
```

TESTS:

```
sage: X = species.SingletonSpecies()
sage: X*X is X*X
True
```

left_factor()

Returns the left factor of this product.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: X = species.SingletonSpecies()
sage: F = P*X
sage: F.left_factor()
Permutation species
```

right_factor()

Returns the right factor of this product.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: X = species.SingletonSpecies()
sage: F = P*X
sage: F.right_factor()
Singleton species
```

weight_ring()

Returns the weight ring for this species. This is determined by asking Sage's coercion model what the result is when you multiply (and add) elements of the weight rings for each of the operands.

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: C = S*S
sage: C.weight_ring()
Rational Field

sage: S = species.SetSpecies(weight=QQ['t'].gen())
sage: C = S*S
sage: C.weight_ring()
Univariate Polynomial Ring in t over Rational Field

sage: S = species.SetSpecies()
sage: C = (S*S).weighted(QQ['t'].gen())
sage: C.weight_ring()
Univariate Polynomial Ring in t over Rational Field
```

class sage.combinat.species.product_species.**ProductSpeciesStructure**(*parent*, *labels*, *subset*, *left*, *right*)

Bases: sage.combinat.species.structure.GenericSpeciesStructure

TESTS:

```
sage: S = species.SetSpecies()
sage: F = S * S
sage: a = F.structures(['a','b','c']).random_element()
sage: a == loads(dumps(a))
True
```

automorphism_group()

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: S = species.SetSpecies()
sage: F = S * S
sage: a = F.structures([1,2,3,4]).random_element(); a
{1}*{2, 3, 4}
sage: a.automorphism_group()
Permutation Group with generators [(2,3), (2,3,4)]

sage: [a.transport(g) for g in a.automorphism_group()]
[{1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4}]

sage: a = F.structures([1,2,3,4]).random_element(); a
{2, 3}*{1, 4}
sage: [a.transport(g) for g in a.automorphism_group()]
[{2, 3}*{1, 4}, {2, 3}*{1, 4}, {2, 3}*{1, 4}, {2, 3}*{1, 4}]
```

canonical_label()

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: F = S * S
sage: S = F.structures(['a','b','c']).list(); S
[{}*{'a', 'b', 'c'},
 {'a'}*{'b', 'c'},
 {'b'}*{'a', 'c'},
 {'c'}*{'a', 'b'},
 {'a', 'b'}*{'c'},
 {'a', 'c'}*{'b'},
 {'b', 'c'}*{'a'},
 {'a', 'b', 'c'}*{}]

sage: F.isotypes(['a','b','c']).cardinality()
4
sage: [s.canonical_label() for s in S]
[{}*{'a', 'b', 'c'},
 {'a'}*{'b', 'c'},
 {'a'}*{'b', 'c'},
 {'a'}*{'b', 'c'},
 {'a', 'b'}*{'c'},
 {'a', 'b'}*{'c'},
 {'a', 'b'}*{'c'},
 {'a', 'b', 'c'}*{}]
```

change_labels(*labels*)

Return a relabelled structure.

INPUT:

- `labels`, a list of labels.

OUTPUT:

A structure with the i-th label of self replaced with the i-th label of the list.

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: F = S * S
sage: a = F.structures(['a','b','c']).random_element(); a
{}*{'a', 'b', 'c'}
sage: a.change_labels([1,2,3])
{}*{1, 2, 3}
```

transport(*perm*)

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: S = species.SetSpecies()
sage: F = S * S
sage: a = F.structures(['a','b','c'])[4]; a
{'a', 'b'}*{'c'}
sage: a.transport(p)
{'a', 'c'}*{'b'}
```

sage.combinat.species.product_species.**ProductSpecies_class**

alias of ProductSpecies

5.1.294 Recursive Species

class sage.combinat.species.recursive_species.**CombinatorialSpecies**

Bases: sage.combinat.species.species.GenericCombinatorialSpecies

EXAMPLES:

```
sage: F = CombinatorialSpecies()
sage: loads(dumps(F))
Combinatorial species
```

```
sage: X = species.SingletonSpecies()
sage: E = species.EmptySetSpecies()
sage: L = CombinatorialSpecies()
sage: L.define(E+X*L)
sage: L.generating_series().coefficients(4)
[1, 1, 1, 1]
sage: LL = loads(dumps(L))
sage: LL.generating_series().coefficients(4)
[1, 1, 1, 1]
```

define(*x*)

Defines self to be equal to the combinatorial species x. This is used to define combinatorial species recursively. All of the real work is done by calling the .set() method for each of the series associated to self.

EXAMPLES: The species of linear orders L can be recursively defined by $L = 1 + X * L$ where 1 represents the empty set species and X represents the singleton species.

```
sage: X = species.SingletonSpecies()
sage: E = species.EmptySetSpecies()
sage: L = CombinatorialSpecies()
sage: L.define(E+X*L)
sage: L.generating_series().coefficients(4)
[1, 1, 1, 1]
sage: L.structures([1,2,3]).cardinality()
```

```
6
sage: L.structures([1,2,3]).list()
[1*(2*(3*{})),
 1*(3*(2*{})),
 2*(1*(3*{})),
 2*(3*(1*{})),
 3*(1*(2*{})),
 3*(2*(1*{}))]

sage: L = species.LinearOrderSpecies()
sage: L.generating_series().coefficients(4)
[1, 1, 1, 1]
sage: L.structures([1,2,3]).cardinality()
6
sage: L.structures([1,2,3]).list()
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

TESTS:

```
sage: A = CombinatorialSpecies()
sage: A.define(E+X*A*A)
sage: A.generating_series().coefficients(6)
[1, 1, 2, 5, 14, 42]
sage: A.generating_series().counts(6)
[1, 1, 4, 30, 336, 5040]
sage: len(A.structures([1,2,3,4]).list())
336
sage: A.isotype_generating_series().coefficients(6)
[1, 1, 2, 5, 14, 42]
sage: len(A.isotypes([1,2,3,4]).list())
14

sage: A = CombinatorialSpecies()
sage: A.define(X+A*A)
sage: A.generating_series().coefficients(6)
[0, 1, 1, 2, 5, 14]
sage: A.generating_series().counts(6)
[0, 1, 2, 12, 120, 1680]
sage: len(A.structures([1,2,3]).list())
12
sage: A.isotype_generating_series().coefficients(6)
[0, 1, 1, 2, 5, 14]
sage: len(A.isotypes([1,2,3,4]).list())
5

sage: X2 = X*X
sage: X5 = X2*X2*X
sage: A = CombinatorialSpecies()
sage: B = CombinatorialSpecies()
sage: C = CombinatorialSpecies()
sage: A.define(X5+B*B)
sage: B.define(X5+C*C)
sage: C.define(X2+C*C+A*A)
sage: A.generating_series().coefficients(Integer(10))
[0, 0, 0, 0, 0, 1, 0, 0, 1, 2]
sage: A.generating_series().coefficients(15)
[0, 0, 0, 0, 0, 1, 0, 0, 1, 2, 5, 4, 14, 10, 48]
sage: B.generating_series().coefficients(15)
[0, 0, 0, 0, 1, 1, 2, 0, 5, 0, 14, 0, 44, 0, 138]
```

```
sage: C.generating_series().coefficients(15)
[0, 0, 1, 0, 1, 0, 2, 0, 5, 0, 15, 0, 44, 2, 142]

sage: F = CombinatorialSpecies()
sage: F.define(E+X+(X*F+X*X*F))
sage: F.generating_series().counts(10)
[1, 2, 6, 30, 192, 1560, 15120, 171360, 2217600, 32296320]
sage: F.generating_series().coefficients(10)
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
sage: F.isotype_generating_series().coefficients(10)
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

weight_ring()

EXAMPLES:

```
sage: F = species.CombinatorialSpecies()
sage: F.weight_ring()
Rational Field

sage: X = species.SingletonSpecies()
sage: E = species.EmptySetSpecies()
sage: L = CombinatorialSpecies()
sage: L.define(E+X*L)
sage: L.weight_ring()
Rational Field
```

class sage.combinat.species.recursive_species.**CombinatorialSpeciesStructure**(*parent*, *s*, ***options*)

Bases: sage.combinat.species.structure.SpeciesStructureWrapper

This is a class for the structures of species such as the sum species that do not provide "additional" structure. For example, if you have the sum C of species A and B, then a structure of C will either be either something from A or B. Instead of just returning one of these directly, a "wrapper" is put around them so that they have their parent is C rather than A or B:

```
sage: X = species.SingletonSpecies()
sage: X2 = X+X
sage: s = X2.structures([1]).random_element(); s
1
sage: s.parent()
Sum of (Singleton species) and (Singleton species)
sage: from sage.combinat.species.structure import SpeciesStructureWrapper
sage: issubclass(type(s), SpeciesStructureWrapper)
True
```

EXAMPLES:

```
sage: E = species.SetSpecies(); B = E+E
sage: s = B.structures([1,2,3]).random_element()
sage: s.parent()
Sum of (Set species) and (Set species)
sage: s == loads(dumps(s))
True
```

5.1.295 Lazy Power Series

This file provides an implementation of lazy univariate power series, which uses the stream class for its internal data structure. The lazy power series keep track of their approximate order as much as possible without forcing the computation of any additional coefficients. This is required for recursively defined power series.

This code is based on the work of Ralf Hemmecke and Martin Rubey's Aldor-Combinat, which can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/aldor/combinat/index.html. In particular, the relevant section for this file can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/AldorCombinat/combinatse9.html.

class sage.combinat.species.series.**LazyPowerSeries**(*A*, *stream=None*, *order=None*, *aorder=None*, *aorder_changed=True*, *is_initialized=False*, *name=None*)

Bases: sage.structure.element.AlgebraElement

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L()
sage: loads(dumps(f))
Uninitialized lazy power series
```

add(*y*)

EXAMPLES: Test Plus 1

```
sage: from sage.combinat.species.series import *
sage: from sage.combinat.species.stream import Stream
sage: L = LazyPowerSeriesRing(QQ)
sage: gs0 = L([0])
sage: gs1 = L([1])
sage: sum1 = gs0 + gs1
sage: sum2 = gs1 + gs1
sage: sum3 = gs1 + gs0
sage: [gs0.coefficient(i) for i in range(11)]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
sage: [gs1.coefficient(i) for i in range(11)]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
sage: [sum1.coefficient(i) for i in range(11)]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
sage: [sum2.coefficient(i) for i in range(11)]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
sage: [sum3.coefficient(i) for i in range(11)]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Test Plus 2

```
sage: gs1 = L([1,2,4,8,0])
sage: gs2 = L([-1, 0,-1,-9,22,0])
sage: sum = gs1 + gs2
sage: sum2 = gs2 + gs1
sage: [ sum.coefficient(i) for i in range(5) ]
[0,  2, 3, -1, 22]
sage: [ sum.coefficient(i) for i in range(5, 11) ]
[0, 0, 0, 0, 0, 0]
sage: [ sum2.coefficient(i) for i in range(5) ]
[0,  2, 3, -1, 22]
sage: [ sum2.coefficient(i) for i in range(5, 11) ]
[0, 0, 0, 0, 0, 0]
```

coefficient(*n*)

Returns the coefficient of xn in self.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L(ZZ)
sage: [f.coefficient(i) for i in range(5)]
[0, 1, -1, 2, -2]
```

coefficients(*n*)

Returns the first n coefficients of self.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L([1,2,3,0])
sage: f.coefficients(5)
[1, 2, 3, 0, 0]
```

compose(*y*)

Returns the composition of this power series and the power series y.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: s = L([1])
sage: t = L([0,0,1])
sage: u = s(t)
sage: u.coefficients(11)
[1, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Test Compose 2

```
sage: s = L([1])
sage: t = L([0,0,1,0])
sage: u = s(t)
sage: u.aorder
0
sage: u.order
Unknown series order
sage: u.coefficients(10)
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
sage: u.aorder
0
sage: u.order
0
```

Test Compose 3 s = 1/(1-x), t = x/(1-x) s(t) = (1-x)/(1-2x)

```
sage: s = L([1])
sage: t = L([0,1])
sage: u = s(t)
sage: u.coefficients(14)
[1, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
```

composition(*y*)

Returns the composition of this power series and the power series y.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: s = L([1])
sage: t = L([0,0,1])
sage: u = s(t)
sage: u.coefficients(11)
[1, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Test Compose 2

```
sage: s = L([1])
sage: t = L([0,0,1,0])
sage: u = s(t)
sage: u.aorder
0
sage: u.order
Unknown series order
sage: u.coefficients(10)
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
sage: u.aorder
0
sage: u.order
0
```

Test Compose 3 s = 1/(1-x), t = x/(1-x) s(t) = (1-x)/(1-2x)

```
sage: s = L([1])
sage: t = L([0,1])
sage: u = s(t)
sage: u.coefficients(14)
[1, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
```

compute_aorder(**args*, ***kwargs*)

The default compute_aorder does nothing.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L(1)
sage: a.compute_aorder() is None
True
```

compute_coefficients(*i*)

Computes all the coefficients of self up to i.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L([1,2,3])
sage: a.compute_coefficients(5)
sage: a
1 + 2*x + 3*x^2 + 3*x^3 + 3*x^4 + 3*x^5 + ...
```

define(*x*)

EXAMPLES: Test Recursive 0

```
sage: L = LazyPowerSeriesRing(QQ)
sage: one = L(1)
sage: monom = L.gen()
sage: s = L()
sage: s._name = 's'
sage: s.define(one+monom*s)
```

```
sage: s.aorder
0
sage: s.order
Unknown series order
sage: [s.coefficient(i) for i in range(6)]
[1, 1, 1, 1, 1, 1]
```

Test Recursive 1

```
sage: s = L()
sage: s._name = 's'
sage: s.define(one+monom*s*s)
sage: s.aorder
0
sage: s.order
Unknown series order
sage: [s.coefficient(i) for i in range(6)]
[1, 1, 2, 5, 14, 42]
```

Test Recursive 1b

```
sage: s = L()
sage: s._name = 's'
sage: s.define(monom + s*s)
sage: s.aorder
1
sage: s.order
Unknown series order
sage: [s.coefficient(i) for i in range(7)]
[0, 1, 1, 2, 5, 14, 42]
```

Test Recursive 2

```
sage: s = L()
sage: s._name = 's'
sage: t = L()
sage: t._name = 't'
sage: s.define(one+monom*t*t*t)
sage: t.define(one+monom*s*s)
sage: [s.coefficient(i) for i in range(9)]
[1, 1, 3, 9, 34, 132, 546, 2327, 10191]
sage: [t.coefficient(i) for i in range(9)]
[1, 1, 2, 7, 24, 95, 386, 1641, 7150]
```

Test Recursive 2b

```
sage: s = L()
sage: s._name = 's'
sage: t = L()
sage: t._name = 't'
sage: s.define(monom + t*t*t)
sage: t.define(monom + s*s)
sage: [s.coefficient(i) for i in range(9)]
[0, 1, 0, 1, 3, 3, 7, 30, 63]
sage: [t.coefficient(i) for i in range(9)]
[0, 1, 1, 0, 2, 6, 7, 20, 75]
```

Test Recursive 3

```
sage: s = L()
sage: s._name = 's'
sage: s.define(one+monom*s*s*s)
sage: [s.coefficient(i) for i in range(10)]
[1, 1, 3, 12, 55, 273, 1428, 7752, 43263, 246675]
```

derivative()

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: L = LazyPowerSeriesRing(QQ)
sage: one = L(1)
sage: monom = L.gen()
sage: s = L([1])
sage: u = s.derivative()
sage: u.coefficients(10)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

sage: s = L()
sage: s._name = 's'
sage: s.define(one+monom*s*s)
sage: u = s.derivative()
sage: u.coefficients(5) #[1*1, 2*2, 3*5, 4*14, 5*42]
[1, 4, 15, 56, 210]

sage: s = L([1])
sage: t = L([0,1])
sage: u = s(t).derivative()
sage: v = (s.derivative().compose(t))*t.derivative()
sage: u.coefficients(11)
[1, 4, 12, 32, 80, 192, 448, 1024, 2304, 5120, 11264]
sage: v.coefficients(11)
[1, 4, 12, 32, 80, 192, 448, 1024, 2304, 5120, 11264]

sage: s = L(); s._name='s'
sage: t = L(); t._name='t'
sage: s.define(monom+t*t*t)
sage: t.define(monom+s*s)
sage: u = (s*t).derivative()
sage: v = s.derivative()*t + s*t.derivative()
sage: u.coefficients(10)
[0, 2, 3, 4, 30, 72, 133, 552, 1791, 4260]
sage: v.coefficients(10)
[0, 2, 3, 4, 30, 72, 133, 552, 1791, 4260]
sage: u.coefficients(10) == v.coefficients(10)
True

sage: f = L._new_initial(2, Stream([0,0,4,5,6,0]))
sage: d = f.derivative()
sage: d.get_aorder()
1
sage: d.coefficients(5)
[0, 8, 15, 24, 0]
```

exponential()

TESTS:

```
sage: def inv_factorial():
...       q = 1
...       yield 0
...       yield q
...       n = 2
...       while True:
...           q = q / n
...           yield q
...           n += 1
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L(inv_factorial()) #e^(x)-1
sage: u = f.exponential()
sage: g = inv_factorial()
sage: z1 = [1,1,2,5,15,52,203,877,4140,21147,115975]
sage: l1 = [z*next(g) for z in z1]
sage: l1 = [1] + l1[1:]
sage: u.coefficients(11)
[1, 1, 1, 5/6, 5/8, 13/30, 203/720, 877/5040, 23/224, 1007/17280, 4639/145152]
sage: l1 == u.coefficients(11)
True
```

get_aorder()

Returns the approximate order of self.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L.gen()
sage: a.get_aorder()
1
```

get_order()

Returns the order of self.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L.gen()
sage: a.get_order()
1
```

get_stream()

Returns self's underlying Stream object.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L.gen()
sage: s = a.get_stream()
sage: [s[i] for i in range(5)]
[0, 1, 0, 0, 0]
```

initialize_coefficient_stream(*compute_coefficients*)

Initializes the coefficient stream.

INPUT: compute_coefficients

TESTS:

```
sage: from sage.combinat.species.series_order import inf, unk
sage: L = LazyPowerSeriesRing(QQ)
```

```
sage: f = L()
sage: compute_coefficients = lambda ao: iter(ZZ)
sage: f.order = inf
sage: f.aorder = inf
sage: f.initialize_coefficient_stream(compute_coefficients)
sage: f.coefficients(5)
[0, 0, 0, 0, 0]

sage: f = L()
sage: compute_coefficients = lambda ao: iter(ZZ)
sage: f.order = 1
sage: f.aorder = 1
sage: f.initialize_coefficient_stream(compute_coefficients)
sage: f.coefficients(5)
[0, 1, -1, 2, -2]
```

integral(*integration_constant=0*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: zero = L(0)
sage: s = zero
sage: t = s.integral()
sage: t.is_zero()
True

sage: s = zero
sage: t = s.integral(1)
sage: t.coefficients(6)
[1, 0, 0, 0, 0, 0]
sage: t._stream.is_constant()
True

sage: s = L.term(1, 0)
sage: t = s.integral()
sage: t.coefficients(6)
[0, 1, 0, 0, 0, 0]
sage: t._stream.is_constant()
True

sage: s = L.term(1,0)
sage: t = s.integral(1)
sage: t.coefficients(6)
[1, 1, 0, 0, 0, 0]
sage: t._stream.is_constant()
True

sage: s = L.term(1, 4)
sage: t = s.integral()
sage: t.coefficients(10)
[0, 0, 0, 0, 0, 1/5, 0, 0, 0, 0]

sage: s = L.term(1,4)
sage: t = s.integral(1)
sage: t.coefficients(10)
[1, 0, 0, 0, 0, 1/5, 0, 0, 0, 0]
```

TESTS:

```
sage: from sage.combinat.species.stream import Stream
sage: f = L._new_initial(2, Stream([0,0,4,5,6,0]))
sage: i = f.derivative().integral()
sage: i.get_aorder()
2
sage: i.coefficients(5)
[0, 0, 4, 5, 6]
sage: i = f.derivative().integral(1)
sage: i.get_aorder()
0
sage: i.coefficients(5)
[1, 0, 4, 5, 6]
```

is_finite(*n=None*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L([0,0,1,0,0]); a
O(1)
sage: a.is_finite()
False
sage: c = a[4]
sage: a.is_finite()
False
sage: a.is_finite(4)
False
sage: c = a[5]
sage: a.is_finite()
True
sage: a.is_finite(4)
True
```

is_zero()

Returns True if and only if self is zero.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: s = L([0,2,3,0])
sage: s.is_zero()
False

sage: s = L(0)
sage: s.is_zero()
True

sage: s = L([0])
sage: s.is_zero()
False
sage: s.coefficient(0)
0
sage: s.coefficient(1)
0
sage: s.is_zero()
True
```

iterator(*n=0*, *initial=None*)

Returns an iterator for the coefficients of self starting at n.

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L(range(10))
sage: g = f.iterator(2)
sage: [next(g) for i in range(5)]
[2, 3, 4, 5, 6]
sage: g = f.iterator(2, initial=[0,0])
sage: [next(g) for i in range(5)]
[0, 0, 2, 3, 4]
```

refine_aorder()

Refines the approximate order of self as much as possible without computing any coefficients.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L([0,0,0,0,1])
sage: a.aorder
0
sage: a.coefficient(2)
0
sage: a.aorder
0
sage: a.refine_aorder()
sage: a.aorder
3

sage: a = L([0,0])
sage: a.aorder
0
sage: a.coefficient(5)
0
sage: a.refine_aorder()
sage: a.aorder
Infinite series order

sage: a = L([0,0,1,0,0,0])
sage: a[4]
0
sage: a.refine_aorder()
sage: a.aorder
2
```

restricted(*min=None*, *max=None*)

Returns the power series restricted to the coefficients starting at min and going up to, but not including max. If min is not specified, then it is assumed to be zero. If max is not specified, then it is assumed to be infinity.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L([1])
sage: a.restricted().coefficients(10)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
sage: a.restricted(min=2).coefficients(10)
[0, 0, 1, 1, 1, 1, 1, 1, 1, 1]
sage: a.restricted(max=5).coefficients(10)
[1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
sage: a.restricted(min=2, max=6).coefficients(10)
```

```
[0, 0, 1, 1, 1, 1, 0, 0, 0, 0]
```

set_approximate_order(*new_order*)

Sets the approximate order of self and returns True if the approximate order has changed otherwise it will return False.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L([0,0,0,3,2,1,0])
sage: f.get_aorder()
0
sage: f.set_approximate_order(3)
True
sage: f.set_approximate_order(3)
False
```

tail()

Returns the power series whose coefficients obtained by subtracting the constant term from this series and then dividing by x.

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L(range(20))
sage: g = f.tail()
sage: g.coefficients(10)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

times(*y*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: gs0 = L(0)
sage: gs1 = L([1])

sage: prod0 = gs0 * gs1
sage: [prod0.coefficient(i) for i in range(11)]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

sage: prod1 = gs1 * gs0
sage: [prod1.coefficient(i) for i in range(11)]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

sage: prod2 = gs1 * gs1
sage: [prod2.coefficient(i) for i in range(11)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

sage: gs1 = L([1,2,4,8,0])
sage: gs2 = L([-1, 0,-1,-9,22,0])

sage: prod1 = gs1 * gs2
sage: [prod1.coefficient(i) for i in range(11)]
[-1, -2, -5, -19, 0, 0, 16, 176, 0, 0, 0]

sage: prod2 = gs2 * gs1
sage: [prod2.coefficient(i) for i in range(11)]
[-1, -2, -5, -19, 0, 0, 16, 176, 0, 0, 0]
```

class sage.combinat.species.series.**LazyPowerSeriesRing**(*R*, *element_class=None*, *names=None*)

Bases: sage.rings.ring.Algebra

TESTS:

```
sage: from sage.combinat.species.series import LazyPowerSeriesRing
sage: L = LazyPowerSeriesRing(QQ)
sage: loads(dumps(L))
Lazy Power Series Ring over Rational Field
```

gen(*i=0*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: L.gen().coefficients(5)
[0, 1, 0, 0, 0]
```

identity_element()

Returns the one power series.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: L.identity_element()
1
```

ngens()

EXAMPLES:

```
sage: LazyPowerSeriesRing(QQ).ngens()
1
```

product_generator(*g*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: s1 = L([1,1,0])
sage: s2 = L([1,0,1,0])
sage: s3 = L([1,0,0,1,0])
sage: s4 = L([1,0,0,0,1,0])
sage: s5 = L([1,0,0,0,0,1,0])
sage: s6 = L([1,0,0,0,0,0,1,0])
sage: s = [s1, s2, s3, s4, s5, s6]
sage: def g():
...       for a in s:
...           yield a
sage: p = L.product_generator(g())
sage: p.coefficients(26)
[1, 1, 1, 2, 2, 3, 4, 4, 4, 5, 5, 5, 5, 4, 4, 4, 3, 2, 2, 1, 1, 1, 0, 0, 0, 0]

sage: def m(n):
...       yield 1
...       while True:
...           for i in range(n-1):
...               yield 0
...           yield 1
...
sage: def s(n):
...       q = 1/n
...       yield 0
...       while True:
```

```
...            for i in range(n-1):
...                yield 0
...            yield q
...

sage: def lhs_gen():
...       n = 1
...       while True:
...           yield L(m(n))
...           n += 1
...

sage: def rhs_gen():
...       n = 1
...       while True:
...           yield L(s(n))
...           n += 1
...
sage: lhs = L.product_generator(lhs_gen())
sage: rhs = L.sum_generator(rhs_gen()).exponential()
sage: lhs.coefficients(10)
[1, 1, 2, 3, 5, 7, 11, 15, 22, 30]
sage: rhs.coefficients(10)
[1, 1, 2, 3, 5, 7, 11, 15, 22, 30]
```

sum(*a*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: l = [L(ZZ)]*3
sage: L.sum(l).coefficients(10)
[0, 3, -3, 6, -6, 9, -9, 12, -12, 15]
```

sum_generator(*g*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: g = [L([1])]*6 + [L(0)]
sage: t = L.sum_generator(g)
sage: t.coefficients(10)
[1, 2, 3, 4, 5, 6, 6, 6, 6, 6]

sage: s = L([1])
sage: def g():
...       while True:
...           yield s
sage: t = L.sum_generator(g())
sage: t.coefficients(9)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

term(*r*, *n*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: L.term(0,0)
0
sage: L.term(3,2).coefficients(5)
[0, 0, 3, 0, 0]
```

zero()
Returns the zero power series.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: L.zero()
0
```

TESTS:

Check that the method $zero_element$ raises a warning (trac ticket #17694):

```
sage: L.zero_element()
doctest:...: DeprecationWarning: zero_element is deprecated. Please use zero instead.
See http://trac.sagemath.org/17694 for details.
0
```

zero_element(*args, **kwds)
Deprecated: Use `zero()` instead. See trac ticket #17694 for details.

sage.combinat.species.series.**uninitialized**()
EXAMPLES:

```
sage: from sage.combinat.species.series import uninitialized
sage: uninitialized()
Traceback (most recent call last):
...
RuntimeError: we should never be here
```

5.1.296 Series Order

This file provides some utility classes which are useful when working with unknown, known, and infinite series orders for univariate power series.

This code is based on the work of Ralf Hemmecke and Martin Rubey's Aldor-Combinat, which can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/aldor/combinat/index.html. In particular, the relevant section for this file can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/AldorCombinat/combinatsu30.html.

class sage.combinat.species.series_order.**InfiniteSeriesOrder**
Bases: sage.combinat.species.series_order.SeriesOrderElement

class sage.combinat.species.series_order.**SeriesOrderElement**

class sage.combinat.species.series_order.**UnknownSeriesOrder**
Bases: sage.combinat.species.series_order.SeriesOrderElement

sage.combinat.species.series_order.**bounded_decrement**(*x*)
EXAMPLES:

```
sage: from sage.combinat.species.series_order import *
sage: u = UnknownSeriesOrder()
sage: bounded_decrement(u)
Unknown series order
sage: bounded_decrement(4)
3
sage: bounded_decrement(0)
0
```

sage.combinat.species.series_order.**increment**(*x*)
EXAMPLES:

```
sage: from sage.combinat.species.series_order import *
sage: u = UnknownSeriesOrder()
sage: increment(u)
Unknown series order
sage: increment(2)
3
```

5.1.297 Set Species

class sage.combinat.species.set_species.**SetSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: sage.combinat.species.species.GenericCombinatorialSpecies, sage.structure.unique_representation.UniqueRepresentation

Returns the species of sets.

EXAMPLES:

```
sage: E = species.SetSpecies()
sage: E.structures([1,2,3]).list()
[{1, 2, 3}]
sage: E.isotype_generating_series().coefficients(4)
[1, 1, 1, 1]

sage: S = species.SetSpecies()
sage: c = S.generating_series().coefficients(3)
sage: S._check()
True
sage: S == loads(dumps(S))
True
```

class sage.combinat.species.set_species.**SetSpeciesStructure**(*parent*, *labels*, *list*)

Bases: sage.combinat.species.structure.GenericSpeciesStructure

This is a base class from which the classes for the structures inherit.

EXAMPLES:

```
sage: from sage.combinat.species.structure import GenericSpeciesStructure
sage: a = GenericSpeciesStructure(None, [2,3,4], [1,2,3])
sage: a
[2, 3, 4]
sage: a.parent() is None
True
sage: a == loads(dumps(a))
True
```

automorphism_group()

Returns the group of permutations whose action on this set leave it fixed. For the species of sets, there is only one isomorphism class, so every permutation is in its automorphism group.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
sage: a.automorphism_group()
Symmetric group of order 3! as a permutation group
```

canonical_label()
EXAMPLES:

```
sage: S = species.SetSpecies()
sage: a = S.structures(["a","b","c"]).random_element(); a
{'a', 'b', 'c'}
sage: a.canonical_label()
{'a', 'b', 'c'}
```

transport(*perm*)
Returns the transport of this set along the permutation perm.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
sage: p = PermutationGroupElement((1,2))
sage: a.transport(p)
{'a', 'b', 'c'}
```

sage.combinat.species.set_species.**SetSpecies_class**
alias of SetSpecies

5.1.298 Combinatorial Species

This file defines the main classes for working with combinatorial species, operations on them, as well as some implementations of basic species required for other constructions.

This code is based on the work of Ralf Hemmecke and Martin Rubey's Aldor-Combinat, which can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/aldor/combinat/index.html. In particular, the relevant section for this file can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/AldorCombinat/combinatse8.html.

Weighted Species:

As a first application of weighted species, we count unlabeled ordered trees by total number of nodes and number of internal nodes. To achieve this, we assign a weight of 1 to the leaves and of q to internal nodes:

```
sage: q = QQ['q'].gen()
sage: leaf = species.SingletonSpecies()
sage: internal_node = species.SingletonSpecies(weight=q)
sage: L = species.LinearOrderSpecies(min=1)
sage: T = species.CombinatorialSpecies()
sage: T.define(leaf + internal_node*L(T))
sage: T.isotype_generating_series().coefficients(6)
[0, 1, q, q^2 + q, q^3 + 3*q^2 + q, q^4 + 6*q^3 + 6*q^2 + q]
```

Consider the following:

```
sage: T.isotype_generating_series().coefficient(4)
q^3 + 3*q^2 + q
```

This means that, among the trees on 4 nodes, one has a single internal node, three have two internal nodes, and one has three internal nodes.

class sage.combinat.species.species.**GenericCombinatorialSpecies**(*min=None*, *max=None*, *weight=None*)
Bases: sage.structure.sage_object.SageObject

TESTS:

```
sage: P = species.PermutationSpecies(size=3)
sage: P._weight
1
sage: P._min
3
sage: P._max
4
```

algebraic_equation_system()

Returns a system of algebraic equations satisfied by this species. The nodes are numbered in the order that they appear as vertices of the associated digraph.

EXAMPLES:

```
sage: B = species.BinaryTreeSpecies()
sage: B.algebraic_equation_system()
[-node3^2 + node1, -node1 + node3 - z]

sage: B.digraph().vertices()
[Combinatorial species,
 Product of (Combinatorial species) and (Combinatorial species),
 Singleton species,
 Sum of (Singleton species) and (Product of (Combinatorial species) and (Combinatorial speci

sage: B.algebraic_equation_system()[0].parent()
Multivariate Polynomial Ring in node0, node1, node2, node3 over Fraction Field of Univariate
```

composition(*g*)

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: S(S)
Composition of (Set species) and (Set species)
```

cycle_index_series(*base_ring=None*)

Returns the cycle index series for this species.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: g = P.cycle_index_series()
sage: g.coefficients(4)
[p[], p[1], p[1, 1] + p[2], p[1, 1, 1] + p[2, 1] + p[3]]
```

digraph()

Returns a directed graph where the vertices are the individual species that make up this one.

EXAMPLES:

```
sage: X = species.SingletonSpecies()
sage: B = species.CombinatorialSpecies()
sage: B.define(X+B*B)
sage: g = B.digraph(); g
Multi-digraph on 4 vertices

sage: g_c, labels = g.canonical_label(certify=True)
sage: g.relabel()
sage: g_r = g.canonical_label()
sage: g_c == g_r
```

```
True
sage: list(sorted(labels.keys()))
[Combinatorial species,
 Product of (Combinatorial species) and (Combinatorial species),
 Singleton species,
 Sum of (Singleton species) and (Product of (Combinatorial species) and (Combinatorial speci
sage: list(sorted(labels.values()))
[0, 1, 2, 3]
```

functorial_composition(*g*)

Returns the functorial composition of self with g.

EXAMPLES:

```
sage: E = species.SetSpecies()
sage: E2 = E.restricted(min=2, max=3)
sage: WP = species.SubsetSpecies()
sage: P2 = E2*E
sage: G = WP.functorial_composition(P2)
sage: G.isotype_generating_series().coefficients(5)
[1, 1, 2, 4, 11]
```

generating_series(*base_ring=None*)

Returns the generating series for this species. This is an exponential generating series so the nth coefficient of the series corresponds to the number of labeled structures with n labels divided by n!.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: g = P.generating_series()
sage: g.coefficients(4)
[1, 1, 1, 1]
sage: g.counts(4)
[1, 1, 2, 6]
sage: P.structures([1,2,3]).list()
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
sage: len(_)
6
```

is_weighted()

Returns True if this species has a nontrivial weighting associated with it.

EXAMPLES:

```
sage: C = species.CycleSpecies()
sage: C.is_weighted()
False
```

isotype_generating_series(*base_ring=None*)

Returns the isotype generating series for this species. The nth coefficient of this series corresponds to the number of isomorphism types for the structures on n labels.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: g = P.isotype_generating_series()
sage: g.coefficients(4)
[1, 1, 2, 3]
sage: g.counts(4)
[1, 1, 2, 3]
sage: P.isotypes([1,2,3]).list()
```

```
[[2, 3, 1], [2, 1, 3], [1, 2, 3]]
sage: len(_)
3
```

isotypes(*labels*, *structure_class=None*)

EXAMPLES:

```
sage: F = CombinatorialSpecies()
sage: F.isotypes([1,2,3]).list()
Traceback (most recent call last):
...
NotImplementedError
```

product(*g*)

Returns the product of self and g.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: F = P * P; F
Product of (Permutation species) and (Permutation species)
```

restricted(**args*, ***kwds*)

EXAMPLES:

```
sage: S = species.SetSpecies().restricted(min=3); S
Set species with min=3
sage: S.structures([1,2]).list()
[]
sage: S.generating_series().coefficients(5)
[0, 0, 0, 1/6, 1/24]
```

structures(*labels*, *structure_class=None*)

EXAMPLES:

```
sage: F = CombinatorialSpecies()
sage: F.structures([1,2,3]).list()
Traceback (most recent call last):
...
NotImplementedError
```

sum(*g*)

Returns the sum of self and g.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: F = P + P; F
Sum of (Permutation species) and (Permutation species)
sage: F.structures([1,2]).list()
[[1, 2], [2, 1], [1, 2], [2, 1]]
```

weight_ring()

Returns the ring in which the weights of this species occur.

By default, this is just the field of rational numbers.

EXAMPLES:

```
sage: species.SetSpecies().weight_ring()
Rational Field
```

weighted(*weight*)
Returns a version of this species with the specified weight.

EXAMPLES:

```
sage: t = ZZ['t'].gen()
sage: C = species.CycleSpecies(); C
Cyclic permutation species
sage: C.weighted(t)
Cyclic permutation species with weight=t
```

5.1.299 Streams or Infinite Arrays

This code is based on the work of Ralf Hemmecke and Martin Rubey's Aldor-Combinat, which can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/aldor/combinat/index.html. In particular, the relevant section for this file can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/AldorCombinat/combinatse12.html.

sage.combinat.species.stream.**Stream**(*x=None*, *const=None*)
Returns a stream.

EXAMPLES: We can create a constant stream by just passing a

```
sage: from sage.combinat.species.stream import Stream
sage: s = Stream(const=0)
sage: [s[i] for i in range(10)]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

class sage.combinat.species.stream.**Stream_class**(*gen=None*, *const=None*, *func=None*)
Bases: sage.structure.sage_object.SageObject

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: from itertools import izip
sage: s = Stream(const=0)
sage: len(s)
1
sage: [x for (x,i) in izip(s, range(4))]
[0, 0, 0, 0]
sage: len(s)
1

sage: s = Stream(const=4)
sage: g = iter(s)
sage: l1 = [x for (x,i) in izip(g, range(10))]
sage: l = [4 for k in range(10)]
sage: l == l1
True

sage: h = lambda l: 1 if len(l) < 2 else l[-1] + l[-2]
sage: fib = Stream(h)
sage: [x for (x,i) in izip(fib, range(11))]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

sage: r = [4, 3, 5, 2, 6, 1, 1, 1, 1, 1]
sage: l = [4, 3, 5, 2, 6, 1]
sage: s = Stream(l)
sage: s[3] = -1
sage: [x for (x,i) in izip(s, r)]
```

```
[4, 3, 5, -1, 6, 1, 1, 1, 1, 1]
sage: s[5] = -2
sage: [x for (x,i) in izip(s, r)]
[4, 3, 5, -1, 6, -2, 1, 1, 1, 1]
sage: s[6] = -3
sage: [x for (x,i) in izip(s, r)]
[4, 3, 5, -1, 6, -2, -3, 1, 1, 1]
sage: s[8] = -4
sage: [x for (x,i) in izip(s, r)]
[4, 3, 5, -1, 6, -2, -3, 1, -4, 1]
sage: a = Stream(const=0)
sage: a[2] = 3
sage: [x for (x,i) in izip(a, range(4))]
[0, 0, 3, 0]
```

data()

Returns a list of all the coefficients computed so far.

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream, _integers_from
sage: s = Stream(_integers_from(3))
sage: s.data()
[]
sage: s[5]
8
sage: s.data()
[3, 4, 5, 6, 7, 8]
```

is_constant()

Returns True if and only if

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: s = Stream([1,2,3])
sage: s.is_constant()
False
sage: s[3]
3
sage: s.data()
[1, 2, 3]
sage: s.is_constant()
True
```

TESTS:

```
sage: l = [2,3,5,7,11,0]
sage: s = Stream(l)
sage: s.is_constant()
False
sage: s[3]
7
sage: s.is_constant()
False
sage: s[5]
0
sage: s.is_constant()
False
sage: s[6]
```

```
0
sage: s.is_constant()
True

sage: s = Stream(const='I am constant.')
sage: s.is_constant()
True
```

map(*f*)

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: s = Stream(ZZ)
sage: square = lambda x: x^2
sage: ss = s.map(square)
sage: [ss[i] for i in range(10)]
[0, 1, 1, 4, 4, 9, 9, 16, 16, 25]
```

TESTS:

```
sage: from itertools import izip
sage: f = lambda l: 0 if len(l) == 0 else l[-1] + 1
sage: o = Stream(f)
sage: [x for (x,i) in izip(o, range(10))]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: double = lambda z: 2*z
sage: t = o.map(double)
sage: [x for (x,i) in izip(t, range(10))]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

sage: double = lambda z: 2*z
sage: o = Stream([0,1,2,3])
sage: [x for (x,i) in izip(o, range(6))]
[0, 1, 2, 3, 3, 3]
sage: t = o.map(double)
sage: [x for (x,i) in izip(t, range(6))]
[0, 2, 4, 6, 6, 6]
```

number_computed()

Returns the number of coefficients computed so far.

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: l = [4,3,5,7,4,1,9,7]
sage: s = Stream(l)
sage: s[3]
7
sage: len(s)
4
sage: s[3]
7
sage: len(s)
4
sage: s[1]
3
sage: len(s)
4
sage: s[4]
4
```

```
sage: len(s)
5
```

TESTS:

```
sage: l = ['Hello', ' ', 'World', '!']
sage: s = Stream(l)
sage: len(s)
0
sage: s[2]
'World'
sage: len(s)
3
sage: u = ""
sage: for i in range(len(s)): u += s[i]
sage: u
'Hello World'
sage: v = ""
sage: for i in range(10): v += s[i]
sage: v
'Hello World!!!!!!!'
sage: len(s)
4
```

set_gen(*gen*)

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: from itertools import izip
sage: fib = Stream()
sage: def g():
...           yield 1
...           yield 1
...           n = 0
...           while True:
...               yield fib[n] + fib[n+1]
...               n += 1

sage: fib.set_gen(g())
sage: [x for (x,i) in izip(fib, range(11))]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

sage: l = [4,3,5,2,6,1]
sage: s = Stream(l)
sage: s[3]
2
sage: len(s)
4
sage: g = iter(l)
sage: s.set_gen(g)
sage: s[5]
3
sage: len(s)
6
```

stretch(*k*)

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: s = Stream(range(1, 10))
sage: s2 = s.stretch(2)
sage: [s2[i] for i in range(10)]
[1, 0, 2, 0, 3, 0, 4, 0, 5, 0]
```

5.1.300 Species structures

We will illustrate the use of the structure classes using the "balls and bars" model for integer compositions. An integer composition of 6 such as [2, 1, 3] can be represented in this model as 'oooooo' where the 6 o's correspond to the balls and the 2 's correspond to the bars. If BB is our species for this model, the it satisfies the following recursive definition:

BB = o + o*BB + o*|*BB

Here we define this species using the default structures:

```
sage: ball = species.SingletonSpecies(); o = var('o')
sage: bar = species.EmptySetSpecies()
sage: BB = CombinatorialSpecies()
sage: BB.define(ball + ball*BB + ball*bar*BB)
sage: BB.isotypes([o]*3).list()
[o*(o*o), o*((o*{})*o), (o*{})*(o*o), (o*{})*((o*{})*o)]
```

If we ignore the parentheses, we can read off that the integer compositions are [3], [2, 1], [1, 2], and [1, 1, 1].

class sage.combinat.species.structure.**GenericSpeciesStructure**(*parent*, *labels*, *list*)

Bases: sage.combinat.combinat.CombinatorialObject

This is a base class from which the classes for the structures inherit.

EXAMPLES:

```
sage: from sage.combinat.species.structure import GenericSpeciesStructure
sage: a = GenericSpeciesStructure(None, [2,3,4], [1,2,3])
sage: a
[2, 3, 4]
sage: a.parent() is None
True
sage: a == loads(dumps(a))
True
```

change_labels(*labels*)

Return a relabelled structure.

INPUT:

- labels, a list of labels.

OUTPUT:

A structure with the i-th label of self replaced with the i-th label of the list.

EXAMPLES:

```
sage: P = species.SubsetSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.change_labels([1,2,3]) for s in S]
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
```

is_isomorphic(*x*)

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: a = S.structures([1,2,3]).random_element(); a
{1, 2, 3}
sage: b = S.structures(['a','b','c']).random_element(); b
{'a', 'b', 'c'}
sage: a.is_isomorphic(b)
True
```

labels()

Returns the labels used for this structure.

Note: This includes labels which may not "appear" in this particular structure.

EXAMPLES:

```
sage: P = species.SubsetSpecies()
sage: s = P.structures(["a", "b", "c"]).random_element()
sage: s.labels()
['a', 'b', 'c']
```

parent()

Returns the species that this structure is associated with.

EXAMPLES:

```
sage: L = species.LinearOrderSpecies()
sage: a,b = L.structures([1,2])
sage: a.parent()
Linear order species
```

class sage.combinat.species.structure.**IsotypesWrapper**(*species*, *labels*, *structure_class*)

Bases: sage.combinat.species.structure.SpeciesWrapper

A base class for the set of isotypes of a species with given set of labels. An object of this type is returned when you call the `isotypes()` method of a species.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: S = F.isotypes([1,2,3])
sage: S == loads(dumps(S))
True
```

class sage.combinat.species.structure.**SimpleIsotypesWrapper**(*species*, *labels*, *structure_class*)

Bases: sage.combinat.species.structure.SpeciesWrapper

Warning: This is deprecated and currently not used for anything.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: S = F.structures([1,2,3])
sage: S == loads(dumps(S))
True
```

class sage.combinat.species.structure.**SimpleStructuresWrapper**(*species*, *labels*, *structure_class*)

Bases: sage.combinat.species.structure.SpeciesWrapper

Warning: This is deprecated and currently not used for anything.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: S = F.structures([1,2,3])
sage: S == loads(dumps(S))
True
```

sage.combinat.species.structure.**SpeciesStructure**

alias of GenericSpeciesStructure

class sage.combinat.species.structure.**SpeciesStructureWrapper**(*parent*, *s*, ***options*)

Bases: sage.combinat.species.structure.GenericSpeciesStructure

This is a class for the structures of species such as the sum species that do not provide "additional" structure. For example, if you have the sum C of species A and B, then a structure of C will either be either something from A or B. Instead of just returning one of these directly, a "wrapper" is put around them so that they have their parent is C rather than A or B:

```
sage: X = species.SingletonSpecies()
sage: X2 = X+X
sage: s = X2.structures([1]).random_element(); s
1
sage: s.parent()
Sum of (Singleton species) and (Singleton species)
sage: from sage.combinat.species.structure import SpeciesStructureWrapper
sage: issubclass(type(s), SpeciesStructureWrapper)
True
```

EXAMPLES:

```
sage: E = species.SetSpecies(); B = E+E
sage: s = B.structures([1,2,3]).random_element()
sage: s.parent()
Sum of (Set species) and (Set species)
sage: s == loads(dumps(s))
True
```

canonical_label()

EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: s = (P+P).structures([1,2,3]).random_element(); s
{{1, 3}, {2}}
sage: s.canonical_label()
{{1, 2}, {3}}
```

change_labels(*labels*)

Return a relabelled structure.

INPUT:

- `labels`, a list of labels.

OUTPUT:

A structure with the i-th label of self replaced with the i-th label of the list.

EXAMPLES:

```
sage: X = species.SingletonSpecies()
sage: X2 = X+X
sage: s = X2.structures([1]).random_element(); s
1
sage: s.change_labels(['a'])
'a'
```

transport(*perm*)

EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: s = (P+P).structures([1,2,3]).random_element(); s
{{1, 3}, {2}}
sage: s.transport(PermutationGroupElement((2,3)))
{{1, 2}, {3}}
```

class sage.combinat.species.structure.**SpeciesWrapper**(*species*, *labels*, *iterator*, *generating_series*, *name*, *structure_class*)

Bases: sage.combinat.combinat.CombinatorialClass

This is a abstract base class for the set of structures of a species as well as the set of isotypes of the species.

Note: One typically does not use SpeciesWrapper directly, but instead instantiates one of its subclasses: StructuresWrapper or IsotypesWrapper.

EXAMPLES:

```
sage: from sage.combinat.species.structure import SpeciesWrapper
sage: F = species.SetSpecies()
sage: S = SpeciesWrapper(F, [1,2,3], "_structures", "generating_series", 'Structures', None)
sage: S
Structures for Set species with labels [1, 2, 3]
sage: S.list()
[{1, 2, 3}]
sage: S.cardinality()
1
```

cardinality()

Returns the number of structures in this set.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: F.structures([1,2,3]).cardinality()
1
```

labels()

Returns the labels used on these structures. If X is the species, then labels() returns the preimage of these structures under the functor X.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: F.structures([1,2,3]).labels()
[1, 2, 3]
```

class sage.combinat.species.structure.**StructuresWrapper**(*species*, *labels*, *structure_class*)

Bases: sage.combinat.species.structure.SpeciesWrapper

A base class for the set of structures of a species with given set of labels. An object of this type is returned when you call the structures() method of a species.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: S = F.structures([1,2,3])
sage: S == loads(dumps(S))
True
```

5.1.301 Subset Species

class sage.combinat.species.subset_species.**SubsetSpecies**(*min=None*, *max=None*, *weight=None*)

Bases: sage.combinat.species.species.GenericCombinatorialSpecies, sage.structure.unique_representation.UniqueRepresentation

Returns the species of subsets.

EXAMPLES:

```
sage: S = species.SubsetSpecies()
sage: S.generating_series().coefficients(5)
[1, 2, 2, 4/3, 2/3]
sage: S.isotype_generating_series().coefficients(5)
[1, 2, 3, 4, 5]

sage: S = species.SubsetSpecies()
sage: c = S.generating_series().coefficients(3)
sage: S._check()
True
sage: S == loads(dumps(S))
True
```

class sage.combinat.species.subset_species.**SubsetSpeciesStructure**(*parent*, *labels*, *list*)

Bases: sage.combinat.species.structure.GenericSpeciesStructure

This is a base class from which the classes for the structures inherit.

EXAMPLES:

```
sage: from sage.combinat.species.structure import GenericSpeciesStructure
sage: a = GenericSpeciesStructure(None, [2,3,4], [1,2,3])
sage: a
[2, 3, 4]
sage: a.parent() is None
True
sage: a == loads(dumps(a))
True
```

automorphism_group()

Returns the group of permutations whose action on this subset leave it fixed.

EXAMPLES:

```
sage: F = species.SubsetSpecies()
sage: a = F.structures([1,2,3,4])[6]; a
{1, 3}
```

```
sage: a.automorphism_group()
Permutation Group with generators [(2,4), (1,3)]

sage: [a.transport(g) for g in a.automorphism_group()]
[{1, 3}, {1, 3}, {1, 3}, {1, 3}]
```

canonical_label()

EXAMPLES:

```
sage: P = species.SubsetSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.canonical_label() for s in S]
[{}, {'a'}, {'a'}, {'a'}, {'a', 'b'}, {'a', 'b'}, {'a', 'b'}, {'a', 'b', 'c'}]
```

complement()

EXAMPLES:

```
sage: F = species.SubsetSpecies()
sage: a = F.structures(["a", "b", "c"])[5]; a
{'a', 'c'}
sage: a.complement()
{'b'}
```

label_subset()

Returns a subset of the labels that "appear" in this structure.

EXAMPLES:

```
sage: P = species.SubsetSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.label_subset() for s in S]
[[], ['a'], ['b'], ['c'], ['a', 'b'], ['a', 'c'], ['b', 'c'], ['a', 'b', 'c']]
```

transport(*perm*)

Returns the transport of this subset along the permutation perm.

EXAMPLES:

```
sage: F = species.SubsetSpecies()
sage: a = F.structures(["a", "b", "c"])[5]; a
{'a', 'c'}
sage: p = PermutationGroupElement((1,2))
sage: a.transport(p)
{'b', 'c'}
sage: p = PermutationGroupElement((1,3))
sage: a.transport(p)
{'a', 'c'}
```

sage.combinat.species.subset_species.**SubsetSpecies_class**

alias of SubsetSpecies

5.1.302 Sum species

class sage.combinat.species.sum_species.**SumSpecies**(*F*, *G*, *min=None*, *max=None*, *weight=None*)

Bases: sage.combinat.species.species.GenericCombinatorialSpecies, sage.structure.unique_representation.UniqueRepresentation

Returns the sum of two species.

EXAMPLES:

```
sage: S = species.PermutationSpecies()
sage: A = S+S
sage: A.generating_series().coefficients(5)
[2, 2, 2, 2, 2]

sage: P = species.PermutationSpecies()
sage: F = P + P
sage: F._check()
True
sage: F == loads(dumps(F))
True
```

TESTS:

```
sage: A = species.SingletonSpecies() + species.SingletonSpecies()
sage: B = species.SingletonSpecies() + species.SingletonSpecies()
sage: C = species.SingletonSpecies() + species.SingletonSpecies(min=2)
sage: A is B
True
sage: (A is C) or (A == C)
False
```

left_summand()

Returns the left summand of this species.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: F = P + P*P
sage: F.left_summand()
Permutation species
```

right_summand()

Returns the right summand of this species.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: F = P + P*P
sage: F.right_summand()
Product of (Permutation species) and (Permutation species)
```

weight_ring()

Returns the weight ring for this species. This is determined by asking Sage's coercion model what the result is when you add elements of the weight rings for each of the operands.

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: C = S+S
sage: C.weight_ring()
Rational Field

sage: S = species.SetSpecies(weight=QQ['t'].gen())
sage: C = S + S
sage: C.weight_ring()
Univariate Polynomial Ring in t over Rational Field
```

class sage.combinat.species.sum_species.**SumSpeciesStructure**(*parent*, *s*, ***options*)

Bases: sage.combinat.species.structure.SpeciesStructureWrapper

This is a class for the structures of species such as the sum species that do not provide "additional" structure. For example, if you have the sum C of species A and B, then a structure of C will either be either something from A or B. Instead of just returning one of these directly, a "wrapper" is put around them so that they have their parent is C rather than A or B:

```
sage: X = species.SingletonSpecies()
sage: X2 = X+X
sage: s = X2.structures([1]).random_element(); s
1
sage: s.parent()
Sum of (Singleton species) and (Singleton species)
sage: from sage.combinat.species.structure import SpeciesStructureWrapper
sage: issubclass(type(s), SpeciesStructureWrapper)
True
```

EXAMPLES:

```
sage: E = species.SetSpecies(); B = E+E
sage: s = B.structures([1,2,3]).random_element()
sage: s.parent()
Sum of (Set species) and (Set species)
sage: s == loads(dumps(s))
True
```

sage.combinat.species.sum_species.**SumSpecies_class**
alias of SumSpecies

5.1.303 Subsets

The set of subsets of a finite set. The set can be given as a list or a Set or else as an integer n which encodes the set $\{1, 2, ..., n\}$. See Subsets for more information and examples.

AUTHORS:

- Mike Hansen: initial version
- Florent Hivert (2009/02/06): doc improvements + new methods

class sage.combinat.subset.**SubMultiset_s**(*s*)
Bases: sage.structure.parent.Parent

The combinatorial class of the sub multisets of s.

EXAMPLES:

```
sage: S = Subsets([1,2,2,3], submultiset=True)
sage: S.cardinality()
12
sage: S.list()
[[],
 [1],
 [2],
 [3],
 [1, 2],
 [1, 3],
 [2, 2],
 [2, 3],
 [1, 2, 2],
 [1, 2, 3],
 [2, 2, 3],
 [1, 2, 2, 3]]
```

```
sage: S.first()
[]
sage: S.last()
[1, 2, 2, 3]
```

cardinality()

Return the cardinality of self

EXAMPLES:

```
sage: S = Subsets([1,1,2,3],submultiset=True)
sage: S.cardinality()
12
sage: len(S.list())
12

sage: S = Subsets([1,1,2,2,3],submultiset=True)
sage: S.cardinality()
18
sage: len(S.list())
18

sage: S = Subsets([1,1,1,2,2,3],submultiset=True)
sage: S.cardinality()
24
sage: len(S.list())
24
```

generating_serie(*variable='x'*)

Return the serie (here a polynom) associated to the counting of the element of self weighted by the number of element they contain.

EXAMPLES:

```
sage: Subsets([1,1],submultiset=True).generating_serie()
x^2 + x + 1
sage: Subsets([1,1,2,3],submultiset=True).generating_serie()
x^4 + 3*x^3 + 4*x^2 + 3*x + 1
sage: Subsets([1,1,1,2,2,3,3,4],submultiset=True).generating_serie()
x^8 + 4*x^7 + 9*x^6 + 14*x^5 + 16*x^4 + 14*x^3 + 9*x^2 + 4*x + 1

sage: S = Subsets([1,1,1,2,2,3,3,4],submultiset=True)
sage: S.cardinality()
72
sage: sum(S.generating_serie())
72
```

random_element()

Return a random element of self with uniform law

EXAMPLES:

```
sage: S = Subsets([1,1,2,3], submultiset=True)
sage: S.random_element()
[2]
```

class sage.combinat.subset.**SubMultiset_sk**(*s*, *k*)

Bases: sage.combinat.subset.SubMultiset_s

The combinatorial class of the subsets of size k of a multiset s. Note that each subset is represented by a list of

the elements rather than a set since we can have multiplicities (no multiset data structure yet in sage).

EXAMPLES:

```
sage: S = Subsets([1,2,3,3],2,submultiset=True)
sage: S._k
2
sage: S.cardinality()
4
sage: S.first()
[1, 2]
sage: S.last()
[3, 3]
sage: [sub for sub in S]
[[1, 2], [1, 3], [2, 3], [3, 3]]
```

cardinality()

Return the cardinality of self

EXAMPLES:

```
sage: S = Subsets([1,2,2,3,3,3],4,submultiset=True)
sage: S.cardinality()
5
sage: len(list(S))
5

sage: S = Subsets([1,2,2,3,3,3],3,submultiset=True)
sage: S.cardinality()
6
sage: len(list(S))
6
```

generating_serie(*variable='x'*)

Return the serie (this case a polynom) associated to the counting of the element of self weighted by the number of element they contains

EXAMPLES:

```
sage: x = ZZ['x'].gen()
sage: l = [1,1,1,1,2,2,3]
sage: for k in xrange(len(l)):
....:     S = Subsets(l,k,submultiset=True)
....:     print S.generating_serie(x) == S.cardinality()*x**k
True
True
True
True
True
True
True
```

random_element()

Return a random submultiset of given length

EXAMPLES:

```
sage: Subsets(7,3).random_element()
{1, 4, 7}
sage: Subsets(7,5).random_element()
{1, 3, 4, 5, 7}
```

sage.combinat.subset.**Subsets**(*s*, *k=None*, *submultiset=False*)

Return the combinatorial class of the subsets of the finite set `s`. The set can be given as a list, Set or any iterable convertible to a set. Alternatively, a non-negative integer n can be provided in place of `s`; in this case, the result is the combinatorial class of the subsets of the set $\{1, 2, \ldots, n\}$ (i.e. of the Sage `range(1,n+1)`).

A second optional parameter `k` can be given. In this case, `Subsets` returns the combinatorial class of subsets of `s` of size `k`.

Warning: The subsets are returned as Sets. Do not assume that these Sets are ordered; they often are not! (E.g., `Subsets(10).list()[619]` returns `{10, 4, 5, 6, 7}` on my system.) See `SubsetsSorted` for a similar class which returns the subsets as sorted tuples.

Finally the option `submultiset` allows one to deal with sets with repeated elements, usually called multisets. The method then returns the class of all multisets in which every element is contained at most as often as it is contained in `s`. These multisets are encoded as lists.

EXAMPLES:

```
sage: S = Subsets([1, 2, 3]); S
Subsets of {1, 2, 3}
sage: S.cardinality()
8
sage: S.first()
{}
sage: S.last()
{1, 2, 3}
sage: S.random_element()  # random
{2}
sage: S.list()
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
```

Here is the same example where the set is given as an integer:

```
sage: S = Subsets(3)
sage: S.list()
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
```

We demonstrate various the effect of the various options:

```
sage: S = Subsets(3, 2); S
Subsets of {1, 2, 3} of size 2
sage: S.list()
[{1, 2}, {1, 3}, {2, 3}]

sage: S = Subsets([1, 2, 2], submultiset=True); S
SubMultiset of [1, 2, 2]
sage: S.list()
[[], [1], [2], [1, 2], [2, 2], [1, 2, 2]]

sage: S = Subsets([1, 2, 2, 3], 3, submultiset=True); S
SubMultiset of [1, 2, 2, 3] of size 3
sage: S.list()
[[1, 2, 2], [1, 2, 3], [2, 2, 3]]

sage: S = Subsets(['a','b','a','b'], 2, submultiset=True); S.list()
[['a', 'a'], ['a', 'b'], ['b', 'b']]
```

And it is possible to play with subsets of subsets:

```
sage: S = Subsets(3)
sage: S2 = Subsets(S); S2
Subsets of Subsets of {1, 2, 3}
sage: S2.cardinality()
256
sage: it = iter(S2)
sage: [next(it) for _ in xrange(8)]
[{}, {{}}, {{1}}, {{2}}, {{3}}, {{1, 2}},  {{1, 3}}, {{2, 3}}]
sage: S2.random_element()     # random
{{2}, {1, 2, 3}, {}}
sage: [S2.unrank(k) for k in xrange(256)] == S2.list()
True

sage: S3 = Subsets(S2)
sage: S3.cardinality()
115792089237316195423570985008687907853269984665640564039457584007913129639936
sage: S3.unrank(14123091480)
{{{1, 3}, {1, 2, 3}, {2}, {1}},
 {{2}, {1, 2, 3}, {}, {1, 2}},
 {},
 {{2}, {1, 2, 3}, {}, {3}, {1, 2}},
 {{1, 2, 3}, {}, {1}}, {{2}, {2, 3}, {}, {1, 2}}}

sage: T = Subsets(S2, 10)
sage: T.cardinality()
278826214642518400
sage: T.unrank(1441231049)
{{{3}, {1, 2}, {}, {2, 3}, {1}, {1, 3}, ..., {{2, 3}, {}}, {{}}}
```

class sage.combinat.subset.**SubsetsSorted**(*s*)

Bases: sage.combinat.subset.Subsets_s

Lightweight class of all subsets of some set S, with each subset being encoded as a sorted tuple.

Used to model indices of algebras given by subsets (so we don't have to explicitly build all 2^n subsets in memory). For example, CliffordAlgebra.

first()

Return the first element of self.

EXAMPLES:

```
sage: from sage.combinat.subset import SubsetsSorted
sage: S = SubsetsSorted(range(3))
sage: S.first()
()
```

last()

Return the last element of self.

EXAMPLES:

```
sage: from sage.combinat.subset import SubsetsSorted
sage: S = SubsetsSorted(range(3))
sage: S.last()
(0, 1, 2)
```

random_element()

Return a random element of self.

EXAMPLES:

```
sage: from sage.combinat.subset import SubsetsSorted
sage: S = SubsetsSorted(range(3))
sage: isinstance(S.random_element(), tuple)
True
```

unrank(*r*)

Return the subset which has rank `r`.

EXAMPLES:

```
sage: from sage.combinat.subset import SubsetsSorted
sage: S = SubsetsSorted(range(3))
sage: S.unrank(4)
(0, 1)
```

class sage.combinat.subset.**Subsets_s**(*s*)

Bases: sage.structure.parent.Parent

Subsets of a given set.

EXAMPLES:

```
sage: S = Subsets(4); S
Subsets of {1, 2, 3, 4}
sage: S.cardinality()
16
sage: Subsets(4).list()
[{}, {1}, {2}, {3}, {4},
 {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4},
 {1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4},
 {1, 2, 3, 4}]

sage: S = Subsets(Subsets(Subsets(GF(3)))); S
Subsets of Subsets of Subsets of Finite Field of size 3
sage: S.cardinality()
115792089237316195423570985008687907853269984665640564039457584007913129639936
sage: S.unrank(3149254230)
{{{1, 2}, {0, 1, 2}, {0, 2}, {0, 1}},
 {{1, 2}, {}, {0, 2}, {1}, {0, 1, 2}, {2}},
 {{1, 2}, {0}}, {{1, 2}, {0, 1}, {0, 1, 2}, {1}},
 {{0, 2}, {1}}}
```

an_element()

Returns an example of subset.

EXAMPLES:

```
sage: Subsets(0).an_element()
{}
sage: Subsets(3).an_element()
{1, 2}
sage: Subsets([2,4,5]).an_element()
{2, 4}
```

cardinality()

Return the number of subsets of the set `s`.

This is given by $2^{|s|}$.

EXAMPLES:

```
sage: Subsets(Set([1,2,3])).cardinality()
8
sage: Subsets([1,2,3,3]).cardinality()
8
sage: Subsets(3).cardinality()
8
```

element_class
alias of Set_object_enumerated

first()
Returns the first subset of s. Since we aren't restricted to subsets of a certain size, this is always the empty set.

EXAMPLES:

```
sage: Subsets([1,2,3]).first()
{}
sage: Subsets(3).first()
{}
```

last()
Return the last subset of s. Since we aren't restricted to subsets of a certain size, this is always the set s itself.

EXAMPLES:

```
sage: Subsets([1,2,3]).last()
{1, 2, 3}
sage: Subsets(3).last()
{1, 2, 3}
```

random_element()
Return a random element of the class of subsets of s (in other words, a random subset of s).

EXAMPLES:

```
sage: Subsets(3).random_element()           # random
{2}
sage: Subsets([4,5,6]).random_element()     # random
{5}

sage: S = Subsets(Subsets(Subsets([0,1,2])))
sage: S.cardinality()
115792089237316195423570985008687907853269984665640564039457584007913129639936
sage: s = S.random_element()
sage: s     # random
{{{1, 2}, {2}, {0}, {1}}, {{1, 2}, {0, 1, 2}, {0, 2}, {0}, {0, 1}}, ..., {{1, 2}, {2}, {1}},
sage: s in S
True
```

rank(*sub*)
Return the rank of sub as a subset of s.

EXAMPLES:

```
sage: Subsets(3).rank([])
0
sage: Subsets(3).rank([1,2])
4
sage: Subsets(3).rank([1,2,3])
```

```
7
sage: Subsets(3).rank([2,3,4])
Traceback (most recent call last):
...
ValueError: {2, 3, 4} is not a subset of {1, 2, 3}
```

underlying_set()
Return the set of elements.

EXAMPLES:

```
sage: Subsets(GF(13)).underlying_set()
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
```

unrank(*r*)
Return the subset of `s` that has rank `k`.

EXAMPLES:

```
sage: Subsets(3).unrank(0)
{}
sage: Subsets([2,4,5]).unrank(1)
{2}
sage: Subsets([1,2,3]).unrank(257)
Traceback (most recent call last):
...
IndexError: index out of range
```

class sage.combinat.subset.**Subsets_sk**(*s*, *k*)
Bases: sage.combinat.subset.Subsets_s

Subsets of fixed size of a set.

EXAMPLES:

```
sage: S = Subsets([0,1,2,5,7], 3); S
Subsets of {0, 1, 2, 5, 7} of size 3
sage: S.cardinality()
10
sage: S.first(), S.last()
({0, 1, 2}, {2, 5, 7})
sage: S.random_element()  # random
{0, 5, 7}
sage: S([0,2,7])
{0, 2, 7}
sage: S([0,3,5])
Traceback (most recent call last):
...
ValueError: {0, 3, 5} not in Subsets of {0, 1, 2, 5, 7} of size 3
sage: S([0])
Traceback (most recent call last):
...
ValueError: {0} not in Subsets of {0, 1, 2, 5, 7} of size 3
```

an_element()
Returns an example of subset.

EXAMPLES:

```
sage: Subsets(0,0).an_element()
{}
```

```
sage: Subsets(3,2).an_element()
{1, 3}
sage: Subsets([2,4,5],2).an_element()
{2, 5}
```

cardinality()

EXAMPLES:

```
sage: Subsets(Set([1,2,3]), 2).cardinality()
3
sage: Subsets([1,2,3,3], 2).cardinality()
3
sage: Subsets([1,2,3], 1).cardinality()
3
sage: Subsets([1,2,3], 3).cardinality()
1
sage: Subsets([1,2,3], 0).cardinality()
1
sage: Subsets([1,2,3], 4).cardinality()
0
sage: Subsets(3,2).cardinality()
3
sage: Subsets(3,4).cardinality()
0
```

first()

Returns the first subset of s of size k.

EXAMPLES:

```
sage: Subsets(Set([1,2,3]), 2).first()
{1, 2}
sage: Subsets([1,2,3,3], 2).first()
{1, 2}
sage: Subsets(3,2).first()
{1, 2}
sage: Subsets(3,4).first()
Traceback (most recent call last):
...
EmptySetError
```

last()

Returns the last subset of s of size k.

EXAMPLES:

```
sage: Subsets(Set([1,2,3]), 2).last()
{2, 3}
sage: Subsets([1,2,3,3], 2).last()
{2, 3}
sage: Subsets(3,2).last()
{2, 3}
sage: Subsets(3,4).last()
Traceback (most recent call last):
...
EmptySetError
```

random_element()

Return a random element of the class of subsets of `s` of size `k` (in other words, a random subset of `s` of

size k).

EXAMPLES:

```
sage: Subsets(3, 2).random_element()
{1, 2}
sage: Subsets(3,4).random_element()
Traceback (most recent call last):
...
EmptySetError
```

rank(*sub*)

Return the rank of sub as a subset of s of size k.

EXAMPLES:

```
sage: Subsets(3,2).rank([1,2])
0
sage: Subsets([2,3,4],2).rank([3,4])
2
sage: Subsets([2,3,4],2).rank([2])
Traceback (most recent call last):
...
ValueError: {2} is not a subset of length 2 of {2, 3, 4}
sage: Subsets([2,3,4],4).rank([2,3,4,5])
Traceback (most recent call last):
...
ValueError: {2, 3, 4, 5} is not a subset of length 4 of {2, 3, 4}
```

unrank(*r*)

Return the subset of s of size k that has rank r.

EXAMPLES:

```
sage: Subsets(3,2).unrank(0)
{1, 2}
sage: Subsets([2,4,5],2).unrank(0)
{2, 4}
sage: Subsets([1,2,8],3).unrank(42)
Traceback (most recent call last):
...
IndexError: index out of range
```

sage.combinat.subset.**dict_to_list**(*d*)

Return a list whose elements are the elements of i of d repeated with multiplicity d[i].

EXAMPLES:

```
sage: from sage.combinat.subset import dict_to_list
sage: dict_to_list({'a':1, 'b':3})
['a', 'b', 'b', 'b']
```

sage.combinat.subset.**list_to_dict**(*l*)

Return a dictionary whose keys are the elements of l and values are the multiplicity they appear in l.

EXAMPLES:

```
sage: from sage.combinat.subset import list_to_dict
sage: list_to_dict(['a', 'b', 'b', 'b'])
{'a': 1, 'b': 3}
```

5.1.304 Subsets satisfying a hereditary property

sage.combinat.subsets_hereditary.**subsets_with_hereditary_property**(*f*, *X*, *max_obstruction_size=None*, *ncpus=1*)

Return all subsets S of X such that $f(S)$ is true.

The boolean function f must be decreasing, i.e. $f(S) \Rightarrow f(S')$ if $S' \subseteq S$.

This function is implemented to call f as few times as possible. More precisely, f will be called on all sets S such that $f(S)$ is true, as well as on all inclusionwise minimal sets S such that $f(S)$ is false.

The problem that this function answers is also known as the learning problem on monotone boolean functions, or as computing the set of winning coalitions in a simple game.

INPUT:

- `f` – a boolean function which takes as input a list of elements from `X`.
- `X` – a list/iterable.
- `max_obstruction_size` (integer) – if you know that there is a k such that $f(S)$ is true if and only if $f(S')$ is true for all $S' \subseteq S$ with $S' \leq k$, set `max_obstruction_size=k`. It may dramatically decrease the number of calls to f. Set to `None` by default, meaning $k = |X|$.
- `ncpus` – number of cpus to use for this computation. Note that changing the value from 1 (default) to anything different *enables* parallel computations which can have a cost by itself, so it is not necessarily a good move. In some cases, however, it is a *great* move. Set to `None` to automatically detect and use the maximum number of cpus available.

Note: Parallel computations are performed through the `parallel()` decorator. See its documentation for more information, in particular with respect to the memory context.

EXAMPLE:

Sets whose elements all have the same remainder mod 2:

```
sage: from sage.combinat.subsets_hereditary import subsets_with_hereditary_property
sage: f = lambda x: (not x) or all(xx%2 == x[0]%2 for xx in x)
sage: list(subsets_with_hereditary_property(f,range(4)))
[[], [0], [1], [2], [3], [0, 2], [1, 3]]
```

Same, on two threads:

```
sage: sorted(list(subsets_with_hereditary_property(f,range(4),ncpus=2)))
[[], [0], [0, 2], [1], [1, 3], [2], [3]]
```

One can use this function to compute the independent sets of a graph. We know, however, that in this case the maximum obstructions are the edges, and have size 2. We can thus set `max_obstruction_size=2`, which reduces the number of calls to f from 91 to 56:

```
sage: num_calls=0
sage: g = graphs.PetersenGraph()
sage: def is_independent_set(S):
....:     global num_calls
....:     num_calls+=1
....:     return g.subgraph(S).size()==0
sage: l1=list(subsets_with_hereditary_property(is_independent_set,g.vertices()))
sage: num_calls
91
sage: num_calls=0
sage: l2=list(subsets_with_hereditary_property(is_independent_set,g.vertices(),max_obstruction_s
```

```
sage: num_calls
56
sage: l1==l2
True
```

TESTS:

```
sage: list(subsets_with_hereditary_property(lambda x:False,range(4)))
[]
sage: list(subsets_with_hereditary_property(lambda x:len(x)<1,range(4)))
[[]]
sage: print list(subsets_with_hereditary_property(lambda x:True,range(2)))
[[], [0], [1], [0, 1]]
```

5.1.305 Subsets whose elements satisfy a predicate pairwise

class sage.combinat.subsets_pairwise.**PairwiseCompatibleSubsets**(*ambient*, *predicate*, *maximal=False*, *element_class=<class 'sage.sets.set.Set_object_enumerated'>*)

Bases: sage.combinat.backtrack.SearchForest

The set of all subsets of `ambient` whose elements satisfy `predicate` pairwise

INPUT:

- `ambient` – a set (or iterable)
- `predicate` – a binary predicate

Assumptions: `predicate` is symmetric (`predicate(x,y) == predicate(y,x)`) and reflexive (`predicate(x,x) == True`).

Note: in fact, `predicate(x,x)` is never called.

Warning: The current name is suboptimal and is subject to change. Suggestions for a good name, and a good user entry point are welcome. Maybe `Subsets(..., independant = predicate)`.

EXAMPLES:

We construct the set of all subsets of $\{4,5,6,8,9\}$ whose elements are pairwise relatively prime:

```
sage: from sage.combinat.subsets_pairwise import PairwiseCompatibleSubsets
sage: def predicate(x,y): return gcd(x,y) == 1
sage: P = PairwiseCompatibleSubsets( [4,5,6,8,9], predicate); P
An enumerated set with a forest structure
sage: P.list()
[{}, {4}, {4, 5}, {9, 4, 5}, {9, 4}, {5}, {5, 6}, {8, 5}, {8, 9, 5}, {9, 5}, {6}, {8}, {8, 9}, {
sage: P.cardinality()
14
sage: P.category()
Category of finite enumerated sets
```

Here we consider only those subsets which are maximal for inclusion (not yet implemented):

```
sage: P = PairwiseCompatibleSubsets( [4,5,6,8,9], predicate, maximal = True); P
An enumerated set with a forest structure
```

```
sage: P.list()                    # todo: not implemented
[{9, 4, 5}, {5, 6}, {8, 9, 5}]
sage: P.cardinality()             # todo: not implemented
14
sage: P.category()
Category of finite enumerated sets
```

Algorithm

In the following, we order the elements of the ambient set by order of apparition. The elements of `self` are generated by organizing them in a search tree. Each node of this tree is of the form `(subset, rest)`, where:

- `subset` represents an element of `self`, represented by an increasing tuple
- `rest` is the set of all y's such that y appears after x in the ambient set and `predicate(x,y)` holds, represented by a decreasing tuple

The root of this tree is `( (), ambient )`. All the other elements are generated by recursive depth first search, which gives lexicographic order.

children (*subset_rest*)

Returns the children of a node in the tree.

TESTS:

```
sage: from sage.combinat.subsets_pairwise import PairwiseCompatibleSubsets
sage: def predicate(x,y): return gcd(x,y) == 1
sage: P = PairwiseCompatibleSubsets( [3,5,7,11,14], predicate); P
An enumerated set with a forest structure
sage: list(P.children( ((3,5), [14,11,7]) ))
[((3, 5, 7), (11,)), ((3, 5, 11), (14,)), ((3, 5, 14), ())]
```

post_process (*subset_rest*)

TESTS:

```
sage: from sage.combinat.subsets_pairwise import PairwiseCompatibleSubsets
sage: def predicate(x,y): return gcd(x,y) == 1
sage: P = PairwiseCompatibleSubsets( [4,5,6,8,9], predicate); P
An enumerated set with a forest structure
sage: P.post_process( ((4,5), (9)) )
{4, 5}
sage: P.post_process( ((4,5), ()) )
{4, 5}
```

5.1.306 Subwords

A subword of a word w is a word obtained by deleting the letters at some (non necessarily adjacent) positions in w. It is not to be confused with the notion of factor where one keeps adjacent positions in w. Sometimes it is useful to allow repeated uses of the same letter of w in a "generalized" subword. We call this a subword with repetitions.

For example:

- "bnjr" is a subword of the word "bonjour" but not a factor;
- "njo" is both a factor and a subword of the word "bonjour";
- "nr" is a subword of "bonjour";

- "rn" is not a subword of "bonjour";
- "nnu" is not a subword of "bonjour";
- "nnu" is a subword with repetitions of "bonjour";

A word can be given either as a string, as a list or as a tuple.

As repetition can occur in the initial word, the subwords of a given words is not a set in general but an enumerated multiset!

Todo

- implement subwords with repetitions
- implement the category of EnumeratedMultiset and inheritate from when needed (i.e. the initial word has repeated letters)

AUTHORS:

- Mike Hansen: initial version
- Florent Hivert (2009/02/06): doc improvements + new methods + bug fixes

`sage.combinat.subword.`**`Subwords`**(*w*, *k=None*, *element_constructor=None*)

Return the set of subwords of `w`.

INPUT:

- `w` – a word (can be a list, a string, a tuple or a word)
- `k` – an optional integer to specify the length of subwords
- `element_constructor` – an optional function that will be used to build the subwords

EXAMPLES:

```
sage: S = Subwords(['a','b','c']); S
Subwords of ['a', 'b', 'c']
sage: S.first()
[]
sage: S.last()
['a', 'b', 'c']
sage: S.list()
[[], ['a'], ['b'], ['c'], ['a', 'b'], ['a', 'c'], ['b', 'c'], ['a', 'b', 'c']]
```

The same example using string, tuple or a word:

```
sage: S = Subwords('abc'); S
Subwords of 'abc'
sage: S.list()
['', 'a', 'b', 'c', 'ab', 'ac', 'bc', 'abc']

sage: S = Subwords((1,2,3)); S
Subwords of (1, 2, 3)
sage: S.list()
[(), (1,), (2,), (3,), (1, 2), (1, 3), (2, 3), (1, 2, 3)]

sage: w = Word([1,2,3])
sage: S = Subwords(w); S
Subwords of word: 123
sage: S.list()
[word: , word: 1, word: 2, word: 3, word: 12, word: 13, word: 23, word: 123]
```

Using word with specified length:

```
sage: S = Subwords(['a','b','c'], 2); S
Subwords of ['a', 'b', 'c'] of length 2
sage: S.list()
[['a', 'b'], ['a', 'c'], ['b', 'c']]
```

An example that uses the `element_constructor` argument:

```
sage: p = Permutation([3,2,1])
sage: Subwords(p, element_constructor=tuple).list()
[(), (3,), (2,), (1,), (3, 2), (3, 1), (2, 1), (3, 2, 1)]
sage: Subwords(p, 2, element_constructor=tuple).list()
[(3, 2), (3, 1), (2, 1)]
```

class sage.combinat.subword.**Subwords_w**(*w*, *element_constructor*)

Bases: sage.structure.parent.Parent

Subwords of a given word.

cardinality()

EXAMPLES:

```
sage: Subwords([1,2,3]).cardinality()
8
```

first()

EXAMPLES:

```
sage: Subwords([1,2,3]).first()
[]
sage: Subwords((1,2,3)).first()
()
sage: Subwords('123').first()
''
```

last()

EXAMPLES:

```
sage: Subwords([1,2,3]).last()
[1, 2, 3]
sage: Subwords((1,2,3)).last()
(1, 2, 3)
sage: Subwords('123').last()
'123'
```

random_element()

Return a random subword with uniform law.

EXAMPLES:

```
sage: S1 = Subwords([1,2,3,2,1,3])
sage: S2 = Subwords([4,6,6,6,7,4,5,5])
sage: for i in xrange(100):
....:   w = S1.random_element()
....:   if w in S2:
....:       assert(w == [])
sage: for i in xrange(100):
....:   w = S2.random_element()
....:   if w in S1:
....:       assert(w == [])
```

class sage.combinat.subword.**Subwords_wk**(*w*, *k*, *element_constructor*)
Bases: sage.combinat.subword.Subwords_w

Subwords with fixed length of a given word.

cardinality()
Returns the number of subwords of w of length k.

EXAMPLES:

```
sage: Subwords([1,2,3], 2).cardinality()
3
```

first()
EXAMPLES:

```
sage: Subwords([1,2,3],2).first()
[1, 2]
sage: Subwords([1,2,3],0).first()
[]
sage: Subwords((1,2,3),2).first()
(1, 2)
sage: Subwords((1,2,3),0).first()
()
sage: Subwords('123',2).first()
'12'
sage: Subwords('123',0).first()
''
```

last()
EXAMPLES:

```
sage: Subwords([1,2,3],2).last()
[2, 3]
sage: Subwords([1,2,3],0).last()
[]
sage: Subwords((1,2,3),2).last()
(2, 3)
sage: Subwords((1,2,3),0).last()
()
sage: Subwords('123',2).last()
'23'
sage: Subwords('123',0).last()
''
```

TESTS:

```
sage: Subwords('123', 0).last()  # trac 10534
''
```

random_element()
Return a random subword of given length with uniform law.

EXAMPLES:

```
sage: S1 = Subwords([1,2,3,2,1],3)
sage: S2 = Subwords([4,4,5,5,4,5,4,4],3)
sage: for i in xrange(100):
....:   w = S1.random_element()
....:   if w in S2:
....:       assert(w == [])
sage: for i in xrange(100):
```

```
....:    w = S2.random_element()
....:    if w in S1:
....:        assert(w == [])
```

sage.combinat.subword.**smallest_positions**(*word*, *subword*, *pos=0*)

Return the smallest positions for which `subword` appears as a subword of `word`. If `pos` is specified, then it returns the positions of the first appearance of subword starting at `pos`.

If `subword` is not found in `word`, then return `False`.

EXAMPLES:

```
sage: sage.combinat.subword.smallest_positions([1,2,3,4], [2,4])
[1, 3]
sage: sage.combinat.subword.smallest_positions([1,2,3,4,4], [2,4])
[1, 3]
sage: sage.combinat.subword.smallest_positions([1,2,3,3,4,4], [3,4])
[2, 4]
sage: sage.combinat.subword.smallest_positions([1,2,3,3,4,4], [3,4],2)
[2, 4]
sage: sage.combinat.subword.smallest_positions([1,2,3,3,4,4], [3,4],3)
[3, 4]
sage: sage.combinat.subword.smallest_positions([1,2,3,4], [2,3])
[1, 2]
sage: sage.combinat.subword.smallest_positions([1,2,3,4], [5,5])
False
sage: sage.combinat.subword.smallest_positions([1,3,3,4,5],[3,5])
[1, 4]
sage: sage.combinat.subword.smallest_positions([1,3,3,5,4,5,3,5],[3,5,3])
[1, 3, 6]
sage: sage.combinat.subword.smallest_positions([1,3,3,5,4,5,3,5],[3,5,3],2)
[2, 3, 6]
sage: sage.combinat.subword.smallest_positions([1,2,3,4,3,4,4],[2,3,3,1])
False
sage: sage.combinat.subword.smallest_positions([1,3,3,5,4,5,3,5],[3,5,3],3)
False
```

TEST:

We check for trac ticket #5534:

```
sage: w = ["a", "b", "c", "d"]; ww = ["b", "d"]
sage: x = sage.combinat.subword.smallest_positions(w, ww); ww
['b', 'd']
```

5.1.307 Subword complex

Fix a Coxeter system (W, S). The subword complex $\mathcal{SC}(Q, w)$ associated to a word $Q \in S^*$ and an element $w \in W$ is the simplicial complex whose ground set is the set of positions in Q and whose facets are complements of sets of positions defining a reduced expression for w.

A subword complex is a shellable sphere if and only if the Demazure product of Q equals w, otherwise it is a shellable ball.

AUTHORS:

- Christian Stump: initial version
- Vincent Pilaud: greedy flip algorithm, minor improvements, documentation

REFERENCES:

class sage.combinat.subword_complex.**SubwordComplex**(Q, w, *algorithm='inductive'*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.homology.simplicial_complex.SimplicialComplex

Fix a Coxeter system (W, S). The subword complex $\mathcal{SC}(Q, w)$ associated to a word $Q \in S^*$ and an element $w \in W$ is the simplicial complex whose ground set is the set of positions in Q and whose facets are complements of sets of positions defining a reduced expression for w.

A subword complex is a shellable sphere if and only if the Demazure product of Q equals w, otherwise it is a shellable ball.

Warning: This implementation only works for groups build using CoxeterGroup, and does not work with groups build using WeylGroup.

EXAMPLES:

As an example, dual associahedra are subword complexes in type A_{n-1} given by the word $[1, \ldots, n, 1, \ldots, n, 1, \ldots, n-1, \ldots, 1, 2, 1]$ and the permutation w_0.

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w); SC
Subword complex of type ['A', 2] for Q = (1, 2, 1, 2, 1) and pi = [1, 2, 1]
sage: SC.facets()
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]
```

REFERENCES: [KnuMil], [PilStu]

Element

alias of SubwordComplexFacet

barycenter()

Return the barycenter of the brick polytope of self.

See also:

brick_polytope()

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2], base_ring=QQ)
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.barycenter()
(4/3, 8/3)
```

brick_fan()

Return the brick fan of self.

It is the normal fan of the brick polytope of self. It is formed by the cones generated by the weight configurations of the facets of self.

See also:

weight_cone

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2], base_ring=QQ)
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
```

```
sage: SC.brick_fan()
Rational polyhedral fan in 2-d lattice N
```

brick_polytope(*coefficients=None*)

Return the brick polytope of `self`.

This polytope is the convex hull of the brick vectors of `self`.

INPUT:

- coefficients – (optional) a list of coefficients used to scale the fundamental weights

See also:

`brick_vectors()`

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2], base_ring=QQ)
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)

sage: X = SC.brick_polytope(); X
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 5 vertices

sage: Y = SC.brick_polytope(coefficients=[1,2]); Y
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 5 vertices

sage: X == Y
False
```

brick_vectors(*coefficients=None*)

Return the list of all brick vectors of facets of `self`.

INPUT:

- coefficients – (optional) a list of coefficients used to scale the fundamental weights

See also:

`brick_vector`

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2], base_ring=QQ)
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.brick_vectors()
[(10/3, 14/3), (10/3, 2/3), (4/3, 14/3), (-2/3, 8/3), (-2/3, 2/3)]
sage: SC.brick_vectors(coefficients=(1,2))
[(14/3, 22/3), (14/3, 4/3), (8/3, 22/3), (-4/3, 10/3), (-4/3, 4/3)]
```

cartan_type()

Return the Cartan type of `self`.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.cartan_type()
['A', 2]
```

cover_relations(*label=False*)

Return the set of cover relations in the associated poset.

INPUT:

- label – boolean (default `False`) whether or not to label the cover relations by the position of flip

OUTPUT:

a list of pairs of facets

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2], base_ring=QQ)
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.cover_relations()
[((0, 1), (1, 2)),
 ((0, 1), (0, 4)),
 ((1, 2), (2, 3)),
 ((0, 4), (3, 4)),
 ((2, 3), (3, 4))]
```

dimension()

Return the dimension of `self`.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.dimension()
1
```

facets()

Return all facets of `self`.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.facets()
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]
```

greedy_facet(*side='positive'*)

Return the negative (or positive) greedy facet of `self`.

This is the lexicographically last (or first) facet of `self`.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.greedy_facet(side="positive")
(0, 1)
sage: SC.greedy_facet(side="negative")
(3, 4)
```

group()

Return the group associated to `self`.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2], base_ring=QQ)
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.group()
```

```
Finite Coxeter group over Rational Field with Coxeter matrix:
[1 3]
[3 1]
```

increasing_flip_graph(*label=True*)

Return the increasing flip graph of the subword complex.

OUTPUT:

a directed graph

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.increasing_flip_graph()
Digraph on 5 vertices
```

increasing_flip_poset()

Return the increasing flip poset of the subword complex.

OUTPUT:

a poset

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.increasing_flip_poset()
Finite poset containing 5 elements
```

interval(*I*, *J*)

Return the interval [I,J] in the increasing flip graph subword complex.

INPUT:

- I, J – two facets

OUTPUT

a set of facets

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: F = SC([1,2])
sage: SC.interval(F, F)
{(1, 2)}
```

is_ball()

Return `True` if the subword complex `self` is a ball.

This is the case if and only if it is not a sphere.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], index_set=[1,2,3])
sage: w = W.from_reduced_word([2,3,2])
sage: SC = SubwordComplex([3,2,3,2,3], w)
sage: SC.is_ball()
False
```

```
sage: SC = SubwordComplex([3,2,1,3,2,3], w)
sage: SC.is_ball()
True
```

is_double_root_free()

Return `True` if `self` is double-root-free.

This means that the root configurations of all facets do not contain a root twice.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.is_double_root_free()
True

sage: SC = SubwordComplex([1,1,2,2,1,1], w)
sage: SC.is_double_root_free()
True

sage: SC = SubwordComplex([1,2,1,2,1,2], w)
sage: SC.is_double_root_free()
False
```

is_pure()

Return `True` since all subword complexes are pure.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], index_set=[1,2,3])
sage: w = W.from_reduced_word([2,3,2])
sage: SC = SubwordComplex([3,2,3,2,3], w)
sage: SC.is_pure()
True
```

is_root_independent()

Return `True` if `self` is root-independent.

This means that the root configuration of any (or equivalently all) facets is linearly independent.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.is_root_independent()
True

sage: SC = SubwordComplex([1,2,1,2,1,2], W.w0)
sage: SC.is_root_independent()
False
```

is_sphere()

Return `True` if the subword complex `self` is a sphere.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], index_set=[1,2,3])
sage: w = W.from_reduced_word([2,3,2])
sage: SC = SubwordComplex([3,2,3,2,3], w)
sage: SC.is_sphere()
True
```

```
sage: SC = SubwordComplex([3,2,1,3,2,3], w)
sage: SC.is_sphere()
False
```

kappa_preimages()

Return a dictionary containing facets of `self` as keys, and list of elements of `self.group()` as values.

See also:

`kappa_preimage`

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: kappa = SC.kappa_preimages()
sage: for F in SC: print F, [w.reduced_word() for w in kappa[F]]
(0, 1) [[]]
(0, 4) [[2], [2, 1]]
(1, 2) [[1]]
(2, 3) [[1, 2]]
(3, 4) [[1, 2, 1]]
```

list()

Return the list of facets of `self`.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: list(SC)
[(0, 1), (0, 4), (1, 2), (2, 3), (3, 4)]
```

minkowski_summand(i)

Return the i th Minkowski summand of `self`.

INPUT:

i – an integer defining a position in the word Q

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2], base_ring=QQ)
sage: SC = SubwordComplex([1,2,1,2,1], W.w0)
sage: SC.minkowski_summand(1)
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex
```

pi()

Return the element in the Coxeter group associated to `self`.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.pi().reduced_word()
[1, 2, 1]
```

word()
Return the word in the simple generators associated to `self`.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: SC.word()
(1, 2, 1, 2, 1)
```

class sage.combinat.subword_complex.**SubwordComplexFacet**(*parent*, *positions*, *facet_test=True*)
Bases: sage.homology.simplicial_complex.Simplex, sage.structure.element.Element

A facet of a subword complex.

Facets of the subword complex $\mathcal{SC}(Q, w)$ are complements of sets of positions in Q defining a reduced expression for w.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1], w)
sage: F = SC[0]
sage: F
(0, 1)
sage: type(F)
<class 'sage.combinat.subword_complex.SubwordComplex_with_category.element_class'>
```

brick_vector(*coefficients=None*)
Return the brick vector of `self`.

This is the sum of the weight vectors in the extended weight configuration.

INPUT:

- coefficients – (optional) a list of coefficients used to scale the fundamental weights

See also:

`extended_weight_configuration()`

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.extended_weight_configuration()
[(4/3, 2/3), (2/3, 4/3), (-2/3, 2/3), (2/3, 4/3), (-2/3, 2/3)]
sage: F.brick_vector()
(4/3, 14/3)
sage: F.brick_vector(coefficients=[1,2])
(8/3, 22/3)
```

extended_root_configuration()
Return the extended root configuration of `self`.

Let $Q = q_1 \dots q_m \in S^*$ and $w \in W$. The extended root configuration of a facet I of $\mathcal{SC}(Q, w)$ is the sequence $\mathsf{r}(I, 1), \dots, \mathsf{r}(I, m)$ of roots defined by $\mathsf{r}(I, k) = \Pi Q_{[k-1] \setminus I}(\alpha_{q_k})$, where $\Pi Q_{[k-1] \setminus I}$ is the

product of the simple reflections q_i for $i \in [k-1] \setminus I$ in this order.

The extended root configuration is used to perform flips efficiently.

See also:

`flip()`

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.extended_root_configuration()
[(1, 0), (1, 1), (-1, 0), (1, 1), (0, 1)]
```

extended_weight_configuration(*coefficients=None*)

Return the extended weight configuration of `self`.

Let $Q = q_1 \dots q_m \in S^*$ and $w \in W$. The extended weight configuration of a facet I of $\mathcal{SC}(Q, w)$ is the sequence $\mathsf{w}(I, 1), \dots, \mathsf{w}(I, m)$ of weights defined by $\mathsf{w}(I, k) = \Pi Q_{[k-1] \setminus I}(\omega_{q_k})$, where $\Pi Q_{[k-1] \setminus I}$ is the product of the simple reflections q_i for $i \in [k-1] \setminus I$ in this order.

The extended weight configuration is used to compute the brick vector.

INPUT:

- coefficients – (optional) a list of coefficients used to scale the fundamental weights

See also:

`brick_vector()`

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2], base_ring=QQ)
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2])
sage: F.extended_weight_configuration()
[(4/3, 2/3), (2/3, 4/3), (-2/3, 2/3), (2/3, 4/3), (-2/3, 2/3)]
sage: F.extended_weight_configuration(coefficients=(1,2))
[(4/3, 2/3), (4/3, 8/3), (-2/3, 2/3), (4/3, 8/3), (-2/3, 2/3)]
```

flip(*i*, *return_position=False*)

Return the facet obtained after flipping position `i` in `self`.

INPUT:

- `i` – position in the word Q (integer).
- `return_position` – boolean (default: `False`) tells whether the new position should be returned as well.

OUTPUT:

- The new subword complex facet.
- The new position if `return_position` is `True`.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.flip(1)
(2, 3)
sage: F.flip(1, return_position=True)
((2, 3), 3)
```

is_vertex()

Return `True` if `self` is a vertex of the brick polytope of `self.parent`.

A facet is a vertex of the brick polytope if its root cone is pointed. Note that this property is always satisfied for root-independent subword complexes.

See also:

`root_cone()`

EXAMPLES:

```
sage: W = CoxeterGroup(['A',1], index_set=[1])
sage: w = W.from_reduced_word([1])
sage: SC = SubwordComplex([1,1,1],w)
sage: F = SC([0,1]); F.is_vertex()
True
sage: F = SC([0,2]); F.is_vertex()
False

sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1,2,1],w)
sage: F = SC([0,1,2,3]); F.is_vertex()
True
sage: F = SC([0,1,2,6]); F.is_vertex()
False
```

kappa_preimage()

Return the fiber of `self` under the κ map.

The κ map sends an element $w \in W$ to the unique facet of $I \in \mathcal{SC}(Q, w)$ such that the cone generated by $w(\Phi^+)$ is contained in the cone generated by the root configuration of I.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2],index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)

sage: F = SC([1,2]); F
(1, 2)
sage: F.kappa_preimage()
[
[-1  1]
[ 0  1]
]

sage: F = SC([0,4]); F
(0, 4)
sage: F.kappa_preimage()
```

```
[
[ 1  0]  [-1  1]
[ 1 -1], [-1  0]
]
```

plot(*list_colors=[ ]*, *labels=[ ]*, *thickness=3*, *fontsize=14*, *shift=(0, 0)*, *compact=False*, *roots=True*, ***args*)

In type A or B, plot a pseudoline arrangement representing the facet `self`.

Pseudoline arrangements are graphical representations of facets of types A or B subword complexes.

INPUT:

- `list_colors` – list (default: `[]`) to change the colors of the pseudolines.
- `labels` – list (default: `[]`) to change the labels of the pseudolines.
- `thickness` – integer (default: `3`) for the thickness of the pseudolines.
- `fontsize` – integer (default: `14`) for the size of the font used for labels.
- `shift` – couple of coordinates (default: `(0,0)`) to change the origin.
- `compact` – boolean (default: `False`) to require a more compact representation.
- `roots` – boolean (default: `True`) to print the extended root configuration.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F.plot()
Graphics object consisting of 26 graphics primitives

sage: W = CoxeterGroup(['B',3])
sage: c = W.from_reduced_word([1,2,3])
sage: Q = c.reduced_word()*2 + W.w0.coxeter_sorting_word(c)
sage: SC = SubwordComplex(Q, W.w0)
sage: F = SC[15]; F.plot()
Graphics object consisting of 53 graphics primitives
```

TESTS:

```
sage: W = CoxeterGroup(['D',4])
sage: c = W.from_reduced_word([1,2,3,4])
sage: Q = c.reduced_word() + W.w0.coxeter_sorting_word(c)
sage: SC = SubwordComplex(Q, W.w0)
sage: F = SC[1]; F.plot()
Traceback (most recent call last):
...
ValueError: Plotting is currently only implemented for irreducibles types A, B, and C.

sage: W = CoxeterGroup(CoxeterMatrix((['A',2],['A',2])))
sage: c = W.from_reduced_word([1,2,3,4])
sage: Q = c.reduced_word() + W.w0.coxeter_sorting_word(c)
sage: SC = SubwordComplex(Q, W.w0)
sage: F = SC[1]; F.plot()
Traceback (most recent call last):
...
ValueError: Plotting is currently only implemented for irreducibles types A, B, and C.
```

REFERENCES: [PilStu]

root_cone()

Return the polyhedral cone generated by the root configuration of `self`.

See also:

`root_configuration()`

EXAMPLES:

```
sage: W = CoxeterGroup(['A',1], index_set=[1])
sage: w = W.from_reduced_word([1])
sage: SC = SubwordComplex([1,1,1],w)
sage: F = SC([0,2]); F.root_cone()
1-d cone in 1-d lattice N
```

root_configuration()

Return the root configuration of `self`.

Let $Q = q_1 \dots q_m \in S^*$ and $w \in W$. The root configuration of a facet $I = [i_1, \dots, i_n]$ of $\mathcal{SC}(Q, w)$ is the sequence $\mathsf{r}(I, i_1), \dots, \mathsf{r}(I, i_n)$ of roots defined by $\mathsf{r}(I, k) = \Pi Q_{[k-1] \setminus I}(\alpha_{q_k})$, where $\Pi Q_{[k-1] \setminus I}$ is the product of the simple reflections q_i for $i \in [k-1] \setminus I$ in this order.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.root_configuration()
[(1, 1), (-1, 0)]
```

show(**kwds*, ***args*)

Show the facet `self`.

See also:

`plot()`

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F.show()
```

upper_root_configuration()

Return the positive roots of the root configuration of `self`.

EXAMPLE:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.root_configuration()
[(1, 1), (-1, 0)]
sage: F.upper_root_configuration()
[(1, 0)]
```

weight_cone()
Return the polyhedral cone generated by the weight configuration of `self`.

See also:

`weight_configuration()`

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: WC = F.weight_cone(); WC
2-d cone in 2-d lattice N
sage: WC.rays()
N( 1, 2),
N(-1, 1)
in 2-d lattice N
```

weight_configuration()
Return the weight configuration of `self`.

Let $Q = q_1 \dots q_m \in S^*$ and $w \in W$. The weight configuration of a facet $I = [i_1, \dots, i_n]$ of $\mathcal{SC}(Q, w)$ is the sequence $\mathsf{w}(I, i_1), \dots, \mathsf{w}(I, i_n)$ of weights defined by $\mathsf{w}(I, k) = \Pi Q_{[k-1] \setminus I}(\omega_{q_k})$, where $\Pi Q_{[k-1] \setminus I}$ is the product of the simple reflections q_i for $i \in [k-1] \setminus I$ in this order.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',2], index_set=[1,2])
sage: w = W.from_reduced_word([1,2,1])
sage: SC = SubwordComplex([1,2,1,2,1],w)
sage: F = SC([1,2]); F
(1, 2)
sage: F.weight_configuration()
[(2/3, 4/3), (-2/3, 2/3)]
```

5.1.308 Symmetric Group Algebra

sage.combinat.symmetric_group_algebra.**HeckeAlgebraSymmetricGroupT**(*R*, *n*, *q=None*)
Return the Hecke algebra of the symmetric group S_n on the T-basis with quantum parameter `q` over the ring R.

If R is a commutative ring and q is an invertible element of R, and if n is a nonnegative integer, then the Hecke algebra of the symmetric group S_n over R with quantum parameter q is defined as the algebra generated by the generators $T_1, T_2, \dots, T_{n-1}$ with relations

$$T_i T_{i+1} T_i = T_{i+1} T_i T_{i+1}$$

for all $i < n - 1$ ("braid relations"),

$$T_i T_j = T_j T_i$$

for all i and j such that $|i - j| > 1$ ("locality relations"), and

$$T_i^2 = q + (q - 1) T_i$$

for all i (the "quadratic relations", also known in the form $(T_i + 1)(T_i - q) = 0$). (This is only one of several existing definitions in literature, not all of which are fully equivalent. We are following the conventions

of [GS93].) For any permutation $w \in S_n$, we can define an element T_w of this Hecke algebra by setting $T_w = T_{i_1} T_{i_2} \cdots T_{i_k}$, where $w = s_{i_1} s_{i_2} \cdots s_{i_k}$ is a reduced word for w (with s_i meaning the transposition $(i, i+1)$, and the product of permutations being evaluated by first applying s_{i_k}, then $s_{i_{k-1}}$, etc.). This element is independent of the choice of the reduced decomposition, and can be computed in Sage by calling `H[w]` where `H` is the Hecke algebra and `w` is the permutation.

The Hecke algebra of the symmetric group S_n with quantum parameter q over R can be seen as a deformation of the group algebra RS_n; indeed, it becomes RS_n when $q = 1$.

Warning: The multiplication on the Hecke algebra of the symmetric group does *not* follow the global option `mult` of the `Permutations` class (see `global_options()`). It is always as defined above. It does not match the default option (`mult=l2r`) of the symmetric group algebra!

REFERENCES:

EXAMPLES:

```
sage: HeckeAlgebraSymmetricGroupT(QQ, 3)
Hecke algebra of the symmetric group of order 3 on the T basis over Univariate Polynomial Ring i

sage: HeckeAlgebraSymmetricGroupT(QQ, 3, 2)
Hecke algebra of the symmetric group of order 3 with q=2 on the T basis over Rational Field
```

The multiplication on the Hecke algebra follows a different convention than the one on the symmetric group algebra does by default:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ, 3)
sage: H3([1,3,2]) * H3([2,1,3])
T[3, 1, 2]
sage: S3 = SymmetricGroupAlgebra(QQ, 3)
sage: S3([1,3,2]) * S3([2,1,3])
[2, 3, 1]

sage: TestSuite(H3).run()
```

class sage.combinat.symmetric_group_algebra.**HeckeAlgebraSymmetricGroup_generic**(*R*, *n*, *q=None*)

Bases: sage.combinat.combinatorial_algebra.CombinatorialAlgebra

TESTS:

```
sage: HeckeAlgebraSymmetricGroupT(QQ, 3)
Hecke algebra of the symmetric group of order 3 on the T basis over Univariate Polynomial Ring i

sage: HeckeAlgebraSymmetricGroupT(QQ, 3, q=1)
Hecke algebra of the symmetric group of order 3 with q=1 on the T basis over Rational Field
```

q()

EXAMPLES:

```
sage: HeckeAlgebraSymmetricGroupT(QQ, 3).q()
q
sage: HeckeAlgebraSymmetricGroupT(QQ, 3, 2).q()
2
```

class sage.combinat.symmetric_group_algebra.**HeckeAlgebraSymmetricGroup_t**(*R*, *n*, *q=None*)

Bases: sage.combinat.symmetric_group_algebra.HeckeAlgebraSymmetricGroup_generic

TESTS:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ, 3)
sage: H3 == loads(dumps(H3))
True
```

algebra_generators()

Return the generators of the algebra.

EXAMPLES:

```
sage: HeckeAlgebraSymmetricGroupT(QQ,3).algebra_generators()
[T[2, 1, 3], T[1, 3, 2]]
```

jucys_murphy(k)

Return the Jucys-Murphy element J_k of the Hecke algebra.

These Jucys-Murphy elements are defined by

$$J_k = (T_{k-1}T_{k-2}\cdots T_1)(T_1T_2\cdots T_{k-1}).$$

More explicitly,

$$J_k = q^{k-1} + \sum_{l=1}^{k-1}(q^l - q^{l-1})T_{(l,k)}.$$

For generic q, the J_k generate a maximal commutative sub-algebra of the Hecke algebra.

Warning: The specialization $q = 1$ does *not* map these elements J_k to the Young-Jucys-Murphy elements of the group algebra RS_n. (Instead, it maps the "reduced" Jucys-Murphy elements $(J_k - q^{k-1})/(q-1)$ to the Young-Jucys-Murphy elements of RS_n.)

EXAMPLES:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ,3)
sage: j2 = H3.jucys_murphy(2); j2
q*T[1, 2, 3] + (q-1)*T[2, 1, 3]
sage: j3 = H3.jucys_murphy(3); j3
q^2*T[1, 2, 3] + (q^2-q)*T[1, 3, 2] + (q-1)*T[3, 2, 1]
sage: j2*j3 == j3*j2
True
sage: j0 = H3.jucys_murphy(1); j0 == H3.one()
True
sage: H3.jucys_murphy(0)
Traceback (most recent call last):
...
ValueError: k (= 0) must be between 1 and n (= 3)
```

t(i)

Return the element T_i of the Hecke algebra `self`.

EXAMPLES:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ,3)
sage: H3.t(1)
T[2, 1, 3]
sage: H3.t(2)
T[1, 3, 2]
sage: H3.t(0)
Traceback (most recent call last):
```

```
...
ValueError: i (= 0) must be between 1 and n-1 (= 2)
```

t_action(*a*, *i*)

Return the product $T_i \cdot a$.

EXAMPLES:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ, 3)
sage: a = H3([2,1,3])+2*H3([1,2,3])
sage: H3.t_action(a, 1)
q*T[1, 2, 3] + (q+1)*T[2, 1, 3]
sage: H3.t(1)*a
q*T[1, 2, 3] + (q+1)*T[2, 1, 3]
```

t_action_on_basis(*perm*, *i*)

Return the product $T_i \cdot T_{perm}$, where `perm` is a permutation in the symmetric group S_n.

EXAMPLES:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ, 3)
sage: H3.t_action_on_basis(Permutation([2,1,3]), 1)
q*T[1, 2, 3] + (q-1)*T[2, 1, 3]
sage: H3.t_action_on_basis(Permutation([1,2,3]), 1)
T[2, 1, 3]
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ, 3, 1)
sage: H3.t_action_on_basis(Permutation([2,1,3]), 1)
T[1, 2, 3]
sage: H3.t_action_on_basis(Permutation([1,3,2]), 2)
T[1, 2, 3]
```

sage.combinat.symmetric_group_algebra.**SymmetricGroupAlgebra**(*R*, *W*, *category=None*)

Return the symmetric group algebra of order `W` over the ring `R`.

INPUT:

- `W` – a symmetric group; alternatively an integer n can be provided, as shorthand for `Permutations(n)`.
- `R` – a base ring
- `category` – a category (default: the category of `W`)

This supports several implementations of the symmetric group. At this point this has been tested with `W=Permutations(n)` and `W=SymmetricGroup(n)`.

Warning: Some features are failing in the latter case, in particular if the domain of the symmetric group is not $1, \ldots, n$.

Note: The brave can also try setting `W=WeylGroup(['A',n-1])`, but little support for this currently exists.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3); QS3
Symmetric group algebra of order 3 over Rational Field
sage: QS3(1)
[1, 2, 3]
sage: QS3(2)
2*[1, 2, 3]
sage: basis = [QS3(p) for p in Permutations(3)]
```

```
sage: a = sum(basis); a
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
sage: a^2
6*[1, 2, 3] + 6*[1, 3, 2] + 6*[2, 1, 3] + 6*[2, 3, 1] + 6*[3, 1, 2] + 6*[3, 2, 1]
sage: a^2 == 6*a
True
sage: b = QS3([3, 1, 2])
sage: b
[3, 1, 2]
sage: b*a
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
sage: b*a == a
True
```

We now construct the symmetric group algebra by providing explicitly the underlying group:

```
sage: SGA = SymmetricGroupAlgebra(QQ, Permutations(4)); SGA
Symmetric group algebra of order 4 over Rational Field
sage: SGA.group()
Standard permutations of 4
sage: SGA.an_element()
[1, 2, 3, 4] + 2*[1, 2, 4, 3] + 3*[1, 3, 2, 4] + [4, 1, 2, 3]

sage: SGA = SymmetricGroupAlgebra(QQ, SymmetricGroup(4)); SGA
Symmetric group algebra of order 4 over Rational Field
sage: SGA.group()
Symmetric group of order 4! as a permutation group
sage: SGA.an_element()
() + 2*(1,2) + 4*(1,2,3,4)

sage: SGA = SymmetricGroupAlgebra(QQ, WeylGroup(["A",3], prefix='s')); SGA
Symmetric group algebra of order 4 over Rational Field
sage: SGA.group()
Weyl Group of type ['A', 3] (as a matrix group acting on the ambient space)
sage: SGA.an_element()
2*s1*s2*s3*s2*s1 + 3*s1*s2*s3*s1 + s1*s2*s3 + 1
```

The preferred way to construct the symmetric group algebra is to go through the usual `algebra` method:

```
sage: SGA = Permutations(3).algebra(QQ); SGA
Symmetric group algebra of order 3 over Rational Field
sage: SGA.group()
Standard permutations of 3

sage: SGA = SymmetricGroup(3).algebra(QQ); SGA
Symmetric group algebra of order 3 over Rational Field
sage: SGA.group()
Symmetric group of order 3! as a permutation group
```

The canonical embedding from the symmetric group algebra of order n to the symmetric group algebra of order $p > n$ is available as a coercion:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: QS4 = SymmetricGroupAlgebra(QQ, 4)
sage: QS4.coerce_map_from(QS3)
Generic morphism:
  From: Symmetric group algebra of order 3 over Rational Field
  To:   Symmetric group algebra of order 4 over Rational Field
```

```
sage: x3  = QS3([3,1,2]) + 2 * QS3([2,3,1]); x3
2*[2, 3, 1] + [3, 1, 2]
sage: QS4(x3)
2*[2, 3, 1, 4] + [3, 1, 2, 4]
```

This allows for mixed expressions:

```
sage: x4  = 3*QS4([3, 1, 4, 2])
sage: x3 + x4
2*[2, 3, 1, 4] + [3, 1, 2, 4] + 3*[3, 1, 4, 2]

sage: QS0 = SymmetricGroupAlgebra(QQ, 0)
sage: QS1 = SymmetricGroupAlgebra(QQ, 1)
sage: x0 = QS0([])
sage: x1 = QS1([1])
sage: x0 * x1
[1]
sage: x3 - (2*x0 + x1) - x4
-3*[1, 2, 3, 4] + 2*[2, 3, 1, 4] + [3, 1, 2, 4] - 3*[3, 1, 4, 2]
```

Caveat: to achieve this, constructing `SymmetricGroupAlgebra(QQ, 10)` currently triggers the construction of all symmetric group algebras of smaller order. Is this a feature we really want to have?

Warning: The semantics of multiplication in symmetric group algebras with index set `Permutations(n)` is determined by the order in which permutations are multiplied, which currently defaults to "in such a way that multiplication is associative with permutations acting on integers from the right", but can be changed to the opposite order at runtime by setting the global variable `Permutations.global_options['mult']` (see `sage.combinat.permutation.Permutations.global_options()`). On the other hand, the semantics of multiplication in symmetric group algebras with index set `SymmetricGroup(n)` does not depend on this global variable. (This has the awkward consequence that the coercions between these two sorts of symmetric group algebras do not respect multiplication when this global variable is set to `'r2l'`.) In view of this, it is recommended that code not rely on the usual multiplication function, but rather use the methods `left_action_product()` and `right_action_product()` for multiplying permutations (these methods don't depend on the setting). See trac ticket #14885 for more information.

We conclude by constructing the algebra of the symmetric group as a monoid algebra:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3, category=Monoids())
sage: QS3.category()
Category of finite dimensional monoid algebras over Rational Field
sage: TestSuite(QS3).run()
```

TESTS:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: TestSuite(QS3).run()

sage: QS3.group()
Standard permutations of 3

sage: QS3.one_basis()
[1, 2, 3]

sage: p1 = Permutation([1,2,3])
sage: p2 = Permutation([2,1,3])
sage: QS3.product_on_basis(p1,p2)
[2, 1, 3]
```

```
sage: W = WeylGroup(["A",3])
sage: SGA = SymmetricGroupAlgebra(QQ, W)
sage: SGA.group() is W
True
sage: TestSuite(SGA).run()

sage: SG = SymmetricGroupAlgebra(ZZ, 3)
sage: SG.group().conjugacy_classes_representatives()
[[1, 2, 3], [2, 1, 3], [2, 3, 1]]

sage: SGg = SymmetricGroup(3).algebra(ZZ)
sage: SGg.group().conjugacy_classes_representatives()
[(), (1,2), (1,2,3)]
```

class sage.combinat.symmetric_group_algebra.**SymmetricGroupAlgebra_n**(*R*, *W*, *category*)

Bases: sage.combinat.free_module.CombinatorialFreeModule

TESTS:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: TestSuite(QS3).run()

sage: QS3 in GroupAlgebras(QQ)
True
sage: QS3 in FiniteDimensionalAlgebrasWithBasis(QQ)
True
```

Check that trac ticket #16926 works:

```
sage: S = SymmetricGroup(4)
sage: SGA = S.algebra(QQ)
sage: TestSuite(SGA).run()
```

Checking that coercion works between equivalent indexing sets:

```
sage: G = SymmetricGroup(4).algebra(QQ)
sage: S = SymmetricGroupAlgebra(QQ,4)
sage: S(G.an_element())
[1, 2, 3, 4] + 2*[2, 1, 3, 4] + 4*[2, 3, 4, 1]
sage: G(S.an_element())
() + 2*(3,4) + 3*(2,3) + (1,4,3,2)
```

algebra_generators()

Return generators of this group algebra (as algebra) as a list of permutations.

The generators used for the group algebra of S_n are the transposition $(2, 1)$ and the n-cycle $(1, 2, \ldots, n)$, unless $n \leq 1$ (in which case no generators are needed).

EXAMPLES:

```
sage: SymmetricGroupAlgebra(ZZ,5).algebra_generators()
Family ([2, 1, 3, 4, 5], [2, 3, 4, 5, 1])

sage: SymmetricGroupAlgebra(QQ,0).algebra_generators()
Family ()

sage: SymmetricGroupAlgebra(QQ,1).algebra_generators()
Family ()
```

TESTS:

Check that trac ticket #15309 is fixed:

```
sage: S3 = SymmetricGroupAlgebra(QQ, 3)
sage: S3.algebra_generators()
Family ([2, 1, 3], [2, 3, 1])
sage: C = CombinatorialFreeModule(ZZ, ZZ)
sage: M = C.module_morphism(lambda x: S3.zero(), codomain=S3)
sage: M.register_as_coercion()
```

antipode(x)

Return the image of the element `x` of `self` under the antipode of the Hopf algebra `self` (where the comultiplication is the usual one on a group algebra).

Explicitly, this is obtained by replacing each permutation σ by σ^{-1} in `x` while keeping all coefficients as they are.

EXAMPLES:

```
sage: QS4 = SymmetricGroupAlgebra(QQ, 4)
sage: QS4.antipode(2 * QS4([1, 3, 4, 2]) - 1/2 * QS4([1, 4, 2, 3]))
-1/2*[1, 3, 4, 2] + 2*[1, 4, 2, 3]
sage: all( QS4.antipode(QS4(p)) == QS4(p.inverse())
....:      for p in Permutations(4) )
True

sage: ZS3 = SymmetricGroupAlgebra(ZZ, 3)
sage: ZS3.antipode(ZS3.zero())
0
sage: ZS3.antipode(-ZS3(Permutation([2, 3, 1])))
-[3, 1, 2]
```

binary_unshuffle_sum(k)

Return the k-th binary unshuffle sum in the group algebra `self`.

The k-th binary unshuffle sum in the symmetric group algebra RS_n over a ring R is defined as the sum of all permutations $\sigma \in S_n$ satisfying $\sigma(1) < \sigma(2) < \cdots < \sigma(k)$ and $\sigma(k+1) < \sigma(k+2) < \cdots < \sigma(n)$.

This element has the property that, if it is denoted by t_k, and if the k-th semi-RSW element (see `semi_rsw_element()`) is denoted by s_k, then $s_k S(t_k)$ and $t_k S(s_k)$ both equal the k-th Reiner-Saliola-Welker shuffling element of RS_n (see `rsw_shuffling_element()`).

The k-th binary unshuffle sum is the image of the complete non-commutative symmetric function $S^{(k,n-k)}$ in the ring of non-commutative symmetric functions under the canonical projection on the symmetric group algebra (through the descent algebra).

EXAMPLES:

The binary unshuffle sums on $\mathbf{Q}S_3$:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: QS3.binary_unshuffle_sum(0)
[1, 2, 3]
sage: QS3.binary_unshuffle_sum(1)
[1, 2, 3] + [2, 1, 3] + [3, 1, 2]
sage: QS3.binary_unshuffle_sum(2)
[1, 2, 3] + [1, 3, 2] + [2, 3, 1]
sage: QS3.binary_unshuffle_sum(3)
[1, 2, 3]
sage: QS3.binary_unshuffle_sum(4)
0
```

Let us check the relation with the k-th Reiner-Saliola-Welker shuffling element stated in the docstring:

```
sage: def test_rsw(n):
....:     ZSn = SymmetricGroupAlgebra(ZZ, n)
....:     for k in range(1, n):
....:         a = ZSn.semi_rsw_element(k)
....:         b = ZSn.binary_unshuffle_sum(k)
....:         c = ZSn.left_action_product(a, ZSn.antipode(b))
....:         d = ZSn.left_action_product(b, ZSn.antipode(a))
....:         e = ZSn.rsw_shuffling_element(k)
....:         if c != e or d != e:
....:             return False
....:     return True
sage: test_rsw(3)
True
sage: test_rsw(4)  # long time
True
sage: test_rsw(5)  # long time
True
```

Let us also check the statement about the complete non-commutative symmetric function:

```
sage: def test_rsw_ncsf(n):
....:     ZSn = SymmetricGroupAlgebra(ZZ, n)
....:     NSym = NonCommutativeSymmetricFunctions(ZZ)
....:     S = NSym.S()
....:     for k in range(1, n):
....:         a = S(Composition([k, n-k])).to_symmetric_group_algebra()
....:         if a != ZSn.binary_unshuffle_sum(k):
....:             return False
....:     return True
sage: test_rsw_ncsf(3)
True
sage: test_rsw_ncsf(4)
True
sage: test_rsw_ncsf(5)  # long time
True
```

canonical_embedding(*other*)

Return the canonical coercion of `self` into a symmetric group algebra `other`.

INPUT:

- `other` – a symmetric group algebra with order p satisfying $p \geq n$, where n is the order of `self`, over a ground ring into which the ground ring of `self` coerces.

EXAMPLES:

```
sage: QS2 = SymmetricGroupAlgebra(QQ, 2)
sage: QS4 = SymmetricGroupAlgebra(QQ, 4)
sage: phi = QS2.canonical_embedding(QS4); phi
Generic morphism:
  From: Symmetric group algebra of order 2 over Rational Field
  To:   Symmetric group algebra of order 4 over Rational Field

sage: x = QS2([2,1]) + 2 * QS2([1,2])
sage: phi(x)
2*[1, 2, 3, 4] + [2, 1, 3, 4]

sage: loads(dumps(phi))
Generic morphism:
```

```
  From: Symmetric group algebra of order 2 over Rational Field
  To:   Symmetric group algebra of order 4 over Rational Field

sage: ZS2 = SymmetricGroupAlgebra(ZZ, 2)
sage: phi = ZS2.canonical_embedding(QS4); phi
Generic morphism:
  From: Symmetric group algebra of order 2 over Integer Ring
  To:   Symmetric group algebra of order 4 over Rational Field

sage: phi = ZS2.canonical_embedding(QS2); phi
Generic morphism:
  From: Symmetric group algebra of order 2 over Integer Ring
  To:   Symmetric group algebra of order 2 over Rational Field

sage: QS4.canonical_embedding(QS2)
Traceback (most recent call last):
...
ValueError: There is no canonical embedding from Symmetric group
 algebra of order 2 over Rational Field to Symmetric group
 algebra of order 4 over Rational Field

sage: QS4g = SymmetricGroup(4).algebra(QQ)
sage: QS4.canonical_embedding(QS4g)(QS4([1,3,2,4]))
(2,3)
sage: QS4g.canonical_embedding(QS4)(QS4g((2,3)))
[1, 3, 2, 4]
sage: ZS2.canonical_embedding(QS4g)(ZS2([2,1]))
(1,2)
sage: ZS2g = SymmetricGroup(2).algebra(ZZ)
sage: ZS2g.canonical_embedding(QS4)(ZS2g((1,2)))
[2, 1, 3, 4]
```

cpi(*p*)

Return the centrally primitive idempotent for the symmetric group of order n corresponding to the irreducible representation indexed by the partition `p`.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ,3)
sage: QS3.cpi([2,1])
2/3*[1, 2, 3] - 1/3*[2, 3, 1] - 1/3*[3, 1, 2]
sage: QS3.cpi([3])
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] + 1/6*[3, 2, 1
sage: QS3.cpi([1,1,1])
1/6*[1, 2, 3] - 1/6*[1, 3, 2] - 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] - 1/6*[3, 2, 1

sage: QS0 = SymmetricGroupAlgebra(QQ, 0)
sage: QS0.cpi(Partition([]))
[]
```

TESTS:

```
sage: QS3.cpi([2,2])
Traceback (most recent call last):
...
TypeError: p (= [2, 2]) must be a partition of n (= 3)
```

cpis()

Return a list of the centrally primitive idempotents of `self`.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ,3)
sage: a = QS3.cpis()
sage: a[0]  # [3]
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] + 1/6*[3, 2, 1
sage: a[1]  # [2, 1]
2/3*[1, 2, 3] - 1/3*[2, 3, 1] - 1/3*[3, 1, 2]
```

TESTS:

Check this works with other indexing sets:

```
sage: G = SymmetricGroup(3).algebra(QQ)
sage: a = G.cpis()
sage: a[0]
1/6*() + 1/6*(2,3) + 1/6*(1,2) + 1/6*(1,2,3) + 1/6*(1,3,2) + 1/6*(1,3)
sage: a[1]
2/3*() - 1/3*(1,2,3) - 1/3*(1,3,2)
```

dft (*form='seminormal'*, *mult='l2r'*)

Return the discrete Fourier transform for `self`.

INPUT:

- `mult` – string (default: $l2r$). If set to $r2l$, this causes the method to use the antipodes (`antipode()`) of the seminormal basis instead of the seminormal basis.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: QS3.dft()
[   1    1    1    1    1    1]
[   1  1/2   -1 -1/2 -1/2  1/2]
[   0  3/4    0  3/4 -3/4 -3/4]
[   0    1    0   -1    1   -1]
[   1 -1/2    1 -1/2 -1/2 -1/2]
[   1   -1   -1    1    1   -1]
```

epsilon_ik (*itab*, *ktab*, *star=0*, *mult='l2r'*)

Return the seminormal basis element of `self` corresponding to the pair of tableaux `itab` and `ktab` (or restrictions of these tableaux, if the optional variable `star` is set).

INPUT:

- `itab`, `ktab` – two standard tableaux of size n.
- `star` – integer (default: 0).
- `mult` – string (default: $l2r$). If set to $r2l$, this causes the method to return the antipode (`antipode()`) of $\epsilon(I, K)$ instead of $\epsilon(I, K)$ itself.

OUTPUT:

The element $\epsilon(I, K)$, where I and K are the tableaux obtained by removing all entries higher than $n - \mathrm{star}$ from `itab` and `ktab`, respectively. Here, we are using the notations from `seminormal_basis()`.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: a = QS3.epsilon_ik([[1,2,3]], [[1,2,3]]); a
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] + 1/6*[3, 2, 1
sage: QS3.dft()*vector(a)
(1, 0, 0, 0, 0, 0)
```

```
sage: a = QS3.epsilon_ik([[1,2],[3]], [[1,2],[3]]); a
1/3*[1, 2, 3] - 1/6*[1, 3, 2] + 1/3*[2, 1, 3] - 1/6*[2, 3, 1] - 1/6*[3, 1, 2] - 1/6*[3, 2, 1
sage: QS3.dft()*vector(a)
(0, 0, 0, 0, 1, 0)
```

Let us take some properties of the seminormal basis listed in the docstring of `seminormal_basis()`, and verify them on the situation of S_3.

First, check the formula

$$\epsilon(T) = \frac{1}{\kappa_{\mathrm{sh}(T)}} \epsilon(\overline{T}) e(T) \epsilon(\overline{T}).$$

In fact:

```
sage: from sage.combinat.symmetric_group_algebra import e
sage: def test_sn1(n):
....:     QSn = SymmetricGroupAlgebra(QQ, n)
....:     QSn1 = SymmetricGroupAlgebra(QQ, n - 1)
....:     for T in StandardTableaux(n):
....:         TT = T.restrict(n-1)
....:         eTT = QSn1.epsilon_ik(TT, TT)
....:         eT = QSn.epsilon_ik(T, T)
....:         kT = prod(T.shape().hooks())
....:         if kT * eT != eTT * e(T) * eTT:
....:             return False
....:     return True
sage: test_sn1(3)
True
sage: test_sn1(4)    # long time
True
```

Next, we check the identity

$$\epsilon(T, S) = \frac{1}{\kappa_{\mathrm{sh}(T)}} \epsilon(\overline{S}) \pi_{T,S} e(T) \epsilon(\overline{T})$$

which we used to define $\epsilon(T, S)$. In fact:

```
sage: from sage.combinat.symmetric_group_algebra import e
sage: def test_sn2(n):
....:     QSn = SymmetricGroupAlgebra(QQ, n)
....:     mul = QSn.left_action_product
....:     QSn1 = SymmetricGroupAlgebra(QQ, n - 1)
....:     for lam in Partitions(n):
....:         k = prod(lam.hooks())
....:         for T in StandardTableaux(lam):
....:             for S in StandardTableaux(lam):
....:                 TT = T.restrict(n-1)
....:                 SS = S.restrict(n-1)
....:                 eTT = QSn1.epsilon_ik(TT, TT)
....:                 eSS = QSn1.epsilon_ik(SS, SS)
....:                 eTS = QSn.epsilon_ik(T, S)
....:                 piTS = [0] * n
....:                 for (i, j) in T.cells():
....:                     piTS[T[i][j] - 1] = S[i][j]
....:                 piTS = QSn(Permutation(piTS))
....:                 if k * eTS != mul(mul(eSS, piTS), mul(e(T), eTT)):
....:                     return False
....:     return True
```

```
sage: test_sn2(3)
True
sage: test_sn2(4)    # long time
True
```

Let us finally check the identity

$$\epsilon(T,S)\epsilon(U,V) = \delta_{T,V}\epsilon(U,S)$$

In fact:

```
sage: def test_sn3(lam):
....:     n = lam.size()
....:     QSn = SymmetricGroupAlgebra(QQ, n)
....:     mul = QSn.left_action_product
....:     for T in StandardTableaux(lam):
....:         for S in StandardTableaux(lam):
....:             for U in StandardTableaux(lam):
....:                 for V in StandardTableaux(lam):
....:                     lhs = mul(QSn.epsilon_ik(T, S), QSn.epsilon_ik(U, V))
....:                     if T == V:
....:                         rhs = QSn.epsilon_ik(U, S)
....:                     else:
....:                         rhs = QSn.zero()
....:                     if rhs != lhs:
....:                         return False
....:     return True
sage: all( test_sn3(lam) for lam in Partitions(3) )
True
sage: all( test_sn3(lam) for lam in Partitions(4) )    # long time
True
```

jucys_murphy(*k*)

Return the Jucys-Murphy element J_k (also known as a Young-Jucys-Murphy element) for the symmetric group algebra `self`.

The Jucys-Murphy element J_k in the symmetric group algebra RS_n is defined for every $k \in \{1, 2, \ldots, n\}$ by

$$J_k = (1,k) + (2,k) + \cdots + (k-1,k) \in RS_n,$$

where the addends are transpositions in S_n (regarded as elements of RS_n). We note that there is not a dependence on n, so it is often surpressed in the notation.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: QS3.jucys_murphy(1)
0
sage: QS3.jucys_murphy(2)
[2, 1, 3]
sage: QS3.jucys_murphy(3)
[1, 3, 2] + [3, 2, 1]

sage: QS4 = SymmetricGroupAlgebra(QQ, 4)
sage: j3 = QS4.jucys_murphy(3); j3
[1, 3, 2, 4] + [3, 2, 1, 4]
sage: j4 = QS4.jucys_murphy(4); j4
[1, 2, 4, 3] + [1, 4, 3, 2] + [4, 2, 3, 1]
sage: j3*j4 == j4*j3
```

```
True

sage: QS5 = SymmetricGroupAlgebra(QQ, 5)
sage: QS5.jucys_murphy(4)
[1, 2, 4, 3, 5] + [1, 4, 3, 2, 5] + [4, 2, 3, 1, 5]
```

TESTS:

```
sage: QS3.jucys_murphy(4)
Traceback (most recent call last):
...
ValueError: k (= 4) must be between 1 and n (= 3) (inclusive)
```

left_action_product(*left*, *right*)

Return the product of two elements `left` and `right` of `self`, where multiplication is defined in such a way that for two permutations p and q, the product pq is the permutation obtained by first applying q and then applying p. This definition of multiplication is tailored to make multiplication of permutations associative with their action on numbers if permutations are to act on numbers from the left.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: p1 = Permutation([2, 1, 3])
sage: p2 = Permutation([3, 1, 2])
sage: QS3.left_action_product(QS3(p1), QS3(p2))
[3, 2, 1]
sage: x = QS3([1, 2, 3]) - 2*QS3([1, 3, 2])
sage: y = 1/2 * QS3([3, 1, 2]) + 3*QS3([1, 2, 3])
sage: QS3.left_action_product(x, y)
3*[1, 2, 3] - 6*[1, 3, 2] - [2, 1, 3] + 1/2*[3, 1, 2]
sage: QS3.left_action_product(0, x)
0
```

The method coerces its input into the algebra `self`:

```
sage: QS4 = SymmetricGroupAlgebra(QQ, 4)
sage: QS4.left_action_product(QS3([1, 2, 3]), QS3([2, 1, 3]))
[2, 1, 3, 4]
sage: QS4.left_action_product(1, Permutation([4, 1, 2, 3]))
[4, 1, 2, 3]
```

TESTS:

```
sage: QS4 = SymmetricGroup(4).algebra(QQ)
sage: QS4.left_action_product(QS4((1,2)), QS4((2,3)))
(1,2,3)
sage: QS4.left_action_product(1, QS4((1,2)))
(1,2)
```

Warning: Note that coercion presently works from permutations of `n` into the `n`-th symmetric group algebra, and also from all smaller symmetric group algebras into the `n`-th symmetric group algebra, but not from permutations of integers smaller than `n` into the `n`-th symmetric group algebra.

monomial_from_smaller_permutation(*permutation*)

Convert `permutation` into a permutation, possibly extending it to the appropriate size, and return the corresponding basis element of `self`.

EXAMPLES:

```
sage: QS5 = SymmetricGroupAlgebra(QQ, 5)
sage: QS5.monomial_from_smaller_permutation([])
[1, 2, 3, 4, 5]
sage: QS5.monomial_from_smaller_permutation(Permutation([3,1,2]))
[3, 1, 2, 4, 5]
sage: QS5.monomial_from_smaller_permutation([5,3,4,1,2])
[5, 3, 4, 1, 2]
sage: QS5.monomial_from_smaller_permutation(SymmetricGroup(2)((1,2)))
[2, 1, 3, 4, 5]

sage: QS5g = SymmetricGroup(5).algebra(QQ)
sage: QS5g.monomial_from_smaller_permutation([2,1])
(1,2)
```

TESTS:

```
sage: QS5.monomial_from_smaller_permutation([5,3,4,1,2]).parent()
Symmetric group algebra of order 5 over Rational Field
```

retract_direct_product (*f*, *m*)

Return the direct-product retract of the element $f \in RS_n$ to RS_m, where $m \leq n$ (and where RS_n is `self`).

If m is a nonnegative integer less or equal to n, then the direct-product retract from S_n to S_m is defined as an R-linear map $S_n \to S_m$ which sends every permutation $p \in S_n$ to

$$\begin{cases} \mathrm{dret}(p) & \text{if } \mathrm{dret}(p) \text{ is defined;} \\ 0 & \text{otherwise} \end{cases}.$$

Here $\mathrm{dret}(p)$ denotes the direct-product retract of the permutation p to S_m, which is defined in `retract_direct_product()`.

EXAMPLES:

```
sage: SGA3 = SymmetricGroupAlgebra(QQ, 3)
sage: SGA3.retract_direct_product(2*SGA3([1,2,3]) - 4*SGA3([2,1,3]) + 7*SGA3([1,3,2]), 2)
2*[1, 2] - 4*[2, 1]
sage: SGA3.retract_direct_product(2*SGA3([1,3,2]) - 5*SGA3([2,3,1]), 2)
0

sage: SGA5 = SymmetricGroupAlgebra(QQ, 5)
sage: SGA5.retract_direct_product(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) + 11*SGA5([3,2,1
11*[3, 2, 1, 4]
sage: SGA5.retract_direct_product(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) + 11*SGA5([3,2,1
-6*[1, 3, 2] + 11*[3, 2, 1]
sage: SGA5.retract_direct_product(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) + 11*SGA5([3,2,1
0
sage: SGA5.retract_direct_product(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) + 11*SGA5([3,2,1
2*[1]

sage: SGA5.retract_direct_product(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]), 3)
8*[1, 2, 3] - 6*[1, 3, 2]
sage: SGA5.retract_direct_product(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]), 1)
2*[1]
sage: SGA5.retract_direct_product(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]), 0)
2*[]
```

TESTS:

Check this works with other indexing sets:

```
sage: G = SymmetricGroup(4).algebra(QQ)
sage: G.retract_direct_product(G.an_element(), 3)
() + 2*(1,2)
```

See also:

retract_plain(), retract_okounkov_vershik()

retract_okounkov_vershik(*f*, *m*)

Return the Okounkov-Vershik retract of the element $f \in RS_n$ to RS_m, where $m \leq n$ (and where RS_n is `self`).

If m is a nonnegative integer less or equal to n, then the Okounkov-Vershik retract from S_n to S_m is defined as an R-linear map $S_n \to S_m$ which sends every permutation $p \in S_n$ to the Okounkov-Vershik retract of the permutation p to S_m, which is defined in retract_okounkov_vershik().

EXAMPLES:

```
sage: SGA3 = SymmetricGroupAlgebra(QQ, 3)
sage: SGA3.retract_okounkov_vershik(2*SGA3([1,2,3]) - 4*SGA3([2,1,3]) + 7*SGA3([1,3,2]), 2)
9*[1, 2] - 4*[2, 1]
sage: SGA3.retract_okounkov_vershik(2*SGA3([1,3,2]) - 5*SGA3([2,3,1]), 2)
2*[1, 2] - 5*[2, 1]

sage: SGA5 = SymmetricGroupAlgebra(QQ, 5)
sage: SGA5.retract_okounkov_vershik(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) + 11*SGA5([3,2
-6*[1, 3, 2, 4] + 8*[1, 4, 2, 3] + 11*[3, 2, 1, 4]
sage: SGA5.retract_okounkov_vershik(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) + 11*SGA5([3,2
2*[1, 3, 2] + 11*[3, 2, 1]
sage: SGA5.retract_okounkov_vershik(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) + 11*SGA5([3,2
13*[1, 2]
sage: SGA5.retract_okounkov_vershik(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) + 11*SGA5([3,2
13*[1]

sage: SGA5.retract_okounkov_vershik(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]), 3)
8*[1, 2, 3] - 6*[1, 3, 2]
sage: SGA5.retract_okounkov_vershik(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]), 1)
2*[1]
sage: SGA5.retract_okounkov_vershik(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]), 0)
2*[]
```

TESTS:

Check this works with other indexing sets:

```
sage: G = SymmetricGroup(4).algebra(QQ)
sage: G.retract_okounkov_vershik(G.an_element(), 3)
() + 2*(1,2) + 4*(1,2,3)
```

See also:

retract_plain(), retract_direct_product()

retract_plain(*f*, *m*)

Return the plain retract of the element $f \in RS_n$ to RS_m, where $m \leq n$ (and where RS_n is `self`).

If m is a nonnegative integer less or equal to n, then the plain retract from S_n to S_m is defined as an

R-linear map $S_n \to S_m$ which sends every permutation $p \in S_n$ to

$$\begin{cases} \mathrm{pret}(p) & \text{if } \mathrm{pret}(p) \text{ is defined;} \\ 0 & \text{otherwise} \end{cases}.$$

Here $\mathrm{pret}(p)$ denotes the plain retract of the permutation p to S_m, which is defined in `retract_plain()`.

EXAMPLES:

```
sage: SGA3 = SymmetricGroupAlgebra(QQ, 3)
sage: SGA3.retract_plain(2*SGA3([1,2,3]) - 4*SGA3([2,1,3]) + 7*SGA3([1,3,2]), 2)
2*[1, 2] - 4*[2, 1]
sage: SGA3.retract_plain(2*SGA3([1,3,2]) - 5*SGA3([2,3,1]), 2)
0

sage: SGA5 = SymmetricGroupAlgebra(QQ, 5)
sage: SGA5.retract_plain(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) + 11*SGA5([3,2,1,4,5]), 4
11*[3, 2, 1, 4]
sage: SGA5.retract_plain(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) + 11*SGA5([3,2,1,4,5]), 3
11*[3, 2, 1]
sage: SGA5.retract_plain(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) + 11*SGA5([3,2,1,4,5]), 2
0
sage: SGA5.retract_plain(8*SGA5([1,4,2,5,3]) - 6*SGA5([1,3,2,5,4]) + 11*SGA5([3,2,1,4,5]), 1
0

sage: SGA5.retract_plain(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]), 3)
8*[1, 2, 3] - 6*[1, 3, 2]
sage: SGA5.retract_plain(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]), 1)
8*[1]
sage: SGA5.retract_plain(8*SGA5([1,2,3,4,5]) - 6*SGA5([1,3,2,4,5]), 0)
8*[]
```

TESTS:

Check this works with other indexing sets:

```
sage: G = SymmetricGroup(4).algebra(QQ)
sage: G.retract_plain(G.an_element(), 3)
() + 2*(1,2)
```

See also:

`retract_direct_product()`, `retract_okounkov_vershik()`

right_action_product (*left*, *right*)

Return the product of two elements `left` and `right` of `self`, where multiplication is defined in such a way that for two permutations p and q, the product pq is the permutation obtained by first applying p and then applying q. This definition of multiplication is tailored to make multiplication of permutations associative with their action on numbers if permutations are to act on numbers from the right.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: p1 = Permutation([2, 1, 3])
sage: p2 = Permutation([3, 1, 2])
sage: QS3.right_action_product(QS3(p1), QS3(p2))
[1, 3, 2]
sage: x = QS3([1, 2, 3]) - 2*QS3([1, 3, 2])
sage: y = 1/2 * QS3([3, 1, 2]) + 3*QS3([1, 2, 3])
sage: QS3.right_action_product(x, y)
```

```
3*[1, 2, 3] - 6*[1, 3, 2] + 1/2*[3, 1, 2] - [3, 2, 1]
sage: QS3.right_action_product(0, x)
0
```

The method coerces its input into the algebra `self`:

```
sage: QS4 = SymmetricGroupAlgebra(QQ, 4)
sage: QS4.right_action_product(QS3([1, 2, 3]), QS3([2, 1, 3]))
[2, 1, 3, 4]
sage: QS4.right_action_product(1, Permutation([4, 1, 2, 3]))
[4, 1, 2, 3]
```

TESTS:

```
sage: QS4 = SymmetricGroup(4).algebra(QQ)
sage: QS4.right_action_product(QS4((1,2)), QS4((2,3)))
(1,3,2)
sage: QS4.right_action_product(1, QS4((1,2)))
(1,2)
```

Warning: Note that coercion presently works from permutations of `n` into the `n`-th symmetric group algebra, and also from all smaller symmetric group algebras into the `n`-th symmetric group algebra, but not from permutations of integers smaller than `n` into the `n`-th symmetric group algebra.

rsw_shuffling_element(k)

Return the k-th Reiner-Saliola-Welker shuffling element in the group algebra `self`.

The k-th Reiner-Saliola-Welker shuffling element in the symmetric group algebra RS_n over a ring R is defined as the sum $\sum_{\sigma \in S_n} \mathrm{noninv}_k(\sigma) \cdot \sigma$, where for every permutation σ, the number $\mathrm{noninv}_k(\sigma)$ is the number of all k-noninversions of σ (that is, the number of all k-element subsets of $\{1, 2, \ldots, n\}$ on which σ restricts to a strictly increasing map). See `sage.combinat.permutation.number_of_noninversions()` for the noninv map.

This element is more or less the operator $\nu_{k,1^{n-k}}$ introduced in [RSW2011]; more precisely, $\nu_{k,1^{n-k}}$ is the left multiplication by this element.

It is a nontrivial theorem (Theorem 1.1 in [RSW2011]) that the operators $\nu_{k,1^{n-k}}$ (for fixed n and varying k) pairwise commute. It is a conjecture (Conjecture 1.2 in [RSW2011]) that all their eigenvalues are integers (which, in light of their commutativity and easily established symmetry, yields that they can be simultaneously diagonalized over $\mathbf{Q}$ with only integer eigenvalues).

EXAMPLES:

The Reiner-Saliola-Welker shuffling elements on $\mathbf{Q}S_3$:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: QS3.rsw_shuffling_element(0)
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
sage: QS3.rsw_shuffling_element(1)
3*[1, 2, 3] + 3*[1, 3, 2] + 3*[2, 1, 3] + 3*[2, 3, 1] + 3*[3, 1, 2] + 3*[3, 2, 1]
sage: QS3.rsw_shuffling_element(2)
3*[1, 2, 3] + 2*[1, 3, 2] + 2*[2, 1, 3] + [2, 3, 1] + [3, 1, 2]
sage: QS3.rsw_shuffling_element(3)
[1, 2, 3]
sage: QS3.rsw_shuffling_element(4)
0
```

Checking the commutativity of Reiner-Saliola-Welker shuffling elements (we leave out the ones for which it is trivial):

```
sage: def test_rsw_comm(n):
....:     QSn = SymmetricGroupAlgebra(QQ, n)
....:     rsws = [QSn.rsw_shuffling_element(k) for k in range(2, n)]
....:     return all( all( rsws[i] * rsws[j] == rsws[j] * rsws[i]
....:                      for j in range(i) )
....:                 for i in range(len(rsws)) )
sage: test_rsw_comm(3)
True
sage: test_rsw_comm(4)
True
sage: test_rsw_comm(5)    # long time
True
```

Note: For large `k` (relative to `n`), it might be faster to call `QSn.left_action_product(QSn.semi_rsw_element(k), QSn.antipode(binary_unshuffle_sum(k)))` than `QSn.rsw_shuffling_element(n)`.

See also:

`semi_rsw_element()`, `binary_unshuffle_sum()`

semi_rsw_element(k)

Return the k-th semi-RSW element in the group algebra `self`.

The k-th semi-RSW element in the symmetric group algebra RS_n over a ring R is defined as the sum of all permutations $\sigma \in S_n$ satisfying $\sigma(1) < \sigma(2) < \cdots < \sigma(k)$.

This element has the property that, if it is denoted by s_k, then $s_k S(s_k)$ is $(n-k)!$ times the k-th Reiner-Saliola-Welker shuffling element of RS_n (see `rsw_shuffling_element()`). Here, S denotes the antipode of the group algebra RS_n.

The k-th semi-RSW element is the image of the complete non-commutative symmetric function $S^{(k,1^{n-k})}$ in the ring of non-commutative symmetric functions under the canonical projection on the symmetric group algebra (through the descent algebra).

EXAMPLES:

The semi-RSW elements on $\mathbf{Q}S_3$:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: QS3.semi_rsw_element(0)
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
sage: QS3.semi_rsw_element(1)
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
sage: QS3.semi_rsw_element(2)
[1, 2, 3] + [1, 3, 2] + [2, 3, 1]
sage: QS3.semi_rsw_element(3)
[1, 2, 3]
sage: QS3.semi_rsw_element(4)
0
```

Let us check the relation with the k-th Reiner-Saliola-Welker shuffling element stated in the docstring:

```
sage: def test_rsw(n):
....:     ZSn = SymmetricGroupAlgebra(ZZ, n)
....:     for k in range(1, n):
....:         a = ZSn.semi_rsw_element(k)
....:         b = ZSn.left_action_product(a, ZSn.antipode(a))
....:         if factorial(n-k) * ZSn.rsw_shuffling_element(k) != b:
....:             return False
```

```
....:     return True
sage: test_rsw(3)
True
sage: test_rsw(4)
True
sage: test_rsw(5)  # long time
True
```

Let us also check the statement about the complete non-commutative symmetric function:

```
sage: def test_rsw_ncsf(n):
....:     ZSn = SymmetricGroupAlgebra(ZZ, n)
....:     NSym = NonCommutativeSymmetricFunctions(ZZ)
....:     S = NSym.S()
....:     for k in range(1, n):
....:         a = S(Composition([k] + [1]*(n-k))).to_symmetric_group_algebra()
....:         if a != ZSn.semi_rsw_element(k):
....:             return False
....:     return True
sage: test_rsw_ncsf(3)
True
sage: test_rsw_ncsf(4)
True
sage: test_rsw_ncsf(5)  # long time
True
```

seminormal_basis(*mult='l2r'*)

Return a list of the seminormal basis elements of `self`.

The seminormal basis of a symmetric group algebra is defined as follows:

Let n be a nonnegative integer. Let R be a $\mathbf{Q}$-algebra. In the following, we will use the "left action" convention for multiplying permutations. This means that for all permutations p and q in S_n, the product pq is defined in such a way that $(pq)(i) = p(q(i))$ for each $i \in \{1, 2, \ldots, n\}$ (this is the same convention as in `left_action_product()`, but not the default semantics of the $*$ operator on permutations in Sage). Thus, for instance, $s_2 s_1$ is the permutation obtained by first transposing 1 with 2 and then transposing 2 with 3 (where $s_i = (i, i+1)$).

For every partition λ of n, let

$$\kappa_\lambda = \frac{n!}{f^\lambda}$$

where f^λ is the number of standard Young tableaux of shape λ. Note that κ_λ is an integer, namely the product of all hook lengths of λ (by the hook length formula). In Sage, this integer can be computed by using `sage.combinat.symmetric_group_algebra.kappa()`.

Let T be a standard tableau of size n.

Let $a(T)$ denote the formal sum (in RS_n) of all permutations in S_n which stabilize the rows of T (as sets), i. e., which map each entry i of T to an entry in the same row as i. (See `sage.combinat.symmetric_group_algebra.a()` for an implementation of this.)

Let $b(T)$ denote the signed formal sum (in RS_n) of all permutations in S_n which stabilize the columns of T (as sets). Here, "signed" means that each permutation is multiplied with its sign. (This is implemented in `sage.combinat.symmetric_group_algebra.b()`.)

Define an element $e(T)$ of RS_n to be $a(T)b(T)$. (This is implemented in `sage.combinat.symmetric_group_algebra.e()` for $R = \mathbf{Q}$.)

Let $\mathrm{sh}(T)$ denote the shape of T. (See `shape()`.)

Let $\overline{T}$ denote the standard tableau of size $n-1$ obtained by removing the letter n (along with its cell) from T (if $n \geq 1$).

Now, we define an element $\epsilon(T)$ of RS_n. We define it by induction on the size n of T, so we set $\epsilon(\emptyset) = 1$ and only need to define $\epsilon(T)$ for $n \geq 1$, assuming that $\epsilon(\overline{T})$ is already defined. We do this by setting

$$\epsilon(T) = \frac{1}{\kappa_{\mathrm{sh}(T)}} \epsilon(\overline{T}) e(T) \epsilon(\overline{T}).$$

This element $\epsilon(T)$ is implemented as `sage.combinat.symmetric_group_algebra.epsilon()` for $R = \mathbf{Q}$, but it is also a particular case of the elements $\epsilon(T, S)$ defined below.

Now let S be a further tableau of the same shape as T (possibly equal to T). Let $\pi_{T,S}$ denote the permutation in S_n such that applying this permutation to the entries of T yields the tableau S. Define an element $\epsilon(T, S)$ of RS_n by

$$\epsilon(T, S) = \frac{1}{\kappa_{\mathrm{sh}(T)}} \epsilon(\overline{S}) \pi_{T,S} e(T) \epsilon(\overline{T}) = \frac{1}{\kappa_{\mathrm{sh}(T)}} \epsilon(\overline{S}) a(S) \pi_{T,S} b(T) \epsilon(\overline{T}).$$

This element $\epsilon(T, S)$ is called *Young's seminormal unit corresponding to the bitableau '(T, S)'*, and is the return value of `epsilon_ik()` applied to `T` and `S`. Note that $\epsilon(T, T) = \epsilon(T)$.

If we let λ run through all partitions of n, and (T, S) run through all pairs of tableaux of shape λ, then the elements $\epsilon(T, S)$ form a basis of RS_n. This basis is called *Young's seminormal basis* and has the properties that

$$\epsilon(T, S) \epsilon(U, V) = \delta_{T,V} \epsilon(U, S)$$

(where δ stands for the Kronecker delta).

Warning: Because of our convention, we are multiplying our elements in reverse of those given in some papers, for example [Ram1997]. Using the other convention of multiplying permutations, we would instead have $\epsilon(U, V)\epsilon(T, S) = \delta_{T,V}\epsilon(U, S)$.

In other words, Young's seminormal basis consists of the matrix units in a (particular) Artin-Wedderburn decomposition of RS_n into a direct product of matrix algebras over $\mathbf{Q}$.

The output of `seminormal_basis()` is a list of all elements of the seminormal basis of `self`.

INPUT:

- `mult` – string (default: `'l2r'`). If set to `'r2l'`, this causes the method to return the list of the antipodes (`antipode()`) of all $\epsilon(T, S)$ instead of the $\epsilon(T, S)$ themselves.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ,3)
sage: QS3.seminormal_basis()
[1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] + 1/6*[3, 2,
1/3*[1, 2, 3] + 1/6*[1, 3, 2] - 1/3*[2, 1, 3] - 1/6*[2, 3, 1] - 1/6*[3, 1, 2] + 1/6*[3, 2, 1
1/3*[1, 3, 2] + 1/3*[2, 3, 1] - 1/3*[3, 1, 2] - 1/3*[3, 2, 1],
1/4*[1, 3, 2] - 1/4*[2, 3, 1] + 1/4*[3, 1, 2] - 1/4*[3, 2, 1],
1/3*[1, 2, 3] - 1/6*[1, 3, 2] + 1/3*[2, 1, 3] - 1/6*[2, 3, 1] - 1/6*[3, 1, 2] - 1/6*[3, 2, 1
1/6*[1, 2, 3] - 1/6*[1, 3, 2] - 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] - 1/6*[3, 2, 1
```

TESTS:

```
sage: QS3g = SymmetricGroup(3).algebra(QQ)
sage: QS3g.seminormal_basis()
[1/6*() + 1/6*(2,3) + 1/6*(1,2) + 1/6*(1,2,3) + 1/6*(1,3,2) + 1/6*(1,3),
 1/3*() + 1/6*(2,3) - 1/3*(1,2) - 1/6*(1,2,3) - 1/6*(1,3,2) + 1/6*(1,3),
```

```
1/3*(2,3) + 1/3*(1,2,3) - 1/3*(1,3,2) - 1/3*(1,3),
1/4*(2,3) - 1/4*(1,2,3) + 1/4*(1,3,2) - 1/4*(1,3),
1/3*() - 1/6*(2,3) + 1/3*(1,2) - 1/6*(1,2,3) - 1/6*(1,3,2) - 1/6*(1,3),
1/6*() - 1/6*(2,3) - 1/6*(1,2) + 1/6*(1,2,3) + 1/6*(1,3,2) - 1/6*(1,3)]
```

REFERENCES:

sage.combinat.symmetric_group_algebra.**a**(*tableau*, *star=0*, *base_ring=Rational Field*)

The row projection operator corresponding to the Young tableau `tableau` (which is supposed to contain every integer from 1 to its size precisely once, but may and may not be standard).

This is the sum (in the group algebra of the relevant symmetric group over $\mathbf{Q}$) of all the permutations which preserve the rows of `tableau`. It is called a_{tableau} in [EtRT], Section 4.2.

REFERENCES:

INPUT:

- `tableau` – Young tableau which contains every integer from 1 to its size precisely once.
- `star` – nonnegative integer (default: 0). When this optional variable is set, the method computes not the row projection operator of `tableau`, but the row projection operator of the restriction of `tableau` to the entries `1, 2, ..., tableau.size() - star` instead.
- `base_ring` – commutative ring (default: `QQ`). When this optional variable is set, the row projection operator is computed over a user-determined base ring instead of $\mathbf{Q}$. (Note that symmetric group algebras currently don't preserve coercion, so e. g. a symmetric group algebra over $\mathbf{Z}$ does not coerce into the corresponding one over $\mathbf{Q}$; so convert manually or choose your base rings wisely!)

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import a
sage: a([[1,2]])
[1, 2] + [2, 1]
sage: a([[1],[2]])
[1, 2]
sage: a([])
[]
sage: a([[1, 5], [2, 3], [4]])
[1, 2, 3, 4, 5] + [1, 3, 2, 4, 5] + [5, 2, 3, 4, 1] + [5, 3, 2, 4, 1]
sage: a([[1,4], [2,3]], base_ring=ZZ)
[1, 2, 3, 4] + [1, 3, 2, 4] + [4, 2, 3, 1] + [4, 3, 2, 1]
```

sage.combinat.symmetric_group_algebra.**b**(*tableau*, *star=0*, *base_ring=Rational Field*)

The column projection operator corresponding to the Young tableau `tableau` (which is supposed to contain every integer from 1 to its size precisely once, but may and may not be standard).

This is the signed sum (in the group algebra of the relevant symmetric group over $\mathbf{Q}$) of all the permutations which preserve the column of `tableau` (where the signs are the usual signs of the permutations). It is called b_{tableau} in [EtRT], Section 4.2.

INPUT:

- `tableau` – Young tableau which contains every integer from 1 to its size precisely once.
- `star` – nonnegative integer (default: 0). When this optional variable is set, the method computes not the column projection operator of `tableau`, but the column projection operator of the restriction of `tableau` to the entries `1, 2, ..., tableau.size() - star` instead.
- `base_ring` – commutative ring (default: `QQ`). When this optional variable is set, the column projection operator is computed over a user-determined base ring instead of $\mathbf{Q}$. (Note that symmetric group algebras

currently don't preserve coercion, so e. g. a symmetric group algebra over $\mathbf{Z}$ does not coerce into the corresponding one over $\mathbf{Q}$; so convert manually or choose your base rings wisely!)

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import b
sage: b([[1,2]])
[1, 2]
sage: b([[1],[2]])
[1, 2] - [2, 1]
sage: b([])
[]
sage: b([[1, 2, 4], [5, 3]])
[1, 2, 3, 4, 5] - [1, 3, 2, 4, 5] - [5, 2, 3, 4, 1] + [5, 3, 2, 4, 1]
sage: b([[1, 4], [2, 3]], base_ring=ZZ)
[1, 2, 3, 4] - [1, 2, 4, 3] - [2, 1, 3, 4] + [2, 1, 4, 3]
sage: b([[1, 4], [2, 3]], base_ring=Integers(5))
[1, 2, 3, 4] + 4*[1, 2, 4, 3] + 4*[2, 1, 3, 4] + [2, 1, 4, 3]
```

With the `l2r` setting for multiplication, the unnormalized Young symmetrizer `e(tableau)` should be the product `b(tableau) * a(tableau)` for every `tableau`. Let us check this on the standard tableaux of size 5:

```
sage: from sage.combinat.symmetric_group_algebra import a, b, e
sage: all( e(t) == b(t) * a(t) for t in StandardTableaux(5) )
True
```

sage.combinat.symmetric_group_algebra.**e**(*tableau*, *star=0*)

The unnormalized Young projection operator corresponding to the Young tableau `tableau` (which is supposed to contain every integer from 1 to its size precisely once, but may and may not be standard).

If n is a nonnegative integer, and T is a Young tableau containing every integer from 1 to n exactly once, then the unnormalized Young projection operator $e(T)$ is defined by

$$e(T) = a(T)b(T) \in \mathbf{Q}S_n,$$

where $a(T) \in \mathbf{Q}S_n$ is the sum of all permutations in S_n which fix the rows of T (as sets), and $b(T) \in \mathbf{Q}S_n$ is the signed sum of all permutations in S_n which fix the columns of T (as sets). Here, "signed" means that each permutation is multiplied with its sign; and the product on the group S_n is defined in such a way that $(pq)(i) = p(q(i))$ for any permutations p and q and any $1 \leq i \leq n$.

Note that the definition of $e(T)$ is not uniform across literature. Others define it as $b(T)a(T)$ instead, or include certain scalar factors (we do not, whence "unnormalized").

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import e
sage: e([[1,2]])
[1, 2] + [2, 1]
sage: e([[1],[2]])
[1, 2] - [2, 1]
sage: e([])
[]
```

There are differing conventions for the order of the symmetrizers and antisymmetrizers. This example illustrates our conventions:

```
sage: e([[1,2],[3]])
[1, 2, 3] + [2, 1, 3] - [3, 1, 2] - [3, 2, 1]
```

To obtain the product $b(T)a(T)$, one has to take the antipode of this:

```
sage: QS3 = parent(e([[1,2],[3]]))
sage: QS3.antipode(e([[1,2],[3]]))
[1, 2, 3] + [2, 1, 3] - [2, 3, 1] - [3, 2, 1]
```

See also:

e_hat()

sage.combinat.symmetric_group_algebra.**e_hat**(*tab*, *star=0*)

The Young projection operator corresponding to the Young tableau `tab` (which is supposed to contain every integer from 1 to its size precisely once, but may and may not be standard). This is an idempotent in the rational group algebra.

If n is a nonnegative integer, and T is a Young tableau containing every integer from 1 to n exactly once, then the Young projection operator $\widehat{e}(T)$ is defined by

$$\widehat{e}(T) = \frac{1}{\kappa_\lambda} a(T) b(T) \in \mathbf{Q}S_n,$$

where λ is the shape of T, where κ_λ is $n!$ divided by the number of standard tableaux of shape λ, where $a(T) \in \mathbf{Q}S_n$ is the sum of all permutations in S_n which fix the rows of T (as sets), and where $b(T) \in \mathbf{Q}S_n$ is the signed sum of all permutations in S_n which fix the columns of T (as sets). Here, "signed" means that each permutation is multiplied with its sign; and the product on the group S_n is defined in such a way that $(pq)(i) = p(q(i))$ for any permutations p and q and any $1 \leq i \leq n$.

Note that the definition of $\widehat{e}(T)$ is not uniform across literature. Others define it as $\frac{1}{\kappa_\lambda} b(T) a(T)$ instead.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import e_hat
sage: e_hat([[1,2,3]])
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] + 1/6*[3, 2, 1]
sage: e_hat([[1],[2]])
1/2*[1, 2] - 1/2*[2, 1]
```

There are differing conventions for the order of the symmetrizers and antisymmetrizers. This example illustrates our conventions:

```
sage: e_hat([[1,2],[3]])
1/3*[1, 2, 3] + 1/3*[2, 1, 3] - 1/3*[3, 1, 2] - 1/3*[3, 2, 1]
```

See also:

e()

sage.combinat.symmetric_group_algebra.**e_ik**(*itab*, *ktab*, *star=0*)

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import e_ik
sage: e_ik([[1,2,3]], [[1,2,3]])
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
sage: e_ik([[1,2,3]], [[1,2,3]], star=1)
[1, 2] + [2, 1]
```

sage.combinat.symmetric_group_algebra.**epsilon**(*tab*, *star=0*)

The (T, T)-th element of the seminormal basis of the group algebra $\mathbf{Q}[S_n]$, where T is the tableau `tab` (with its `star` highest entries removed if the optional variable `star` is set).

See the docstring of seminormal_basis() for the notation used herein.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import epsilon
sage: epsilon([[1,2]])
1/2*[1, 2] + 1/2*[2, 1]
sage: epsilon([[1],[2]])
1/2*[1, 2] - 1/2*[2, 1]
```

sage.combinat.symmetric_group_algebra.**epsilon_ik**(*itab*, *ktab*, *star=0*)

Return the seminormal basis element of the symmetric group algebra $\mathbf{Q}S_n$ corresponding to the pair of tableaux `itab` and `ktab` (or restrictions of these tableaux, if the optional variable `star` is set).

INPUT:

- `itab`, `ktab` – two standard tableaux of same size.
- `star` – integer (default: 0).

OUTPUT:

The element $\epsilon(I, K) \in \mathbf{Q}S_n$, where I and K are the tableaux obtained by removing all entries higher than $n -$ star from `itab` and `ktab`, respectively (where n is the size of `itab` and `ktab`). Here, we are using the notations from `seminormal_basis()`.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import epsilon_ik
sage: epsilon_ik([[1,2],[3]], [[1,3],[2]])
1/4*[1, 3, 2] - 1/4*[2, 3, 1] + 1/4*[3, 1, 2] - 1/4*[3, 2, 1]
sage: epsilon_ik([[1,2],[3]], [[1,3],[2]], star=1)
Traceback (most recent call last):
...
ValueError: the two tableaux must be of the same shape
```

sage.combinat.symmetric_group_algebra.**kappa**(*alpha*)

Return κ_α, which is $n!$ divided by the number of standard tableaux of shape α (where α is a partition of n).

INPUT:

- `alpha` – integer partition (can be encoded as a list).

OUTPUT:

The factorial of the size of `alpha`, divided by the number of standard tableaux of shape `alpha`. Equivalently, the product of all hook lengths of `alpha`.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import kappa
sage: kappa(Partition([2,1]))
3
sage: kappa([2,1])
3
```

sage.combinat.symmetric_group_algebra.**pi_ik**(*itab*, *ktab*)

Return the permutation p which sends every entry of the tableau `itab` to the respective entry of the tableau `ktab`, as an element of the corresponding symmetric group algebra.

This assumes that `itab` and `ktab` are tableaux (possibly given just as lists of lists) of the same shape.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import pi_ik
sage: pi_ik([[1,3],[2]], [[1,2],[3]])
[1, 3, 2]
```

sage.combinat.symmetric_group_algebra.**seminormal_test**(*n*)

Run a variety of tests to verify that the construction of the seminormal basis works as desired. The numbers appearing are results in James and Kerber's 'Representation Theory of the Symmetric Group' [JamesKerber].

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import seminormal_test
sage: seminormal_test(3)
True
```

5.1.309 Representations of the Symmetric Group

Todo

- construct the product of two irreducible representations.
- implement Induction/Restriction of representations.

Warning: This code uses a different convention than in Sagan's book "The Symmetric Group"

class sage.combinat.symmetric_group_representations.**SpechtRepresentation**(*partition*, *ring=None*, *cache_matrices=True*)

Bases: sage.combinat.symmetric_group_representations.SymmetricGroupRepresentation_generic

An irreducible representation of the symmetric group corresponding to `partition`.

For more information, see the documentation for `SymmetricGroupRepresentation()`.

EXAMPLES:

```
sage: spc = SymmetricGroupRepresentation([3])
sage: spc([3,2,1])
[1]
sage: spc == loads(dumps(spc))
True

sage: spc = SymmetricGroupRepresentation([3], cache_matrices=False)
sage: spc([3,2,1])
[1]
sage: spc == loads(dumps(spc))
True
```

representation_matrix(*permutation*)

Returns the matrix representing the `permutation` in this irreducible representation.

Note: This method caches the results.

EXAMPLES:

```
sage: spc = SymmetricGroupRepresentation([3,1], 'specht')
sage: spc.representation_matrix(Permutation([2,1,3,4]))
[ 0 -1  0]
[-1  0  0]
[ 0  0  1]
sage: spc.representation_matrix(Permutation([3,2,1,4]))
[0 0 1]
```

```
[0 1 0]
[1 0 0]
```

scalar_product(*u*, *v*)

Return 0 if u+v is not a permutation, and the signature of the permutation otherwise.

This is the scalar product of a vertex u of the underlying Yang-Baxter graph with the vertex v in the 'dual' Yang-Baxter graph.

EXAMPLES:

```
sage: spc = SymmetricGroupRepresentation([3,2], 'specht')
sage: spc.scalar_product((1,0,2,1,0),(0,3,0,3,0))
-1
sage: spc.scalar_product((1,0,2,1,0),(3,0,0,3,0))
0
```

scalar_product_matrix(*permutation=None*)

Return the scalar product matrix corresponding to permutation.

The entries are given by the scalar products of u and permutation.action(v), where u is a vertex in the underlying Yang-Baxter graph and v is a vertex in the dual graph.

EXAMPLES:

```
sage: spc = SymmetricGroupRepresentation([3,1], 'specht')
sage: spc.scalar_product_matrix()
[ 1  0  0]
[ 0 -1  0]
[ 0  0  1]
```

class sage.combinat.symmetric_group_representations.**SpechtRepresentations**(*n*, *ring=None*, *cache_matrices=True*)

Bases: sage.combinat.symmetric_group_representations.SymmetricGroupRepresentations_class

Irreducible representations of the symmetric group.

See the documentation for SymmetricGroupRepresentations() for more information.

EXAMPLES:

```
sage: snorm = SymmetricGroupRepresentations(3, "seminormal")
sage: snorm == loads(dumps(snorm))
True
```

object_class

alias of SpechtRepresentation

sage.combinat.symmetric_group_representations.**SymmetricGroupRepresentation**(*partition*, *implementation='specht'*, *ring=None*, *cache_matrices=True*)

The irreducible representation of the symmetric group corresponding to partition.

INPUT:

- partition – a partition of a positive integer

- **implementation – string (default: "specht"), one of:**
  - "seminormal" - for Young's seminormal representation
  - "orthogonal" - for Young's orthogonal representation
  - "specht" - for Specht's representation
- ring – the ring over which the representation is defined.
- cache_matrices – boolean (default: True) if True, then any representation matrices that are computed are cached.

EXAMPLES:

Young's orthogonal representation: the matrices are orthogonal.

```
sage: orth = SymmetricGroupRepresentation([2,1], "orthogonal"); orth
Orthogonal representation of the symmetric group corresponding to [2, 1]
sage: all(a*a.transpose() == a.parent().identity_matrix() for a in orth)
True

sage: orth = SymmetricGroupRepresentation([3,2], "orthogonal"); orth
Orthogonal representation of the symmetric group corresponding to [3, 2]
sage: orth([2,1,3,4,5])
[ 1  0  0  0  0]
[ 0  1  0  0  0]
[ 0  0 -1  0  0]
[ 0  0  0  1  0]
[ 0  0  0  0 -1]
sage: orth([1,3,2,4,5])
[           1            0            0            0            0]
[           0        -1/2 1/2*sqrt(3)            0            0]
[           0 1/2*sqrt(3)          1/2            0            0]
[           0            0            0        -1/2 1/2*sqrt(3)]
[           0            0            0 1/2*sqrt(3)          1/2]
sage: orth([1,2,4,3,5])
[       -1/3 2/3*sqrt(2)            0            0            0]
[2/3*sqrt(2)          1/3            0            0            0]
[           0            0            1            0            0]
[           0            0            0            1            0]
[           0            0            0            0           -1]
```

The Specht Representation:

```
sage: spc = SymmetricGroupRepresentation([3,2], "specht")
sage: spc.scalar_product_matrix(Permutation([1,2,3,4,5]))
[ 1  0  0  0  0]
[ 0 -1  0  0  0]
[ 0  0  1  0  0]
[ 0  0  0  1  0]
[-1  0  0  0 -1]
sage: spc.scalar_product_matrix(Permutation([5,4,3,2,1]))
[ 1 -1  0  1  0]
[ 0  0  1  0 -1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[-1  0  0  0 -1]
sage: spc([5,4,3,2,1])
[ 1 -1  0  1  0]
[ 0  0 -1  0  1]
[ 0  0  0 -1  1]
```

```
[ 0  1 -1 -1  1]
[ 0  1  0 -1  1]
sage: spc.verify_representation()
True
```

By default, any representation matrices that are computed are cached:

```
sage: spc = SymmetricGroupRepresentation([3,2], "specht")
sage: spc([5,4,3,2,1])
[ 1 -1  0  1  0]
[ 0  0 -1  0  1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[ 0  1  0 -1  1]
sage: spc._cache__representation_matrix
{(([5, 4, 3, 2, 1],), ()): [ 1 -1  0  1  0]
[ 0  0 -1  0  1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[ 0  1  0 -1  1]}
```

This can be turned off with the keyword cache_matrices:

```
sage: spc = SymmetricGroupRepresentation([3,2], "specht", cache_matrices=False)
sage: spc([5,4,3,2,1])
[ 1 -1  0  1  0]
[ 0  0 -1  0  1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[ 0  1  0 -1  1]
sage: hasattr(spc, '_cache__representation_matrix')
False
```

Note: The implementation is based on the paper [Las].

REFERENCES:

AUTHORS:

- Franco Saliola (2009-04-23)

class sage.combinat.symmetric_group_representations.**SymmetricGroupRepresentation_generic_clas**

Bases: sage.structure.sage_object.SageObject

Generic methods for a representation of the symmetric group.

to_character()

Return the character of the representation.

EXAMPLES:

The trivial character:

```
sage: rho = SymmetricGroupRepresentation([3])
sage: chi = rho.to_character(); chi
Character of Symmetric group of order 3! as a permutation group
sage: chi.values()
[1, 1, 1]
```

```
sage: all(chi(g) == 1 for g in SymmetricGroup(3))
True
```

The sign character:

```
sage: rho = SymmetricGroupRepresentation([1,1,1])
sage: chi = rho.to_character(); chi
Character of Symmetric group of order 3! as a permutation group
sage: chi.values()
[1, -1, 1]
sage: all(chi(g) == g.sign() for g in SymmetricGroup(3))
True
```

The defining representation:

```
sage: triv = SymmetricGroupRepresentation([4])
sage: hook = SymmetricGroupRepresentation([3,1])
sage: def_rep = lambda p : triv(p).block_sum(hook(p)).trace()
sage: map(def_rep, Permutations(4))
[4, 2, 2, 1, 1, 2, 2, 0, 1, 0, 0, 1, 1, 0, 2, 1, 0, 0, 0, 1, 1, 2, 0, 0]
sage: [p.to_matrix().trace() for p in Permutations(4)]
[4, 2, 2, 1, 1, 2, 2, 0, 1, 0, 0, 1, 1, 0, 2, 1, 0, 0, 0, 1, 1, 2, 0, 0]
```

verify_representation()

Verify the representation: tests that the images of the simple transpositions are involutions and tests that the braid relations hold.

EXAMPLES:

```
sage: spc = SymmetricGroupRepresentation([1,1,1])
sage: spc.verify_representation()
True
sage: spc = SymmetricGroupRepresentation([4,2,1])
sage: spc.verify_representation()
True
```

sage.combinat.symmetric_group_representations.**SymmetricGroupRepresentations**(*n*, *implementation='specht'*, *ring=None*, *cache_matrices=True*

Irreducible representations of the symmetric group.

INPUT:

- n – positive integer
- **implementation – string (default: "specht"), one of:**
  - "seminormal" - for Young's seminormal representation
  - "orthogonal" - for Young's orthogonal representation
  - "specht" - for Specht's representation
- ring – the ring over which the representation is defined.
- cache_matrices – boolean (default: True) if True, then any representation matrices that are computed are cached.

EXAMPLES:

Young's orthogonal representation: the matrices are orthogonal.

```
sage: orth = SymmetricGroupRepresentations(3, "orthogonal"); orth
Orthogonal representations of the symmetric group of order 3! over Symbolic Ring
sage: orth.list()
[Orthogonal representation of the symmetric group corresponding to [3], Orthogonal representatic
sage: orth([2,1])([1,2,3])
[1 0]
[0 1]
```

Young's seminormal representation.

```
sage: snorm = SymmetricGroupRepresentations(3, "seminormal"); snorm
Seminormal representations of the symmetric group of order 3! over Rational Field
sage: sgn = snorm([1,1,1]); sgn
Seminormal representation of the symmetric group corresponding to [1, 1, 1]
sage: map(sgn, Permutations(3))
[[1], [-1], [-1], [1], [1], [-1]]
```

The Specht Representation.

```
sage: spc = SymmetricGroupRepresentations(5, "specht"); spc
Specht representations of the symmetric group of order 5! over Integer Ring
sage: spc([3,2])([5,4,3,2,1])
[ 1 -1  0  1  0]
[ 0  0 -1  0  1]
[ 0  0  0 -1  1]
[ 0  1 -1 -1  1]
[ 0  1  0 -1  1]
```

Note: The implementation is based on the paper [Las].

AUTHORS:

- Franco Saliola (2009-04-23)

class sage.combinat.symmetric_group_representations.**SymmetricGroupRepresentations_class**(*n*, *ring=N*, *cache_*

Bases: sage.combinat.combinat.CombinatorialClass

Generic methods for the CombinatorialClass of irreducible representations of the symmetric group.

class sage.combinat.symmetric_group_representations.**YoungRepresentation_Orthogonal**(*partition*, *ring=None*, *cache_matrice*

Bases: sage.combinat.symmetric_group_representations.YoungRepresentation_generic

An irreducible representation of the symmetric group corresponding to `partition`.

For more information, see the documentation for SymmetricGroupRepresentation().

EXAMPLES:

```
sage: spc = SymmetricGroupRepresentation([3])
sage: spc([3,2,1])
[1]
sage: spc == loads(dumps(spc))
True
```

```
sage: spc = SymmetricGroupRepresentation([3], cache_matrices=False)
sage: spc([3,2,1])
[1]
sage: spc == loads(dumps(spc))
True
```

class sage.combinat.symmetric_group_representations.**YoungRepresentation_Seminormal**(*partition*, *ring=None*, *cache_matrice*

Bases: sage.combinat.symmetric_group_representations.YoungRepresentation_generic

An irreducible representation of the symmetric group corresponding to `partition`.

For more information, see the documentation for `SymmetricGroupRepresentation()`.

EXAMPLES:

```
sage: spc = SymmetricGroupRepresentation([3])
sage: spc([3,2,1])
[1]
sage: spc == loads(dumps(spc))
True

sage: spc = SymmetricGroupRepresentation([3], cache_matrices=False)
sage: spc([3,2,1])
[1]
sage: spc == loads(dumps(spc))
True
```

class sage.combinat.symmetric_group_representations.**YoungRepresentation_generic**(*partition*, *ring=None*, *cache_matrices=T*

Bases: sage.combinat.symmetric_group_representations.SymmetricGroupRepresentation_generic

Generic methods for Young's representations of the symmetric group.

representation_matrix(*permutation*)

Return the matrix representing `permutation`.

EXAMPLES:

```
sage: orth = SymmetricGroupRepresentation([2,1], "orthogonal")
sage: orth.representation_matrix(Permutation([2,1,3]))
[ 1  0]
[ 0 -1]
sage: orth.representation_matrix(Permutation([1,3,2]))
[       -1/2 1/2*sqrt(3)]
[1/2*sqrt(3)         1/2]

sage: norm = SymmetricGroupRepresentation([2,1], "seminormal")
sage: p = PermutationGroupElement([2,1,3])
sage: norm.representation_matrix(p)
[ 1  0]
[ 0 -1]
sage: p = PermutationGroupElement([1,3,2])
sage: norm.representation_matrix(p)
[-1/2  3/2]
[ 1/2  1/2]
```

representation_matrix_for_simple_transposition(*i*)

Return the matrix representing the transposition that swaps `i` and `i+1`.

EXAMPLES:

```
sage: orth = SymmetricGroupRepresentation([2,1], "orthogonal")
sage: orth.representation_matrix_for_simple_transposition(1)
[ 1  0]
[ 0 -1]
sage: orth.representation_matrix_for_simple_transposition(2)
[       -1/2 1/2*sqrt(3)]
[1/2*sqrt(3)         1/2]

sage: norm = SymmetricGroupRepresentation([2,1], "seminormal")
sage: norm.representation_matrix_for_simple_transposition(1)
[ 1  0]
[ 0 -1]
sage: norm.representation_matrix_for_simple_transposition(2)
[-1/2  3/2]
[ 1/2  1/2]
```

class sage.combinat.symmetric_group_representations.**YoungRepresentations_Orthogonal**(*n*, *ring=None*, *cache_matri*

Bases: sage.combinat.symmetric_group_representations.SymmetricGroupRepresentations_class

Irreducible representations of the symmetric group.

See the documentation for SymmetricGroupRepresentations() for more information.

EXAMPLES:

```
sage: snorm = SymmetricGroupRepresentations(3, "seminormal")
sage: snorm == loads(dumps(snorm))
True
```

object_class

alias of YoungRepresentation_Orthogonal

class sage.combinat.symmetric_group_representations.**YoungRepresentations_Seminormal**(*n*, *ring=None*, *cache_matri*

Bases: sage.combinat.symmetric_group_representations.SymmetricGroupRepresentations_class

Irreducible representations of the symmetric group.

See the documentation for SymmetricGroupRepresentations() for more information.

EXAMPLES:

```
sage: snorm = SymmetricGroupRepresentations(3, "seminormal")
sage: snorm == loads(dumps(snorm))
True
```

object_class

alias of YoungRepresentation_Seminormal

sage.combinat.symmetric_group_representations.**partition_to_vector_of_contents**(*partition*, *reverse=False*)

Returns the "vector of contents" associated to partition.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_representations import partition_to_vector_of_contents
sage: partition_to_vector_of_contents([3,2])
(0, 1, 2, -1, 0)
```

5.1.310 Tableaux

AUTHORS:

- Mike Hansen (2007): initial version
- Jason Bandlow (2011): updated to use Parent/Element model, and many minor fixes
- Andrew Mathas (2012-13): completed the transition to the parent/element model begun by Jason Bandlow
- Travis Scrimshaw (11-22-2012): Added tuple options, changed `*katabolism*` to `*catabolism*`. Cleaned up documentation.

This file consists of the following major classes:

Element classes:

- `Tableau`
- `SemistandardTableau`
- `StandardTableau`

Factory classes:

- `Tableaux`
- `SemistandardTableaux`
- `StandardTableaux`

Parent classes:

- `Tableaux_all`
- `Tableaux_size`
- `SemistandardTableaux_all` (facade class)
- `SemistandardTableaux_size`
- `SemistandardTableaux_size_inf`
- `SemistandardTableaux_size_weight`
- `SemistandardTableaux_shape`
- `SemistandardTableaux_shape_inf`
- `SemistandardTableaux_shape_weight`
- `StandardTableaux_all` (facade class)
- `StandardTableaux_size`
- `StandardTableaux_shape`

For display options, see `Tableaux.global_options()`.

class `sage.combinat.tableau.`**`SemistandardTableau`**(*parent*, *t*)
Bases: `sage.combinat.tableau.Tableau`

A class to model a semistandard tableau.

INPUT:

- t – a tableau, a list of iterables, or an empty list

OUTPUT:

- A SemistandardTableau object constructed from t.

A semistandard tableau is a tableau whose entries are positive integers, which are weakly increasing in rows and strictly increasing down columns.

EXAMPLES:

```
sage: t = SemistandardTableau([[1,2,3],[2,3]]); t
[[1, 2, 3], [2, 3]]
sage: t.shape()
[3, 2]
sage: t.pp() # pretty print
1 2 3
2 3
sage: t = Tableau([[1,2],[2]])
sage: s = SemistandardTableau(t); s
[[1, 2], [2]]
sage: SemistandardTableau([]) # The empty tableau
[]
```

When using code that will generate a lot of tableaux, it is slightly more efficient to construct a SemistandardTableau from the appropriate Parent object:

```
sage: SST = SemistandardTableaux()
sage: SST([[1, 2, 3], [4, 5]])
[[1, 2, 3], [4, 5]]
```

TESTS:

```
sage: SemistandardTableau([[1,2,3],[1]])
Traceback (most recent call last):
...
ValueError: [[1, 2, 3], [1]] is not a column strict tableau

sage: SemistandardTableau([[1,2,1]])
Traceback (most recent call last):
...
ValueError: The rows of [[1, 2, 1]] are not weakly increasing

sage: SemistandardTableau([[0,1]])
Traceback (most recent call last):
...
ValueError: entries must be positive integers
```

class sage.combinat.tableau.**SemistandardTableaux**(***kwds*)

Bases: sage.combinat.tableau.Tableaux

A factory class for the various classes of semistandard tableaux.

INPUT:

Keyword arguments:

- size – The size of the tableaux
- shape – The shape of the tableaux
- eval – The weight (also called content or evaluation) of the tableaux

- `max_entry` – A maximum entry for the tableaux. This can be a positive integer or infinity (`oo`). If `size` or `shape` are specified, `max_entry` defaults to be `size` or the size of `shape`.

Positional arguments:

- The first argument is interpreted as either `size` or `shape` according to whether it is an integer or a partition
- The second keyword argument will always be interpreted as `eval`

OUTPUT:

- The appropriate class, after checking basic consistency tests. (For example, specifying `eval` implies a value for max_entry).

A semistandard tableau is a tableau whose entries are positive integers, which are weakly increasing in rows and strictly increasing down columns. Note that Sage uses the English convention for partitions and tableaux; the longer rows are displayed on top.

Classes of semistandard tableaux can be iterated over if and only if there is some restriction.

EXAMPLES:

```
sage: SST = SemistandardTableaux([2,1]); SST
Semistandard tableaux of shape [2, 1] and maximum entry 3
sage: SST.list()
[[[1, 1], [2]],
 [[1, 1], [3]],
 [[1, 2], [2]],
 [[1, 2], [3]],
 [[1, 3], [2]],
 [[1, 3], [3]],
 [[2, 2], [3]],
 [[2, 3], [3]]]

sage: SST = SemistandardTableaux(3); SST
Semistandard tableaux of size 3 and maximum entry 3
sage: SST.list()
[[[1, 1, 1]],
 [[1, 1, 2]],
 [[1, 1, 3]],
 [[1, 2, 2]],
 [[1, 2, 3]],
 [[1, 3, 3]],
 [[2, 2, 2]],
 [[2, 2, 3]],
 [[2, 3, 3]],
 [[3, 3, 3]],
 [[1, 1], [2]],
 [[1, 1], [3]],
 [[1, 2], [2]],
 [[1, 2], [3]],
 [[1, 3], [2]],
 [[1, 3], [3]],
 [[2, 2], [3]],
 [[2, 3], [3]],
 [[1], [2], [3]]]

sage: SST = SemistandardTableaux(3, max_entry=2); SST
Semistandard tableaux of size 3 and maximum entry 2
sage: SST.list()
[[[1, 1, 1]],
```

```
 [[1, 1, 2]],
 [[1, 2, 2]],
 [[2, 2, 2]],
 [[1, 1], [2]],
 [[1, 2], [2]]]

sage: SST = SemistandardTableaux(3, max_entry=oo); SST
Semistandard tableaux of size 3
sage: SST[123]
[[3, 4], [6]]

sage: SemistandardTableaux(max_entry=2)[11]
[[1, 1], [2]]

sage: SemistandardTableaux()[0]
[]
```

Element
alias of SemistandardTableau

class sage.combinat.tableau.**SemistandardTableaux_all**(*max_entry=None*)
Bases: sage.combinat.tableau.SemistandardTableaux, sage.sets.disjoint_union_enumerated_set

All semistandard tableaux.

Warning: Input is not checked; please use SemistandardTableaux to ensure the options are properly parsed.

list()
TESTS:

```
sage: SemistandardTableaux().list()
Traceback (most recent call last):
...
NotImplementedError
```

class sage.combinat.tableau.**SemistandardTableaux_shape**(*p*, *max_entry=None*)
Bases: sage.combinat.tableau.SemistandardTableaux

Semistandard tableaux of fixed shape p with a given max entry.

A semistandard tableau with max entry i is required to have all its entries less or equal to i. It is not required to actually contain an entry i.

INPUT:

- p – A partition
- max_entry – The max entry; defaults to the size of p.

Warning: Input is not checked; please use SemistandardTableaux to ensure the options are properly parsed.

cardinality(*algorithm='hook'*)
Return the cardinality of self.

INPUT:

- algorithm – (default: 'hook') any one of the following:

- `'hook'` – use Stanley's hook length formula
- `'sum'` – sum over the compositions of `max_entry` the number of semistandard tableau with `shape` and given weight vector

This is computed using *Stanley's hook length formula*:

$$f_\lambda = \prod_{u \in \lambda} \frac{n + c(u)}{h(u)}.$$

where n is the `max_entry`, $c(u)$ is the content of u, and $h(u)$ is the hook length of u. See [Sta-EC2] Corollary 7.21.4.

EXAMPLES:

```
sage: SemistandardTableaux([2,1]).cardinality()
8
sage: SemistandardTableaux([2,2,1]).cardinality()
75
sage: SymmetricFunctions(QQ).schur()([2,2,1]).expand(5)(1,1,1,1,1) # cross check
75
sage: SemistandardTableaux([5]).cardinality()
126
sage: SemistandardTableaux([3,2,1]).cardinality()
896
sage: SemistandardTableaux([3,2,1], max_entry=7).cardinality()
2352
sage: SemistandardTableaux([6,5,4,3,2,1], max_entry=30).cardinality()
208361017592001331200
sage: ssts = [SemistandardTableaux(p, max_entry=6) for p in Partitions(5)]
sage: all(sst.cardinality() == sst.cardinality(algorithm='sum')
....:     for sst in ssts)
True
```

random_element()

Return a uniformly distributed random tableau of the given `shape` and `max_entry`.

Uses the algorithm from [Krat99] based on the Novelli-Pak-Stoyanovskii bijection

EXAMPLES:

```
sage: SemistandardTableaux([2, 2, 1, 1]).random_element()
[[1, 1], [2, 3], [3], [5]]
sage: SemistandardTableaux([2, 2, 1, 1], max_entry=7).random_element()
[[1, 4], [5, 5], [6], [7]]
```

REFERENCES:

class sage.combinat.tableau.**SemistandardTableaux_shape_inf**(*p*)

Bases: sage.combinat.tableau.SemistandardTableaux

Semistandard tableaux of fixed shape p and no maximum entry.

Warning: Input is not checked; please use `SemistandardTableaux` to ensure the options are properly parsed.

class sage.combinat.tableau.**SemistandardTableaux_shape_weight**(*p*, *mu*)

Bases: sage.combinat.tableau.SemistandardTableaux_shape

Semistandard tableaux of fixed shape p and weight μ.

Warning: Input is not checked; please use SemistandardTableaux to ensure the options are properly parsed.

cardinality()

Returns the number of semistandard tableaux of the given shape and weight, as computed by kostka_number function of symmetrica.

EXAMPLES:

```
sage: SemistandardTableaux([2,2], [2, 1, 1]).cardinality()
1
sage: SemistandardTableaux([2,2,2], [2, 2, 1,1]).cardinality()
1
sage: SemistandardTableaux([2,2,2], [2, 2, 2]).cardinality()
1
sage: SemistandardTableaux([3,2,1], [2, 2, 2]).cardinality()
2
```

list()

Return a list of all semistandard tableaux in self generated by symmetrica.

EXAMPLES:

```
sage: SemistandardTableaux([2,2], [2, 1, 1]).list()
[[[1, 1], [2, 3]]]
sage: SemistandardTableaux([2,2,2], [2, 2, 1,1]).list()
[[[1, 1], [2, 2], [3, 4]]]
sage: SemistandardTableaux([2,2,2], [2, 2, 2]).list()
[[[1, 1], [2, 2], [3, 3]]]
sage: SemistandardTableaux([3,2,1], [2, 2, 2]).list()
[[[1, 1, 2], [2, 3], [3]], [[1, 1, 3], [2, 2], [3]]]
```

class sage.combinat.tableau.**SemistandardTableaux_size**(*n*, *max_entry=None*)

Bases: sage.combinat.tableau.SemistandardTableaux

Semistandard tableaux of fixed size n.

Warning: Input is not checked; please use SemistandardTableaux to ensure the options are properly parsed.

cardinality()

Return the cardinality of self.

EXAMPLES:

```
sage: SemistandardTableaux(3).cardinality()
19
sage: SemistandardTableaux(4).cardinality()
116
sage: SemistandardTableaux(4, max_entry=2).cardinality()
9
sage: SemistandardTableaux(4, max_entry=10).cardinality()
4225
sage: ns = range(1, 6)
sage: ssts = [ SemistandardTableaux(n) for n in ns ]
sage: all([sst.cardinality() == len(sst.list()) for sst in ssts])
True
```

random_element()

Generate a random SemistandardTableau with uniform probability.

The RSK algorithm gives a bijection between symmetric $k \times k$ matrices of nonnegative integers that sum to n and semistandard tableaux with size n and maximum entry k.

The number of $k \times k$ symmetric matrices of nonnegative integers having sum of elements on the diagonal i and sum of elements above the diagonal j is $\binom{k+i-1}{k-1}\binom{\binom{k}{2}+j-1}{\binom{k}{2}-1}$. We first choose the sum of the elements on the diagonal randomly weighted by the number of matrices having that trace. We then create random integer vectors of length k having that sum and use them to generate a $k \times k$ diagonal matrix. Then we take a random integer vector of length $\binom{k}{2}$ summing to half the remainder and distribute it symmetrically to the remainder of the matrix.

Applying RSK to the random symmetric matrix gives us a pair of identical SemistandardTableau of which we choose the first.

EXAMPLES:

```
sage: SemistandardTableaux(6).random_element() # random
[[1, 1, 2], [3, 5, 5]]
sage: SemistandardTableaux(6, max_entry=7).random_element() # random
[[2, 4, 4, 6, 6, 6]]
```

class sage.combinat.tableau.**SemistandardTableaux_size_inf**(n)

Bases: sage.combinat.tableau.SemistandardTableaux

Semistandard tableaux of fixed size n with no maximum entry.

Warning: Input is not checked; please use SemistandardTableaux to ensure the options are properly parsed.

list()

TESTS:

```
sage: SemistandardTableaux(3, max_entry=oo).list()
Traceback (most recent call last):
...
NotImplementedError
```

class sage.combinat.tableau.**SemistandardTableaux_size_weight**(n, mu)

Bases: sage.combinat.tableau.SemistandardTableaux

Semistandard tableaux of fixed size n and weight μ.

Warning: Input is not checked; please use SemistandardTableaux to ensure the options are properly parsed.

cardinality()

Return the cardinality of self.

EXAMPLES:

```
sage: SemistandardTableaux(3, [2,1]).cardinality()
2
sage: SemistandardTableaux(4, [2,2]).cardinality()
3
```

class sage.combinat.tableau.**StandardTableau**(*parent*, *t*)

Bases: sage.combinat.tableau.SemistandardTableau

A class to model a standard tableau.

INPUT:

- `t` – a Tableau, a list of iterables, or an empty list

OUTPUT:

- A StandardTableau object constructed from `t`.

A standard tableau is a semistandard tableau whose entries are exactly the positive integers from 1 to n, where n is the size of the tableau.

EXAMPLES:

```
sage: t = StandardTableau([[1,2,3],[4,5]]); t
[[1, 2, 3], [4, 5]]
sage: t.shape()
[3, 2]
sage: t.pp() # pretty print
1 2 3
4 5
sage: t.is_standard()
True
sage: StandardTableau([]) # The empty tableau
[]
```

When using code that will generate a lot of tableaux, it is slightly more efficient to construct a StandardTableau from the appropriate `Parent` object:

```
sage: ST = StandardTableaux()
sage: ST([[1, 2, 3], [4, 5]])
[[1, 2, 3], [4, 5]]
```

content (*k, multicharge=[0]*)

Returns the content of `k` in a standard tableau. That is, if `k` appears in row r and column c of the tableau then we return $c - r$.

The `multicharge` is a list of length 1 which gives an offset for all of the contents. It is included mainly for compatibility with `TableauTuple`.

EXAMPLES:

```
sage: StandardTableau([[1,2],[3,4]]).content(3)
-1

sage: StandardTableau([[1,2],[3,4]]).content(6)
Traceback (most recent call last):
...
ValueError: 6 does not appear in tableau
```

dominates (*t*)

Return `True` if `self` dominates the tableau `t`. That is, if the shape of the tableau restricted to k dominates the shape of `t` restricted to k, for $k = 1, 2, \ldots, n$.

When the two tableaux have the same shape, then this ordering coincides with the Bruhat ordering for the corresponding permutations.

INPUT:

- `t` – A tableau

EXAMPLES:

```
sage: s=StandardTableau([[1,2,3],[4,5]])
sage: t=StandardTableau([[1,2],[3,5],[4]])
sage: s.dominates(t)
True
sage: t.dominates(s)
False
sage: all(StandardTableau(s).dominates(t) for t in StandardTableaux([3,2]))
True
sage: s.dominates([[1,2,3,4,5]])
False
```

down()

An iterator for all the standard tableaux that can be obtained from `self` by removing a cell. Note that this iterates just over a single tableau (or nothing if `self` is empty).

EXAMPLES:

```
sage: t = StandardTableau([[1,2],[3]])
sage: [x for x in t.down()]
[[[1, 2]]]
sage: t = StandardTableau([])
sage: [x for x in t.down()]
[]
```

down_list()

Return a list of all the standard tableaux that can be obtained from `self` by removing a cell. Note that this is just a singleton list if `self` is nonempty, and an empty list otherwise.

EXAMPLES:

```
sage: t = StandardTableau([[1,2],[3]])
sage: t.down_list()
[[[1, 2]]]
sage: t = StandardTableau([])
sage: t.down_list()
[]
```

is_standard()

Return `True` since `self` is a standard tableau.

EXAMPLES:

```
sage: StandardTableau([[1, 3], [2, 4]]).is_standard()
True
```

promotion(*n=None*)

Return the image of `self` under the promotion operator.

The promotion operator, applied to a standard tableau t, does the following:

Remove the letter n from t, thus leaving a hole where it used to be. Apply jeu de taquin to move this hole southwest (in French notation) until it reaches the inner boundary of t. Fill 0 into the hole once jeu de taquin has completed. Finally, add 1 to each letter in the tableau. The resulting standard tableau is the image of t under the promotion operator.

This definition of promotion is precisely the one given in [Hai1992] (p. 90). It is the inverse of the maps called "promotion" in [Sg2011] (p. 23) and in [Stan2009].

See the `promotion()` method for a more general operator.

EXAMPLES:

```
sage: ST = StandardTableaux(7)
sage: all( st.promotion().promotion_inverse() == st for st in ST ) # long time
True
sage: all( st.promotion_inverse().promotion() == st for st in ST ) # long time
True
sage: st = StandardTableau([[1,2,5],[3,4]])
sage: parent(st.promotion())
Standard tableaux
```

promotion_inverse(*n=None*)

Return the image of `self` under the inverse promotion operator. The optional variable m should be set to the size of `self` minus 1 for a minimal speedup; otherwise, it defaults to this number.

The inverse promotion operator, applied to a standard tableau t, does the following:

Remove the letter 1 from t, thus leaving a hole where it used to be. Apply jeu de taquin to move this hole northeast (in French notation) until it reaches the outer boundary of t. Fill $n+1$ into this hole, where n is the size of t. Finally, subtract 1 from each letter in the tableau. This yields a new standard tableau.

This definition of inverse promotion is the map called "promotion" in [Sg2011] (p. 23) and in [Stan2009], and is the inverse of the map called "promotion" in [Hai1992] (p. 90).

See the `promotion_inverse()` method for a more general operator.

EXAMPLES:

```
sage: t = StandardTableau([[1,3],[2,4]])
sage: t.promotion_inverse()
[[1, 2], [3, 4]]
```

We check the equivalence of two definitions of inverse promotion on standard tableaux:

```
sage: ST = StandardTableaux(7)
sage: def bk_promotion_inverse7(st):
....:     st2 = st
....:     for i in range(1, 7):
....:         st2 = st2.bender_knuth_involution(i, check=False)
....:     return st2
sage: all( bk_promotion_inverse7(st) == st.promotion_inverse() for st in ST ) # long time
True
```

standard_descents()

Return a list of the integers i such that i appears strictly further north than $i+1$ in `self` (this is not to say that i and $i+1$ must be in the same column). The list is sorted in increasing order.

EXAMPLES:

```
sage: StandardTableau( [[1,3,4],[2,5]] ).standard_descents()
[1, 4]
sage: StandardTableau( [[1,2],[3,4]] ).standard_descents()
[2]
sage: StandardTableau( [[1,2,5],[3,4],[6,7],[8],[9]] ).standard_descents()
[2, 5, 7, 8]
sage: StandardTableau( [] ).standard_descents()
[]
```

standard_major_index()

Return the major index of the standard tableau `self` in the standard meaning of the word. The major index is defined to be the sum of the descents of `self` (see `standard_descents()` for their definition).

EXAMPLES:

```
sage: StandardTableau( [[1,4,5],[2,6],[3]] ).standard_major_index()
8
sage: StandardTableau( [[1,2],[3,4]] ).standard_major_index()
2
sage: StandardTableau( [[1,2,3],[4,5]] ).standard_major_index()
3
```

standard_number_of_descents()

Return the number of all integers i such that i appears strictly further north than $i+1$ in `self` (this is not to say that i and $i+1$ must be in the same column). A list of these integers can be obtained using the `standard_descents()` method.

EXAMPLES:

```
sage: StandardTableau( [[1,2],[3,4],[5]] ).standard_number_of_descents()
2
sage: StandardTableau( [] ).standard_number_of_descents()
0
sage: tabs = StandardTableaux(5)
sage: all( t.standard_number_of_descents() == t.schuetzenberger_involution().standard_number
True
```

up()

An iterator for all the standard tableaux that can be obtained from `self` by adding a cell.

EXAMPLES:

```
sage: t = StandardTableau([[1,2]])
sage: [x for x in t.up()]
[[[1, 2, 3]], [[1, 2], [3]]]
```

up_list()

Return a list of all the standard tableaux that can be obtained from `self` by adding a cell.

EXAMPLES:

```
sage: t = StandardTableau([[1,2]])
sage: t.up_list()
[[[1, 2, 3]], [[1, 2], [3]]]
```

class sage.combinat.tableau.**StandardTableaux**(***kwds*)

Bases: sage.combinat.tableau.SemistandardTableaux

A factory for the various classes of standard tableaux.

INPUT:

- Either a non-negative integer (possibly specified with the keyword `n`) or a partition.

OUTPUT:

- With no argument, the class of all standard tableaux
- With a non-negative integer argument, `n`, the class of all standard tableaux of size `n`
- With a partition argument, the class of all standard tableaux of that shape.

A standard tableau is a semistandard tableaux which contains each of the entries from 1 to `n` exactly once.

All classes of standard tableaux are iterable.

EXAMPLES:

```
sage: ST = StandardTableaux(3); ST
Standard tableaux of size 3
sage: ST.first()
[[1, 2, 3]]
sage: ST.last()
[[1], [2], [3]]
sage: ST.cardinality()
4
sage: ST.list()
[[[1, 2, 3]], [[1, 3], [2]], [[1, 2], [3]], [[1], [2], [3]]]
```

TESTS:

```
sage: StandardTableaux()([])
[]
sage: ST = StandardTableaux([2,2]); ST
Standard tableaux of shape [2, 2]
sage: ST.first()
[[1, 3], [2, 4]]
sage: ST.last()
[[1, 2], [3, 4]]
sage: ST.cardinality()
2
sage: ST.list()
[[[1, 3], [2, 4]], [[1, 2], [3, 4]]]
```

Element
alias of StandardTableau

class sage.combinat.tableau.**StandardTableaux_all**
Bases: sage.combinat.tableau.StandardTableaux, sage.sets.disjoint_union_enumerated_sets.Di

All standard tableaux.

class sage.combinat.tableau.**StandardTableaux_shape**(*p*)
Bases: sage.combinat.tableau.StandardTableaux

Semistandard tableaux of a fixed shape p.

Warning: Input is not checked; please use SemistandardTableaux to ensure the options are properly parsed.

cardinality()
Return the number of standard Young tableaux of this shape.

This method uses the so-called *hook length formula*, a formula for the number of Young tableaux associated with a given partition. The formula says the following: Let λ be a partition. For each cell c of the Young diagram of λ, let the *hook length* of c be defined as 1 plus the number of cells horizontally to the right of c plus the number of cells vertically below c. The number of standard Young tableaux of shape λ is then $n!$ divided by the product of the hook lengths of the shape of λ, where $n = |\lambda|$.

For example, consider the partition [3,2,1] of 6 with Ferrers diagram:

```
# # #
# #
#
```

When we fill in the cells with their respective hook lengths, we obtain:

```
5 3 1
3 1
1
```

The hook length formula returns

$$\frac{6!}{5 \cdot 3 \cdot 1 \cdot 3 \cdot 1 \cdot 1} = 16.$$

EXAMPLES:

```
sage: StandardTableaux([3,2,1]).cardinality()
16
sage: StandardTableaux([2,2]).cardinality()
2
sage: StandardTableaux([5]).cardinality()
1
sage: StandardTableaux([6,5,5,3]).cardinality()
6651216
sage: StandardTableaux([]).cardinality()
1
```

REFERENCES:

- http://mathworld.wolfram.com/HookLengthFormula.html

list()

Return a list of the standard Young tableaux of the specified shape.

EXAMPLES:

```
sage: StandardTableaux([2,2]).list()
[[[1, 3], [2, 4]], [[1, 2], [3, 4]]]
sage: StandardTableaux([5]).list()
[[[1, 2, 3, 4, 5]]]
sage: StandardTableaux([3,2,1]).list()
[[[1, 4, 6], [2, 5], [3]],
 [[1, 3, 6], [2, 5], [4]],
 [[1, 2, 6], [3, 5], [4]],
 [[1, 3, 6], [2, 4], [5]],
 [[1, 2, 6], [3, 4], [5]],
 [[1, 4, 5], [2, 6], [3]],
 [[1, 3, 5], [2, 6], [4]],
 [[1, 2, 5], [3, 6], [4]],
 [[1, 3, 4], [2, 6], [5]],
 [[1, 2, 4], [3, 6], [5]],
 [[1, 2, 3], [4, 6], [5]],
 [[1, 3, 5], [2, 4], [6]],
 [[1, 2, 5], [3, 4], [6]],
 [[1, 3, 4], [2, 5], [6]],
 [[1, 2, 4], [3, 5], [6]],
 [[1, 2, 3], [4, 5], [6]]]
```

random_element()

Return a random standard tableau of the given shape using the Greene-Nijenhuis-Wilf Algorithm.

EXAMPLES:

```
sage: StandardTableaux([2,2]).random_element()
[[1, 2], [3, 4]]
sage: StandardTableaux([]).random_element()
[]
```

class sage.combinat.tableau.**StandardTableaux_size**(*n*)

Bases: sage.combinat.tableau.StandardTableaux

Standard tableaux of fixed size n.

Warning: Input is not checked; please use StandardTableaux to ensure the options are properly parsed.

cardinality()

Return the number of all standard tableaux of size n.

The number of standard tableaux of size n is equal to the number of involutions in the symmetric group S_n. This is a consequence of the symmetry of the RSK correspondence, that if $\sigma \mapsto (P, Q)$, then $\sigma^{-1} \mapsto (Q, P)$. For more information, see Wikipedia article Robinson-Schensted-Knuth_correspondence#Symmetry.

ALGORITHM:

The algorithm uses the fact that standard tableaux of size n are in bijection with the involutions of size n, (see page 41 in section 4.1 of [Ful1997]). For each number of fixed points, you count the number of ways to choose those fixed points multiplied by the number of perfect matchings on the remaining values.

REFERENCES:

EXAMPLES:

```
sage: StandardTableaux(3).cardinality()
4
sage: ns = [1,2,3,4,5,6]
sage: sts = [StandardTableaux(n) for n in ns]
sage: all([st.cardinality() == len(st.list()) for st in sts])
True
sage: StandardTableaux(50).cardinality()
278869956053423428391046158692597776
```

TESTS:

```
sage: def cardinality_using_hook_formula(n):
....:     c = 0
....:     for p in Partitions(n):
....:         c += StandardTableaux(p).cardinality()
....:     return c
sage: all([cardinality_using_hook_formula(i) == StandardTableaux(i).cardinality() for i in r
True
```

random_element()

Return a random StandardTableau with uniform probability.

This algorithm uses the fact that the Robinson-Schensted correspondence returns a pair of identical standard Young tableaux (SYTs) if and only if the permutation was an involution. Thus, generating a random SYT is equivalent to generating a random involution.

To generate an involution, we first need to choose its number of fixed points k (if the size of the involution is even, the number of fixed points will be even, and if the size is odd, the number of fixed points will be odd). To do this, we choose a random integer r between 0 and the number N of all involutions of size n. We then decompose the interval $\{1, 2, \ldots, N\}$ into subintervals whose lengths are the numbers of involutions of size n with respectively 0, 1, ..., $\lfloor N/2 \rfloor$ fixed points. The interval in which our random integer r lies then decides how many fixed points our random involution will have. We then place those

fixed points randomly and then compute a perfect matching (an involution without fixed points) on the remaining values.

EXAMPLES:

```
sage: StandardTableaux(5).random_element() # random
[[1, 4, 5], [2], [3]]
sage: StandardTableaux(0).random_element()
[]
sage: StandardTableaux(1).random_element()
[[1]]
```

TESTS:

```
sage: all([StandardTableaux(10).random_element() in StandardTableaux(10) for i in range(20)]
True
```

class sage.combinat.tableau.**Tableau**(*parent*, *t*)

Bases: sage.structure.list_clone.ClonableList

A class to model a tableau.

INPUT:

- t – a Tableau, a list of iterables, or an empty list

OUTPUT:

- A Tableau object constructed from t.

A tableau is abstractly a mapping from the cells in a partition to arbitrary objects (called entries). It is often represented as a finite list of nonempty lists (or generally an iterable of iterables) of weakly decreasing lengths. This list, in particular, can be empty, representing the empty tableau.

Note that Sage uses the English convention for partitions and tableaux; the longer rows are displayed on top.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[4,5]]); t
[[1, 2, 3], [4, 5]]
sage: t.shape()
[3, 2]
sage: t.pp() # pretty print
1 2 3
4 5
sage: t.is_standard()
True

sage: Tableau([['a','c','b'],[],(2,1)]])
[['a', 'c', 'b'], [], (2, 1)]]
sage: Tableau([]) # The empty tableau
[]
```

When using code that will generate a lot of tableaux, it is slightly more efficient to construct a Tableau from the appropriate Parent object:

```
sage: T = Tableaux()
sage: T([[1, 2, 3], [4, 5]])
[[1, 2, 3], [4, 5]]
```

TESTS:

```
sage: Tableau([[1],[2,3]])
Traceback (most recent call last):
...
ValueError: A tableau must be a list of iterables of weakly decreasing length.
sage: Tableau([1,2,3])
Traceback (most recent call last):
...
ValueError: A tableau must be a list of iterables.
```

add_entry(*cell*, *m*)

Return the result of setting the entry in cell `cell` equal to `m` in the tableau `self`.

This tableau has larger size than `self` if `cell` does not belong to the shape of `self`; otherwise, the tableau has the same shape as `self` and has the appropriate entry replaced.

INPUT:

- `cell` – a pair of nonnegative integers

OUTPUT:

The tableau `self` with the entry in cell `cell` set to `m`. This entry overwrites an existing entry if `cell` already belongs to `self`, or is added to the tableau if `cell` is a cocorner of the shape `self`. (Either way, the input is not modified.)

Note: Both coordinates of `cell` are interpreted as starting at 0. So, `cell == (0, 0)` corresponds to the northwesternmost cell.

EXAMPLES:

```
sage: s=StandardTableau([[1,2,5],[3,4]]); s.pp()
  1  2  5
  3  4
sage: t=s.add_entry( (1,2), 6); t.pp()
  1  2  5
  3  4  6
sage: t.category()
Category of elements of Standard tableaux
sage: s.add_entry( (2,0), 6).pp()
  1  2  5
  3  4
  6
sage: u=s.add_entry( (1,2), 3); u.pp()
  1  2  5
  3  4  3
sage: u.category()
Category of elements of Tableaux
sage: s.add_entry( (2,2),3)
Traceback (most recent call last):
...
IndexError: (2, 2) is not an addable cell of the tableau
```

anti_restrict(*n*)

Return the skew tableau formed by removing all of the cells from `self` that are filled with a number at most n.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[4,5]]); t
[[1, 2, 3], [4, 5]]
```

```
sage: t.anti_restrict(1)
[[None, 2, 3], [4, 5]]
sage: t.anti_restrict(2)
[[None, None, 3], [4, 5]]
sage: t.anti_restrict(3)
[[None, None, None], [4, 5]]
sage: t.anti_restrict(4)
[[None, None, None], [None, 5]]
sage: t.anti_restrict(5)
[[None, None, None], [None, None]]
```

atom()

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).atom()
[2, 2]
sage: Tableau([[1,2,3],[4,5],[6]]).atom()
[3, 2, 1]
```

bender_knuth_involution(*k*, *rows=None*, *check=True*)

Return the image of `self` under the k-th Bender–Knuth involution, assuming `self` is a semistandard tableau.

Let T be a tableau, then a *lower free 'k' in 'T'* means a cell of T which is filled with the integer k and whose direct lower neighbor is not filled with the integer $k+1$ (in particular, this lower neighbor might not exist at all). Let an *upper free 'k + 1' in 'T'* mean a cell of T which is filled with the integer $k+1$ and whose direct upper neighbor is not filled with the integer k (in particular, this neighbor might not exist at all). It is clear that for any row r of T, the lower free k's and the upper free $k+1$'s in r together form a contiguous interval or r.

The *'k'-th Bender–Knuth switch at row 'i'* changes the entries of the cells in this interval in such a way that if it used to have a entries of k and b entries of $k+1$, it will now have b entries of k and a entries of $k+1$. For fixed k, the k-th Bender–Knuth switches for different i commute. The composition of the k-th Bender–Knuth switches for all rows is called the *'k'-th Bender-Knuth involution*. This is used to show that the Schur functions defined by semistandard tableaux are symmetric functions.

INPUT:

- `k` – an integer
- `rows` – (Default `None`) When set to `None`, the method computes the k-th Bender–Knuth involution as defined above. When an iterable, this computes the composition of the k-th Bender–Knuth switches at row i over all i in `rows`. When set to an integer i, the method computes the k-th Bender–Knuth switch at row i. Note the indexing of the rows starts with 1.
- `check` – (Default: `True`) Check to make sure `self` is semistandard. Set to `False` to avoid this check.

OUTPUT:

The image of `self` under either the k-th Bender–Knuth involution, the k-th Bender–Knuth switch at a certain row, or the composition of such switches, as detailed in the INPUT section.

EXAMPLES:

```
sage: t = Tableau([[1,1,3,4,4,5,6,7],[2,2,4,6,7,7,7],[3,4,5,8,8,9],[6,6,7,10],[7,8,8,11],[8]
sage: t.bender_knuth_involution(1) == t
True
sage: t.bender_knuth_involution(2)
[[1, 1, 2, 4, 4, 5, 6, 7], [2, 3, 4, 6, 7, 7, 7], [3, 4, 5, 8, 8, 9], [6, 6, 7, 10], [7, 8,
sage: t.bender_knuth_involution(3)
```

```
[[1, 1, 3, 3, 3, 5, 6, 7], [2, 2, 4, 6, 7, 7, 7], [3, 4, 5, 8, 8, 9], [6, 6, 7, 10], [7, 8,
sage: t.bender_knuth_involution(4)
[[1, 1, 3, 4, 5, 5, 6, 7], [2, 2, 4, 6, 7, 7, 7], [3, 5, 5, 8, 8, 9], [6, 6, 7, 10], [7, 8,
sage: t.bender_knuth_involution(5)
[[1, 1, 3, 4, 4, 5, 6, 7], [2, 2, 4, 5, 7, 7, 7], [3, 4, 6, 8, 8, 9], [5, 5, 7, 10], [7, 8,
sage: t.bender_knuth_involution(666) == t
True
sage: t.bender_knuth_involution(4, 2) == t
True
sage: t.bender_knuth_involution(4, 3)
[[1, 1, 3, 4, 4, 5, 6, 7], [2, 2, 4, 6, 7, 7, 7], [3, 5, 5, 8, 8, 9], [6, 6, 7, 10], [7, 8,
```

The `rows` keyword can be an iterator:

```
sage: t.bender_knuth_involution(6, iter([1,2])) == t
False
sage: t.bender_knuth_involution(6, iter([3,4])) == t
True
```

The Bender–Knuth involution is an involution:

```
sage: T = SemistandardTableaux(shape=[3,1,1], max_entry=4)
sage: all(all(t.bender_knuth_involution(k).bender_knuth_involution(k) == t for k in range(1,
True
```

The same holds for the single switches:

```
sage: all(all(t.bender_knuth_involution(k, j).bender_knuth_involution(k, j) == t for k in ra
True
```

Locality of the Bender–Knuth involutions:

```
sage: all(all(t.bender_knuth_involution(k).bender_knuth_involution(l) == t.bender_knuth_invo
True
```

Berenstein and Kirillov [BerKilGGI] have shown that $(s_1 s_2)^6 = id$ (for tableaux of straight shape):

```
sage: p = lambda t, k: t.bender_knuth_involution(k).bender_knuth_involution(k + 1)
sage: all(p(p(p(p(p(p(t,1),1),1),1),1),1) == t for t in T)
True
```

However, $(s_2 s_3)^6 = id$ is false:

```
sage: p = lambda t, k: t.bender_knuth_involution(k).bender_knuth_involution(k + 1)
sage: t = Tableau([[1,2,2],[3,4]])
sage: x = t
sage: for i in range(6): x = p(x, 2)
sage: x
[[1, 2, 3], [2, 4]]
sage: x == t
False
```

TESTS:

```
sage: t = Tableau([])
sage: t.bender_knuth_involution(3)
[]
```

REFERENCES:

bump (x)

Insert `x` into `self` using Schensted's row-bumping (or row-insertion) algorithm.

EXAMPLES:

```
sage: t = Tableau([[1,2],[3]])
sage: t.bump(1)
[[1, 1], [2], [3]]
sage: t
[[1, 2], [3]]
sage: t.bump(2)
[[1, 2, 2], [3]]
sage: t.bump(3)
[[1, 2, 3], [3]]
sage: t
[[1, 2], [3]]
sage: t = Tableau([[1,2,2,3],[2,3,5,5],[4,4,6],[5,6]])
sage: t.bump(2)
[[1, 2, 2, 2], [2, 3, 3, 5], [4, 4, 5], [5, 6, 6]]
sage: t.bump(1)
[[1, 1, 2, 3], [2, 2, 5, 5], [3, 4, 6], [4, 6], [5]]
```

bump_multiply(*left*, *right*)

Multiply two tableaux using Schensted's bump.

This product makes the set of semistandard tableaux into an associative monoid. The empty tableau is the unit in this monoid. See pp. 11-12 of [Ful1997].

The same product operation is implemented in a different way in `slide_multiply()`.

EXAMPLES:

```
sage: t = Tableau([[1,2,2,3],[2,3,5,5],[4,4,6],[5,6]])
sage: t2 = Tableau([[1,2],[3]])
sage: t.bump_multiply(t2)
[[1, 1, 2, 2, 3], [2, 2, 3, 5], [3, 4, 5], [4, 6, 6], [5]]
```

catabolism()

Remove the top row of `self` and insert it back in using column Schensted insertion (starting with the largest letter).

EXAMPLES:

```
sage: Tableau([]).catabolism()
[]
sage: Tableau([[1,2,3,4,5]]).catabolism()
[[1, 2, 3, 4, 5]]
sage: Tableau([[1,1,3,3],[2,3],[3]]).catabolism()
[[1, 1, 2, 3, 3, 3], [3]]
sage: Tableau([[1, 1, 2, 3, 3, 3], [3]]).catabolism()
[[1, 1, 2, 3, 3, 3, 3]]
```

catabolism_projector(*parts*)

EXAMPLES:

```
sage: t = Tableau([[1,1,3,3],[2,3],[3]])
sage: t.catabolism_projector([[4,2,1]])
[[1, 1, 3, 3], [2, 3], [3]]
sage: t.catabolism_projector([[1]])
[]
sage: t.catabolism_projector([[2,1],[1]])
[]
```

```
sage: t.catabolism_projector([[1,1],[4,1]])
[[1, 1, 3, 3], [2, 3], [3]]
```

catabolism_sequence()

Perform `catabolism()` on `self` until it returns a tableau consisting of a single row.

EXAMPLES:

```
sage: t = Tableau([[1,2,3,4,5,6,8],[7,9]])
sage: t.catabolism_sequence()
[[[1, 2, 3, 4, 5, 6, 8], [7, 9]],
 [[1, 2, 3, 4, 5, 6, 7, 9], [8]],
 [[1, 2, 3, 4, 5, 6, 7, 8], [9]],
 [[1, 2, 3, 4, 5, 6, 7, 8, 9]]]
sage: Tableau([]).catabolism_sequence()
[[]]
```

cells()

Return a list of the coordinates of the cells of `self`.

Coordinates start at 0, so the northwesternmost cell (in English notation) has coordinates $(0, 0)$.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).cells()
[(0, 0), (0, 1), (1, 0), (1, 1)]
```

cells_containing(i)

Return the list of cells in which the letter i appears in the tableau `self`. The list is ordered with cells appearing from left to right.

Cells are given as pairs of coordinates (a, b), where both rows and columns are counted from 0 (so $a = 0$ means the cell lies in the leftmost column of the tableau, etc.).

EXAMPLES:

```
sage: t = Tableau([[1,1,3],[2,3,5],[4,5]])
sage: t.cells_containing(5)
[(2, 1), (1, 2)]
sage: t.cells_containing(4)
[(2, 0)]
sage: t.cells_containing(6)
[]

sage: t = Tableau([[1,1,2,4],[2,4,4],[4]])
sage: t.cells_containing(4)
[(2, 0), (1, 1), (1, 2), (0, 3)]

sage: t = Tableau([[1,1,2,8,9],[2,5,6,11],[3,7,7,13],[4,8,9],[5],[13],[14]])
sage: t.cells_containing(8)
[(3, 1), (0, 3)]

sage: Tableau([]).cells_containing(3)
[]
```

charge()

Return the charge of the reading word of `self`. See `charge()` for more information.

EXAMPLES:

```
sage: Tableau([[1,1],[2,2],[3]]).charge()
0
sage: Tableau([[1,1,3],[2,2]]).charge()
1
sage: Tableau([[1,1,2],[2],[3]]).charge()
1
sage: Tableau([[1,1,2],[2,3]]).charge()
2
sage: Tableau([[1,1,2,3],[2]]).charge()
2
sage: Tableau([[1,1,2,2],[3]]).charge()
3
sage: Tableau([[1,1,2,2,3]]).charge()
4
```

check()

Check that `self` is a valid straight-shape tableau.

EXAMPLES:

```
sage: t = Tableau([[1,1],[2]])
sage: t.check()

sage: t = Tableau([[None, None, 1], [2, 4], [3, 4, 5]])
Traceback (most recent call last):
...
ValueError: A tableau must be a list of iterables of weakly decreasing length.
```

cocharge()

Return the cocharge of the reading word of `self`. See `cocharge()` for more information.

EXAMPLES:

```
sage: Tableau([[1,1],[2,2],[3]]).cocharge()
4
sage: Tableau([[1,1,3],[2,2]]).cocharge()
3
sage: Tableau([[1,1,2],[2],[3]]).cocharge()
3
sage: Tableau([[1,1,2],[2,3]]).cocharge()
2
sage: Tableau([[1,1,2,3],[2]]).cocharge()
2
sage: Tableau([[1,1,2,2],[3]]).cocharge()
1
sage: Tableau([[1,1,2,2,3]]).cocharge()
0
```

column_stabilizer()

Return the PermutationGroup corresponding to the column stabilizer of `self`.

This assumes that every integer from 1 to the size of `self` appears exactly once in `self`.

EXAMPLES:

```
sage: cs = Tableau([[1,2,3],[4,5]]).column_stabilizer()
sage: cs.order() == factorial(2)*factorial(2)
True
sage: PermutationGroupElement([(1,3,2),(4,5)]) in cs
False
```

```
sage: PermutationGroupElement([(1,4)]) in cs
True
```

components()

This function returns a list containing itself. It exists mainly for compatibility with `TableauTuple` as it allows constructions like the example below.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[4,5]]);
sage: for s in t.components(): print s.to_list()
[[1, 2, 3], [4, 5]]
```

conjugate()

Return the conjugate of `self`.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).conjugate()
[[1, 3], [2, 4]]
sage: c = StandardTableau([[1,2],[3,4]]).conjugate()
sage: c.parent()
Standard tableaux
```

corners()

Return the corners of the tableau `self`.

EXAMPLES:

```
sage: Tableau([[1, 4, 6], [2, 5], [3]]).corners()
[(0, 2), (1, 1), (2, 0)]
sage: Tableau([[1, 3], [2, 4]]).corners()
[(1, 1)]
```

descents()

Return a list of the cells `(i,j)` such that `self[i][j] > self[i-1][j]`.

Warning: This is not to be confused with the descents of a standard tableau.

EXAMPLES:

```
sage: Tableau( [[1,4],[2,3]] ).descents()
[(1, 0)]
sage: Tableau( [[1,2],[3,4]] ).descents()
[(1, 0), (1, 1)]
sage: Tableau( [[1,2,3],[4,5]] ).descents()
[(1, 0), (1, 1)]
```

entries()

Return the tuple of all entries of `self`, in the order obtained by reading across the rows from top to bottom (in English notation).

EXAMPLES:

```
sage: t = Tableau([[1,3], [2]])
sage: t.entries()
(1, 3, 2)
```

entry(*cell*)

Returns the entry of cell `cell` in the tableau `self`. Here, `cell` should be given as a tuple (i, j) of

zero-based coordinates (so the northwesternmost cell in English notation is $(0, 0)$).

EXAMPLES:

```
sage: t = Tableau([[1,2],[3,4]])
sage: t.entry( (0,0) )
1
sage: t.entry( (1,1) )
4
```

evacuation(*n=None*, *check=True*)

Return the evacuation of the tableau `self`.

This is an alias for `schuetzenberger_involution()`.

This method relies on the analogous method on words, which reverts the word and then complements all letters within the underlying ordered alphabet. If n is specified, the underlying alphabet is assumed to be $[1, 2, \ldots, n]$. If no alphabet is specified, n is the maximal letter appearing in `self`.

INPUT:

- `n` – an integer specifying the maximal letter in the alphabet (optional)
- `check` – (Default: `True`) Check to make sure `self` is semistandard. Set to `False` to avoid this check. (optional)

OUTPUT:

- a tableau, the evacuation of `self`

EXAMPLES:

```
sage: t = Tableau([[1,1,1],[2,2]])
sage: t.evacuation(3)
[[2, 2, 3], [3, 3]]

sage: t = Tableau([[1,2,3],[4,5]])
sage: t.evacuation()
[[1, 2, 5], [3, 4]]

sage: t = Tableau([[1,3,5,7],[2,4,6],[8,9]])
sage: t.evacuation()
[[1, 2, 6, 8], [3, 4, 9], [5, 7]]

sage: t = Tableau([])
sage: t.evacuation()
[]

sage: t = StandardTableau([[1,2,3],[4,5]])
sage: s = t.evacuation()
sage: s.parent()
Standard tableaux
```

evaluation()

Return the weight of the tableau `self`. Trailing zeroes are omitted when returning the weight.

The weight of a tableau T is the sequence $(a_1, a_2, a_3, \ldots)$, where a_k is the number of entries of T equal to k. This sequence contains only finitely many nonzero entries.

The weight of a tableau T is the same as the weight of the reading word of T, for any reading order.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).weight()
[1, 1, 1, 1]

sage: Tableau([]).weight()
[]

sage: Tableau([[1,3,3,7],[4,2],[2,3]]).weight()
[1, 2, 3, 1, 0, 0, 1]
```

TESTS:

We check that this agrees with going to the word:

```
sage: t = Tableau([[1,3,4,7],[6,2],[2,3]])
sage: def by_word(T):
....:     ed = T.to_word().evaluation_dict()
....:     m = max(ed.keys()) + 1
....:     return [ed.get(k,0) for k in range(1,m)]
sage: by_word(t) == t.weight()
True
sage: SST = SemistandardTableaux(shape=[3,1,1])
sage: all(by_word(t) == t.weight() for t in SST)
True
```

flush()

Return the number of flush segments in `self`, as in [S14].

Let $1 \leq i < k \leq r+1$ and suppose ℓ is the smallest integer greater than k such that there exists an ℓ-segment in the $(i+1)$-st row of T. A k-segment in the i-th row of T is called *flush* if the leftmost box in the k-segment and the leftmost box of the ℓ-segment are in the same column of T. If, however, no such ℓ exists, then this k-segment is said to be *flush* if the number of boxes in the k-segment is equal to θ_i, where $\theta_i = \lambda_i - \lambda_{i+1}$ and the shape of T is $\lambda = (\lambda_1 > \lambda_2 > \cdots > \lambda_r)$. Denote the number of flush k-segments in T by $\mathrm{flush}(T)$.

EXAMPLES:

```
sage: t = Tableau([[1,1,2,3,5],[2,3,5,5],[3,4]])
sage: t.flush()
3

sage: B = crystals.Tableaux("A4",shape=[4,3,2,1])
sage: t = B[32].to_tableau()
sage: t.flush()
4
```

height()

Return the height of `self`.

EXAMPLES:

```
sage: Tableau([[1,2,3],[4,5]]).height()
2
sage: Tableau([[1,2,3]]).height()
1
sage: Tableau([]).height()
0
```

insert_word(*w*, *left=False*)

Insert the word `w` into the tableau `self` letter by letter using Schensted insertion. By default, the word `w`

is being processed from left to right, and the insertion used is row insertion. If the optional keyword `left` is set to `True`, the word `w` is being processed from right to left, and column insertion is used instead.

EXAMPLES:

```
sage: t0 = Tableau([])
sage: w = [1,1,2,3,3,3,3]
sage: t0.insert_word(w)
[[1, 1, 2, 3, 3, 3, 3]]
sage: t0.insert_word(w,left=True)
[[1, 1, 2, 3, 3, 3, 3]]
sage: w.reverse()
sage: t0.insert_word(w)
[[1, 1, 3, 3], [2, 3], [3]]
sage: t0.insert_word(w,left=True)
[[1, 1, 3, 3], [2, 3], [3]]
sage: t1 = Tableau([[1,3],[2]])
sage: t1.insert_word([4,5])
[[1, 3, 4, 5], [2]]
sage: t1.insert_word([4,5], left=True)
[[1, 3], [2, 5], [4]]
```

inversion_number()

Return the inversion number of `self`.

The inversion number is defined to be the number of inversions of `self` minus the sum of the arm lengths of the descents of `self` (see the `inversions()` and `descents()` methods for the relevant definitions).

Warning: This has none of the meanings in which the word "inversion" is used in the theory of standard tableaux.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[2,5]])
sage: t.inversion_number()
0
sage: t = Tableau([[1,2,4],[3,5]])
sage: t.inversion_number()
0
```

inversions()

Return a list of the inversions of `self`.

Let T be a tableau. An inversion is an attacking pair (c, d) of the shape of T (see `attacking_pairs()` for a definition of this) such that the entry of c in T is greater than the entry of d.

Warning: Do not mistake this for the inversions of a standard tableau.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[2,5]])
sage: t.inversions()
[((1, 1), (0, 0))]
sage: t = Tableau([[1,4,3],[5,2],[2,6],[3]])
sage: t.inversions()
[((0, 1), (0, 2)), ((1, 0), (1, 1)), ((1, 1), (0, 0)), ((2, 1), (1, 0))]
```

is_column_increasing(*weak=False*)

Return `True` if the entries in each column are in increasing order, and `False` otherwise.

By default, this checks for strictly increasing columns. Set `weak` to `True` to test for weakly increasing columns.

EXAMPLES:

```
sage: T = Tableau([[1, 1, 3], [1, 2]])
sage: T.is_column_increasing(weak=True)
True
sage: T.is_column_increasing()
False
sage: Tableau([[2], [1]]).is_column_increasing(weak=True)
False
```

is_column_strict()

Return `True` if `self` is a column strict tableau and `False` otherwise.

A tableau is column strict if the entries in each column are in (strictly) increasing order.

EXAMPLES:

```
sage: Tableau([[1, 3], [2, 4]]).is_column_strict()
True
sage: Tableau([[1, 2], [2, 4]]).is_column_strict()
True
sage: Tableau([[2, 3], [2, 4]]).is_column_strict()
False
sage: Tableau([[5, 3], [2, 4]]).is_column_strict()
False
sage: Tableau([]).is_column_strict()
True
sage: Tableau([[1, 4, 2]]).is_column_strict()
True
sage: Tableau([[1, 4, 2], [2, 5]]).is_column_strict()
True
sage: Tableau([[1, 4, 2], [2, 3]]).is_column_strict()
False
```

is_increasing()

Return `True` if `self` is an increasing tableau and `False` otherwise.

A tableau is increasing if it is both row strict and column strict.

EXAMPLES:

```
sage: Tableau([[1, 3], [2, 4]]).is_increasing()
True
sage: Tableau([[1, 2], [2, 4]]).is_increasing()
True
sage: Tableau([[2, 3], [2, 4]]).is_increasing()
False
sage: Tableau([[5, 3], [2, 4]]).is_increasing()
False
sage: Tableau([[1, 2, 3], [2, 3], [3]]).is_increasing()
True
```

is_k_tableau(*k*)

Checks whether `self` is a valid weak k-tableau.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[2,3],[3]])
sage: t.is_k_tableau(3)
True
sage: t = Tableau([[1,1,3],[2,2],[3]])
sage: t.is_k_tableau(3)
False
```

is_key_tableau()

Return `True` if `self` is a key tableau or `False` otherwise.

A tableau is a *key tableau* if the set of entries in the j-th column is a subset of the set of entries in the $(j-1)$-st column.

REFERENCES:

EXAMPLES:

```
sage: t = Tableau([[1,1,1],[2,3],[3]])
sage: t.is_key_tableau()
True

sage: t = Tableau([[1,1,2],[2,3],[3]])
sage: t.is_key_tableau()
False
```

is_rectangular()

Return `True` if the tableau `self` is rectangular and `False` otherwise.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).is_rectangular()
True
sage: Tableau([[1,2,3],[4,5],[6]]).is_rectangular()
False
sage: Tableau([]).is_rectangular()
True
```

is_row_increasing(*weak=False*)

Return `True` if the entries in each row are in increasing order, and `False` otherwise.

By default, this checks for strictly increasing rows. Set `weak` to `True` to test for weakly increasing rows.

EXAMPLES:

```
sage: T = Tableau([[1, 1, 3], [1, 2]])
sage: T.is_row_increasing(weak=True)
True
sage: T.is_row_increasing()
False
sage: Tableau([[2, 1]]).is_row_increasing(weak=True)
False
```

is_row_strict()

Return `True` if `self` is a row strict tableau and `False` otherwise.

A tableau is row strict if the entries in each row are in (strictly) increasing order.

EXAMPLES:

```
sage: Tableau([[1, 3], [2, 4]]).is_row_strict()
True
sage: Tableau([[1, 2], [2, 4]]).is_row_strict()
```

```
True
sage: Tableau([[2, 3], [2, 4]]).is_row_strict()
True
sage: Tableau([[5, 3], [2, 4]]).is_row_strict()
False
```

is_semistandard()

Return `True` if `self` is a semistandard tableau, and `False` otherwise.

A tableau is semistandard if its rows weakly increase and its columns strictly increase.

EXAMPLES:

```
sage: Tableau([[1,1],[1,2]]).is_semistandard()
False
sage: Tableau([[1,2],[1,2]]).is_semistandard()
False
sage: Tableau([[1,1],[2,2]]).is_semistandard()
True
sage: Tableau([[1,2],[2,3]]).is_semistandard()
True
sage: Tableau([[4,1],[3,2]]).is_semistandard()
False
```

is_standard()

Return `True` if `self` is a standard tableau and `False` otherwise.

EXAMPLES:

```
sage: Tableau([[1, 3], [2, 4]]).is_standard()
True
sage: Tableau([[1, 2], [2, 4]]).is_standard()
False
sage: Tableau([[2, 3], [2, 4]]).is_standard()
False
sage: Tableau([[5, 3], [2, 4]]).is_standard()
False
```

k_weight(k)

Return the k-weight of `self`.

A tableau has k-weight $\alpha = (\alpha_1, ..., \alpha_n)$ if there are exactly α_i distinct residues for the cells occupied by the letter i for each i. The residue of a cell in position (a, b) is $a - b$ modulo $k + 1$.

This definition is the one used in [Ive2012] (p. 12).

REFERENCES:

EXAMPLES:

```
sage: Tableau([[1,2],[2,3]]).k_weight(1)
[1, 1, 1]
sage: Tableau([[1,2],[2,3]]).k_weight(2)
[1, 2, 1]
sage: t = Tableau([[1,1,1,2,5],[2,3,6],[3],[4]])
sage: t.k_weight(1)
[2, 1, 1, 1, 1, 1]
sage: t.k_weight(2)
[3, 2, 2, 1, 1, 1]
sage: t.k_weight(3)
[3, 1, 2, 1, 1, 1]
sage: t.k_weight(4)
```

```
[3, 2, 2, 1, 1, 1]
sage: t.k_weight(5)
[3, 2, 2, 1, 1, 1]
```

lambda_catabolism(*part*)

Return the `part`-catabolism of `self`, where `part` is a partition (which can be just given as an array).

For a partition λ and a tableau T, the λ-catabolism of T is defined by performing the following steps.

1. Truncate the parts of λ so that λ is contained in the shape of T. Let m be the length of this partition.

2. Let T_a be the first m rows of T, and T_b be the remaining rows.

3. Let S_a be the skew tableau T_a/λ.

4. Concatenate the reading words of S_a and T_b, and insert into a tableau.

EXAMPLES:

```
sage: Tableau([[1,1,3],[2,4,5]]).lambda_catabolism([2,1])
[[3, 5], [4]]
sage: t = Tableau([[1,1,3,3],[2,3],[3]])
sage: t.lambda_catabolism([])
[[1, 1, 3, 3], [2, 3], [3]]
sage: t.lambda_catabolism([1])
[[1, 2, 3, 3, 3], [3]]
sage: t.lambda_catabolism([1,1])
[[1, 3, 3, 3], [3]]
sage: t.lambda_catabolism([2,1])
[[3, 3, 3, 3]]
sage: t.lambda_catabolism([4,2,1])
[]
sage: t.lambda_catabolism([5,1])
[[3, 3]]
sage: t.lambda_catabolism([4,1])
[[3, 3]]
```

last_letter_lequal(*tab2*)

Return `True` if `self` is less than or equal to `tab2` in the last letter ordering.

EXAMPLES:

```
sage: st = StandardTableaux([3,2])
sage: f = lambda b: 1 if b else 0
sage: matrix( [ [ f(t1.last_letter_lequal(t2)) for t2 in st] for t1 in st] )
[1 1 1 1 1]
[0 1 1 1 1]
[0 0 1 1 1]
[0 0 0 1 1]
[0 0 0 0 1]
```

left_key_tableau()

Return the left key tableau of `self`.

The left key tableau of a tableau T is the key tableau whose entries are weakly lesser than the corresponding entries in T, and whose column reading word is subject to certain conditions. See [LS90] for the full definition.

ALGORITHM:

The following algorithm follows [Willis10]. Note that if T is a key tableau then the output of the algorithm is T.

To compute the left key tableau L of a tableau T we iterate over the columns of T. Let T_j be the j-th column of T and iterate over the entires in T_j from bottom to top. Initialize the corresponding entry k in L as the largest entry in T_j. Scan the columns to the left of T_j and with each column update k to be the lowest entry in that column which is weakly less than k. Update T_j and all columns to the left by removing all scanned entries.

See also:

- `is_key_tableau()`

EXAMPLES:

```
sage: t = Tableau([[1,2],[2,3]])
sage: t.left_key_tableau()
[[1, 1], [2, 2]]
sage: t = Tableau([[1,1,2,4],[2,3,3],[4],[5]])
sage: t.left_key_tableau()
[[1, 1, 1, 2], [2, 2, 2], [4], [5]]
```

TESTS:

We check that if we have a key tableau, we return the same tableau:

```
sage: t = Tableau([[1,1,1,2], [2,2,2], [4], [5]])
sage: t.is_key_tableau()
True
sage: t.left_key_tableau() == t
True
```

leq(*secondtab*)

Check whether each entry of `self` is less-or-equal to the corresponding entry of a further tableau `secondtab`.

INPUT:

- `secondtab` – a tableau of the same shape as `self`

EXAMPLES:

sage: T = Tableau([[1, 2], [3]]) sage: S = Tableau([[1, 3], [3]]) sage: G = Tableau([[2, 1], [4]]) sage: H = Tableau([[1, 2], [4]]) sage: T.leq(S) True sage: T.leq(T) True sage: T.leq(G) False sage: T.leq(H) True sage: S.leq(T) False sage: S.leq(G) False sage: S.leq(H) False sage: G.leq(H) False sage: H.leq(G) False

TESTS:

```
sage: StandardTableau(T).leq(S)
True
sage: T.leq(SemistandardTableau(S))
True
```

level()

Returns the level of `self`, which is always 1.

This function exists mainly for compatibility with `TableauTuple`.

EXAMPLE:

```
sage: Tableau([[1,2,3],[4,5]]).level()
1
```

major_index()

Return the major index of `self`.

The major index of a tableau T is defined to be the sum of the number of descents of `T` (defined in `descents()`) with the sum of their legs' lengths.

> **Warning:** This is not to be confused with the major index of a standard tableau.

EXAMPLES:

```
sage: Tableau( [[1,4],[2,3]] ).major_index()
1
sage: Tableau( [[1,2],[3,4]] ).major_index()
2
```

If the major index would be defined in the sense of standard tableaux theory, then the following would give 3 for a result:

```
sage: Tableau( [[1,2,3],[4,5]] ).major_index()
2
```

pp()

Returns a pretty print string of the tableau.

EXAMPLES:

```
sage: T = Tableau([[1,2,3],[3,4],[5]])
sage: T.pp()
  1  2  3
  3  4
  5
sage: Tableaux.global_options(convention="french")
sage: T.pp()
  5
  3  4
  1  2  3
sage: Tableaux.global_options.reset()
```

promotion(n)

Return the image of `self` under the promotion operator.

> **Warning:** You might know this operator as the inverse promotion operator – literature does not agree on the name. You might also be looking for the Lapointe-Lascoux-Morse promotion operator (`promotion_operator()`).

The promotion operator, applied to a tableau t, does the following:

Iterate over all letters $n+1$ in the tableau t, from left to right. For each of these letters, do the following:

- Remove the letter from t, thus leaving a hole where it used to be.
- Apply jeu de taquin to move this hole southwest (in French notation) until it reaches the inner boundary of t.
- Fill 0 into the hole once jeu de taquin has completed.

Once this all is done, add 1 to each letter in the tableau. This is not always well-defined. Restricted to the class of semistandard tableaux whose entries are all $\leq n+1$, this is the usual promotion operator defined on this class.

When `self` is a standard tableau of size `n + 1`, this definition of promotion is precisely the one given in [Hai1992] (p. 90). It is the inverse of the maps called "promotion" in [Sg2011] (p. 23) and in [Stan2009].

Warning: To my (Darij's) knowledge, the fact that the above promotion operator really is the inverse of the "inverse promotion operator" `promotion_inverse()` for semistandard tableaux has never been proven in literature. Corrections are welcome.

REFERENCES:

EXAMPLES:

```
sage: t = Tableau([[1,2],[3,3]])
sage: t.promotion(2)
[[1, 1], [2, 3]]

sage: t = Tableau([[1,1,1],[2,2,3],[3,4,4]])
sage: t.promotion(3)
[[1, 1, 2], [2, 2, 3], [3, 4, 4]]

sage: t = Tableau([[1,2],[2]])
sage: t.promotion(3)
[[2, 3], [3]]

sage: t = Tableau([[1,1,3],[2,2]])
sage: t.promotion(2)
[[1, 2, 2], [3, 3]]

sage: t = Tableau([[1,1,3],[2,3]])
sage: t.promotion(2)
[[1, 1, 2], [2, 3]]

sage: t = Tableau([])
sage: t.promotion(2)
[]
```

TESTS:

We check the equivalence of two definitions of promotion on semistandard tableaux:

```
sage: ST = SemistandardTableaux(shape=[3,2,2,1], max_entry=6)
sage: def bk_promotion6(st):
....:     st2 = st
....:     for i in range(5, 0, -1):
....:         st2 = st2.bender_knuth_involution(i, check=False)
....:     return st2
sage: all( bk_promotion6(st) == st.promotion(5) for st in ST ) # long time
True
sage: ST = SemistandardTableaux(shape=[4,4], max_entry=6)
sage: all( bk_promotion6(st) == st.promotion(5) for st in ST ) # long time
True
```

We also check `promotion_inverse()` is the inverse of `promotion()`:

```
sage: ST = SemistandardTableaux(shape=[3,2,1], max_entry=7)
sage: all( st.promotion(6).promotion_inverse(6) == st for st in ST ) # long time
True
```

promotion_inverse(*n*)

Return the image of `self` under the inverse promotion operator.

Warning: You might know this operator as the promotion operator (without "inverse") – literature does not agree on the name.

The inverse promotion operator, applied to a tableau t, does the following:

Iterate over all letters 1 in the tableau t, from right to left. For each of these letters, do the following:

- Remove the letter from t, thus leaving a hole where it used to be.
- Apply jeu de taquin to move this hole northeast (in French notation) until it reaches the outer boundary of t.
- Fill $n + 2$ into the hole once jeu de taquin has completed.

Once this all is done, subtract 1 from each letter in the tableau. This is not always well-defined. Restricted to the class of semistandard tableaux whose entries are all $\leq n + 1$, this is the usual inverse promotion operator defined on this class.

When `self` is a standard tableau of size `n + 1`, this definition of inverse promotion is the map called "promotion" in [Sg2011] (p. 23) and in [Stan2009], and is the inverse of the map called "promotion" in [Hai1992] (p. 90).

Warning: To my (Darij's) knowledge, the fact that the above "inverse promotion operator" really is the inverse of the promotion operator `promotion()` for semistandard tableaux has never been proven in literature. Corrections are welcome.

EXAMPLES:

```
sage: t = Tableau([[1,2],[3,3]])
sage: t.promotion_inverse(2)
[[1, 2], [2, 3]]

sage: t = Tableau([[1,2],[2,3]])
sage: t.promotion_inverse(2)
[[1, 1], [2, 3]]

sage: t = Tableau([[1,2,5],[3,3,6],[4,7]])
sage: t.promotion_inverse(8)
[[1, 2, 4], [2, 5, 9], [3, 6]]

sage: t = Tableau([])
sage: t.promotion_inverse(2)
[]
```

TESTS:

We check the equivalence of two definitions of inverse promotion on semistandard tableaux:

```
sage: ST = SemistandardTableaux(shape=[4,2,1], max_entry=7)
sage: def bk_promotion_inverse7(st):
....:     st2 = st
....:     for i in range(1, 7):
....:         st2 = st2.bender_knuth_involution(i, check=False)
....:     return st2
sage: all( bk_promotion_inverse7(st) == st.promotion_inverse(6) for st in ST ) # long time
True
sage: ST = SemistandardTableaux(shape=[2,2,2], max_entry=7)
sage: all( bk_promotion_inverse7(st) == st.promotion_inverse(6) for st in ST ) # long time
True
```

A test for trac ticket #13203:

```
sage: T = Tableau([[1]])
sage: type(T.promotion_inverse(2)[0][0])
<type 'sage.rings.integer.Integer'>
```

promotion_operator(*i*)

Return a list of semistandard tableaux obtained by the i-th Lapointe-Lascoux-Morse promotion operator from the semistandard tableau `self`.

This operator is defined by taking the maximum entry m of T, then adding a horizontal i-strip to T in all possible ways, each time filling this strip with $m+1$'s, and finally letting the permutation $\sigma_1\sigma_2\cdots\sigma_m = (2, 3, \ldots, m+1, 1)$ act on each of the resulting tableaux via the Lascoux-Schuetzenberger action (`symmetric_group_action_on_values()`). This method returns the list of all resulting tableaux. See [LLM01] for the purpose of this operator.

REFERENCES:

EXAMPLES:

```
sage: t = Tableau([[1,2],[3]])
sage: t.promotion_operator(1)
[[[1, 2], [3], [4]], [[1, 2], [3, 4]], [[1, 2, 4], [3]]]
sage: t.promotion_operator(2)
[[[1, 1], [2, 3], [4]],
 [[1, 1, 2], [3], [4]],
 [[1, 1, 4], [2, 3]],
 [[1, 1, 2, 4], [3]]]
sage: Tableau([[1]]).promotion_operator(2)
[[[1, 1], [2]], [[1, 1, 2]]]
sage: Tableau([[1,1],[2]]).promotion_operator(3)
[[[1, 1, 1], [2, 2], [3]],
 [[1, 1, 1, 2], [2], [3]],
 [[1, 1, 1, 3], [2, 2]],
 [[1, 1, 1, 2, 3], [2]]]
```

The example from [LLM01] p. 12:

```
sage: Tableau([[1,1],[2,2]]).promotion_operator(3)
[[[1, 1, 1], [2, 2], [3, 3]],
 [[1, 1, 1, 3], [2, 2], [3]],
 [[1, 1, 1, 3, 3], [2, 2]]]
```

TESTS:

```
sage: Tableau([]).promotion_operator(2)
[[[1, 1]]]
sage: Tableau([]).promotion_operator(1)
[[[1]]]
```

raise_action_from_words(*f*, **args*)

EXAMPLES:

```
sage: from sage.combinat.tableau import symmetric_group_action_on_values
sage: import functools
sage: t = Tableau([[1,1,3,3],[2,3],[3]])
sage: f = functools.partial(t.raise_action_from_words, symmetric_group_action_on_values)
sage: f([1,2,3])
[[1, 1, 3, 3], [2, 3], [3]]
sage: f([3,2,1])
[[1, 1, 1, 1], [2, 3], [3]]
```

```
sage: f([1,3,2])
[[1, 1, 2, 2], [2, 2], [3]]
```

reading_word_permutation()

Return the permutation obtained by reading the entries of the standardization of `self` row by row, starting with the bottommost row (in English notation).

EXAMPLES:

```
sage: StandardTableau([[1,2],[3,4]]).reading_word_permutation()
[3, 4, 1, 2]
```

Check that trac ticket #14724 is fixed:

```
sage: SemistandardTableau([[1,1]]).reading_word_permutation()
[1, 2]
```

reduced_lambda_catabolism(*part*)

EXAMPLES:

```
sage: t = Tableau([[1,1,3,3],[2,3],[3]])
sage: t.reduced_lambda_catabolism([])
[[1, 1, 3, 3], [2, 3], [3]]
sage: t.reduced_lambda_catabolism([1])
[[1, 2, 3, 3, 3], [3]]
sage: t.reduced_lambda_catabolism([1,1])
[[1, 3, 3, 3], [3]]
sage: t.reduced_lambda_catabolism([2,1])
[[3, 3, 3, 3]]
sage: t.reduced_lambda_catabolism([4,2,1])
[]
sage: t.reduced_lambda_catabolism([5,1])
0
sage: t.reduced_lambda_catabolism([4,1])
0
```

restrict(*n*)

Return the restriction of the semistandard tableau `self` to `n`. If possible, the restricted tableau will have the same parent as this tableau.

If T is a semistandard tableau and n is a nonnegative integer, then the restriction of T to n is defined as the (semistandard) tableau obtained by removing all cells filled with entries greater than n from T.

Note: If only the shape of the restriction, rather than the whole restriction, is needed, then the faster method `restriction_shape()` is preferred.

EXAMPLES:

```
sage: Tableau([[1,2],[3],[4]]).restrict(3)
[[1, 2], [3]]
sage: StandardTableau([[1,2],[3],[4]]).restrict(2)
[[1, 2]]
sage: Tableau([[1,2,3],[2,4,4],[3]]).restrict(0)
[]
sage: Tableau([[1,2,3],[2,4,4],[3]]).restrict(2)
[[1, 2], [2]]
sage: Tableau([[1,2,3],[2,4,4],[3]]).restrict(3)
[[1, 2, 3], [2], [3]]
sage: Tableau([[1,2,3],[2,4,4],[3]]).restrict(5)
```

```
[[1, 2, 3], [2, 4, 4], [3]]
```

If possible the restricted tableau will belong to the same category as the original tableau:

```
sage: S=StandardTableau([[1,2,4,7],[3,5],[6]]); S.category()
Category of elements of Standard tableaux
sage: S.restrict(4).category()
Category of elements of Standard tableaux
sage: SS=StandardTableaux([4,2,1])([[1,2,4,7],[3,5],[6]]); SS.category()
Category of elements of Standard tableaux of shape [4, 2, 1]
sage: SS.restrict(4).category()
Category of elements of Standard tableaux

sage: Tableau([[1,2],[3],[4]]).restrict(3)
[[1, 2], [3]]
sage: Tableau([[1,2],[3],[4]]).restrict(2)
[[1, 2]]
sage: SemistandardTableau([[1,1],[2]]).restrict(1)
[[1, 1]]
sage: _.category()
Category of elements of Semistandard tableaux
```

restriction_shape(*n*)

Return the shape of the restriction of the semistandard tableau `self` to `n`.

If T is a semistandard tableau and n is a nonnegative integer, then the restriction of T to n is defined as the (semistandard) tableau obtained by removing all cells filled with entries greater than n from T.

This method computes merely the shape of the restriction. For the restriction itself, use `restrict()`.

EXAMPLES:

```
sage: Tableau([[1,2],[2,3],[3,4]]).restriction_shape(3)
[2, 2, 1]
sage: StandardTableau([[1,2],[3],[4],[5]]).restriction_shape(2)
[2]
sage: Tableau([[1,3,3,5],[2,4,4],[17]]).restriction_shape(0)
[]
sage: Tableau([[1,3,3,5],[2,4,4],[17]]).restriction_shape(2)
[1, 1]
sage: Tableau([[1,3,3,5],[2,4,4],[17]]).restriction_shape(3)
[3, 1]
sage: Tableau([[1,3,3,5],[2,4,4],[17]]).restriction_shape(5)
[4, 3]

sage: all( T.restriction_shape(i) == T.restrict(i).shape()
....:      for T in StandardTableaux(5) for i in range(1, 5) )
True
```

reverse_bump(*loc*)

Reverse row bump the entry of `self` at the specified location `loc` (given as a row index or a corner `(r, c)` of the tableau).

This is the reverse of Schensted's row-insertion algorithm. See Section 1.1, page 8, of Fulton's [Ful1997].

INPUT:

- `loc` – Can be either of the following:
  - The coordinates `(r, c)` of the square to reverse-bump (which must be a corner of the tableau);

–The row index r of this square.

Note that both r and c are 0-based, i.e., the topmost row and the leftmost column are the 0-th row and the 0-th column.

OUTPUT:

An ordered pair consisting of:

1. The resulting (smaller) tableau;
2. The entry bumped out at the end of the process.

See also:

bump()

EXAMPLES:

This is the reverse of Schensted's bump:

```
sage: T = Tableau([[1, 1, 2, 2, 4], [2, 3, 3], [3, 4], [4]])
sage: T.reverse_bump(2)
([[1, 1, 2, 3, 4], [2, 3, 4], [3], [4]], 2)
sage: T == T.reverse_bump(2)[0].bump(2)
True
sage: T.reverse_bump((3, 0))
([[1, 2, 2, 2, 4], [3, 3, 3], [4, 4]], 1)
```

Some errors caused by wrong input:

```
sage: T.reverse_bump((3, 1))
Traceback (most recent call last):
...
ValueError: invalid corner
sage: T.reverse_bump(4)
Traceback (most recent call last):
...
IndexError: list index out of range
sage: Tableau([[2, 2, 1], [3, 3]]).reverse_bump(0)
Traceback (most recent call last):
...
ValueError: Reverse bumping is only defined for semistandard tableaux
```

Some edge cases:

```
sage: Tableau([[1]]).reverse_bump(0)
([], 1)
sage: Tableau([[1,1]]).reverse_bump(0)
([[1]], 1)
sage: Tableau([]).reverse_bump(0)
Traceback (most recent call last):
...
IndexError: list index out of range
```

Note: Reverse row bumping is only implemented for tableaux with weakly increasing and strictly increasing columns (though the tableau does not need to be an instance of class SemistandardTableau).

---

right_key_tableau()

Return the right key tableau of self.

The right key tableau of a tableau T is a key tableau whose entries are weakly greater than the corresponding entries in T, and whose column reading word is subject to certain conditions. See [LS90] for the full definition.

ALGORITHM:

The following algorithm follows [Willis10]. Note that if T is a key tableau then the output of the algorithm is T.

To compute the right key tableau R of a tableau T we iterate over the columns of T. Let T_j be the j-th column of T and iterate over the entires in T_j from bottom to top. Initialize the corresponding entry k in R to be the largest entry in T_j. Scan the bottom of each column of T to the right of T_j, updating k to be the scanned entry whenever the scanned entry is weakly greater than k. Update T_j and all columns to the right by removing all scanned entries.

See also:

- is_key_tableau()

EXAMPLES:

```
sage: t = Tableau([[1,2],[2,3]])
sage: t.right_key_tableau()
[[2, 2], [3, 3]]
sage: t = Tableau([[1,1,2,4],[2,3,3],[4],[5]])
sage: t.right_key_tableau()
[[2, 2, 2, 4], [3, 4, 4], [4], [5]]
```

TESTS:

We check that if we have a key tableau, we return the same tableau:

```
sage: t = Tableau([[1,1,1,2], [2,2,2], [4], [5]])
sage: t.is_key_tableau()
True
sage: t.right_key_tableau() == t
True
```

rotate_180()

Return the tableau obtained by rotating `self` by 180 degrees.

This only works for rectangular tableaux.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).rotate_180()
[[4, 3], [2, 1]]
```

row_stabilizer()

Return the PermutationGroup corresponding to the row stabilizer of `self`.

This assumes that every integer from 1 to the size of `self` appears exactly once in `self`.

EXAMPLES:

```
sage: rs = Tableau([[1,2,3],[4,5]]).row_stabilizer()
sage: rs.order() == factorial(3)*factorial(2)
True
sage: PermutationGroupElement([(1,3,2),(4,5)]) in rs
True
sage: PermutationGroupElement([(1,4)]) in rs
False
sage: rs = Tableau([[1, 2],[3]]).row_stabilizer()
```

```
sage: PermutationGroupElement([(1,2),(3,)]) in rs
True
sage: rs.one().domain()
[1, 2, 3]
sage: rs = Tableau([[1],[2],[3]]).row_stabilizer()
sage: rs.order()
1
sage: rs = Tableau([[2,4,5],[1,3]]).row_stabilizer()
sage: rs.order()
12
sage: rs = Tableau([]).row_stabilizer()
sage: rs.order()
1
```

schensted_insert(*i*, *left=False*)

Insert `i` into `self` using Schensted's row-bumping (or row-insertion) algorithm.

INPUT:

- `i` – a number to insert
- `left` – (default: `False`) boolean; if set to `True`, the insertion will be done from the left. That is, if one thinks of the algorithm as appending a letter to the reading word of `self`, we append the letter to the left instead of the right

EXAMPLES:

```
sage: t = Tableau([[3,5],[7]])
sage: t.schensted_insert(8)
[[3, 5, 8], [7]]
sage: t.schensted_insert(8, left=True)
[[3, 5], [7], [8]]
```

schuetzenberger_involution(*n=None*, *check=True*)

Return the Schuetzenberger involution of the tableau `self`.

This method relies on the analogous method on words, which reverts the word and then complements all letters within the underlying ordered alphabet. If n is specified, the underlying alphabet is assumed to be $[1, 2, \ldots, n]$. If no alphabet is specified, n is the maximal letter appearing in `self`.

INPUT:

- `n` – an integer specifying the maximal letter in the alphabet (optional)
- `check` – (Default: `True`) Check to make sure `self` is semistandard. Set to `False` to avoid this check. (optional)

OUTPUT:

- a tableau, the Schuetzenberger involution of `self`

EXAMPLES:

```
sage: t = Tableau([[1,1,1],[2,2]])
sage: t.schuetzenberger_involution(3)
[[2, 2, 3], [3, 3]]

sage: t = Tableau([[1,2,3],[4,5]])
sage: t.schuetzenberger_involution()
[[1, 2, 5], [3, 4]]

sage: t = Tableau([[1,3,5,7],[2,4,6],[8,9]])
```

```
sage: t.schuetzenberger_involution()
[[1, 2, 6, 8], [3, 4, 9], [5, 7]]

sage: t = Tableau([])
sage: t.schuetzenberger_involution()
[]

sage: t = StandardTableau([[1,2,3],[4,5]])
sage: s = t.schuetzenberger_involution()
sage: s.parent()
Standard tableaux
```

seg()

Return the total number of segments in `self`, as in [S14].

Let T be a tableaux. We define a *k-segment* of T (in the i-th row) to be a maximal consecutive sequence of k-boxes in the i-th row for any $i+1 \leq k \leq r+1$. Denote the total number of k-segments in T by $\mathrm{seg}(T)$.

REFERENCES:

EXAMPLES:

```
sage: t = Tableau([[1,1,2,3,5],[2,3,5,5],[3,4]])
sage: t.seg()
6

sage: B = crystals.Tableaux("A4",shape=[4,3,2,1])
sage: t = B[31].to_tableau()
sage: t.seg()
3
```

shape()

Return the shape of a tableau `self`.

EXAMPLES:

```
sage: Tableau([[1,2,3],[4,5],[6]]).shape()
[3, 2, 1]
```

size()

Return the size of the shape of the tableau `self`.

EXAMPLES:

```
sage: Tableau([[1, 4, 6], [2, 5], [3]]).size()
6
sage: Tableau([[1, 3], [2, 4]]).size()
4
```

slide_multiply(*left*, *right*)

Multiply two tableaux using jeu de taquin.

This product makes the set of semistandard tableaux into an associative monoid. The empty tableau is the unit in this monoid.

See pp. 15 of [Ful1997].

The same product operation is implemented in a different way in `bump_multiply()`.

EXAMPLES:

```
sage: t = Tableau([[1,2,2,3],[2,3,5,5],[4,4,6],[5,6]])
sage: t2 = Tableau([[1,2],[3]])
sage: t.slide_multiply(t2)
[[1, 1, 2, 2, 3], [2, 2, 3, 5], [3, 4, 5], [4, 6, 6], [5]]
```

socle()

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).socle()
2
sage: Tableau([[1,2,3,4]]).socle()
4
```

standardization(*check=True*)

Return the standardization of `self`, assuming `self` is a semistandard tableau.

The standardization of a semistandard tableau T is the standard tableau $\mathrm{st}(T)$ of the same shape as T whose reversed reading word is the standardization of the reversed reading word of T.

The standardization of a word w can be formed by replacing all 1's in w by $1, 2, \ldots, k_1$ from left to right, all 2's in w by $k_1 + 1, k_1 + 2, \ldots, k_2$, and repeating for all letters which appear in w. See also `Word.standard_permutation()`.

INPUT:

- `check` – (Default: `True`) Check to make sure `self` is semistandard. Set to `False` to avoid this check.

EXAMPLES:

```
sage: t = Tableau([[1,3,3,4],[2,4,4],[5,16]])
sage: t.standardization()
[[1, 3, 4, 7], [2, 5, 6], [8, 9]]
```

Standard tableaux are fixed under standardization:

```
sage: all((t == t.standardization() for t in StandardTableaux(6)))
True
sage: t = Tableau([])
sage: t.standardization()
[]
```

The reading word of the standardization is the standardization of the reading word:

```
sage: T = SemistandardTableaux(shape=[6,3,3,1], max_entry=5)
sage: all(t.to_word().standard_permutation() == t.standardization().reading_word_permutation
True
```

symmetric_group_action_on_entries(*w*)

Return the tableau obtained form this tableau by acting by the permutation `w`.

Let T be a standard tableau of size n, then the action of $w \in S_n$ is defined by permuting the entries of T (recall they are $1, 2, \ldots, n$). In particular, suppose the entry at cell (i, j) is a, then the entry becomes $w(a)$. In general, the resulting tableau wT may *not* be standard.

Note: This is different than `symmetric_group_action_on_values()` which is defined on semistandard tableaux and is guaranteed to return a semistandard tableau.

INPUT:

- `w` – a permutation

EXAMPLES:

```
sage: StandardTableau([[1,2,4],[3,5]]).symmetric_group_action_on_entries( Permutation(((4,5)
[[1, 2, 5], [3, 4]]
sage: _.category()
Category of elements of Standard tableaux
sage: StandardTableau([[1,2,4],[3,5]]).symmetric_group_action_on_entries( Permutation(((1,2)
[[2, 1, 4], [3, 5]]
sage: _.category()
Category of elements of Tableaux
```

symmetric_group_action_on_values(*perm*)

Return the image of the semistandard tableau `self` under the action of the permutation `perm` using the Lascoux-Schuetzenberger action of the symmetric group S_n on the semistandard tableaux with ceiling n.

If n is a nonnegative integer, then the Lascoux-Schuetzenberger action is a group action of the symmetric group S_n on the set of semistandard Young tableaux with ceiling n (that is, with entries taken from the set $\{1, 2, \ldots, n\}$). It is defined as follows:

Let $i \in \{1, 2, \ldots, n-1\}$, and let T be a semistandard tableau with ceiling n. Let w be the reading word (`to_word()`) of T. Replace all letters i in w by closing parentheses, and all letters $i+1$ in w by opening parentheses. Whenever an opening parenthesis stands left of a closing parenthesis without there being any parentheses inbetween (it is allowed to have letters inbetween as long as they are not parentheses), consider these two parentheses as matched with each other, and replace them back by the letters $i+1$ and i. Repeat this procedure until there are no more opening parentheses standing left of closing parentheses. Then, let a be the number of opening parentheses in the word, and b the number of closing parentheses (notice that all opening parentheses are left of all closing parentheses). Replace the first a parentheses by the letters i, and replace the remaining b parentheses by the letters $i+1$. Let w' be the resulting word. Let T' be the tableau with the same shape as T but with reading word w'. This tableau T' can be shown to be semistandard. We define the image of T under the action of the simple transposition $s_i = (i, i+1) \in S_n$ to be this tableau T'. It can be shown that these actions $s_1, s_2, \ldots, s_{n-1}$ satisfy the Moore-Coxeter relations of S_n, and thus this extends to a unique action of the symmetric group S_n on the set of semistandard tableaux with ceiling n. This is the Lascoux-Schuetzenberger action.

This action of the symmetric group S_n on the set of all semistandard tableaux of given shape λ with entries in $\{1, 2, \ldots, n\}$ is the one defined in [Loth02] Theorem 5.6.3. In particular, the action of s_i is denoted by σ_i in said source. (Beware of the typo in the definition of σ_i: it should say $\sigma_i(a_i^r a_{i+1}^s) = a_i^s a_{i+1}^r$, not $\sigma_i(a_i^r a_{i+1}^s) = a_i^s a_{i+1}^s$.)

EXAMPLES:

```
sage: t = Tableau([[1,1,3,3],[2,3],[3]])
sage: t.symmetric_group_action_on_values([1,2,3])
[[1, 1, 3, 3], [2, 3], [3]]
sage: t.symmetric_group_action_on_values([2,1,3])
[[1, 2, 3, 3], [2, 3], [3]]
sage: t.symmetric_group_action_on_values([3,1,2])
[[1, 2, 2, 2], [2, 3], [3]]
sage: t.symmetric_group_action_on_values([2,3,1])
[[1, 1, 1, 1], [2, 2], [3]]
sage: t.symmetric_group_action_on_values([3,2,1])
[[1, 1, 1, 1], [2, 3], [3]]
sage: t.symmetric_group_action_on_values([1,3,2])
[[1, 1, 2, 2], [2, 2], [3]]
```

TESTS:

```
sage: t = Tableau([])
sage: t.symmetric_group_action_on_values([])
[]
```

to_Gelfand_Tsetlin_pattern()

Return the Gelfand-Tsetlin pattern corresponding to self when semistandard.

EXAMPLES:

```
sage: T = Tableau([[1,2,3],[2,3],[3]])
sage: G = T.to_Gelfand_Tsetlin_pattern(); G
[[3, 2, 1], [2, 1], [1]]
sage: G.to_tableau() == T
True
sage: T = Tableau([[1,3],[2]])
sage: T.to_Gelfand_Tsetlin_pattern()
[[2, 1, 0], [1, 1], [1]]
```

to_chain(*max_entry=None*)

Return the chain of partitions corresponding to the (semi)standard tableau self.

The optional keyword parameter max_entry can be used to customize the length of the chain. Specifically, if this parameter is set to a nonnegative integer n, then the chain is constructed from the positions of the letters $1, 2, \ldots, n$ in the tableau.

EXAMPLES:

```
sage: Tableau([[1,2],[3],[4]]).to_chain()
[[], [1], [2], [2, 1], [2, 1, 1]]
sage: Tableau([[1,1],[2]]).to_chain()
[[], [2], [2, 1]]
sage: Tableau([[1,1],[3]]).to_chain()
[[], [2], [2], [2, 1]]
sage: Tableau([]).to_chain()
[[]]
sage: Tableau([[1,1],[2],[3]]).to_chain(max_entry=2)
[[], [2], [2, 1]]
sage: Tableau([[1,1],[2],[3]]).to_chain(max_entry=3)
[[], [2], [2, 1], [2, 1, 1]]
sage: Tableau([[1,1],[2],[3]]).to_chain(max_entry=4)
[[], [2], [2, 1], [2, 1, 1], [2, 1, 1]]
sage: Tableau([[1,1,2],[2,3],[4,5]]).to_chain(max_entry=6)
[[], [2], [3, 1], [3, 2], [3, 2, 1], [3, 2, 2], [3, 2, 2]]
```

to_list()

Return self as a list of lists (not tuples!).

EXAMPLES:

```
sage: t = Tableau([[1,2],[3,4]])
sage: l = t.to_list(); l
[[1, 2], [3, 4]]
sage: l[0][0] = 2
sage: t
[[1, 2], [3, 4]]
```

to_sign_matrix(*max_entry=None*)

Return the sign matrix of self.

A sign matrix is an $m \times n$ matrix of 0's, 1's and -1's such that the partial sums of each column is either 0 or 1 and the partial sums of each row is non-negative. [Aval2008]

INPUT:

- max_entry – A non-negative integer, the maximum allowable number in the tableau. Defaults to the largest entry in the tableau if not specified.

EXAMPLES:

```
sage: t = SemistandardTableau([[1,1,1,2,4],[3,3,4],[4,5],[6,6]])
sage: t.to_sign_matrix(6)
[ 0  0  0  1  0  0]
[ 0  1  0 -1  0  0]
[ 1 -1  0  1  0  0]
[ 0  0  1 -1  1  1]
[ 0  0  0  1 -1  0]
sage: t = Tableau([[1,2,4],[3,5]])
sage: t.to_sign_matrix(7)
[ 0  0  0  1  0  0  0]
[ 0  1  0 -1  1  0  0]
[ 1 -1  1  0 -1  0  0]
sage: t=Tableau([(4,5,4,3),(2,1,3)])
sage: t.to_sign_matrix(5)
[ 0  0  1  0  0]
[ 0  0  0  1  0]
[ 1  0 -1 -1  1]
[-1  1  0  1 -1]
sage: s=Tableau([(1,0,-2,4),(3,4,5)])
sage: s.to_sign_matrix(6)
Traceback (most recent call last):
...
ValueError: the entries must be non-negative integers
```

REFERENCES:

to_word()

An alias for to_word_by_row().

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).to_word()
word: 3412
sage: Tableau([[1, 4, 6], [2, 5], [3]]).to_word()
word: 325146
```

to_word_by_column()

Return the word obtained from a column reading of the tableau self (starting with the leftmost column, reading every column from bottom to top).

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).to_word_by_column()
word: 3142
sage: Tableau([[1, 4, 6], [2, 5], [3]]).to_word_by_column()
word: 321546
```

to_word_by_row()

Return the word obtained from a row reading of the tableau self (starting with the lowermost row, reading every row from left to right).

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).to_word_by_row()
word: 3412
sage: Tableau([[1, 4, 6], [2, 5], [3]]).to_word_by_row()
word: 325146
```

vertical_flip()

Return the tableau obtained by vertically flipping the tableau `self`.

This only works for rectangular tableaux.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).vertical_flip()
[[3, 4], [1, 2]]
```

weight()

Return the weight of the tableau `self`. Trailing zeroes are omitted when returning the weight.

The weight of a tableau T is the sequence $(a_1, a_2, a_3, \ldots)$, where a_k is the number of entries of T equal to k. This sequence contains only finitely many nonzero entries.

The weight of a tableau T is the same as the weight of the reading word of T, for any reading order.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).weight()
[1, 1, 1, 1]

sage: Tableau([]).weight()
[]

sage: Tableau([[1,3,3,7],[4,2],[2,3]]).weight()
[1, 2, 3, 1, 0, 0, 1]
```

TESTS:

We check that this agrees with going to the word:

```
sage: t = Tableau([[1,3,4,7],[6,2],[2,3]])
sage: def by_word(T):
....:     ed = T.to_word().evaluation_dict()
....:     m = max(ed.keys()) + 1
....:     return [ed.get(k,0) for k in range(1,m)]
sage: by_word(t) == t.weight()
True
sage: SST = SemistandardTableaux(shape=[3,1,1])
sage: all(by_word(t) == t.weight() for t in SST)
True
```

class sage.combinat.tableau.**Tableau_class**(*parent*, *t*)

Bases: sage.combinat.tableau.Tableau

This exists solely for unpickling `Tableau_class` objects.

class sage.combinat.tableau.**Tableaux**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

A factory class for the various classes of tableaux.

INPUT:

- `n` (optional) – a non-negative integer

OUTPUT:

- If `n` is specified, the class of tableaux of size `n`. Otherwise, the class of all tableaux.

A tableau in Sage is a finite list of lists, whose lengths are weakly decreasing, or an empty list, representing the empty tableau. The entries of a tableau can be any Sage objects. Because of this, no enumeration through the set of Tableaux is possible.

EXAMPLES:

```
sage: T = Tableaux(); T
Tableaux
sage: T3 = Tableaux(3); T3
Tableaux of size 3
sage: [['a','b']] in T
True
sage: [['a','b']] in T3
False
sage: t = T3([[1,1,1]]); t
[[1, 1, 1]]
sage: t in T
True
sage: t.parent()
Tableaux of size 3
sage: T([]) # the empty tableau
[]
sage: T.category()
Category of sets
```

TESTS:

```
sage: TestSuite( Tableaux() ).run()
sage: TestSuite( Tableaux(5) ).run()
sage: t = Tableaux(3)([[1,2],[3]])
sage: t.parent()
Tableaux of size 3
sage: Tableaux(t)
Traceback (most recent call last):
...
ValueError: The argument to Tableaux() must be a non-negative integer.
sage: Tableaux(3)([[1, 1]])
Traceback (most recent call last):
...
ValueError: [[1, 1]] is not an element of Tableaux of size 3.

sage: t0 = Tableau([[1]])
sage: t1 = Tableaux()([[1]])
sage: t2 = Tableaux()(t1)
sage: t0 == t1 == t2
True
sage: t1 in Tableaux()
True
sage: t1 in Tableaux(1)
True
sage: t1 in Tableaux(2)
False

sage: [[1]] in Tableaux()
True
sage: [] in Tableaux(0)
True
```

Check that trac ticket #14145 has been fixed:

```
sage: 1 in Tableaux()
False
```

Element
alias of Tableau

global_options(*get_value, **set_value)
Sets the global options for elements of the tableau, skew_tableau, and tableau tuple classes. The defaults are for tableau to be displayed as a list, latexed as a Young diagram using the English convention.

OPTIONS:

- ascii_art – (default: repr) Controls the ascii art output for tableaux
  - compact – minimal length ascii art
  - repr – display using the diagram string representation
  - table – display as a table
- convention – (default: English) Sets the convention used for displaying tableaux and partitions
  - English – use the English convention
  - French – use the French convention
- display – (default: list) Controls the way in which tableaux are printed
  - array – alias for diagram
  - compact – minimal length string representation
  - diagram – display as Young diagram (similar to pp())
  - ferrers_diagram – alias for diagram
  - list – print tableaux as lists
  - young_diagram – alias for diagram
- latex – (default: diagram) Controls the way in which tableaux are latexed
  - array – alias for diagram
  - diagram – as a Young diagram
  - ferrers_diagram – alias for diagram
  - list – as a list
  - young_diagram – alias for diagram
- notation – alternative name for convention

Note: Changing the convention for tableaux also changes the convention for partitions.

If no parameters are set, then the function returns a copy of the options dictionary.

EXAMPLES:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T
[[1, 2, 3], [4, 5]]
sage: Tableaux.global_options(display="array")
sage: T
  1  2  3
```

```
  4  5
sage: Tableaux.global_options(convention="french")
sage: T
  4  5
  1  2  3
```

Changing the `convention` for tableaux also changes the `convention` for partitions and vice versa:

```
sage: P = Partition([3,3,1])
sage: print P.ferrers_diagram()
*
***
***
sage: Partitions.global_options(convention="english")
sage: print P.ferrers_diagram()
***
***
*
sage: T
  1  2  3
  4  5
```

The ASCII art can also be changed:

```
sage: t = Tableau([[1,2,3],[4,5]])
sage: ascii_art(t)
  1  2  3
  4  5
sage: Tableaux.global_options(ascii_art="table")
sage: ascii_art(t)
+---+---+
| 4 | 5 |
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
sage: Tableaux.global_options(ascii_art="compact")
sage: ascii_art(t)
|4|5|
|1|2|3|
sage: Tableaux.global_options.reset()
```

See `GlobalOptions` for more features of these options.

class sage.combinat.tableau.**Tableaux_all**

Bases: sage.combinat.tableau.Tableaux

Initializes the class of all tableaux

TESTS:

```
sage: T = sage.combinat.tableau.Tableaux_all()
sage: TestSuite(T).run()
```

an_element()

Returns a particular element of the class.

TESTS:

```
sage: T = Tableaux()
sage: T.an_element()
[[1, 1], [1]]
```

class sage.combinat.tableau.**Tableaux_size**(*n*)
Bases: sage.combinat.tableau.Tableaux

Tableaux of a fixed size n.

an_element()
Returns a particular element of the class.

TESTS:

```
sage: T = sage.combinat.tableau.Tableaux_size(3)
sage: T.an_element()
[[1, 1], [1]]
sage: T = sage.combinat.tableau.Tableaux_size(0)
sage: T.an_element()
[]
```

sage.combinat.tableau.**from_chain**(*chain*)
Returns a semistandard tableau from a chain of partitions.

EXAMPLES:

```
sage: from sage.combinat.tableau import from_chain
sage: from_chain([[], [2], [2, 1], [3, 2, 1]])
[[1, 1, 3], [2, 3], [3]]
```

sage.combinat.tableau.**from_shape_and_word**(*shape*, *w*, *convention='French'*)
Returns a tableau from a shape and word.

INPUT:

- `shape` – a partition
- `w` – a word whose length equals that of the partition
- `convention` – a string which can take values `"French"` or `"English"`; the default is `"French"`

OUTPUT:

A tableau, whose shape is `shape` and whose reading word is `w`. If the `convention` is specified as `"French"`, the reading word is to be read starting from the top row in French convention (= the bottom row in English convention). If the `convention` is specified as `"English"`, the reading word is to be read starting with the top row in English convention.

EXAMPLES:

```
sage: from sage.combinat.tableau import from_shape_and_word
sage: t = Tableau([[1, 3], [2], [4]])
sage: shape = t.shape(); shape
[2, 1, 1]
sage: word = t.to_word(); word
word: 4213
sage: from_shape_and_word(shape, word)
[[1, 3], [2], [4]]
sage: word = Word(flatten(t))
sage: from_shape_and_word(shape, word, convention = "English")
[[1, 3], [2], [4]]
```

sage.combinat.tableau.**symmetric_group_action_on_values**(*word*, *perm*)
EXAMPLES:

```
sage: from sage.combinat.tableau import symmetric_group_action_on_values
sage: symmetric_group_action_on_values([1,1,1],[1,3,2])
[1, 1, 1]
sage: symmetric_group_action_on_values([1,1,1],[2,1,3])
[2, 2, 2]
sage: symmetric_group_action_on_values([1,2,1],[2,1,3])
[2, 2, 1]
sage: symmetric_group_action_on_values([2,2,2],[2,1,3])
[1, 1, 1]
sage: symmetric_group_action_on_values([2,1,2],[2,1,3])
[2, 1, 1]
sage: symmetric_group_action_on_values([2,2,3,1,1,2,2,3],[1,3,2])
[2, 3, 3, 1, 1, 2, 3, 3]
sage: symmetric_group_action_on_values([2,1,1],[2,1])
[2, 1, 2]
sage: symmetric_group_action_on_values([2,2,1],[2,1])
[1, 2, 1]
sage: symmetric_group_action_on_values([1,2,1],[2,1])
[2, 2, 1]
```

sage.combinat.tableau.**unmatched_places**(*w*, *open*, *close*)

EXAMPLES:

```
sage: from sage.combinat.tableau import unmatched_places
sage: unmatched_places([2,2,2,1,1,1],2,1)
([], [])
sage: unmatched_places([1,1,1,2,2,2],2,1)
([0, 1, 2], [3, 4, 5])
sage: unmatched_places([], 2, 1)
([], [])
sage: unmatched_places([1,2,4,6,2,1,5,3],2,1)
([0], [1])
sage: unmatched_places([2,2,1,2,4,6,2,1,5,3], 2, 1)
([], [0, 3])
sage: unmatched_places([3,1,1,1,2,1,2], 2, 1)
([1, 2, 3], [6])
```

5.1.311 TableauTuples

A `TableauTuple` is a tuple of tableaux. These objects arise naturally in representation theory of the wreath products of cyclic groups and the symmetric groups where the standard tableau tulpes index bases for the ordinary irreducible representations. This generalises the well-known fact the ordinary irreducible representations of the symmetric groups have bases indexed by the standard tableaux of a given shape. More generally, `TableauTuples`, or multitableaux, appear in the representation theory of the degenerate and non-degenerate cyclotomic Hecke algebras and in the crystal theory of the integral highest weight representations of the affine special linear groups.

A `TableauTuple` is an ordered tuple $(t^{(1)}, t^{(2)}, \ldots, t^{(l)})$ of tableaux. The length of the tuple is its *level* and the tableaux $t^{(1)}, t^{(2)}, \ldots, t^{(l)}$ are the components of the `TableauTuple`.

A tableaux can be thought of as the labelled diagram of a partition. Analogously, a `TableauTuple` is the labelled diagram of a `PartitionTuple`. That is, a `TableauTuple` is a tableau of `PartitionTuple` shape. As much as possible, `TableauTuples` behave in exactly the same way as `Tableaux`. There are obvious differences in that the cells of a partition are ordered pairs (r, c), where r is a row index and c a column index, whereas the cells of a `PartitionTuple` are ordered triples (k, r, c), with r and c as before and k indexes the component.

Frequently, we will call a `TableauTuple` a tableau, or a tableau of `PartitionTuple` shape. If the shape of the tableau is known this should not cause any confusion.

Warning: In sage the convention is that the (k, r, c)-th entry of a tableau tuple t is the entry in row r, column c and component k of the tableau. This is because it makes much more sense to let `t[k]` be component of the tableau. In particular, we want `t(k,r,c) == t[k][r][c]`. In the literature, the cells of a tableau tuple are usually written in the form (r, c, k), where r is the row index, c is the column index, and k is the component index. The same convention applies to the cells of `PartitionTuples`.

Note: As with partitions and tableaux, the cells are 0-based. For example, the (lexicographically) first cell in any non-empty tableau tuple is `[0,0,0]`.

EXAMPLES:

```
sage: TableauTuple([[1,2,3],[4,5]])
[[1, 2, 3], [4, 5]]
sage: t=TableauTuple([ [[6,7],[8,9]],[[1,2,3],[4,5]] ]); t
([[6, 7], [8, 9]], [[1, 2, 3], [4, 5]])
sage: t.pp()
     6  7     1  2  3
     8  9     4  5
sage: t(0,0,1)
7
sage: t(1,0,1)
2
sage: t.shape()
([2, 2], [3, 2])
sage: t.size()
9
sage: t.level()
2
sage: t.components()
[[[6, 7], [8, 9]], [[1, 2, 3], [4, 5]]]
sage: t.entries()
[6, 7, 8, 9, 1, 2, 3, 4, 5]
sage: t.parent()
Tableau tuples
sage: t.category()
Category of elements of Tableau tuples
```

One reason for implementing `TableauTuples` is to be able to consider `StandardTableauTuples`. These objects arise in many areas of algebraic combinatorics. In particular, they index bases for the Specht modules of the cyclotomic Hecke algebras of type $G(r, 1, n)$. A `StandardTableauTuple` of tableau whose entries are increasing along rows and down columns in each component and which contain the numbers $1, 2, \ldots, n$, where the shape of the `StandardTableauTuple` is a `PartitionTuple` of n.

```
sage: s = StandardTableauTuple([ [[1,2],[3]],[[4,5]]])
sage: s.category()
Category of elements of Standard tableau tuples
sage: t = TableauTuple([ [[1,2],[3]],[[4,5]]])
sage: t.is_standard(), t.is_column_strict(), t.is_row_strict()
(True, True, True)
sage: t.category()
Category of elements of Tableau tuples
sage: s == t
True
sage: s is t
False
sage: s == StandardTableauTuple(t)
```

```
True
sage: StandardTableauTuples([ [2,1],[1] ])[:]
[([[1, 2], [3]], [[4]]),
 ([[1, 3], [2]], [[4]]),
 ([[1, 2], [4]], [[3]]),
 ([[1, 3], [4]], [[2]]),
 ([[2, 3], [4]], [[1]]),
 ([[1, 4], [2]], [[3]]),
 ([[1, 4], [3]], [[2]]),
 ([[2, 4], [3]], [[1]])]
```

As tableaux (of partition shape) are in natural bijection with 1-tuples of tableaux all of the `TableauTuple` classes return an ordinary `Tableau` when given `TableauTuple` of level 1.

```
sage: TableauTuples( level=1 ) is Tableaux()
True
sage: TableauTuple([[1,2,3],[4,5]])
[[1, 2, 3], [4, 5]]
sage: TableauTuple([ [[1,2,3],[4,5]] ])
[[1, 2, 3], [4, 5]]
sage: TableauTuple([[1,2,3],[4,5]]) == Tableau([[1,2,3],[4,5]])
True
```

There is one situation where a 1-tuple of tableau is not actually a `Tableau`; tableaux generated by the `StandardTableauTuples()` iterators must have the correct parents, so in this one case 1-tuples of tableaux are different from `Tableaux`:

```
sage: StandardTableauTuples()[:10]
[(),
 ([[1]]),
 ([], []),
 ([[1, 2]]),
 ([[1], [2]]),
 ([[1]], []),
 ([], [[1]]),
 ([], [], []),
 ([[1, 2, 3]]),
 ([[1, 3], [2]])]
```

AUTHORS:

- Andrew Mathas (2012-10-09): Initial version – heavily based on `tableau.py` by Mike Hansen (2007) and Jason Bandlow (2011).

This file consists of the following major classes:

Element classes:

- `TableauTuples`
- `StandardTableauTuples`

Factory classes:

- `TableauTuples`
- `StandardTableauTuples`

Parent classes:

- `TableauTuples_all`

- TableauTuples_level
- TableauTuples_size
- TableauTuples_level_size
- StandardTableauTuples_all
- StandardTableauTuples_level
- StandardTableauTuples_size
- StandardTableauTuples_level_size
- StandardTableauTuples_shape

See also:

- Tableau
- StandardTableau
- Tableaux
- StandardTableaux
- Partitions
- PartitionTuples

Todo

Implement semistandard tableau tuples as defined in [DJM].

Much of the combinatorics implemented here is motivated by this and subsequent papers on the representation theory of these algebras.

REFERENCES:

class sage.combinat.tableau_tuple.**StandardTableauTuple**(*parent*, *t*)
Bases: sage.combinat.tableau_tuple.TableauTuple

A class to model a standard tableau of shape a partition tuple. This is a tuple of standard tableau with entries $1, 2, \ldots, n$, where n is the size of the underlying partition tuple, such that the entries increase along rows and down columns in each component of the tuple.

Note: The tableaux appearing in a StandardTableauTuple are both row and column strict, but individually they are not standard tableaux because the entries in any single component of a StandardTableauTuple will typically not be in bijection with $\{1, 2, \ldots, n\}$.

INPUT:

•t – a tableau, a list of (standard) tableau or an equivalent list

OUTPUT:

•A StandardTableauTuple object constructed from t.

Note: Sage uses the English convention for (tuples of) partitions and tableaux: the longer rows are displayed on top. As with PartitionTuple, in sage the cells, or nodes, of partition tuples are 0-based. For example, the (lexicographically) first cell in any non-empty partition tuple is $[0, 0, 0]$. Further, the coorindates [k,r,c] in a TableauTuple refer to the component, row and column indices, respectively.

EXAMPLES:

```
sage: t=TableauTuple([ [[1,3,4],[7,9]], [[2,8,11],[6]], [[5,10]] ]); t
([[1, 3, 4], [7, 9]], [[2, 8, 11], [6]], [[5, 10]])
sage: t[0][0][0]
1
sage: t[1][1][0]
6
sage: t[2][0][0]
5
sage: t[2][0][1]
10

sage: t = StandardTableauTuple([[[4,5],[7]],[[1,2,3],[6,8]],[[9]]]); t
([[4, 5], [7]], [[1, 2, 3], [6, 8]], [[9]])
sage: t.pp()
 4  5     1  2  3     9
 7        6  8
sage: t.shape()
([2, 1], [3, 2], [1])
sage: t[0].pp() # pretty print
4 5
7
sage: t.is_standard()
True
sage: t[0].is_standard()
False
sage: StandardTableauTuple([[],[],[]]) # An empty tableau tuple
([], [], [])
```

When using code that will generate a lot of tableaux, it is slightly more efficient to construct a StandardTableauTuple from the appropriate parent object:

```
sage: STT = StandardTableauTuples()
sage: STT([[[4,5],[7]],[[1,2,3],[6,8]],[[9]]])
([[4, 5], [7]], [[1, 2, 3], [6, 8]], [[9]])
```

See also:

- Tableau
- Tableaux
- TableauTuples
- TableauTuple
- StandardTableauTuples

TESTS:

```
sage: StandardTableauTuple( [[1,2,3],[4,5]] ).category()  # indirect doctest
Category of elements of Standard tableaux
sage: StandardTableauTuple([[[1,2,3],[4,5]]]).category()  # indirect doctest
Category of elements of Standard tableaux
sage: StandardTableauTuples()([[[1,2,3],[4,5]]]).category()  # indirect doctest
Category of elements of Standard tableaux

sage: StandardTableauTuple([[[1,2,3]],[[1]]])
Traceback (most recent call last):
...
ValueError: entries must be in bijection with {1,2,...,n}
```

```
sage: StandardTableauTuple([[],[[1,2,1]]])
Traceback (most recent call last):
...
ValueError: tableaux must be row strict

sage: StandardTableauTuple([ [[1,2,4],[6]],[[0,1]],[[10]] ])
Traceback (most recent call last):
...
ValueError: entries must be in bijection with {1,2,...,n}

sage: TestSuite(  StandardTableauTuple([[[1,3,4],[6]],[[2],[5]]]) ).run()
sage: TestSuite(  StandardTableauTuple([[[1,3,4],[6]],[], [[2],[5]]]) ).run()
sage: TestSuite(  StandardTableauTuple([[[1,3,4],[6]],[[7]], [[2],[5]]]) ).run()
```

content(*k*, *multicharge*)

Return the content `k` in `self`.

The content of k in a standard tableau. That is, if k appears in row r and column c of the tableau then we return $c - r$ + `multicharge[k]`.

The `multicharge` = $[m_1, \ldots, m_l]$ determines the dominant weight

$$\Lambda = \sum_{i=1}^{l} \Lambda_{a_i}$$

of the affine special linear group. In the combinatorics, the `mutichargе` simply offsets the contents in each component so that the cell (k, r, c) has content $a_k + c - r$.

INPUT:

- `k` – An integer with $1 \leq k \leq n$
- `multicharge` – a sequence of integers of length l.

Here l is the `level()` and n is the `size()` of `self`.

EXAMPLES:

```
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).content(3,[0,0])
-1
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).content(3,[0,1])
0
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).content(3,[0,2])
1
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).content(6,[0,2])
Traceback (most recent call last):
...
ValueError: 6 must be contained in the tableaux
```

dominates(*t*)

Returns `True` if the tableau (tuple) `self` dominates the tableau `t`. The two tableaux do not need to be of the same shape.

EXAMPLES:

```
sage: s=StandardTableauTuple([[1,2,3],[4,5]])
sage: t=StandardTableauTuple([[1,2],[3,5],[4]])
sage: s.dominates(t)
True
sage: t.dominates(s)
False
```

inverse(*k*)

Return the cell containing `k` in the tableau tuple `self`.

EXAMPLES:

```
sage: StandardTableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).inverse(1)
(0, 0, 0)
sage: StandardTableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).inverse(2)
(0, 0, 1)
sage: StandardTableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).inverse(3)
(0, 1, 0)
sage: StandardTableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).inverse(12)
(2, 2, 0)
```

restrict(*m=None*)

Returns the restriction of the standard tableau `self` to `m`, which defaults to one less than the current `size()`.

EXAMPLES:

```
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(6)
([[5]], [[1, 2], [3, 4]])
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(5)
([[5]], [[1, 2], [3, 4]])
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(4)
([], [[1, 2], [3, 4]])
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(3)
([], [[1, 2], [3]])
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(2)
([], [[1, 2]])
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(1)
([], [[1]])
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(0)
([], [])
```

Where possible the restricted tableau belongs to the same category as the tableau `self`:

```
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(3).category()
Category of elements of Tableau tuples
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).restrict(3).category()
Category of elements of Standard tableau tuples
sage: StandardTableauTuples([[1],[2,2]])([[[5]],[[1,2],[3,4]]]).restrict(3).category()
Category of elements of Standard tableau tuples
sage: StandardTableauTuples(level=2)([[[5]],[[1,2],[3,4]]]).restrict(3).category()
Category of elements of Standard tableau tuples of level 2
```

to_chain()

Returns the chain of partitions corresponding to the standard tableau tuple `self`.

EXAMPLES:

```
sage: StandardTableauTuple([[[5]],[[1,2],[3,4]]]).to_chain()
[([], []),
  ([], [1]),
  ([], [2]),
  ([], [2, 1]),
  ([], [2, 2]),
  ([1], [2, 2])]
```

class sage.combinat.tableau_tuple.**StandardTableauTuples**

Bases: sage.combinat.tableau_tuple.TableauTuples

A factory class for the various classes of tuples of standard tableau.

INPUT:

There are three optional arguments:

- level – The level() of the tuples of tableaux
- size – The size() of the tuples of tableaux
- shape – A list or a partition tuple specifying the shape() of the standard tableau tuples

It is not necessary to use the keywords. If they are not used then then first integer argument specifies the level() and the second the size() of the tableau tuples.

OUTPUT:

The appropriate subclass of StandardTableauTuples.

A tuple of standard tableau is a tableau whose entries are positive integers which increase from left to down along the rows, and from top to bottom down the columns, in each component. The entries do NOT need to increase from left to right along the components.

Note: Sage uses the English convention for (tuples of) partitions and tableaux: the longer rows are displayed on top. As with PartitionTuple, in sage the cells, or nodes, of partition tuples are 0-based. For example, the (lexicographically) first cell in any non-empty partition tuple is $[0, 0, 0]$.

EXAMPLES:

```
sage: tabs=StandardTableauTuples([[2],[1],[1,1]]); tabs
Standard tableau tuples of shape ([2], [1], [1, 1])
sage: tabs.cardinality()
30
sage: tabs.list()
[([[1, 2]], [[3]], [[4], [5]]),
 ([[1, 3]], [[2]], [[4], [5]]),
 ([[2, 3]], [[1]], [[4], [5]]),
 ([[1, 2]], [[4]], [[3], [5]]),
 ([[1, 3]], [[4]], [[2], [5]]),
 ([[2, 3]], [[4]], [[1], [5]]),
 ([[1, 4]], [[2]], [[3], [5]]),
 ([[2, 4]], [[1]], [[3], [5]]),
 ([[1, 4]], [[3]], [[2], [5]]),
 ([[2, 4]], [[3]], [[1], [5]]),
 ([[3, 4]], [[1]], [[2], [5]]),
 ([[3, 4]], [[2]], [[1], [5]]),
 ([[1, 2]], [[5]], [[3], [4]]),
 ([[1, 3]], [[5]], [[2], [4]]),
 ([[2, 3]], [[5]], [[1], [4]]),
 ([[1, 4]], [[5]], [[2], [3]]),
 ([[2, 4]], [[5]], [[1], [3]]),
 ([[3, 4]], [[5]], [[1], [2]]),
 ([[1, 5]], [[2]], [[3], [4]]),
 ([[2, 5]], [[1]], [[3], [4]]),
 ([[1, 5]], [[3]], [[2], [4]]),
 ([[2, 5]], [[3]], [[1], [4]]),
 ([[3, 5]], [[1]], [[2], [4]]),
 ([[3, 5]], [[2]], [[1], [4]]),
 ([[1, 5]], [[4]], [[2], [3]]),
```

```
 ([[2, 5]], [[4]], [[1], [3]]),
 ([[3, 5]], [[4]], [[1], [2]]),
 ([[4, 5]], [[1]], [[2], [3]]),
 ([[4, 5]], [[2]], [[1], [3]]),
 ([[4, 5]], [[3]], [[1], [2]])]

sage: tabs=StandardTableauTuples(level=3); tabs
Standard tableau tuples of level 3
sage: tabs[100]
([[1, 2], [3]], [], [[4]])

sage: StandardTableauTuples()[0]
()
```

TESTS:

```
sage: TestSuite( StandardTableauTuples() ).run()
sage: TestSuite( StandardTableauTuples(level=1) ).run()
sage: TestSuite( StandardTableauTuples(level=4) ).run()
sage: TestSuite( StandardTableauTuples(size=0) ).run(max_runs=50) # recursion depth exceeded wit
sage: TestSuite( StandardTableauTuples(size=6) ).run()
sage: TestSuite( StandardTableauTuples(level=1, size=0) ).run()
sage: TestSuite( StandardTableauTuples(level=1, size=0) ).run()
sage: TestSuite( StandardTableauTuples(level=1, size=10) ).run()
sage: TestSuite( StandardTableauTuples(level=4, size=0) ).run()
sage: TestSuite( StandardTableauTuples(level=4, size=0) ).run()
sage: TestSuite( StandardTableauTuples(level=4, size=10) ).run() # long time
sage: TestSuite( StandardTableauTuples(shape=[[1],[3,1],[],[2,1]]) ).run()
```

See also:

- TableauTuples
- Tableau
- StandardTableau
- StandardTableauTuples

Element
alias of StandardTableauTuple

an_element()
Returns a particular element of the class.

EXAMPLES:

```
sage: StandardTableauTuples().an_element()
([[1]], [[2, 3]], [[4, 5, 6, 7]])
```

level_one_parent_class
alias of StandardTableaux_all

shape()
Return the shape of the set of StandardTableauTuples, or None if it is not defined.

EXAMPLES:

```
sage: tabs=StandardTableauTuples(shape=[[5,2],[3,2],[],[1,1,1],[3]]); tabs
Standard tableau tuples of shape ([5, 2], [3, 2], [], [1, 1, 1], [3])
sage: tabs.shape()
```

```
([5, 2], [3, 2], [], [1, 1, 1], [3])
sage: StandardTableauTuples().shape() is None
True
```

class sage.combinat.tableau_tuple.**StandardTableauTuples_all**
Bases: sage.combinat.tableau_tuple.StandardTableauTuples

Default class of all StandardTableauTuples with an arbitrary level() and size().

class sage.combinat.tableau_tuple.**StandardTableauTuples_level**(*level*)
Bases: sage.combinat.tableau_tuple.StandardTableauTuples

Class of all StandardTableauTuples with a fixed level and arbitrary size.

an_element()
Returns a particular element of the class.

EXAMPLES:

```
sage: StandardTableauTuples(2).an_element()
([[1]], [[2, 3]])
sage: StandardTableauTuples(3).an_element()
([[1]], [[2, 3]], [[4, 5, 6, 7]])
```

class sage.combinat.tableau_tuple.**StandardTableauTuples_level_size**(*level*, *size*)
Bases: sage.combinat.tableau_tuple.StandardTableauTuples

Class of all StandardTableauTuples with a fixed level and a fixed size.

an_element()
Returns a particular element of the class.

EXAMPLES:

```
sage: StandardTableauTuples(5,size=2).an_element()
([], [], [], [], [[1], [2]])
sage: StandardTableauTuples(2,size=4).an_element()
([[1]], [[2, 3], [4]])
```

cardinality()
Returns the number of elements in this set of tableaux.

EXAMPLES:

```
sage: StandardTableauTuples(3,2).cardinality()
12
sage: StandardTableauTuples(4,6).cardinality()
31936
```

class sage.combinat.tableau_tuple.**StandardTableauTuples_shape**(*shape*)
Bases: sage.combinat.tableau_tuple.StandardTableauTuples

Class of all StandardTableauTuples of a fixed shape.

an_element()
Returns a particular element of the class.

EXAMPLES:

```
sage: StandardTableauTuples([[2],[2,1]]).an_element()
([[2, 4]], [[1, 3], [5]])
sage: StandardTableauTuples([[10],[],[]]).an_element()
([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]], [], [])
```

cardinality()

Returns the number of standard Young tableau tuples of with the same shape as the partition tuple `self`.

Let $\mu = (\mu^{(1)}, \ldots, \mu^{(l)})$ be the `shape` of the tableaux in `self` and let $m_k = |\mu^{(k)}|$, for $1 \leq k \leq l$. Multiplying by a (unique) coset representative of the Young subgroup $S_{m_1} \times \cdots \times S_{m_l}$ inside the symmetric group S_n, we can assume that t is standard and the numbers $1, 2..., n$ are entered in order from to right along the components of the tableau. Therefore, there are

$$\binom{n}{m_1, \ldots, m_l} \prod_{k=1}^{l} |\mathrm{Std}(\mu^{(k)})|$$

standard tableau tuples of this shape, where $|\mathrm{Std}(\mu^{(k)})|$ is the number of standard tableau of shape $\mu^{(k)}$, for $1 \leq k \leq l$. This is given by the hook length formula.

EXAMPLES:

```
sage: StandardTableauTuples([[3,2,1],[]]).cardinality()
16
sage: StandardTableauTuples([[1],[1],[1]]).cardinality()
6
sage: StandardTableauTuples([[2,1],[1],[1]]).cardinality()
40
sage: StandardTableauTuples([[3,2,1],[3,2]]).cardinality()
36960
```

last()

Returns the last standard tableau tuple in `self`, with respect to the order that they are generated by the iterator. This is just the standard tableau tuple with the numbers $1, 2, \ldots, n$, where n is `size()`, entered in order down the columns form right to left along the components.

EXAMPLES:

```
sage: t=StandardTableauTuples([[2],[2,2]]).last().pp()
    5  6      1  3
              2  4
```

random_element()

Returns a random standard tableau in `self`.

We do this by randomly selecting addable nodes to place $1, 2, \ldots, n$. Of course we could do this recursively, but it's more efficient to keep track of the (changing) list of addable nodes as we go.

EXAMPLES:

```
sage: StandardTableauTuples([[2],[2,1]]).random_element()  # random
([[1, 2]], [[3, 4], [5]])
```

class sage.combinat.tableau_tuple.**StandardTableauTuples_size**(*size*)

Bases: sage.combinat.tableau_tuple.StandardTableauTuples

Class of all `StandardTableauTuples` with an arbitrary `level` and a fixed `size`.

an_element()

Returns a particular element of the class.

EXAMPLES:

```
sage: StandardTableauTuples(size=2).an_element()
([[1]], [[2]], [], [])
sage: StandardTableauTuples(size=4).an_element()
([[1]], [[2, 3, 4]], [], [])
```

class sage.combinat.tableau_tuple.**TableauTuple**(*parent*, *t*)
Bases: sage.combinat.combinat.CombinatorialElement

A class to model a tuple of tableaux.

INPUT:

- t – a list or tuple of Tableau, a list or tuple of lists of lists

OUTPUT:

- The Tableau tuple object constructed from t.

A TableauTuple is a tuple of tableau of shape a PartitionTuple. These combinatorial objects are useful is several areas of algebraic combinatorics. In particular, they are important in:

- the representation theory of the complex reflection groups of type $G(l, 1, n)$ and the representation theory of the associated (degenerate and non-degenerate) Hecke algebras. See, for example, [DJM]
- the crystal theory of (quantum) affine special linear groups and its integral highest weight modules and their canonical bases. See, for example, [BK].

These apparently different and unrelated contexts are, in fact, intimately related as in characteristic zero the cyclotomic Hecke algebras categorify the canonical bases of the integral highest weight modules of the quantum affine special linear groups.

The level() of a tableau tuple is the length of the tuples. This corresponds to the level of the corresponding highest weight module.

In sage a TableauTuple looks an behaves like a real tuple of (level 1) Tableaux. Many of the operations which are defined on Tableau extend to TableauTuples. Tableau tulpes of level 1 are just ordinary Tableau.

In sage, the entries of Tableaux can be very general, including arbitrarily nested lists, so some lists can be interpreted either as a tuple of tableaux or simply as tableaux. If it is possible to interpret the input to TableauTuple as a tuple of tableaux then TableauTuple returns the corresponding tuple. Given a 1-tulpe of tableaux the tableau itself is returned.

EXAMPLES:

```
sage: t = TableauTuple([ [[6,9,10],[11]],[[1,2,3],[4,5]],[[7],[8]] ]); t
([[6, 9, 10], [11]], [[1, 2, 3], [4, 5]], [[7], [8]])
sage: t.level()
3
sage: t.size()
11
sage: t.shape()
([3, 1], [3, 2], [1, 1])
sage: t.is_standard()
True
sage: t.pp() # pretty print
6  9 10     1  2  3     7
11          4  5        8
sage: t.category()
Category of elements of Tableau tuples
sage: t.parent()
Tableau tuples

sage: s = TableauTuple([ [['a','c','b'],['d','e']],[[(2,1)]]]); s
([['a', 'c', 'b'], ['d', 'e']], [[(2, 1)]])
sage: s.shape()
([3, 2], [1])
sage: s.size()
```

```
6

sage: TableauTuple([[],[],[]])  # The empty 3-tuple of tableaux
([], [], [])

sage: TableauTuple([[1,2,3],[4,5]])
[[1, 2, 3], [4, 5]]
sage: TableauTuple([[1,2,3],[4,5]]) == Tableau([[1,2,3],[4,5]])
True
```

See also:

- StandardTableauTuple
- StandardTableauTuples
- StandardTableau
- StandardTableaux
- TableauTuple
- TableauTuples
- Tableau
- Tableaux

TESTS:

```
sage: TableauTuple( [[1,2,3],[4,5]] ).category()
Category of elements of Tableaux
sage: TableauTuple([[[1,2,3],[4,5]]]).category()
Category of elements of Tableaux

sage: TableauTuple([[1],[2,3]])
Traceback (most recent call last):
...
ValueError: A tableau must be a list of iterables.

sage: TestSuite( TableauTuple([ [[1,2],[3,4]], [[1,2],[3,4]] ]) ).run()
sage: TestSuite( TableauTuple([ [[1,2],[3,4]], [], [[1,2],[3,4]] ]) ).run()
sage: TestSuite( TableauTuple([[[1,1],[1]],[[1,1,1]],[[1],[1],[1]],[[1]]]) ).run()
```

Element
alias of Tableau

add_entry(*cell*, *m*)
Set the entry in `cell` equal to `m`. If the cell does not exist then extend the tableau, otherwise just replace the entry.

EXAMPLES:

```
sage: s=StandardTableauTuple([ [[3,4,7],[6,8]], [[9,13],[12]], [[1,5],[2,11],[10]] ]); s.pp(
     3  4  7     9 13     1  5
     6  8       12        2 11
                         10
sage: t=s.add_entry( (0,0,3),14); t.pp(); t.category()
     3  4  7 14     9 13     1  5
     6  8          12        2 11
                            10
```

```
Category of elements of Standard tableau tuples
sage: t=s.add_entry( (0,0,3),15); t.pp(); t.category()
     3  4  7 15     9 13     1  5
     6  8          12        2 11
                             10
Category of elements of Tableau tuples
sage: t=s.add_entry( (1,1,1),14); t.pp(); t.category()
     3  4  7     9 13     1  5
     6  8       12 14     2 11
                         10
Category of elements of Standard tableau tuples
sage: t=s.add_entry( (2,1,1),14); t.pp(); t.category()
     3  4  7     9 13     1  5
     6  8       12        2 14
                         10
Category of elements of Tableau tuples
sage: t=s.add_entry( (2,1,2),14); t.pp(); t.category()
Traceback (most recent call last):
...
IndexError: (2, 1, 2) is not an addable cell of the tableau
```

cells_containing(*m*)

Returns the list of cells in which the letter `m` appears in the tableau `self`.

The list is ordered with cells appearing from left to right.

EXAMPLES:

```
sage: t = TableauTuple([[[4,5]],[[1,1,2,4],[2,4,4],[4]],[[1,3,4],[3,4]]])
sage: t.cells_containing(4)
[(0, 0, 0),
  (1, 2, 0),
  (1, 1, 1),
  (1, 1, 2),
  (1, 0, 3),
  (2, 1, 1),
  (2, 0, 2)]
sage: t.cells_containing(6)
[]
```

charge()

Return the charge of the reading word of `self`.

See `charge()` for more information.

EXAMPLES:

```
sage: TableauTuple([[[4,5]],[[1,1,2,4],[2,4,4],[4]],[[1,3,4],[3,4]]]).charge()
4
```

cocharge()

Return the cocharge of the reading word of `self`.

See `cocharge()` for more information.

EXAMPLES:

```
sage: TableauTuple([[[4,5]],[[1,1,2,4],[2,4,4],[4]],[[1,3,4],[3,4]]]).charge()
4
```

column_stabilizer()

Return the PermutationGroup corresponding to self. That is, return subgroup of the symmetric group of degree size() which is the column stabilizer of self.

EXAMPLES:

```
sage: cs = TableauTuple([[[1,2,3],[4,5]],[[6,7]],[[8],[9]]]).column_stabilizer()
sage: cs.order()
8
sage: PermutationGroupElement([(1,3,2),(4,5)]) in cs
False
sage: PermutationGroupElement([(1,4)]) in cs
True
```

components()

Return a list of the components of tableau tuple self. The *components* are the individual Tableau which are contained in the tuple self.

For compatibility with TableauTuples of level() 1, components() should be used to iterate over the components of TableauTuples.

EXAMPLES:

```
sage: for t in TableauTuple([[1,2,3],[4,5]]).components(): t.pp()
  1  2  3
  4  5
sage: for t in TableauTuple([ [[1,2,3],[4,5]], [[6,7],[8,9]] ]).components(): t.pp()
  1  2  3
  4  5

  6  7
  8  9
```

conjugate()

Returns the conjugate of the tableau tuple self. That is, the TableauTuple obtained from self by reversing the order of the components and conjugating each component – that is, swapping the rows and columns of the all of Tableau in self (see sage.combinat.tableau.Tableau.conjugate()).

EXAMPLES:

```
sage: TableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).conjugate()
([[9, 11, 12], [10]], [[5, 8], [6], [7]], [[1, 3], [2, 4]])
```

entries()

Returns a sorted list of all entries of self, in the order obtained by reading across the rows.

EXAMPLES:

```
sage: TableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).entries()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
sage: TableauTuple([[[1,2],[3,4]],[[9,10],[11],[12]],[[5,6,7],[8]]]).entries()
[1, 2, 3, 4, 9, 10, 11, 12, 5, 6, 7, 8]
```

entry(*l*, *r*, *c*)

Returns the entry of the cell (l, r, c) in self.

A cell is a tuple (l, r, c) of coordinates, where l is the component index, r is the row index, and c is the column index.

EXAMPLES:

```
sage: t = TableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]])
sage: t.entry(1, 0, 0)
```

```
5
sage: t.entry(1, 1, 1)
Traceback (most recent call last):
...
IndexError: tuple index out of range
```

is_column_strict()

Return `True` if the tableau `self` is column strict and `False` otherwise.

A tableau tuple is *column strict* if the entries in each column of each component are in increasing order, when read from top to bottom.

EXAMPLES:

```
sage: TableauTuple([[[5,7],[8]],[[1, 3], [2, 4]],[[6]]]).is_column_strict()
True
sage: TableauTuple([[[1, 2], [2, 4]],[[4,5,6],[7,8]]]).is_column_strict()
True
sage: TableauTuple([[[1]],[[2, 3], [2, 4]]]).is_column_strict()
False
sage: TableauTuple([[[1]],[[2, 2], [4,5]]]).is_column_strict()
True
sage: TableauTuple([[[1,2],[6,7]],[[4,8], [6, 9]],[]]).is_column_strict()
True
```

is_row_strict()

Return `True` if the tableau `self` is row strict and `False` otherwise.

A tableau tuple is *row strict* if the entries in each row of each component are in increasing order, when read from left to right.

EXAMPLES:

```
sage: TableauTuple([[[5,7],[8]],[[1, 3], [2, 4]],[[6]]]).is_row_strict()
True
sage: TableauTuple([[[1, 2], [2, 4]],[[4,5,6],[7,8]]]).is_row_strict()
True
sage: TableauTuple([[[1]],[[2, 3], [2, 4]]]).is_row_strict()
True
sage: TableauTuple([[[1]],[[2, 2], [4,5]]]).is_row_strict()
False
sage: TableauTuple([[[1,2],[6,7]],[[4,8], [6, 9]],[]]).is_row_strict()
True
```

is_standard()

Returns `True` if the tableau `self` is a standard tableau and `False` otherwise.

A tableau tuple is *standard* if it is row standard, column standard and the entries in the tableaux are $1, 2, \ldots, n$, where n is the `size()` of the underlying partition tuple of `self`.

EXAMPLES:

```
sage: TableauTuple([[[5,7],[8]],[[1, 3], [2, 4]],[[6]]]).is_standard()
True
sage: TableauTuple([[[1, 2], [2, 4]],[[4,5,6],[7,8]]]).is_standard()
False
sage: TableauTuple([[[1]],[[2, 3], [2, 4]]]).is_standard()
False
sage: TableauTuple([[[1]],[[2, 2], [4,5]]]).is_row_strict()
False
```

```
sage: TableauTuple([[[1,2],[6,7]],[[4,8], [6, 9]],[]]).is_standard()
False
```

level()

Return the level of the tableau `self`, which is just the number of components in the tableau tuple `self`.

EXAMPLES:

```
sage: TableauTuple([[[7,8,9]],[],[[1,2,3],[4,5],[6]]]).level()
3
```

pp()

Pretty printing for the tableau tuple `self`.

EXAMPLES:

```
sage: TableauTuple([ [[1,2,3],[4,5]], [[1,2,3],[4,5]] ]).pp()
    1  2  3     1  2  3
    4  5        4  5
sage: TableauTuple([ [[1,2],[3],[4]],[],[[6,7,8],[10,11],[12],[13]]]).pp()
    1  2   -     6  7  8
    3           10 11
    4           12
                13
sage: t = TableauTuple([ [[1,2,3],[4,5],[6],[9]], [[1,2,3],[4,5,8]], [[11,12,13],[14]] ])
sage: t.pp()
    1  2  3     1  2  3    11 12 13
    4  5        4  5  8    14
    6
    9
sage: TableauTuples.global_options(convention="french")
sage: t.pp()
     9
     6
     4  5        4  5  8    14
     1  2  3     1  2  3    11 12 13
sage: TableauTuples.global_options.reset()
```

restrict(*m=None*)

Returns the restriction of the standard tableau `self` to `m`.

The restriction is the subtableau of `self` whose entries are less than or equal to `m`.

By default, `m` is one less than the current size.

EXAMPLES:

```
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict()
([], [[1, 2], [3, 4]])
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(6)
([[5]], [[1, 2], [3, 4]])
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(5)
([[5]], [[1, 2], [3, 4]])
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(4)
([], [[1, 2], [3, 4]])
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(3)
([], [[1, 2], [3]])
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(2)
([], [[1, 2]])
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(1)
([], [[1]])
```

```
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(0)
([], [])
```

Where possible the restricted tableau belongs to the same category as the original tableaux:

```
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(3).category()
Category of elements of Tableau tuples
sage: TableauTuple([[[5]],[[1,2],[3,4]]]).restrict(3).category()
Category of elements of Tableau tuples
sage: TableauTuples(level=2)([[[5]],[[1,2],[3,4]]]).restrict(3).category()
Category of elements of Tableau tuples of level 2
```

row_stabilizer()

Return the PermutationGroup corresponding to self. That is, return subgroup of the symmetric group of degree size() which is the row stabilizer of self.

EXAMPLES:

```
sage: rs = TableauTuple([[1,2,3],[4,5]],[[6,7]],[[8],[9]]).row_stabilizer()
sage: rs.order()
24
sage: PermutationGroupElement([(1,3,2),(4,5)]) in rs
True
sage: PermutationGroupElement([(1,4)]) in rs
False
sage: rs.one().domain()
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

shape()

Returns the PartitionTuple which is the shape of the tableau tuple self.

EXAMPLES:

```
sage: TableauTuple([[[7,8,9]],[],[[1,2,3],[4,5],[6]]]).shape()
([3], [], [3, 2, 1])
```

size()

Returns the size of the tableau tuple self, which is just the number of boxes, or the size, of the underlying PartitionTuple.

EXAMPLES:

```
sage: TableauTuple([[[7,8,9]],[],[[1,2,3],[4,5],[6]]]).size()
9
```

symmetric_group_action_on_entries(*w*)

Return the action of a permutation w on self.

Consider a standard tableau tuple $T = (t^{(1)}, t^{(2)}, \dots t^{(l)})$ of size n, then the action of $w \in S_n$ is defined by permuting the entries of T (recall they are $1, 2, \dots, n$). In particular, suppose the entry at cell (k, i, j) is a, then the entry becomes $w(a)$. In general, the resulting tableau tuple wT may *not* be standard.

INPUT:

- w – a permutation

EXAMPLES:

```
sage: TableauTuple([[[1,2],[4]],[[3,5]]]).symmetric_group_action_on_entries( Permutation(((4
([[1, 2], [5]], [[3, 4]])
sage: TableauTuple([[[1,2],[4]],[[3,5]]]).symmetric_group_action_on_entries( Permutation(((1
([[2, 1], [4]], [[3, 5]])
```

to_list()
Return the list representation of the tableaux tuple `self`.

EXAMPLES:

```
sage: TableauTuple([ [[1,2,3],[4,5]], [[6,7],[8,9]] ]).to_list()
[[[1, 2, 3], [4, 5]], [[6, 7], [8, 9]]]
```

to_permutation()
Returns a permutation with the entries in the tableau tuple `self` which is obtained by `self` obtained by reading the entries of the tableau tuple in order from left to right along the rows, and then top to bottom, in each component and then left to right along the components.

EXAMPLES:

```
sage: TableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).to_permutation()
[12, 11, 9, 10, 8, 5, 6, 7, 3, 4, 1, 2]
```

to_word()
Returns a word obtained from a row reading of the tableau tuple `self`.

EXAMPLES:

```
sage: TableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).to_word_by_row()
word: 12,11,9,10,8,5,6,7,3,4,1,2
```

to_word_by_column()
Returns the word obtained from a column reading of the tableau tuple `self`.

EXAMPLES:

```
sage: TableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).to_word_by_column()
word: 12,11,9,10,8,5,6,7,3,1,4,2
```

to_word_by_row()
Returns a word obtained from a row reading of the tableau tuple `self`.

EXAMPLES:

```
sage: TableauTuple([[[1,2],[3,4]],[[5,6,7],[8]],[[9,10],[11],[12]]]).to_word_by_row()
word: 12,11,9,10,8,5,6,7,3,4,1,2
```

up(*n=None*)
An iterator for all the `TableauTuple` that can be obtained from `self` by adding a cell with the label `n`. If `n` is not specified then a cell with label `n` will be added to the tableau tuple, where `n-1` is the size of the tableau tuple before any cells are added.

EXAMPLES:

```
sage: list(TableauTuple([[[1,2]],[[3]]]).up())
[([[1, 2, 4]], [[3]]),
 ([[1, 2], [4]], [[3]]),
 ([[1, 2]], [[3, 4]]),
 ([[1, 2]], [[3], [4]])]
```

class sage.combinat.tableau_tuple.**TableauTuples**
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

A factory class for the various classes of tableau tuples.

INPUT:

There are three optional arguments:

- `shape` – determines a `PartitionTuple` which gives the shape of the `TableauTuples`
- `level` – the level of the tableau tuples (positive integer)
- `size` – the size of the tableau tuples (non-negative integer)

It is not necessary to use the keywords. If they are not specified then the first integer argument specifies the `level` and the second the `size` of the tableaux.

OUTPUT:

- The corresponding class of tableau tuples.

The entries of a tableau can be any sage object. Because of this, no enumeration of the set of `TableauTuples` is possible.

EXAMPLES:

```
sage: T3 = TableauTuples(3); T3
Tableau tuples of level 3
sage: [['a','b']] in TableauTuples()
True
sage: [['a','b']] in TableauTuples(level=3)
False
sage: t = TableauTuples(level=3)([[],[[1,1,1]],[]]); t
([], [[1, 1, 1]], [])
sage: t in T3
True
sage: t in TableauTuples()
True
sage: t in TableauTuples(size=3)
True
sage: t in TableauTuples(size=4)
False
sage: t in StandardTableauTuples()
False
sage: t.parent()
Tableau tuples of level 3
sage: t.category()
Category of elements of Tableau tuples of level 3
```

TESTS:

```
sage: TableauTuples(0)
Traceback (most recent call last):
...
ValueError: the level must be a positive integer

sage: t = TableauTuples(3)([[],[],[[1,2],[3]]])
sage: t.parent()
Tableau tuples of level 3
sage: TableauTuples(t)
Traceback (most recent call last):
...
ValueError: the level must be a positive integer
sage: TableauTuples(3)([[1, 1]])
Traceback (most recent call last):
...
ValueError: [[1, 1]] is not an element of Tableau tuples of level 3
```

```
sage: t0 = Tableau([[1]])
sage: t1 = TableauTuples()([[1]])
sage: t2 = TableauTuples()(t1)
sage: t0 == t1 == t2
True
sage: t1 in TableauTuples()
True
sage: t1 in TableauTuples(1)
True
sage: t1 in TableauTuples(2)
False

sage: [[1]] in TableauTuples()
True
sage: [] in TableauTuples()
True

sage: TableauTuples(level=0)
Traceback (most recent call last):
...
ValueError: the level must be a positive integer

sage: TestSuite( TableauTuples() ).run()
sage: TestSuite( TableauTuples(level=1) ).run()
sage: TestSuite( TableauTuples(level=2) ).run()
sage: TestSuite( TableauTuples(level=6) ).run()
sage: TestSuite( TableauTuples(size=0) ).run()
sage: TestSuite( TableauTuples(size=1) ).run()
sage: TestSuite( TableauTuples(size=2) ).run()
sage: TestSuite( TableauTuples(size=10) ).run()
sage: TestSuite( TableauTuples(level=1, size=0) ).run()
sage: TestSuite( TableauTuples(level=1, size=1) ).run()
sage: TestSuite( TableauTuples(level=1, size=10) ).run()
sage: TestSuite( TableauTuples(level=2, size=0) ).run()
sage: TestSuite( TableauTuples(level=2, size=1) ).run()
sage: TestSuite( TableauTuples(level=2, size=10) ).run()
sage: TestSuite( TableauTuples(level=6, size=0) ).run()
sage: TestSuite( TableauTuples(level=6, size=1) ).run()
sage: TestSuite( TableauTuples(level=6, size=10) ).run()
```

Check that trac ticket #14145 has been fixed:

```
sage: 1 in TableauTuples()
False
```

Element
alias of `TableauTuple`

global_options(**get_value*, ***set_value*)
Sets the global options for elements of the tableau, skew_tableau, and tableau tuple classes. The defaults are for tableau to be displayed as a list, latexed as a Young diagram using the English convention.

OPTIONS:

- `ascii_art` – (default: `repr`) Controls the ascii art output for tableaux
  - `compact` – minimal length ascii art
  - `repr` – display using the diagram string representation

  - –table – display as a table
- convention – (default: English) Sets the convention used for displaying tableaux and partitions
  - –English – use the English convention
  - –French – use the French convention
- display – (default: list) Controls the way in which tableaux are printed
  - –array – alias for diagram
  - –compact – minimal length string representation
  - –diagram – display as Young diagram (similar to pp())
  - –ferrers_diagram – alias for diagram
  - –list – print tableaux as lists
  - –young_diagram – alias for diagram
- latex – (default: diagram) Controls the way in which tableaux are latexed
  - –array – alias for diagram
  - –diagram – as a Young diagram
  - –ferrers_diagram – alias for diagram
  - –list – as a list
  - –young_diagram – alias for diagram
- notation – alternative name for convention

Note: Changing the convention for tableaux also changes the convention for partitions.

If no parameters are set, then the function returns a copy of the options dictionary.

EXAMPLES:

```
sage: T = Tableau([[1,2,3],[4,5]])
sage: T
[[1, 2, 3], [4, 5]]
sage: Tableaux.global_options(display="array")
sage: T
  1  2  3
  4  5
sage: Tableaux.global_options(convention="french")
sage: T
  4  5
  1  2  3
```

Changing the convention for tableaux also changes the convention for partitions and vice versa:

```
sage: P = Partition([3,3,1])
sage: print P.ferrers_diagram()
*
***
***
sage: Partitions.global_options(convention="english")
sage: print P.ferrers_diagram()
***
***
*
```

```
sage: T
  1  2  3
  4  5
```

The ASCII art can also be changed:

```
sage: t = Tableau([[1,2,3],[4,5]])
sage: ascii_art(t)
  1  2  3
  4  5
sage: Tableaux.global_options(ascii_art="table")
sage: ascii_art(t)
+---+---+
| 4 | 5 |
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
sage: Tableaux.global_options(ascii_art="compact")
sage: ascii_art(t)
|4|5|
|1|2|3|
sage: Tableaux.global_options.reset()
```

See `GlobalOptions` for more features of these options.

level()

Return the `level` of a tableau tuple in `self`, or `None` if different tableau tuples in `self` can have different sizes. The `level` of a tableau tuple is just the level of the underlying `PartitionTuple`.

EXAMPLES:

```
sage: TableauTuples().level() is None
True
sage: TableauTuples(7).level()
7
```

level_one_parent_class

alias of `Tableaux_all`

list()

If the set of tableau tuples `self` is finite then this function returns the list of these tableau tuples. If the class is infinite an error is returned.

EXAMPLES:

```
sage: StandardTableauTuples([[2,1],[2]]).list()
[([[1, 2], [3]], [[4, 5]]),
 ([[1, 3], [2]], [[4, 5]]),
 ([[1, 2], [4]], [[3, 5]]),
 ([[1, 3], [4]], [[2, 5]]),
 ([[2, 3], [4]], [[1, 5]]),
 ([[1, 4], [2]], [[3, 5]]),
 ([[1, 4], [3]], [[2, 5]]),
 ([[2, 4], [3]], [[1, 5]]),
 ([[1, 2], [5]], [[3, 4]]),
 ([[1, 3], [5]], [[2, 4]]),
 ([[2, 3], [5]], [[1, 4]]),
 ([[1, 4], [5]], [[2, 3]]),
 ([[2, 4], [5]], [[1, 3]]),
 ([[3, 4], [5]], [[1, 2]]),
```

```
([[1, 5], [2]], [[3, 4]]),
([[1, 5], [3]], [[2, 4]]),
([[2, 5], [3]], [[1, 4]]),
([[1, 5], [4]], [[2, 3]]),
([[2, 5], [4]], [[1, 3]]),
([[3, 5], [4]], [[1, 2]])]
```

size()

Return the `size` of a tableau tuple in `self`, or `None` if different tableau tuples in `self` can have sifferent sizes. The `size` of a tableau tuple is just the size of the underlying `PartitionTuple`.

EXAMPLES:

```
sage: TableauTuples(size=14).size()
14
```

class sage.combinat.tableau_tuple.**TableauTuples_all**

Bases: sage.combinat.tableau_tuple.TableauTuples

The parent class of all `TableauTuples`, with arbitrary `level` and `size`.

an_element()

Returns a particular element of the class.

EXAMPLES:

```
sage: TableauTuples().an_element()
([[1]], [[2]], [[3]], [[4]], [[5]], [[6]], [[7]])
```

class sage.combinat.tableau_tuple.**TableauTuples_level**(*level*)

Bases: sage.combinat.tableau_tuple.TableauTuples

Class of all `TableauTuples` with a fixed `level` and arbitrary `size`.

an_element()

Returns a particular element of the class.

EXAMPLES:

```
sage: TableauTuples(3).an_element()
([], [], [])
sage: TableauTuples(5).an_element()
([], [], [], [], [])
sage: T = TableauTuples(0)
Traceback (most recent call last):
...
ValueError: the level must be a positive integer
```

class sage.combinat.tableau_tuple.**TableauTuples_level_size**(*level*, *size*)

Bases: sage.combinat.tableau_tuple.TableauTuples

Class of all `TableauTuples` with a fixed `level` and a fixed `size`.

an_element()

Returns a particular element of the class.

EXAMPLES:

```
sage: TableauTuples(3,0).an_element()
([], [], [])
sage: TableauTuples(3,1).an_element()
([[1]], [], [])
```

```
sage: TableauTuples(3,2).an_element()
([[1, 2]], [], [])
```

class sage.combinat.tableau_tuple.**TableauTuples_size**(*size*)
Bases: sage.combinat.tableau_tuple.TableauTuples

Class of all TableauTuples with a arbitrary level and fixed size.

an_element()
Returns a particular element of the class.

EXAMPLES:

```
sage: TableauTuples(size=3).an_element()
([], [[1, 2, 3]], [])
sage: TableauTuples(size=0).an_element()
([], [], [])
```

5.1.312 Generalized Tamari lattices

These lattices depend on three parameters a, b and m, where a and b are coprime positive integers and m is a nonnegative integer.

The elements are Dyck paths in the $(a \times b)$-rectangle. The order relation depends on m.

To use the provided functionality, you should import Generalized Tamari lattices by typing:

```
sage: from sage.combinat.tamari_lattices import GeneralizedTamariLattice
```

Then,

```
sage: GeneralizedTamariLattice(3,2)
Finite lattice containing 2 elements
sage: GeneralizedTamariLattice(4,3)
Finite lattice containing 5 elements
```

The classical **Tamari lattices** are special cases of this construction and are also available directly using the catalogue of posets, as follows:

```
sage: posets.TamariLattice(3)
Finite lattice containing 5 elements
```

See also:

For more detailed information see TamariLattice(), GeneralizedTamariLattice().

sage.combinat.tamari_lattices.**GeneralizedTamariLattice**(*a*, *b*, *m=1*)
Return the (a, b)-Tamari lattice of parameter m.

INPUT:

- a and b coprime integers with $a \geq b$
- m a nonnegative integer such that $a \geq b \times m$

OUTPUT:

- a finite lattice (the lattice property is only conjectural in general)

The elements of the lattice are Dyck paths in the $(a \times b)$-rectangle.

The parameter m (slope) is used only to define the covering relations. When the slope m is 0, two paths are comparable if and only if one is always above the other.

The usual Tamari lattice of index b is the special case $a = b + 1$ and $m = 1$.

Other special cases give the m-Tamari lattices studied in [BMFPR].

EXAMPLES:

```
sage: from sage.combinat.tamari_lattices import GeneralizedTamariLattice
sage: GeneralizedTamariLattice(3,2)
Finite lattice containing 2 elements
sage: GeneralizedTamariLattice(4,3)
Finite lattice containing 5 elements
sage: GeneralizedTamariLattice(4,4)
Traceback (most recent call last):
...
ValueError: The numbers a and b must be coprime with a>=b.
sage: GeneralizedTamariLattice(7,5,2)
Traceback (most recent call last):
...
ValueError: The condition a>=b*m does not hold.
sage: P = GeneralizedTamariLattice(5,3);P
Finite lattice containing 7 elements
```

TESTS:

```
sage: P.coxeter_transformation()**18 == 1
True
```

REFERENCES:

sage.combinat.tamari_lattices.**TamariLattice**(n)

Return the n-th Tamari lattice.

INPUT:

- n a nonnegative integer

OUTPUT:

- a finite lattice

The elements of the lattice are Dyck paths in the $(n + 1 \times n)$-rectangle.

See Tamari lattice for mathematical background.

EXAMPLES:

```
sage: posets.TamariLattice(3)
Finite lattice containing 5 elements
```

sage.combinat.tamari_lattices.**paths_in_triangle**(i, j, a, b)

Return all Dyck paths from $(0, 0)$ to (i, j) in the $(a \times b)$-rectangle.

This means that at each step of the path, one has $ay \geq bx$.

A path is represented by a sequence of 0 and 1, where 0 is an horizontal step $(1, 0)$ and 1 is a vertical step $(0, 1)$.

INPUT:

- a and b coprime integers with $a \geq b$
- i and j nonnegative integers with $1 \geq \frac{j}{b} \geq \frac{bi}{a} \geq 0$

OUTPUT:

- a list of paths

EXAMPLES:

```
sage: from sage.combinat.tamari_lattices import paths_in_triangle
sage: paths_in_triangle(2,2,2,2)
[(1, 0, 1, 0), (1, 1, 0, 0)]
sage: paths_in_triangle(2,3,4,4)
[(1, 0, 1, 0, 1), (1, 1, 0, 0, 1), (1, 0, 1, 1, 0),
(1, 1, 0, 1, 0), (1, 1, 1, 0, 0)]
sage: paths_in_triangle(2,1,4,4)
Traceback (most recent call last):
...
ValueError: The endpoint is not valid.
sage: paths_in_triangle(3,2,5,3)
[(1, 0, 1, 0, 0), (1, 1, 0, 0, 0)]
```

sage.combinat.tamari_lattices.**swap**(*p*, *i*, *m=1*)

Perform a covering move in the (a, b)-Tamari lattice of parameter m.

The letter at position i in p must be a 0, followed by at least one 1.

INPUT:

- p a Dyck path in the $(a \times b)$-rectangle
- i an integer between 0 and $a + b - 1$

OUTPUT:

- a Dyck path in the $(a \times b)$-rectangle

EXAMPLES:

```
sage: from sage.combinat.tamari_lattices import swap
sage: swap((1,0,1,0),1)
(1, 1, 0, 0)
sage: swap((1,0,1,0),6)
Traceback (most recent call last):
...
ValueError: The index is greater than the length of the path.
sage: swap((1,1,0,0,1,1,0,0),3)
(1, 1, 0, 1, 1, 0, 0, 0)
sage: swap((1,1,0,0,1,1,0,0),2)
Traceback (most recent call last):
...
ValueError: There is no such covering move.
```

5.1.313 Tiling Solver

Tiling a n-dimensional box into non-intersecting n-dimensional polyominoes.

This uses dancing links code which is in Sage. Dancing links were originally introduced by Donald Knuth in 2000 [Knuth1]. In particular, Knuth used dancing links to solve tilings of a region by 2d pentaminoes. Here we extend the method to any dimension.

In particular, the `sage.games.quantumino` module is based on the Tiling Solver and allows to solve the 3d Quantumino puzzle.

This module defines two classes:

- sage.combinat.tiling.Polyomino class, to represent polyominoes in arbitrary dimension. The goal of this class is to return all the rotated, reflected and/or translated copies of a polyomino that are contained in a certain box.
- sage.combinat.tiling.TilingSolver class, to solve the general problem of tiling a rectangular n-dimensional box with a set of n-dimensional polyominoes. One can specify if rotations and reflections are allowed or not and if pieces can be reused or not. This class convert the tiling data into rows of a matrix that are passed to the DLX solver. It also allows to compute the number of solutions.

AUTHOR:

- Sebastien Labbe, June 2011, initial version
- Sebastien Labbe, July 2015, count solutions up to rotations

EXAMPLES:

2d Easy Example

Here is a 2d example. Let us try to fill the 3×2 rectangle with a 1×2 rectangle and a 2×2 square. Obviously, there are two solutions:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0), (0,1)])
sage: q = Polyomino([(0,0), (0,1), (1,0), (1,1)])
sage: T = TilingSolver([p,q], box=[3,2])
sage: it = T.solve()
sage: next(it)
[Polyomino: [(0, 0), (0, 1), (1, 0), (1, 1)], Color: gray, Polyomino: [(2, 0), (2, 1)], Color: gray]
sage: next(it)
[Polyomino: [(1, 0), (1, 1), (2, 0), (2, 1)], Color: gray, Polyomino: [(0, 0), (0, 1)], Color: gray]
sage: next(it)
Traceback (most recent call last):
...
StopIteration
sage: T.number_of_solutions()
2
```

1d Easy Example

Here is an easy one dimensional example where we try to tile a stick of length 6 with three sticks of length 1, 2 and 3. There are six solutions:

```
sage: p = Polyomino([[0]])
sage: q = Polyomino([[0],[1]])
sage: r = Polyomino([[0],[1],[2]])
sage: T = TilingSolver([p,q,r], box=[6])
sage: len(T.rows())
15
sage: it = T.solve()
sage: next(it)
[Polyomino: [(0,)], Color: gray, Polyomino: [(1,), (2,)], Color: gray, Polyomino: [(3,), (4,), (5,)],
sage: next(it)
[Polyomino: [(0,)], Color: gray, Polyomino: [(1,), (2,), (3,)], Color: gray, Polyomino: [(4,), (5,)],
sage: T.number_of_solutions()
6
```

2d Puzzle allowing reflections

The following is a puzzle owned by Florent Hivert:

```
sage: from sage.combinat.tiling import Polyomino, TilingSolver
sage: L = []
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(1,3)], 'yellow'))
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2)], "black"))
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,3)], "gray"))
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(0,3),(1,0),(1,3)],"cyan"))
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(0,3),(1,0),(1,1)],"red"))
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(0,3),(1,1),(1,2)],"blue"))
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(0,3),(1,1),(1,3)],"green"))
sage: L.append(Polyomino([(0,1),(0,2),(0,3),(1,0),(1,1),(1,3)],"magenta"))
sage: L.append(Polyomino([(0,1),(0,2),(0,3),(1,0),(1,1),(1,2)],"orange"))
sage: L.append(Polyomino([(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)],"pink"))
```

By default, rotations are allowed and reflections are not. In this case, there are no solution:

```
sage: T = TilingSolver(L, (8,8))
sage: T.number_of_solutions()                          # long time (2.5 s)
0
```

If reflections are allowed, there are solutions. Solve the puzzle and show one solution:

```
sage: T = TilingSolver(L, (8,8), reflection=True)
sage: solution = next(T.solve())                                    # long time (7s)
sage: G = sum([piece.show2d() for piece in solution], Graphics()) # long time (<1s)
sage: G.show(aspect_ratio=1, axes=False)                            # long time (2s)
```

Compute the number of solutions:

```
sage: T.number_of_solutions()                            # long time (2.6s)
328
```

Create a animation of all the solutions:

```
sage: a = T.animate()                                 # not tested
sage: a                                               # not tested
Animation with 328 frames
```

3d Puzzle

The same thing done in 3d *without* allowing reflections this time:

```
sage: from sage.combinat.tiling import Polyomino, TilingSolver
sage: L = []
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(0,3,0),(1,0,0),(1,1,0),(1,2,0),(1,3,0)]))
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(0,3,0),(1,0,0),(1,1,0),(1,2,0)]))
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(0,3,0),(1,0,0),(1,1,0),(1,3,0)]))
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(0,3,0),(1,0,0),(1,3,0)]))
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(0,3,0),(1,0,0),(1,1,0)]))
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(0,3,0),(1,1,0),(1,2,0)]))
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(0,3,0),(1,1,0),(1,3,0)]))
sage: L.append(Polyomino([(0,1,0),(0,2,0),(0,3,0),(1,0,0),(1,1,0),(1,3,0)]))
sage: L.append(Polyomino([(0,1,0),(0,2,0),(0,3,0),(1,0,0),(1,1,0),(1,2,0)]))
sage: L.append(Polyomino([(0,0,0),(0,1,0),(0,2,0),(1,0,0),(1,1,0),(1,2,0)]))
```

Solve the puzzle and show one solution:

```
sage: T = TilingSolver(L, (8,8,1))
sage: solution = next(T.solve())                                    # long time (8s)
sage: G = sum([p.show3d(size=0.85) for p in solution], Graphics()) # long time (<1s)
sage: G.show(aspect_ratio=1, viewer='tachyon')                      # long time (2s)
```

Let us compute the number of solutions:

```
sage: T.number_of_solutions()                              # long time (3s)
328
```

Donald Knuth example : the Y pentamino

Donald Knuth [Knuth1] considered the problem of packing 45 Y pentaminoes into a 15×15 square:

```
sage: from sage.combinat.tiling import Polyomino, TilingSolver
sage: y = Polyomino([(0,0),(1,0),(2,0),(3,0),(2,1)])
sage: T = TilingSolver([y], box=(5,10), reusable=True, reflection=True)
sage: T.number_of_solutions()
10
sage: solution = next(T.solve())
sage: G = sum([p.show2d() for p in solution], Graphics())
sage: G.show(aspect_ratio=1)                          # long time (2s)

sage: T = TilingSolver([y], box=(15,15), reusable=True, reflection=True)
sage: T.number_of_solutions()                         # not tested
212
```

Animation of Donald Knuth's dancing links

Animation of the solutions:

```
sage: from sage.combinat.tiling import Polyomino, TilingSolver
sage: Y = Polyomino([(0,0),(1,0),(2,0),(3,0),(2,1)], color='yellow')
sage: T = TilingSolver([Y], box=(15,15), reusable=True, reflection=True)
sage: a = T.animate(stop=40)            # long time  # optional -- ImageMagick
sage: a                                 # long time  # optional -- ImageMagick
Animation with 40 frames
```

Incremental animation of the solutions (one piece is removed/added at a time):

```
sage: a = T.animate('incremental', stop=40)    # long time  # optional -- ImageMagick
sage: a                                        # long time  # optional -- ImageMagick
Animation with 40 frames
sage: a.show(delay=50, iterations=1)           # long time  # optional -- ImageMagick
```

5d Easy Example

Here is a 5d example. Let us try to fill the $2 \times 2 \times 2 \times 2 \times 2$ rectangle with reusable $1 \times 1 \times 1 \times 1 \times 1$ rectangles. Obviously, there is one solution:

```
sage: from sage.combinat.tiling import Polyomino, TilingSolver
sage: p = Polyomino([(0,0,0,0,0)])
sage: T = TilingSolver([p], box=(2,2,2,2,2), reusable=True)
sage: rows = T.rows()                                    # long time (3s)
sage: rows                                               # long time (fast)
[[0], [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [1
sage: T.number_of_solutions()                            # long time (fast)
1
```

REFERENCES:

class sage.combinat.tiling.**Polyomino**(*coords*, *color='gray'*)

Bases: sage.structure.sage_object.SageObject

Return the polyomino defined by a set of coordinates.

The polyomino is the union of the unit square (or cube, or n-cube) centered at those coordinates. Such an object should be connected, but the code do not make this assumption.

INPUT:

- coords - iterable of tuple
- color - string (optional, default: 'gray'), the color

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: Polyomino([(0,0,0), (0,1,0), (1,1,0), (1,1,1)], color='blue')
Polyomino: [(0, 0, 0), (0, 1, 0), (1, 1, 0), (1, 1, 1)], Color: blue
```

boundary()

Return the boundary of a 2d polyomino.

INPUT:

- self - a 2d polyomino

OUTPUT:

- list of edges (an edge is a pair of adjacent 2d coordinates)

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0), (1,0), (0,1), (1,1)])
sage: p.boundary()
[((0.5, 1.5), (1.5, 1.5)), ((-0.5, -0.5), (0.5, -0.5)), ((0.5, -0.5), (1.5, -0.5)), ((-0.5,
sage: len(_)
8
sage: p = Polyomino([(5,5)])
sage: p.boundary()
[((4.5, 5.5), (5.5, 5.5)), ((4.5, 4.5), (5.5, 4.5)), ((4.5, 4.5), (4.5, 5.5)), ((5.5, 4.5),
```

bounding_box()

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0),(1,0,0),(1,1,0),(1,1,1),(1,2,0)], color='deeppink')
sage: p.bounding_box()
[[0, 0, 0], [1, 2, 1]]
```

canonical()

Returns the translated copy of self having minimal and nonnegative coordinates

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0),(1,0,0),(1,1,0),(1,1,1),(1,2,0)], color='deeppink')
sage: p
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color: deeppink
sage: p.canonical()
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color: deeppink
```

TESTS:

```
sage: p
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color: deeppink
sage: p + (3,4,5)
Polyomino: [(3, 4, 5), (4, 4, 5), (4, 5, 5), (4, 5, 6), (4, 6, 5)], Color: deeppink
sage: (p + (3,4,5)).canonical()
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color: deeppink
```

canonical_isometric_copies(*orientation_preserving=True*, *mod_box_isometries=False*)

Return the set of image of self under isometries of the n-cube where the coordinates are all nonnegative and minimal.

INPUT:

- `orientation_preserving` – bool (optional, default: `True`), If True, the group of isometries of the n-cube is restricted to those that preserve the orientation, i.e. of determinant 1.
- `mod_box_isometries` – bool (default: `False`), whether to quotient the group of isometries of the n-cube by the subgroup of isometries of the $a_1 \times a_2 \cdots \times a_n$ rectangular box where are the a_i are assumed to be distinct.

OUTPUT:

set of Polyomino

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0), (0,1,0), (1,1,0), (1,1,1)], color='blue')
sage: s = p.canonical_isometric_copies()
sage: len(s)
12
```

With the non orientation-preserving:

```
sage: s = p.canonical_isometric_copies(orientation_preserving=False)
sage: len(s)
24
```

Modulo rotation by angle 180 degrees:

```
sage: s = p.canonical_isometric_copies(mod_box_isometries=True)
sage: len(s)
3
```

TESTS:

```
sage: from sage.games.quantumino import pentaminos
sage: [len(p.canonical_isometric_copies((5,8,2), mod_box_isometries=False)) for p in pentami
[24, 24, 24, 24, 24, 24, 12, 12, 24, 24, 24, 24, 12, 12, 24, 24, 12]
```

```
sage: [len(p.canonical_isometric_copies((5,8,2), mod_box_isometries=True)) for p in pentamir
[6, 6, 6, 6, 6, 6, 3, 3, 6, 6, 6, 6, 3, 3, 6, 6, 3]
```

canonical_orthogonals(**args*, ***kwds*)

Deprecated: Use `canonical_isometric_copies()` instead. See trac ticket #19107 for details.

center()

Return the center of the polyomino.

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0),(0,0,1)])
sage: p.center()
(0, 0, 1/2)
```

In 3d:

```
sage: p = Polyomino([(0,0,0),(1,0,0),(1,1,0),(1,1,1),(1,2,0)], color='deeppink')
sage: p.center()
(4/5, 4/5, 1/5)
```

In 2d:

```
sage: p = Polyomino([(0,0),(1,0),(1,1),(1,2)])
sage: p.center()
(3/4, 3/4)
```

color()

Return the color of the polyomino.

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0), (0,1,0), (1,1,0), (1,1,1)], color='blue')
sage: p.color()
'blue'
```

isometric_copies(*box*, *orientation_preserving=True*, *mod_box_isometries=False*)

Return the translated and isometric images of self that lies in the box.

INPUT:

- `box` – tuple of integers, size of the box
- `orientation_preserving` – bool (optional, default: `True`), If True, the group of isometries of the n-cube is restricted to those that preserve the orientation, i.e. of determinant 1.
- `mod_box_isometries` – bool (default: `False`), whether to quotient the group of isometries of the n-cube by the subgroup of isometries of the $a_1 \times a_2 \cdots \times a_n$ rectangular box where are the a_i are assumed to be distinct.

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0),(1,0,0),(1,1,0),(1,1,1),(1,2,0)], color='deeppink')
sage: L = list(p.isometric_copies(box=(5,8,2)))
sage: len(L)
360

sage: p = Polyomino([(0,0,0),(1,0,0),(1,1,0),(1,2,0),(1,2,1)], color='orange')
sage: L = list(p.isometric_copies(box=(5,8,2)))
```

```
sage: len(L)
180
sage: L = list(p.isometric_copies((5,8,2), False))
sage: len(L)
360
sage: L = list(p.isometric_copies((5,8,2), mod_box_isometries=True))
sage: len(L)
45
```

neighbor_edges()

Return an iterator over the pairs of neighbor coordinates of the polyomino.

Two points P and Q are neighbor if $P - Q$ has one coordinate equal to $+1$ or -1 and zero everywhere else.

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0),(0,0,1)])
sage: list(sorted(edge) for edge in p.neighbor_edges())
[[(0, 0, 0), (0, 0, 1)]]
```

In 3d:

```
sage: p = Polyomino([(0,0,0),(1,0,0),(1,1,0),(1,1,1),(1,2,0)], color='deeppink')
sage: L = sorted(sorted(edge) for edge in p.neighbor_edges())
sage: for a in L: a
[(0, 0, 0), (1, 0, 0)]
[(1, 0, 0), (1, 1, 0)]
[(1, 1, 0), (1, 1, 1)]
[(1, 1, 0), (1, 2, 0)]
```

In 2d:

```
sage: p = Polyomino([(0,0),(1,0),(1,1),(1,2)])
sage: L = sorted(sorted(edge) for edge in p.neighbor_edges())
sage: for a in L: a
[(0, 0), (1, 0)]
[(1, 0), (1, 1)]
[(1, 1), (1, 2)]
```

show2d(*size=0.7*, *color='black'*, *thickness=1*)

Returns a 2d Graphic object representing the polyomino.

INPUT:

- `self` - a polyomino of dimension 2
- `size` - number (optional, default: `0.7`), the size of each square.
- `color` - color (optional, default: `'black'`), color of the boundary line.
- `thickness` - number (optional, default: `1`), how thick the boundary line is.

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0),(1,0),(1,1),(1,2)], color='deeppink')
sage: p.show2d()              # long time (0.5s)
Graphics object consisting of 17 graphics primitives
```

show3d(*size=1*)

Returns a 3d Graphic object representing the polyomino.

INPUT:

- `self` - a polyomino of dimension 3
- `size` - number (optional, default: `1`), the size of each `1 \times 1 \times 1` cube. This does a homothety with respect to the center of the polyomino.

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0), (0,1,0), (1,1,0), (1,1,1)], color='blue')
sage: p.show3d()                # long time (2s)
Graphics3d Object
```

translated(**args*, ***kwds*)

Deprecated: Use `translated_copies()` instead. See trac ticket #19107 for details.

translated_copies(*box*)

Returns an iterator over the translated images of self inside a box.

INPUT:

- `box` – tuple of integers, size of the box

OUTPUT:

iterator of 3d polyominoes

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino
sage: p = Polyomino([(0,0,0),(1,0,0),(1,1,0),(1,1,1),(1,2,0)], color='deeppink')
sage: for t in p.translated_copies(box=(5,8,2)): t
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color: deeppink
Polyomino: [(0, 1, 0), (1, 1, 0), (1, 2, 0), (1, 2, 1), (1, 3, 0)], Color: deeppink
Polyomino: [(0, 2, 0), (1, 2, 0), (1, 3, 0), (1, 3, 1), (1, 4, 0)], Color: deeppink
Polyomino: [(0, 3, 0), (1, 3, 0), (1, 4, 0), (1, 4, 1), (1, 5, 0)], Color: deeppink
Polyomino: [(0, 4, 0), (1, 4, 0), (1, 5, 0), (1, 5, 1), (1, 6, 0)], Color: deeppink
Polyomino: [(0, 5, 0), (1, 5, 0), (1, 6, 0), (1, 6, 1), (1, 7, 0)], Color: deeppink
Polyomino: [(1, 0, 0), (2, 0, 0), (2, 1, 0), (2, 1, 1), (2, 2, 0)], Color: deeppink
Polyomino: [(1, 1, 0), (2, 1, 0), (2, 2, 0), (2, 2, 1), (2, 3, 0)], Color: deeppink
Polyomino: [(1, 2, 0), (2, 2, 0), (2, 3, 0), (2, 3, 1), (2, 4, 0)], Color: deeppink
Polyomino: [(1, 3, 0), (2, 3, 0), (2, 4, 0), (2, 4, 1), (2, 5, 0)], Color: deeppink
Polyomino: [(1, 4, 0), (2, 4, 0), (2, 5, 0), (2, 5, 1), (2, 6, 0)], Color: deeppink
Polyomino: [(1, 5, 0), (2, 5, 0), (2, 6, 0), (2, 6, 1), (2, 7, 0)], Color: deeppink
Polyomino: [(2, 0, 0), (3, 0, 0), (3, 1, 0), (3, 1, 1), (3, 2, 0)], Color: deeppink
Polyomino: [(2, 1, 0), (3, 1, 0), (3, 2, 0), (3, 2, 1), (3, 3, 0)], Color: deeppink
Polyomino: [(2, 2, 0), (3, 2, 0), (3, 3, 0), (3, 3, 1), (3, 4, 0)], Color: deeppink
Polyomino: [(2, 3, 0), (3, 3, 0), (3, 4, 0), (3, 4, 1), (3, 5, 0)], Color: deeppink
Polyomino: [(2, 4, 0), (3, 4, 0), (3, 5, 0), (3, 5, 1), (3, 6, 0)], Color: deeppink
Polyomino: [(2, 5, 0), (3, 5, 0), (3, 6, 0), (3, 6, 1), (3, 7, 0)], Color: deeppink
Polyomino: [(3, 0, 0), (4, 0, 0), (4, 1, 0), (4, 1, 1), (4, 2, 0)], Color: deeppink
Polyomino: [(3, 1, 0), (4, 1, 0), (4, 2, 0), (4, 2, 1), (4, 3, 0)], Color: deeppink
Polyomino: [(3, 2, 0), (4, 2, 0), (4, 3, 0), (4, 3, 1), (4, 4, 0)], Color: deeppink
Polyomino: [(3, 3, 0), (4, 3, 0), (4, 4, 0), (4, 4, 1), (4, 5, 0)], Color: deeppink
Polyomino: [(3, 4, 0), (4, 4, 0), (4, 5, 0), (4, 5, 1), (4, 6, 0)], Color: deeppink
Polyomino: [(3, 5, 0), (4, 5, 0), (4, 6, 0), (4, 6, 1), (4, 7, 0)], Color: deeppink
```

This method is independant of the translation of the polyomino:

```
sage: q = Polyomino([(0,0,0), (1,0,0)])
sage: list(q.translated_copies((2,2,1)))
[Polyomino: [(0, 0, 0), (1, 0, 0)], Color: gray, Polyomino: [(0, 1, 0), (1, 1, 0)], Color: g
sage: q = Polyomino([(34,7,-9), (35,7,-9)])
sage: list(q.translated_copies((2,2,1)))
[Polyomino: [(0, 0, 0), (1, 0, 0)], Color: gray, Polyomino: [(0, 1, 0), (1, 1, 0)], Color: g
```

Inside smaller boxes:

```
sage: list(p.translated_copies(box=(2,2,3)))
[]
sage: list(p.translated_copies(box=(2,3,2)))
[Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color: deeppink]
sage: list(p.translated_copies(box=(3,2,2)))
[]
sage: list(p.translated_copies(box=(1,1,1)))
[]
sage: list(p.translated_copies(box=(1,1,-1)))
[]
```

translated_orthogonals(**args*, ***kwds*)

Deprecated: Use `isometric_copies()` instead. See trac ticket #19107 for details.

class sage.combinat.tiling.**TilingSolver**(*pieces*, *box*, *rotation=True*, *reflection=False*, *reusable=False*)

Bases: sage.structure.sage_object.SageObject

Tiling solver

Solve the problem of tiling a rectangular box with a certain number of pieces, called polyominoes, where each polyomino must be used exactly once.

INPUT:

- `pieces` - iterable of Polyominoes
- `box` - tuple, size of the box
- `rotation` - bool (optional, default: `True`), whether to allow rotations
- `reflection` - bool (optional, default: `False`), whether to allow reflections
- `reusable` - bool (optional, default: `False`), whether to allow the pieces to be reused

EXAMPLES:

By default, rotations are allowed and reflections are not allowed:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: T
Tiling solver of 3 pieces into the box (1, 1, 6)
Rotation allowed: True
Reflection allowed: False
Reusing pieces allowed: False
```

Solutions are given by an iterator:

```
sage: it = T.solve()
sage: for p in next(it): p
```

```
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2)], Color: gray
Polyomino: [(0, 0, 3), (0, 0, 4), (0, 0, 5)], Color: gray
```

Another solution:

```
sage: for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2), (0, 0, 3)], Color: gray
Polyomino: [(0, 0, 4), (0, 0, 5)], Color: gray
```

TESTS:

```
sage: T = TilingSolver([p,q,r], box=(1,1,6), rotation=False, reflection=True)
Traceback (most recent call last):
...
NotImplementedError: When reflection is allowed and rotation is not allowed
```

animate (*partial=None*, *stop=None*, *size=0.75*, *axes=False*)

Return an animation of evolving solutions.

INPUT:

- `partial` - string (optional, default: `None`), whether to include partial (incomplete) solutions. It can be one of the following:
  - `None` - include only complete solutions
  - `'common_prefix'` - common prefix between two consecutive solutions
  - `'incremental'` - one piece change at a time
- `stop` - integer (optional, default:`None`), number of frames
- `size` - number (optional, default: `0.75`), the size of each `1 \times 1` square. This does a homothety with respect to the center of each polyomino.
- `axes` - bool (optional, default:`False`), whether the x and y axes are shown.

EXAMPLES:

```
sage: from sage.combinat.tiling import Polyomino, TilingSolver
sage: y = Polyomino([(0,0),(1,0),(2,0),(3,0),(2,1)], color='cyan')
sage: T = TilingSolver([y], box=(5,10), reusable=True, reflection=True)
sage: a = T.animate()
sage: a                    # optional -- ImageMagick
Animation with 10 frames
```

Include partial solutions (common prefix between two consecutive solutions):

```
sage: a = T.animate('common_prefix')
sage: a              # optional -- ImageMagick
Animation with 19 frames
```

Incremental solutions (one piece removed or added at a time):

```
sage: a = T.animate('incremental')       # long time (2s)
sage: a                                  # long time (2s)  # optional -- ImageMagick
Animation with 123 frames

sage: a.show()                  # optional -- ImageMagick
```

The show function takes arguments to specify the delay between frames (measured in hundredths of a second, default value 20) and the number of iterations (default value 0, which means to iterate forever). To iterate 4 times with half a second between each frame:

```
sage: a.show(delay=50, iterations=4)  # optional -- ImageMagick
```

Limit the number of frames:

```
sage: a = T.animate('incremental', stop=13)      # not tested
sage: a                                          # not tested
Animation with 13 frames
```

coord_to_int_dict()

Returns a dictionary mapping coordinates to integers.

OUTPUT:

dict

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: A = T.coord_to_int_dict()
sage: sorted(A.iteritems())
[((0, 0, 0), 3), ((0, 0, 1), 4), ((0, 0, 2), 5), ((0, 0, 3), 6), ((0, 0, 4), 7), ((0, 0, 5),
```

Reusable pieces:

```
sage: p = Polyomino([(0,0), (0,1)])
sage: q = Polyomino([(0,0), (0,1), (1,0), (1,1)])
sage: T = TilingSolver([p,q], box=[3,2], reusable=True)
sage: B = T.coord_to_int_dict()
sage: sorted(B.iteritems())
[((0, 0), 0), ((0, 1), 1), ((1, 0), 2), ((1, 1), 3), ((2, 0), 4), ((2, 1), 5)]
```

dlx_solver()

Return the sage DLX solver of that 3d tiling problem.

OUTPUT:

DLX Solver

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: T.dlx_solver()
Dancing links solver for 9 columns and 15 rows
```

int_to_coord_dict()

Returns a dictionary mapping integers to coordinates.

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
```

```
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: B = T.int_to_coord_dict()
sage: sorted(B.iteritems())
[(3, (0, 0, 0)), (4, (0, 0, 1)), (5, (0, 0, 2)), (6, (0, 0, 3)), (7, (0, 0, 4)), (8, (0, 0,
```

Reusable pieces:

```
sage: from sage.combinat.tiling import Polyomino, TilingSolver
sage: p = Polyomino([(0,0), (0,1)])
sage: q = Polyomino([(0,0), (0,1), (1,0), (1,1)])
sage: T = TilingSolver([p,q], box=[3,2], reusable=True)
sage: B = T.int_to_coord_dict()
sage: sorted(B.iteritems())
[(0, (0, 0)), (1, (0, 1)), (2, (1, 0)), (3, (1, 1)), (4, (2, 0)), (5, (2, 1))]
```

TESTS:

The methods `int_to_coord_dict` and `coord_to_int_dict` returns dictionary that are inverse of each other:

```
sage: A = T.coord_to_int_dict()
sage: B = T.int_to_coord_dict()
sage: all(A[B[i]] == i for i in B)
True
sage: all(B[A[i]] == i for i in A)
True
```

is_suitable()

Return whether the volume of the box is equal to sum of the volume of the polyominoes and the number of rows sent to the DLX solver is larger than zero.

If these conditions are not verified, then the problem is not suitable in the sense that there are no solution.

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: T.is_suitable()
True
sage: T = TilingSolver([p,q,r], box=(1,1,7))
sage: T.is_suitable()
False
```

nrows_per_piece()

Return the number of rows necessary by each piece.

OUTPUT:

list

EXAMPLES:

```
sage: from sage.games.quantumino import QuantuminoSolver
sage: q = QuantuminoSolver(0)
sage: T = q.tiling_solver()
sage: T.nrows_per_piece()              # long time (10s)
[360, 360, 360, 360, 360, 180, 180, 672, 672, 360, 360, 180, 180, 360, 360, 180]
```

number_of_solutions()
Return the number of distinct solutions.

OUTPUT:

integer

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0)])
sage: q = Polyomino([(0,0), (0,1)])
sage: r = Polyomino([(0,0), (0,1), (0,2)])
sage: T = TilingSolver([p,q,r], box=(1,6))
sage: T.number_of_solutions()
6

sage: T = TilingSolver([p,q,r], box=(1,7))
sage: T.number_of_solutions()
0
```

pieces()
Return the list of pieces.

OUTPUT:

list of 3d polyominoes

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: for p in T._pieces: p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 0), (0, 0, 1)], Color: gray
Polyomino: [(0, 0, 0), (0, 0, 1), (0, 0, 2)], Color: gray
```

row_to_polyomino(*row_number*)
Return a polyomino associated to a row.

INPUT:

- `row_number` – integer, the i-th row

OUTPUT:

polyomino

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: a = Polyomino([(0,0,0), (0,0,1), (1,0,0)], color='blue')
sage: b = Polyomino([(0,0,0), (1,0,0), (0,1,0)], color='red')
sage: T = TilingSolver([a,b], box=(2,1,3))
sage: len(T.rows())
16

sage: T.row_to_polyomino(7)
Polyomino: [(0, 0, 1), (1, 0, 1), (1, 0, 2)], Color: blue
```

```
sage: T.row_to_polyomino(13)
Polyomino: [(0, 0, 1), (0, 0, 2), (1, 0, 1)], Color: red
```

rows()

Creation of the rows

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: rows = T.rows()
sage: for row in rows: row
[0, 3]
[0, 4]
[0, 5]
[0, 6]
[0, 7]
[0, 8]
[1, 3, 4]
[1, 4, 5]
[1, 5, 6]
[1, 6, 7]
[1, 8, 7]
[2, 3, 4, 5]
[2, 4, 5, 6]
[2, 5, 6, 7]
[2, 8, 6, 7]
```

rows_for_piece(*i*, *mod_box_isometries=False*)

Return the rows for the i-th piece.

INPUT:

- `i` – integer, the i-th piece
- `mod_box_isometries` – bool (default: `False`), whether to consider only rows for positions up to the action of the quotient the group of isometries of the n-cube by the subgroup of isometries of the $a_1 \times a_2 \cdots \times a_n$ rectangular box where are the a_i are assumed to be distinct.

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: T.rows_for_piece(0)
[[0, 3], [0, 4], [0, 5], [0, 6], [0, 7], [0, 8]]
sage: T.rows_for_piece(1)
[[1, 3, 4], [1, 4, 5], [1, 5, 6], [1, 6, 7], [1, 8, 7]]
sage: T.rows_for_piece(2)
[[2, 3, 4, 5], [2, 4, 5, 6], [2, 5, 6, 7], [2, 8, 6, 7]]
```

Less rows when using `mod_box_isometries=True`:

```
sage: a = Polyomino([(0,0,0), (0,0,1), (1,0,0)])
sage: b = Polyomino([(0,0,0), (1,0,0), (0,1,0)])
sage: T = TilingSolver([a,b], box=(2,1,3))
sage: T.rows_for_piece(0)
[[0, 5, 3, 6],
 [0, 7, 4, 6],
 [0, 2, 3, 6],
 [0, 7, 3, 4],
 [0, 5, 2, 3],
 [0, 3, 4, 6],
 [0, 5, 2, 6],
 [0, 7, 3, 6]]
sage: T.rows_for_piece(0, mod_box_isometries=True)
[[0, 2, 3, 6], [0, 7, 3, 4]]
sage: T.rows_for_piece(1, mod_box_isometries=True)
[[1, 2, 3, 6], [1, 7, 3, 4]]
```

solve (*partial=None*)

Returns an iterator of list of 3d polyominoes that are an exact cover of the box.

INPUT:

- `partial` - string (optional, default: `None`), whether to include partial (incomplete) solutions. It can be one of the following:
  - `None` - include only complete solution
  - `'common_prefix'` - common prefix between two consecutive solutions
  - `'incremental'` - one piece change at a time

OUTPUT:

iterator of list of 3d polyominoes

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: it = T.solve()
sage: for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2)], Color: gray
Polyomino: [(0, 0, 3), (0, 0, 4), (0, 0, 5)], Color: gray
sage: for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2), (0, 0, 3)], Color: gray
Polyomino: [(0, 0, 4), (0, 0, 5)], Color: gray
sage: for p in next(it): p
Polyomino: [(0, 0, 0), (0, 0, 1)], Color: gray
Polyomino: [(0, 0, 2), (0, 0, 3), (0, 0, 4)], Color: gray
Polyomino: [(0, 0, 5)], Color: gray
```

Including the partial solutions:

```
sage: it = T.solve(partial='common_prefix')
sage: for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2)], Color: gray
```

```
Polyomino: [(0, 0, 3), (0, 0, 4), (0, 0, 5)], Color: gray
sage: for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
sage: for p in next(it): p
Polyomino: [(0, 0, 0)], Color: gray
Polyomino: [(0, 0, 1), (0, 0, 2), (0, 0, 3)], Color: gray
Polyomino: [(0, 0, 4), (0, 0, 5)], Color: gray
sage: for p in next(it): p
sage: for p in next(it): p
Polyomino: [(0, 0, 0), (0, 0, 1)], Color: gray
Polyomino: [(0, 0, 2), (0, 0, 3), (0, 0, 4)], Color: gray
Polyomino: [(0, 0, 5)], Color: gray
```

Colors are preserved when the polyomino can be reused:

```
sage: p = Polyomino([(0,0,0)], color='yellow')
sage: q = Polyomino([(0,0,0), (0,0,1)], color='yellow')
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)], color='yellow')
sage: T = TilingSolver([p,q,r], box=(1,1,6), reusable=True)
sage: it = T.solve()
sage: for p in next(it): p
Polyomino: [(0, 0, 0)], Color: yellow
Polyomino: [(0, 0, 1)], Color: yellow
Polyomino: [(0, 0, 2)], Color: yellow
Polyomino: [(0, 0, 3)], Color: yellow
Polyomino: [(0, 0, 4)], Color: yellow
Polyomino: [(0, 0, 5)], Color: yellow
```

TESTS:

```
sage: T = TilingSolver([p,q,r], box=(1,1,7))
sage: next(T.solve())
Traceback (most recent call last):
...
StopIteration
```

space()

Returns an iterator over all the non negative integer coordinates contained in the box.

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: list(T.space())
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 0, 3), (0, 0, 4), (0, 0, 5)]
```

starting_rows()

Return the starting rows for each piece.

EXAMPLES:

```
sage: from sage.combinat.tiling import TilingSolver, Polyomino
sage: p = Polyomino([(0,0,0)])
sage: q = Polyomino([(0,0,0), (0,0,1)])
sage: r = Polyomino([(0,0,0), (0,0,1), (0,0,2)])
sage: T = TilingSolver([p,q,r], box=(1,1,6))
sage: T.starting_rows()
```

```
    [0, 6, 11, 15]
```

sage.combinat.tiling.**ncube_isometry_group**(*n*, *orientation_preserving=True*)

Return the isometry group of the n-cube as a list of matrices.

INPUT:

- n – positive integer, dimension of the space
- orientation_preserving – bool (optional, default: True), whether the orientation is preserved

OUTPUT:

list of matrices

EXAMPLES:

```
sage: from sage.combinat.tiling import ncube_isometry_group
sage: ncube_isometry_group(2)
[
[1 0]  [ 0  1]  [-1  0]  [ 0 -1]
[0 1], [-1  0], [ 0 -1], [ 1  0]
]
sage: ncube_isometry_group(2, orientation_preserving=False)
[
[1 0]  [ 0 -1]  [ 1  0]  [ 0  1]  [0 1]  [-1  0]  [ 0 -1]  [-1  0]
[0 1], [-1  0], [ 0 -1], [-1  0], [1 0], [ 0 -1], [ 1  0], [ 0  1]
]
```

There are 24 orientation preserving isometries of the 3-cube:

```
sage: ncube_isometry_group(3)
[
[1 0 0]  [ 1  0  0]  [-1  0  0]  [-1  0  0]  [0 0 1]  [ 0  0 -1]
[0 1 0]  [ 0 -1  0]  [ 0  1  0]  [ 0 -1  0]  [1 0 0]  [ 1  0  0]
[0 0 1], [ 0  0 -1], [ 0  0 -1], [ 0  0  1], [0 1 0], [ 0 -1  0],

[ 0  0 -1]  [ 0  0  1]  [0 1 0]  [ 0 -1  0]  [ 0  1  0]  [ 0 -1  0]
[-1  0  0]  [-1  0  0]  [0 0 1]  [ 0  0 -1]  [ 0  0 -1]  [ 0  0  1]
[ 0  1  0], [ 0 -1  0], [1 0 0], [ 1  0  0], [-1  0  0], [-1  0  0],

[ 0  1  0]  [ 0 -1  0]  [ 0  1  0]  [ 0 -1  0]  [ 1  0  0]  [ 1  0  0]
[ 1  0  0]  [ 1  0  0]  [-1  0  0]  [-1  0  0]  [ 0  0 -1]  [ 0  0  1]
[ 0  0 -1], [ 0  0  1], [ 0  0  1], [ 0  0 -1], [ 0  1  0], [ 0 -1  0],

[-1  0  0]  [-1  0  0]  [ 0  0 -1]  [ 0  0  1]  [ 0  0  1]  [ 0  0 -1]
[ 0  0  1]  [ 0  0 -1]  [ 0  1  0]  [ 0 -1  0]  [ 0  1  0]  [ 0 -1  0]
[ 0  1  0], [ 0 -1  0], [ 1  0  0], [ 1  0  0], [-1  0  0], [-1  0  0]
]
```

TESTS:

```
sage: ncube_isometry_group(1)
[[1]]
sage: ncube_isometry_group(0)
Traceback (most recent call last):
...
ValueError: ['B', 0] is not a valid Cartan type
```

Is deprecated:

```
sage: from sage.combinat.tiling import orthogonal_transformation
sage: L = orthogonal_transformation(2)
doctest:...: DeprecationWarning: orthogonal_transformation is
deprecated. Please use sage.combinat.tiling.ncube_isometry_group
instead. See http://trac.sagemath.org/19107 for details.
```

sage.combinat.tiling.**ncube_isometry_group_cosets**(*n*, *orientation_preserving=True*)

Return the quotient of the isometry group of the n-cube by the the isometry group of the rectangular parallelepiped.

INPUT:

- `n` – positive integer, dimension of the space
- `orientation_preserving` – bool (optional, default: `True`), whether the orientation is preserved

OUTPUT:

list of cosets, each coset being a sorted list of matrices

EXAMPLES:

```
sage: from sage.combinat.tiling import ncube_isometry_group_cosets
sage: sorted(ncube_isometry_group_cosets(2))
[[
[-1  0]  [1 0]
[ 0 -1], [0 1]
], [
[ 0 -1]  [ 0  1]
[ 1  0], [-1  0]
]]
sage: sorted(ncube_isometry_group_cosets(2, False))
[[
[-1  0]  [-1  0]  [ 1  0]  [1 0]
[ 0 -1], [ 0  1], [ 0 -1], [0 1]
], [
[ 0 -1]  [ 0 -1]  [ 0  1]  [0 1]
[-1  0], [ 1  0], [-1  0], [1 0]
]]

sage: sorted(ncube_isometry_group_cosets(3))
[[
[-1  0  0]  [-1  0  0]  [ 1  0  0]  [1 0 0]
[ 0 -1  0]  [ 0  1  0]  [ 0 -1  0]  [0 1 0]
[ 0  0  1], [ 0  0 -1], [ 0  0 -1], [0 0 1]
], [
[-1  0  0]  [-1  0  0]  [ 1  0  0]  [ 1  0  0]
[ 0  0 -1]  [ 0  0  1]  [ 0  0 -1]  [ 0  0  1]
[ 0 -1  0], [ 0  1  0], [ 0  1  0], [ 0 -1  0]
], [
[ 0 -1  0]  [ 0 -1  0]  [ 0  1  0]  [ 0  1  0]
[-1  0  0]  [ 1  0  0]  [-1  0  0]  [ 1  0  0]
[ 0  0 -1], [ 0  0  1], [ 0  0  1], [ 0  0 -1]
], [
[ 0 -1  0]  [ 0 -1  0]  [ 0  1  0]  [0 1 0]
[ 0  0 -1]  [ 0  0  1]  [ 0  0 -1]  [0 0 1]
[ 1  0  0], [-1  0  0], [-1  0  0], [1 0 0]
], [
[ 0  0 -1]  [ 0  0 -1]  [ 0  0  1]  [0 0 1]
[-1  0  0]  [ 1  0  0]  [-1  0  0]  [1 0 0]
```

```
[ 0  1  0], [ 0 -1  0], [ 0 -1  0], [0 1 0]
], [
[ 0  0 -1]  [ 0  0 -1]  [ 0  0  1]  [ 0  0  1]
[ 0 -1  0]  [ 0  1  0]  [ 0 -1  0]  [ 0  1  0]
[-1  0  0], [ 1  0  0], [ 1  0  0], [-1  0  0]
]]
```

TESTS:

```
sage: cosets = ncube_isometry_group_cosets(3, False)
sage: len(cosets)
6
sage: map(len, cosets)
[8, 8, 8, 8, 8, 8]
sage: type(cosets[0][0])
<type 'sage.matrix.matrix_rational_dense.Matrix_rational_dense'>
```

5.1.314 Tools

sage.combinat.tools.**transitive_ideal**(*f*, *x*)

Return a list of all elements reachable from x in the abstract reduction system whose reduction relation is given by the function f.

In more elementary terms: If S is a set, and f is a function sending every element of S to a list of elements of S, then we can define a digraph on the vertex set S by drawing an edge from s to t for every $s \in S$ and every $t \in f(s)$. If $x \in S$, then an element $y \in S$ is said to be reachable from x if there is a path $x \to y$ in this graph. Given f and x, this method computes the list of all elements of S reachable from x.

Note that if there are infinitely many such elements, then this method will never halt.

EXAMPLES:

```
sage: f = lambda x: [x-1] if x > 0 else []
sage: sage.combinat.tools.transitive_ideal(f, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

5.1.315 Tuples

sage.combinat.tuple.**Tuples**(*S*, *k*)

Returns the combinatorial class of ordered tuples of S of length k.

An ordered tuple of length k of set is an ordered selection with repetition and is represented by a list of length k containing elements of set.

EXAMPLES:

```
sage: S = [1,2]
sage: Tuples(S,3).list()
[[1, 1, 1], [2, 1, 1], [1, 2, 1], [2, 2, 1], [1, 1, 2], [2, 1, 2], [1, 2, 2], [2, 2, 2]]
sage: mset = ["s","t","e","i","n"]
sage: Tuples(mset,2).list()
[['s', 's'], ['t', 's'], ['e', 's'], ['i', 's'], ['n', 's'], ['s', 't'], ['t', 't'],
 ['e', 't'], ['i', 't'], ['n', 't'], ['s', 'e'], ['t', 'e'], ['e', 'e'], ['i', 'e'],
 ['n', 'e'], ['s', 'i'], ['t', 'i'], ['e', 'i'], ['i', 'i'], ['n', 'i'], ['s', 'n'],
 ['t', 'n'], ['e', 'n'], ['i', 'n'], ['n', 'n']]
```

```
sage: K.<a> = GF(4, 'a')
sage: mset = [x for x in K if x!=0]
sage: Tuples(mset,2).list()
[[a, a], [a + 1, a], [1, a], [a, a + 1], [a + 1, a + 1], [1, a + 1], [a, 1], [a + 1, 1], [1, 1]]
```

class sage.combinat.tuple.**Tuples_sk**(S, k)

Bases: sage.combinat.combinat.CombinatorialClass

TESTS:

```
sage: T = Tuples([1,2,3],2)
sage: T == loads(dumps(T))
True
```

cardinality()

EXAMPLES:

```
sage: S = [1,2,3,4,5]
sage: Tuples(S,2).cardinality()
25
sage: S = [1,1,2,3,4,5]
sage: Tuples(S,2).cardinality()
25
```

sage.combinat.tuple.**UnorderedTuples**(S, k)

Returns the combinatorial class of unordered tuples of S of length k.

An unordered tuple of length k of set is a unordered selection with repetitions of set and is represented by a sorted list of length k containing elements from set.

EXAMPLES:

```
sage: S = [1,2]
sage: UnorderedTuples(S,3).list()
[[1, 1, 1], [1, 1, 2], [1, 2, 2], [2, 2, 2]]
sage: UnorderedTuples(["a","b","c"],2).list()
[['a', 'a'], ['a', 'b'], ['a', 'c'], ['b', 'b'], ['b', 'c'], ['c', 'c']]
```

class sage.combinat.tuple.**UnorderedTuples_sk**(S, k)

Bases: sage.combinat.combinat.CombinatorialClass

TESTS:

```
sage: T = Tuples([1,2,3],2)
sage: T == loads(dumps(T))
True
```

cardinality()

EXAMPLES:

```
sage: S = [1,2,3,4,5]
sage: UnorderedTuples(S,2).cardinality()
15
```

list()

EXAMPLES:

```
sage: S = [1,2]
sage: UnorderedTuples(S,3).list()
[[1, 1, 1], [1, 1, 2], [1, 2, 2], [2, 2, 2]]
sage: UnorderedTuples(["a","b","c"],2).list()
[['a', 'a'], ['a', 'b'], ['a', 'c'], ['b', 'b'], ['b', 'c'], ['c', 'c']]
```

5.1.316 Introduction to combinatorics in Sage

This thematic tutorial is a translation by Hugh Thomas of the combinatorics chapter written by Nicolas M. Thiéry in the book "Calcul Mathématique avec Sage" [CMS2012]. It covers mainly the treatment in Sage of the following combinatorial problems: enumeration (how many elements are there in a set S?), listing (generate all the elements of S, or iterate through them), and random selection (choosing an element at random from a set S according to a given distribution, for example the uniform distribution). These questions arise naturally in the calculation of probabilities (what is the probability in poker of obtaining a straight or a four-of-a-kind of aces?), in statistical physics, and also in computer algebra (the number of elements in a finite field), or in the analysis of algorithms. Combinatorics covers a much wider domain (partial orders, representation theory, ...) for which we only give a few pointers towards the possibilities offered by Sage.

Todo

Add link to some thematic tutorial on graphs

A characteristic of computational combinatorics is the profusion of types of objects and sets that one wants to manipulate. It would be impossible to describe them all or, a fortiori, to implement them all. After some *Initial examples*, this chapter illustrates the underlying method: supplying the basic building blocks to describe common combinatorial sets *Common enumerated sets*, tools for combining them to construct new examples *Constructions*, and generic algorithms for solving uniformly a large class of problems *Generic algorithms*.

This is a domain in which Sage has much more extensive capabilities than most computer algebra systems, and it is rapidly expanding; at the same time, it is still quite new, and has many unnecessary limitations and incoherences.

Initial examples

Poker and probability

We begin by solving a classic problem: enumerating certain combinations of cards in the game of poker, in order to deduce their probability.

A card in a poker deck is characterized by a suit (hearts, diamonds, spades, or clubs) and a value (2, 3, ..., 10, jack, queen, king, ace). The game is played with a full deck, which consists of the Cartesian product of the set of suits and the set of values:

$$\text{Cards} = \text{Suits} \times \text{Values} = \{(s, v) \mid s \in \text{Suits} \text{ et } v \in \text{Values}\}.$$

We construct these examples in Sage:

```
sage: Suits = Set(["Hearts", "Diamonds", "Spades", "Clubs"])
sage: Values = Set([2, 3, 4, 5, 6, 7, 8, 9, 10,
....:               "Jack", "Queen", "King", "Ace"])
sage: Cards = cartesian_product([Values, Suits])
```

There are 4 suits and 13 possible values, and therefore $4 \times 13 = 52$ cards in the poker deck:

```
sage: Suits.cardinality()
4
sage: Values.cardinality()
13
sage: Cards.cardinality()
52
```

Draw a card at random:

```
sage: Cards.random_element()                          # random
(6, 'Clubs')
```

Now we can define a set of cards:

```
sage: Set([Cards.random_element(), Cards.random_element()])   # random
{(2, 'Hearts'), (4, 'Spades')}
```

This problem should eventually disappear: it is planned to change the implementation of Cartesian products so that their elements are immutable by default.

Returning to our main topic, we will be considering a simplified version of poker, in which each player directly draws five cards, which form his *hand*. The cards are all distinct and the order in which they are drawn is irrelevant; a hand is therefore a subset of size 5 of the set of cards. To draw a hand at random, we first construct the set of all possible hands, and then we ask for a randomly chosen element:

```
sage: Hands = Subsets(Cards, 5)
sage: Hands.random_element()                          # random
{(4, 'Hearts', 4), (9, 'Diamonds'), (8, 'Spades'),
 (9, 'Clubs'), (7, 'Hearts')}
```

The total number of hands is given by the number of subsets of size 5 of a set of size 52, which is given by the binomial coefficient $\binom{52}{5}$:

```
sage: binomial(52,5)
2598960
```

One can also ignore the method of calculation, and simply ask for the size of the set of hands:

```
sage: Hands.cardinality()
2598960
```

The strength of a poker hand depends on the particular combination of cards present. One such combination is the *flush*; this is a hand all of whose cards have the same suit. (In principle, straight flushes should be excluded; this will be the goal of an exercise given below.) Such a hand is therefore characterized by the choice of five values from among the thirteen possibilities, and the choice of one of four suits. We will construct the set of all flushes, so as to determine how many there are:

```
sage: Flushes = cartesian_product([Subsets(Values, 5), Suits])
sage: Flushes.cardinality()
5148
```

The probability of obtaining a flush when drawing a hand at random is therefore:

```
sage: Flushes.cardinality()  / Hands.cardinality()
33/16660
```

or about two in a thousand:

```
sage: 1000.0 * Flushes.cardinality()  / Hands.cardinality()
1.98079231692677
```

We will now attempt a little numerical simulation. The following function tests whether a given hand is a flush or not:

```
sage: def is_flush(hand):
....:     return len(set(suit for (val, suit) in hand)) == 1
```

We now draw 10000 hands at random, and count the number of flushes obtained (this takes about 10 seconds):

```
sage: n = 10000
sage: nflush = 0
sage: for i in range(n):                                # long time
....:     hand = Hands.random_element()
....:     if is_flush(hand):
....:         nflush += 1
sage: print n, nflush                                   # random
10000 18
```

Exercises

A hand containing four cards of the same value is called a *four of a kind*. Construct the set of four of a kind hands (Hint: use `Arrangements` to choose a pair of distinct values at random, then choose a suit for the first value). Calculate the number of four of a kind hand, list them, and then determine the probability of obtaining a four of a kind when drawing a hand at random.
A hand all of whose cards have the same suit, and whose values are consecutive, is called a *straight flush* rather than a *flush*. Count the number of straight flushes, and then deduce the correct probability of obtaining a flush when drawing a hand at random.
Calculate the probability of each of the poker hands (see Wikipedia article Poker_hands), and compare them with the results of simulations.

Enumeration of trees using generating functions

In this section, we discuss the example of complete binary trees, and illustrate in this context many techniques of enumeration in which formal power series play a natural role. These techniques are quite general, and can be applied whenever the combinatorial objects in question admit a recursive definition (grammar) (see *Species, decomposable combinatorial classes* for an automated treatment). The goal is not a formal presentation of these methods; the calculations are rigorous, but most of the justifications will be skipped.

A *complete binary tree* is either a leaf L, or a node to which two complete binary trees are attached (see *Figure: The five complete binary trees with four leaves*).

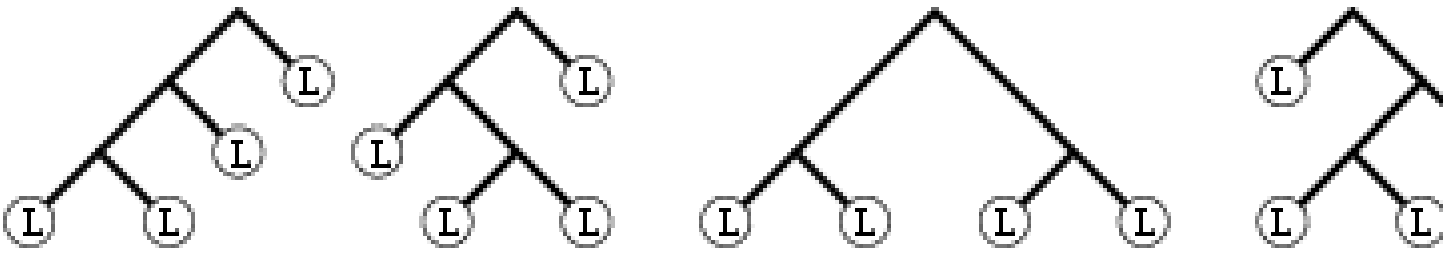


Figure 5.1: Figure: The five complete binary trees with four leaves

Exercise: enumeration of binary trees

Find by hand all the complete binary trees with $n = 1, 2, 3, 4, 5$ leaves (see *Exercise: complete binary tree iterator* to find them using `Sage`).

Our goal is to determine the number c_n of complete binary trees with n leaves (in this section, except when explicitly stated otherwise, "trees" always means complete binary trees). This is a typical situation in which one is not only

interested in a single set, but in a family of sets, typically parameterized by $n \in \mathbf{N}$.

According to the solution of *Exercise: enumeration of binary trees*, the first terms are given by $c_1, \ldots, c_5 = 1, 1, 2, 5, 14$. The simple fact of knowing these few numbers is already very valuable. In fact, this permits research in a gold mine of information: the Online Encyclopedia of Integer Sequences, commonly called "Sloane", the name of its principal author, which contains more than 190000 sequences of integers:

```
sage: oeis([1,1,2,5,14])                                  # optional -- internet
0: A000108: Catalan numbers: C(n) = binomial(2n,n)/(n+1) = (2n)!/(n!(n+1)!). Also called Segner numbe
1: A120588: G.f. satisfies: 3*A(x) = 2 + x + A(x)^2, with a(0) = 1.
2: A080937: Number of Catalan paths (nonnegative, starting and ending at 0, step +/-1) of 2*n steps w
```

The result suggests that the trees are counted by one of the most famous sequences, the Catalan numbers. Looking through the references supplied by the Encyclopedia, we see that this is really the case: the few numbers above form a digital fingerprint of our objects, which enable us to find, in a few seconds, a precise result from within an abundant literature.

Our next goal is to recover this result using Sage. Let C_n be the set of trees with n leaves, so that $c_n = |C_n|$; by convention, we will define $C_0 = \emptyset$ and $c_0 = 0$. The set of all trees is then the disjoint union of the sets C_n:

$$C = \biguplus_{n \in \mathbb{N}} C_n \,.$$

Having named the set C of all trees, we can translate the recursive definition of trees into a set-theoretic equation:

$$C \quad \approx \quad \{\mathrm{L}\} \quad \uplus \quad C \times C \,.$$

In words: a tree t (which is by definition in C) is either a leaf (so in $\{\mathrm{L}\}$) or a node to which two trees t_1 and t_2 have been attached, and which we can therefore identify with the pair (t_1, t_2) (in the Cartesian product $C \times C$).

The founding idea of algebraic combinatorics, introduced by Euler in a letter to Goldbach of 1751 to treat a similar problem , is to manipuate all the numbers c_n simultaneously, by encoding them as coefficients in a formal power series, called the *generating function* of the c_n's:

$$C(z) = \sum_{n \in \mathbb{N}} c_n z^n \,,$$

where z is a formal variable (which means that we do not have to worry about questions of convergence). The beauty of this idea is that set-theoretic operations ($A \uplus B$, $A \times B$) translate naturally into algebraic operations on the corresponding series ($A(z) + B(z)$, $A(z) \cdot B(z)$), in such a way that the set-theoretic equation satisfied by C can be translated directly into an algebraic equation satisfied by $C(z)$:

$$C(z) = z + C(z) \cdot C(z) \,.$$

Now we can solve this equation with Sage. In order to do so, we introduce two variables, C and z, and we define the equation:

```
sage: C, z = var('C,z');
sage: sys = [ C == z + C*C ]
```

There are two solutions, which happen to have closed forms:

```
sage: sol = solve(sys, C, solution_dict=True); sol
[{C: -1/2*sqrt(-4*z + 1) + 1/2}, {C: 1/2*sqrt(-4*z + 1) + 1/2}]
sage: s0 = sol[0][C]; s1 = sol[1][C]
```

and whose Taylor series begin as follows:

```
sage: s0.series(z, 6)
1*z + 1*z^2 + 2*z^3 + 5*z^4 + 14*z^5 + Order(z^6)
sage: s1.series(z, 6)
1 + (-1)*z + (-1)*z^2 + (-2)*z^3 + (-5)*z^4 + (-14)*z^5
+ Order(z^6)
```

The second solution is clearly aberrant, while the first one gives the expected coefficients. Therefore, we set:

```
sage: C = s0
```

We can now calculate the next terms:

```
sage: C.series(z, 11)
1*z + 1*z^2 + 2*z^3 + 5*z^4 + 14*z^5 + 42*z^6 +
132*z^7 + 429*z^8 + 1430*z^9 + 4862*z^10 + Order(z^11)
```

or calculate, more or less instantaneously, the 100-th coefficient:

```
sage: C.series(z, 101).coefficient(z,100)
227508830794229349661819540395688853956041682601541047340
```

It is unfortunate to have to recalculate everything if at some point we wanted the 101-st coefficient. Lazy power series (see `sage.combinat.species.series`) come into their own here, in that one can define them from a system of equations without solving it, and, in particular, without needing a closed form for the answer. We begin by defining the ring of lazy power series:

```
sage: L.<z> = LazyPowerSeriesRing(QQ)
```

Then we create a "free" power series, which we name, and which we then define by a recursive equation:

```
sage: C = L()
sage: C._name = 'C'
sage: C.define( z + C * C );
```

```
sage: [C.coefficient(i) for i in range(11)]
[0, 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

At any point, one can ask for any coefficient without having to redefine C:

```
sage: C.coefficient(100)
227508830794229349661819540395688853956041682601541047340
sage: C.coefficient(200)
129013158064429114001222907669676675134349530552728882499810851598901419013348319045534580850847735
```

We now return to the closed form of $C(z)$:

```
sage: z = var('z');
sage: C = s0; C
-1/2*sqrt(-4*z + 1) + 1/2
```

The n-th coefficient in the Taylor series for $C(z)$ being given by $\frac{1}{n!}C(z)^{(n)}(0)$, we look at the successive derivatives $C(z)^{(n)}(z)$:

```
sage: derivative(C, z, 1)
1/sqrt(-4*z + 1)
sage: derivative(C, z, 2)
2/(-4*z + 1)^(3/2)
```

```
sage: derivative(C, z, 3)
12/(-4*z + 1)^(5/2)
```

This suggests the existence of a simple explicit formula, which we will now seek. The following small function returns $d_n = n!\,c_n$:

```
sage: def d(n): return derivative(s0, n).subs(z=0)
```

Taking successive quotients:

```
sage: [ (d(n+1) / d(n)) for n in range(1,17) ]
[2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62]
```

we observe that d_n satisfies the recurrence relation $d_{n+1} = (4n-2)d_n$, from which we deduce that c_n satisfies the recurrence relation $c_{n+1} = \frac{(4n-2)}{n+1}c_n$. Simplifying, we find that c_n is the $(n-1)$-th Catalan number:

$$c_n = \mathrm{Catalan}(n-1) = \frac{1}{n}\binom{2(n-1)}{n-1}.$$

We check this:

```
sage: n = var('n');
sage: c = 1/n*binomial(2*(n-1),n-1)
sage: [c.subs(n=k) for k in range(1, 11)]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
sage: [catalan_number(k-1) for k in range(1, 11)]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

We can now calculate coefficients much further; here we calculate c_{100000} which has more than 60000 digits:

```
sage: cc = c(n = 100000)
```

This takes a couple of seconds:

```
sage: %time cc = c(100000)                          # not tested
CPU times: user 2.34 s, sys: 0.00 s, total: 2.34 s
Wall time: 2.34 s
sage: ZZ(cc).ndigits()
60198
```

The methods which we have used generalize to all recursively defined objects: the system of set-theoretic equations can be translated into a system of equations on the generating function, which enables the recursive calculation of its coefficients. If the set-theoretic equations are simple enough (for example, if they only involve Cartesian products and disjoint unions), the equation for $C(z)$ is algebraic. This equation has, in general, no closed-form solution. However, using *confinement*, one can deduce a *linear* differential equation which $C(z)$ satisfies. This differential equation, in turn, can be translated into a recurrence relation of fixed length on its coefficients c_n. In this case, the series is called *D-finite*. After the initial calculation of this recurrence relation, the calculation of coefficients is very fast. All these steps are purely algorithmic, and it is planned to port into `Sage` the implementations which exist in `Maple` (the `gfun` and `combstruct` packages) or `MuPAD-Combinat` (the `decomposableObjects` library).

For the moment, we illustrate this general procedure in the case of complete binary trees. The generating function $C(z)$ is a solution to an algebraic equation $P(z, C(z)) = 0$, where $P = P(x, y)$ is a polynomial with coefficients in $\mathbf{Q}$. In the present case, $P = y^2 - y + x$. We formally differentiate this equation with respect to z:

```
sage: x, y, z = var('x, y, z')
sage: P = function('P')(x, y)
sage: C = function('C')(z)
```

```
sage: equation =  P(x=z, y=C) == 0
sage: diff(equation, z)
D[0](C)(z)*D[1](P)(z, C(z)) + D[0](P)(z, C(z)) == 0
```

or, in a more readable format,

$$\frac{dC(z)}{dz}\frac{\partial P}{\partial y}(z, C(z)) + \frac{\partial P}{\partial x}(z, C(z)) = 0$$

From this we deduce:

$$\frac{dC(z)}{dz} = -\frac{\frac{\partial P}{\partial x}}{\frac{\partial P}{\partial y}}(z, C(z))\,.$$

In the case of complete binary trees, this gives:

```
sage: P = y^2 - y + x
sage: Px = diff(P, x); Py = diff(P, y)
sage: - Px / Py
-1/(2*y - 1)
```

Recall that $P(z, C(z)) = 0$. Thus, we can calculate this fraction mod P and, in this way, express the derivative of $C(z)$ as a *polynomial in* $C(z)$ *with coefficients in* $\mathbf{Q}(z)$. In order to achieve this, we construct the quotient ring $R = \mathbf{Q}(x)[y]/(P)$:

```
sage: Qx = QQ['x'].fraction_field()
sage: Qxy = Qx['y']
sage: R = Qxy.quo(P); R
Univariate Quotient Polynomial Ring in ybar
over Fraction Field of Univariate Polynomial Ring in x
over Rational Field with modulus y^2 - y + x
```

Note: `ybar` is the name of the variable y in the quotient ring.

Todo

add link to some tutorial on quotient rings

We continue the calculation of this fraction in R:

```
sage: fraction = - R(Px) / R(Py); fraction
(1/2/(x - 1/4))*ybar - 1/4/(x - 1/4)
```

Note: The following variant does not work yet:

```
sage: fraction = R( - Px / Py ); fraction   # todo: not implemented
Traceback (most recent call last):
...
TypeError: denominator must be a unit
```

We lift the result to $\mathbf{Q}(x)[y]$ and then substitute z and $C(z)$ to obtain an expression for $\frac{d}{dz}C(z)$:

```
sage: fraction = fraction.lift(); fraction
(1/2/(x - 1/4))*y - 1/4/(x - 1/4)
sage: fraction(x=z, y=C)
2*C(z)/(4*z - 1) - 1/(4*z - 1)
```

or, more legibly,

$$\frac{\partial C(z)}{\partial z} = \frac{1}{1-4z} - \frac{2}{1-4z} C(z)\,.$$

In this simple case, we can directly deduce from this expression a linear differential equation with coefficients in $\mathbf{Q}[z]$:

```
sage: equadiff = diff(C,z) == fraction(x=z, y=C)
sage: equadiff
D[0](C)(z) == 2*C(z)/(4*z - 1) - 1/(4*z - 1)
sage: equadiff = equadiff.simplify_rational()
sage: equadiff = equadiff * equadiff.rhs().denominator()
sage: equadiff = equadiff - equadiff.rhs()
sage: equadiff
(4*z - 1)*D[0](C)(z) - 2*C(z) + 1 == 0
```

or, more legibly,

$$(1-4z)\frac{\partial C(z)}{\partial z} + 2C(z) - 1 = 0\,.$$

It is trivial to verify this equation on the closed form:

```
sage: Cf = sage.symbolic.function_factory.function('C')
sage: equadiff.substitute_function(Cf, s0)
doctest:...: DeprecationWarning:...
(4*z - 1)/sqrt(-4*z + 1) + sqrt(-4*z + 1) == 0
sage: bool(equadiff.substitute_function(Cf, s0))
True
```

In the general case, one continues to calculate successive derivatives of $C(z)$. These derivatives are *confined* in the quotient ring $\mathbf{Q}(z)[C]/(P)$ which is of finite dimension $\deg P$ over $\mathbf{Q}(z)$. Therefore, one will eventually find a linear relation among the first $\deg P$ derivatives of $C(z)$. Putting it over a single denominator, we obtain a linear differential equation of degree $\leq \deg P$ with coefficients in $\mathbf{Q}[z]$. By extracting the coefficient of z^n in the differential equation, we obtain the desired recurrence relation on the coefficients; in this case we recover the relation we had already found, based on the closed form:

$$c_{n+1} = \frac{(4n-2)}{n+1} c_n$$

After fixing the correct initial conditions, it becomes possible to calculate the coefficents of $C(z)$ recursively:

```
sage: def C(n): return 1 if n <= 1 else (4*n-6)/n * C(n-1)
sage: [ C(i) for i in range(10) ]
[1, 1, 1, 2, 5, 14, 42, 132, 429, 1430]
```

If n is too large for the explicit calculation of c_n, a sequence asymptotically equivalent to the sequence of coefficients c_n may be sought. Here again, there are generic techniques. The central tool is complex analysis, specifically, the study of the generating function around its singularities. In the present instance, the singularity is at $z_0 = 1/4$ and one would obtain $c_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$.

Summary We see here a general phenomenon of computer algebra: the best *data structure* to describe a complicated mathematical object (a real number, a sequence, a formal power series, a function, a set) is often an equation defining the object (or a system of equations, typically with some initial conditions). Attempting to find a closed-form solution to this equation is not necessarily of interest: on the one hand, such a closed form rarely exists (e.g., the problem of solving a polynomial by radicals), and on the other hand, the equation, in itself, contains all the necessary information to calculate algorithmically the properties of the object under consideration (e.g., a numerical approximation, the initial terms or elements, an asymptotic equivalent), or to calculate with the object itself (e.g., performing arithmetic

on power series). Therefore, instead of solving the equation, we look for the equation describing the object which is best suited to the problem we want to solve.

As we saw in our example, confinement (for example, in a finite dimensional vector space) is a fundamental tool for studying such equations. This notion of confinement is widely applicable in elimination techniques (linear algebra, Gröbner bases, and their algebro-differential generalizations). The same tool is central in algorithms for automatic summation and automatic verification of identities (Gosper's algorithm, Zeilberger's algorithm, and their generalizations; see also *Exercise: alternating sign matrices*).

All these techniques and their many generalizations are at the heart of very active topics of research: automatic combinatorics and analytic combinatorics, with major applications in the analysis of algorithms. It is likely, and desirable, that they will be progressively implemented in `Sage`.

Common enumerated sets

First example: the subsets of a set

Fix a set E of size n and consider the subsets of E of size k. We know that these subsets are counted by the binomial coefficients $\binom{n}{k}$. We can therefore calculate the number of subsets of size $k = 2$ of $E = \{1, 2, 3, 4\}$ with the function `binomial`:

```
sage: binomial(4, 2)
6
```

Alternatively, we can *construct* the set $\mathcal{P}_2(E)$ of all the subsets of size 2 of E, then ask its cardinality:

```
sage: S = Subsets([1,2,3,4], 2)
sage: S.cardinality()
6
```

Once `S` has been constructed, we can also obtain the list of its elements:

```
sage: S.list()
[{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}]
```

or select an element at random:

```
sage: S.random_element()                # random
{1, 4}
```

More precisely, the object `S` models the set $\mathcal{P}_2(E)$ equipped with a fixed order (here, lexicographic order). It is therefore possible to ask for its 5-th element, keeping in mind that, as with `Python` lists, the first element is numbered zero:

```
sage: S.unrank(4)
{2, 4}
```

As a shortcut, in this setting, one can also use the notation:

```
sage: S[4]
{2, 4}
```

but this should be used with care because some sets have a natural indexing other than by $(0, 1, \dots)$.

Conversely, one can calculate the position of an object in this order:

```
sage: s = S([2,4]); s
{2, 4}
sage: S.rank(s)
4
```

Note that `S` is *not* the list of its elements. One can, for example, model the set $\mathcal{P}(\mathcal{P}(\mathcal{P}(E)))$ and calculate its cardinality ($2^{2^{2^4}}$):

```
sage: E = Set([1,2,3,4])
sage: S = Subsets(Subsets(Subsets(E))); S
Subsets of Subsets of Subsets of {1, 2, 3, 4}
sage: n = S.cardinality(); n
2003529930406846464979072351560255750447825475569751419265016973...
```

which is roughly $2 \cdot 10^{19728}$:

```
sage: n.ndigits()
19729
```

or ask for its 237102124-th element:

```
sage: S.unrank(237102123)
{{{2, 4}, {1, 4}, {}, {1, 3, 4}, {1, 2, 4}, {4}, {2, 3}, {1, 3}, {2}},
  {{1, 3}, {2, 4}, {1, 2, 4}, {}, {3, 4}}}
```

It would be physically impossible to construct explicitly all the elements of S, as there are many more of them than there are particles in the universe (estimated at 10^{82}).

Remark: it would be natural in `Python` to use `len(S)` to ask for the cardinality of `S`. This is not possible because `Python` requires that the result of `len` be an integer of type `int`; this could cause overflows, and would not permit the return of {Infinity} for infinite sets:

```
sage: len(S)
Traceback (most recent call last):
...
OverflowError: Python int too large to convert to C long
```

Partitions of integers

We now consider another classic problem: given a positive integer n, in how many ways can it be written in the form of a sum $n = i_1 + i_2 + \cdots + i_\ell$, where $i_1, \ldots, i_\ell$ are positive integers? There are two cases to distinguish:

- the order of the elements in the sum is not important, in which case we call $(i_1, \ldots, i_\ell)$ a *partition* of n;
- the order of the elements in the sum is important, in which case we call $(i_1, \ldots, i_\ell)$ a *composition* of n.

We will begin with the partitions of $n = 5$; as before, we begin by constructing the set of these partitions:

```
sage: P5 = Partitions(5); P5
Partitions of the integer 5
```

then we ask for its cardinality:

```
sage: P5.cardinality()
7
```

We look at these 7 partitions; the order being irrelevant, the entries are ordered, by convention, in decreasing order.

```
sage: P5.list()
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1],
 [1, 1, 1, 1, 1]]
```

The calculation of the number of partitions uses the Rademacher formula (Wikipedia article Partition_(number_theory)), implemented in C and highly optimized, which makes it very fast:

```
sage: Partitions(100000).cardinality()
2749351056977569651267751632098635268817342931598005475820312598430214732811496417305505074166073662
```

Partitions of integers are combinatorial objects naturally equipped with many operations. They are therefore returned as objects that are richer than simple lists:

```
sage: P7 = Partitions(7)
sage: p = P7.unrank(5); p
[4, 2, 1]
sage: type(p)
<class 'sage.combinat.partition.Partitions_n_with_category.element_class'>
```

For example, they can be represented graphically by a Ferrers diagram:

```
sage: print p.ferrers_diagram()
****
**
*
```

We leave it to the user to explore by introspection the available operations.

Note that we can also construct a partition directly by:

```
sage: Partition([4,2,1])
[4, 2, 1]
```

or:

```
sage: P7([4,2,1])
[4, 2, 1]
```

If one wants to restrict the possible values of the parts $i_1, \ldots, i_\ell$ of the partition as, for example, when giving change, one can use `WeightedIntegerVectors`. For example, the following calculation:

```
sage: WeightedIntegerVectors(8, [2,3,5]).list()
[[0, 1, 1], [1, 2, 0], [4, 0, 0]]
```

shows that to make 8 dollars using 2, 3, and 5 dollar bills, one can use a 3 and a 5 dollar bill, or a 2 and two 3 dollar bills, or four 2 dollar bills.

Compositions of integers are manipulated the same way:

```
sage: C5 = Compositions(5); C5
Compositions of 5
sage: C5.cardinality()
16
sage: C5.list()
[[1, 1, 1, 1, 1], [1, 1, 1, 2], [1, 1, 2, 1], [1, 1, 3],
 [1, 2, 1, 1], [1, 2, 2], [1, 3, 1], [1, 4], [2, 1, 1, 1],
 [2, 1, 2], [2, 2, 1], [2, 3], [3, 1, 1], [3, 2], [4, 1], [5]]
```

The number 16 above seems significant and suggests the existence of a formula. We look at the number of compositions of n ranging from 0 to 9:

```
sage: [ Compositions(n).cardinality() for n in range(10) ]
[1, 1, 2, 4, 8, 16, 32, 64, 128, 256]
```

Similarly, if we consider the number of compositions of 5 by length, we find a line of Pascal's triangle:

```
sage: x = var('x')
sage: sum( x^len(c) for c in C5 )
x^5 + 4*x^4 + 6*x^3 + 4*x^2 + x
```

The above example uses a functionality which we have not seen yet: `C5` being iterable, it can be used like a list in a `for` loop or a comprehension (*Set comprehension and iterators*).

> Prove the formulas suggested by the above examples for the number of compositions of n and the number of compositions of n of length k; investigate by introspection whether `Sage` uses these formulas for calculating cardinalities.

Some other finite enumerated sets

Essentially, the principle is the same for all the finite sets with which one wants to do combinatorics in `Sage`; begin by constructing an object which models this set, and then supply appropriate methods, following a uniform interface [5]. We now give a few more typical examples.

Intervals of integers:

```
sage: C = IntegerRange(3, 21, 2); C
{3, 5, ..., 19}
sage: C.cardinality()
9
sage: C.list()
[3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Permutations:

```
sage: C = Permutations(4); C
Standard permutations of 4
sage: C.cardinality()
24
sage: C.list()
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2],
 [1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4], [2, 1, 4, 3],
 [2, 3, 1, 4], [2, 3, 4, 1], [2, 4, 1, 3], [2, 4, 3, 1],
 [3, 1, 2, 4], [3, 1, 4, 2], [3, 2, 1, 4], [3, 2, 4, 1],
 [3, 4, 1, 2], [3, 4, 2, 1], [4, 1, 2, 3], [4, 1, 3, 2],
 [4, 2, 1, 3], [4, 2, 3, 1], [4, 3, 1, 2], [4, 3, 2, 1]]
```

Set partitions:

```
sage: C = SetPartitions([1,2,3])
sage: C
Set partitions of {1, 2, 3}
sage: C.cardinality()
5
sage: C.list()
```

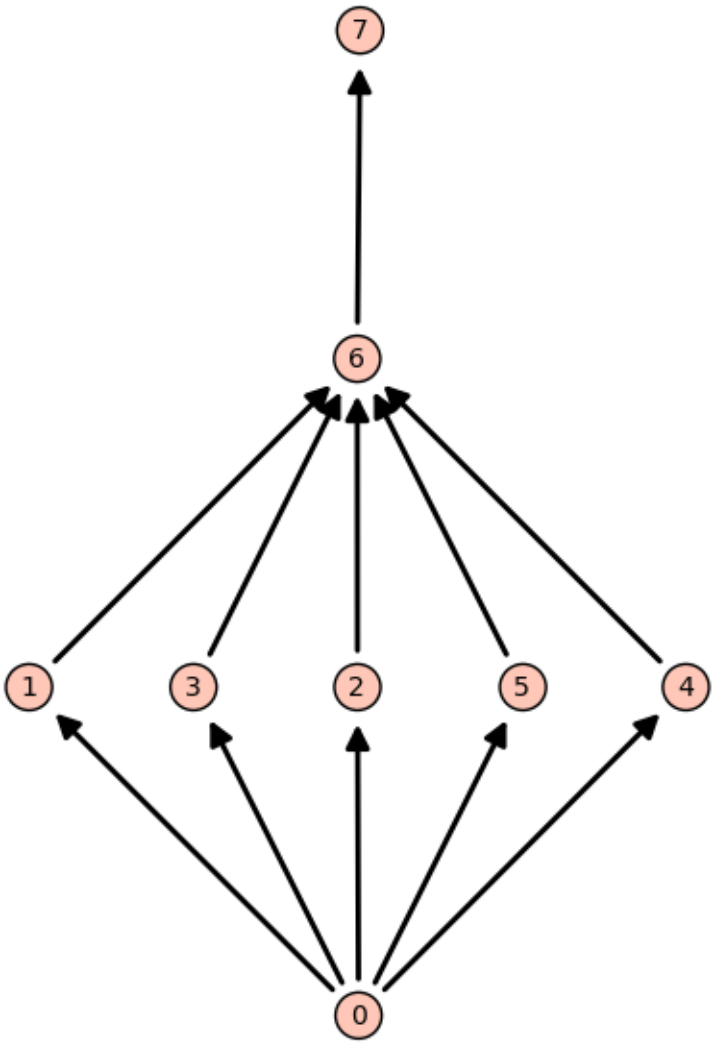
[5] Or at least that should be the case; there are still many corners to clean up.

```
[{{1, 2, 3}}, {{1}, {2, 3}}, {{1, 3}, {2}}, {{1, 2}, {3}},
 {{1}, {2}, {3}}]
```

Partial orders on a set of 8 elements, up to isomorphism:

```
sage: C = Posets(8); C
Posets containing 8 vertices
sage: C.cardinality()
16999
```

```
sage: C.unrank(20).plot()
Graphics object consisting of 20 graphics primitives
```

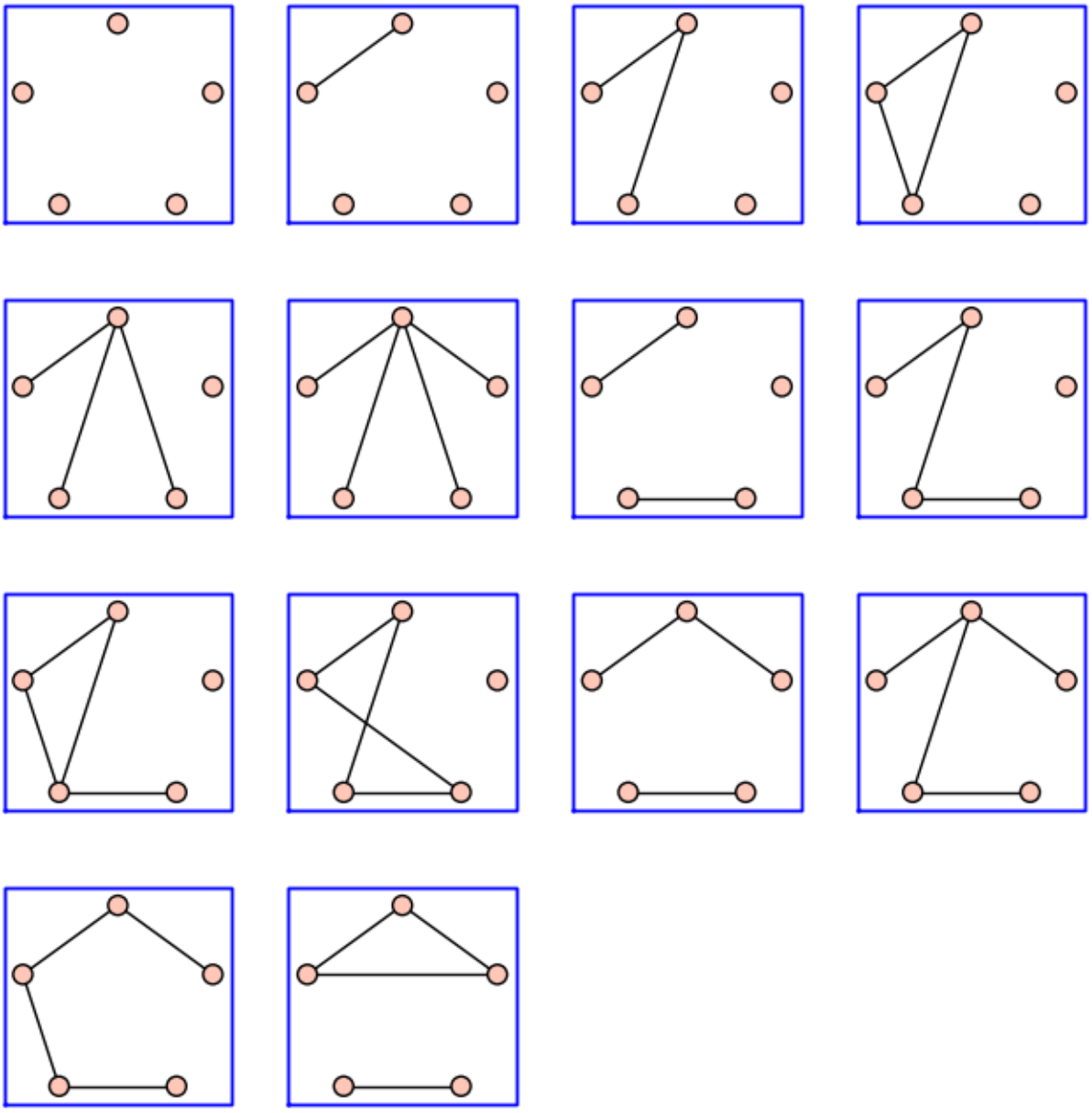


One can iterate through all graphs up to isomorphism. For example, there are 34 simple graphs with 5 vertices:

```
sage: len(list(graphs(5)))
34
```

Here are those with at most 4 edges:

```
sage: show(graphs(5, lambda G: G.size() <= 4))
```



However, the *set* `C` of these graphs is not yet available in `Sage`; as a result, the following commands are not yet implemented:

```
sage: C = Graphs(5)                  # todo: not implemented
sage: C.cardinality()                # todo: not implemented
34
sage: Graphs(19).cardinality()       # todo: not implemented
246378092531250045243830074914327 68
sage: Graphs(19).random_element()    # todo: not implemented
Graph on 19 vertices
```

What we have seen so far also applies, in principle, to finite algebraic structures like the dihedral groups:

```
sage: G = DihedralGroup(4); G
Dihedral group of order 8 as a permutation group
sage: G.cardinality()
8
sage: G.list()
[(), (1,4)(2,3), (1,2,3,4), (1,3)(2,4), (1,3), (2,4), (1,4,3,2), (1,2)(3,4)]
```

or the algebra of 2×2 matrices over the finite field $\mathbf{Z}/2\mathbf{Z}$:

```
sage: C = MatrixSpace(GF(2), 2)
sage: C.list()
[
[0 0]  [1 0]  [0 1]  [0 0]  [0 0]  [1 1]  [1 0]  [1 0]  [0 1]  [0 1]
[0 0], [0 0], [0 0], [1 0], [0 1], [0 0], [1 0], [0 1], [1 0], [0 1],

[0 0]  [1 1]  [1 1]  [1 0]  [0 1]  [1 1]
[1 1], [1 0], [0 1], [1 1], [1 1], [1 1]
]
sage: C.cardinality()
16
```

Exercise

List all the monomials of degree 5 in three variables (see `IntegerVectors`). Manipulate the ordered set partitions `OrderedSetPartitions` and standard tableaux (`StandardTableaux`).

Exercise

List the alternating sign matrices of size 3, 4, and 5 (`AlternatingSignMatrices`), and try to guess the definition. The discovery and proof of the formula for the enumeration of these matrices (see the method `cardinality`), motivated by calculations of determinants in physics, is quite a story. In particular, the first proof, given by Zeilberger in 1992 was automatically produced by a computer program. It was 84 pages long, and required nearly a hundred people to verify it.

Exercise

Calculate by hand the number of vectors in $(\mathbf{Z}/2\mathbf{Z})^5$, and the number of matrices in $GL_3(\mathbf{Z}/2\mathbf{Z})$ (that is to say, the number of invertible 3×3 matrices with coefficients in $\mathbf{Z}/2\mathbf{Z}$). Verify your answer with `Sage`. Generalize to $GL_n(\mathbf{Z}/q\mathbf{Z})$.

Set comprehension and iterators

We will now show some of the possibilities offered by `Python` for constructing (and iterating through) sets, with a notation that is flexible and close to usual mathematical usage, and in particular the benefits this yields in combinatorics.

We begin by constructing the finite set $\{i^2 \parallel i \in \{1, 3, 7\}\}$:

```
sage: [ i^2 for i in [1, 3, 7] ]
[1, 9, 49]
```

and then the same set, but with i running from 1 to 9:

```
sage: [ i^2 for i in range(1,10) ]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

A construction of this form in Python is called *set comprehension*. A clause can be added to keep only those elements with i prime:

```
sage: [ i^2 for i in range(1,10) if is_prime(i) ]
[4, 9, 25, 49]
```

Combining more than one set comprehension, it is possible to construct the set $\{(i, j) \mid 1 \leq j < i < 5\}$:

```
sage: [ (i,j) for i in range(1,6) for j in range(1,i) ]
[(2, 1), (3, 1), (3, 2), (4, 1), (4, 2), (4, 3),
 (5, 1), (5, 2), (5, 3), (5, 4)]
```

or to produce Pascal's triangle:

```
sage: [[binomial(n,i) for i in range(n+1)] for n in range(10)]
[[1],
 [1, 1],
 [1, 2, 1],
 [1, 3, 3, 1],
 [1, 4, 6, 4, 1],
 [1, 5, 10, 10, 5, 1],
 [1, 6, 15, 20, 15, 6, 1],
 [1, 7, 21, 35, 35, 21, 7, 1],
 [1, 8, 28, 56, 70, 56, 28, 8, 1],
 [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]]
```

The execution of a set comprehension is accomplished in two steps; first an *iterator* is constructed, and then a list is filled with the elements successively produced by the iterator. Technically, an *iterator* is an object with a method next which returns a new value each time it is called, until it is exhausted. For example, the following iterator it:

```
sage: it = (binomial(3, i) for i in range(4))
```

returns successively the binomial coefficients $\binom{3}{i}$ with $i = 0, 1, 2, 3$:

```
sage: next(it)
1
sage: next(it)
3
sage: next(it)
3
sage: next(it)
1
```

When the iterator is finally exhausted, an exception is raised:

```
sage: next(it)
Traceback (most recent call last):
  ...
StopIteration
```

More generally, an *iterable* is a Python object L (a list, a set, ...) over whose elements it is possible to iterate. Technically, the iterator is constructed by iter(L). In practice, the commands iter and next are used very rarely, since for loops and list comprehensions provide a much pleasanter syntax:

```
sage: for s in Subsets(3):
....:     print s
{}
{1}
{2}
{3}
{1, 2}
{1, 3}
{2, 3}
{1, 2, 3}

sage: [ s.cardinality() for s in Subsets(3) ]
[0, 1, 1, 1, 2, 2, 2, 3]
```

What is the point of an iterator? Consider the following example:

```
sage: sum( [ binomial(8, i) for i in range(9) ] )
256
```

When it is executed, a list of 9 elements is constructed, and then it is passed as an argument to `sum` to add them up. If, on the other hand, the iterator is passed directly to `sum` (note the absence of square brackets):

```
sage: sum( binomial(8, i) for i in xrange(9) )
256
```

the function `sum` receives the iterator directly, and can short-circuit the construction of the intermediate list. If there are a large number of elements, this avoids allocating a large quantity of memory to fill a list which will be immediately destroyed [6].

Most functions that take a list of elements as input will also accept an iterator (or an iterable) instead. To begin with, one can obtain the list (or the tuple) of elements of an iterator as follows:

```
sage: list(binomial(8, i) for i in xrange(9))
[1, 8, 28, 56, 70, 56, 28, 8, 1]
sage: tuple(binomial(8, i) for i in xrange(9))
(1, 8, 28, 56, 70, 56, 28, 8, 1)
```

We now consider the functions `all` and `any` which denote respectively the n-ary *and* and *or*:

```
sage: all([True, True, True, True])
True
sage: all([True, False, True, True])
False
sage: any([False, False, False, False])
False
sage: any([False, False, True, False])
True
```

The following example verifies that all primes from 3 to 99 are odd:

```
sage: all( is_odd(p) for p in range(3,100) if is_prime(p) )
True
```

A *Mersenne prime* is a prime of the form $2^p - 1$. We verify that, for $p < 1000$, if $2^p - 1$ is prime, then p is also prime:

[6] Technical detail: `xrange` returns an iterator on $\{0, \ldots, 8\}$ while `range` returns the corresponding list. Starting in `Python` 3.0, `range` will behave like `xrange`, and `xrange` will no longer be needed.

```
sage: def mersenne(p): return 2^p -1
sage: [ is_prime(p)
....:   for p in range(1000) if is_prime(mersenne(p)) ]
[True, True, True, True, True, True, True, True, True, True,
 True, True, True, True]
```

Is the converse true?

Exercise

Try the two following commands and explain the considerable difference in the length of the calculations:

```
sage: all(   is_prime(mersenne(p))
....:        for p in range(1000) if is_prime(p)  )
False
sage: all( [ is_prime(mersenne(p))
....:        for p in range(1000) if is_prime(p)] )
False
```

We now try to find the smallest counter-example. In order to do this, we use the `Sage` function `exists`:

```
sage: exists( (p for p in range(1000) if is_prime(p)),
....:         lambda p: not is_prime(mersenne(p)) )
(True, 11)
```

Alternatively, we could construct an iterator on the counter-examples:

```
sage: counter_examples = \
....:   (p for p in range(1000)
....:      if is_prime(p) and not is_prime(mersenne(p)))
sage: next(counter_examples)
11
sage: next(counter_examples)
23
```

Exercice

What do the following commands do?

```
sage: cubes = [t**3 for t in range(-999,1000)]
sage: exists([(x,y) for x in cubes for y in cubes],  # long time (3s, 2012)
....:        lambda (x,y): x+y == 218)
(True, (-125, 343))
sage: exists(((x,y) for x in cubes for y in cubes),  # long time (2s, 2012)
....:        lambda (x,y): x+y == 218)
(True, (-125, 343))
```

Which of the last two is more economical in terms of time? In terms of memory? By how much?

Exercice

Try each of the following commands, and explain its result. If possible, hide the result first and try to guess it before launching the command.

Todo
hide the results by default

Warning: it will be necessary to interrupt the execution of some of the commands

```
sage: x = var('x')
sage: sum( x^len(s) for s in Subsets(8) )
x^8 + 8*x^7 + 28*x^6 + 56*x^5 + 70*x^4 + 56*x^3 + 28*x^2 + 8*x + 1

sage: sum( x^p.length() for p in Permutations(3) )
x^3 + 2*x^2 + 2*x + 1

sage: factor(sum( x^p.length() for p in Permutations(3) ))
(x^2 + x + 1)*(x + 1)

sage: P = Permutations(5)
sage: all( p in P for p in P )
True

sage: for p in GL(2, 2): print p; print
[1 0]
[0 1]

[0 1]
[1 0]

[0 1]
[1 1]

[1 1]
[0 1]

[1 1]
[1 0]

[1 0]
[1 1]

sage: for p in Partitions(3): print p    # not tested
[3]
[2, 1]
[1, 1, 1]
...

sage: for p in Partitions(): print p     # not tested
[]
[1]
[2]
[1, 1]
[3]
...

sage: for p in Primes(): print p         # not tested
2
3
5
7
```

Operations on iterators Python provides numerous tools for manipulating iterators; most of them are in the itertools library, which can be imported by:

```
sage: import itertools
```

We will demonstrate some applications, taking as a starting point the permutations of 3:

```
sage: list(Permutations(3))
[[1, 2, 3], [1, 3, 2], [2, 1, 3],
 [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

We can list the elements of a set by numbering them:

```
sage: list(enumerate(Permutations(3)))
[(0, [1, 2, 3]), (1, [1, 3, 2]), (2, [2, 1, 3]),
 (3, [2, 3, 1]), (4, [3, 1, 2]), (5, [3, 2, 1])]
```

select only the elements in positions 2, 3, and 4 (analogue of l[1:4]):

```
sage: import itertools
sage: list(itertools.islice(Permutations(3), 1, 4))
[[1, 3, 2], [2, 1, 3], [2, 3, 1]]
```

apply a function to all the elements:

```
sage: list(itertools.imap(lambda z: z.cycle_type(),
....:                     Permutations(3)))
[[1, 1, 1], [2, 1], [2, 1], [3], [3], [2, 1]]
```

or select the elements satisfying a certain condition:

```
sage: list(itertools.ifilter(lambda z: z.has_pattern([1,2]),
....:                        Permutations(3)))
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2]]
```

In all these situations, attrcall can be an advantageous alternative to creating an anonymous function:

```
sage: list(itertools.imap(lambda z: z.cycle_type(),
....:                     Permutations(3)))
[[1, 1, 1], [2, 1], [2, 1], [3], [3], [2, 1]]
sage: list(itertools.imap(attrcall("cycle_type"),
....:                     Permutations(3)))
[[1, 1, 1], [2, 1], [2, 1], [3], [3], [2, 1]]
```

Implementation of new iterators It is easy to construct new iterators, using the keyword yield instead of return in a function:

```
sage: def f(n):
....:     for i in range(n):
....:         yield i
```

After the yield, execution is not halted, but only suspended, ready to be continued from the same point. The result of the function is therefore an iterator over the successive values returned by yield:

```
sage: g = f(4)
sage: next(g)
0
```

```
sage: next(g)
1
sage: next(g)
2
sage: next(g)
3

sage: next(g)
Traceback (most recent call last):
  ...
StopIteration
```

The function could be used as follows:

```
sage: [ x for x in f(5) ]
[0, 1, 2, 3, 4]
```

This model of computation, called *continuation*, is very useful in combinatorics, especially when combined with recursion. Here is how to generate all words of a given length on a given alphabet:

```
sage: def words(alphabet,l):
....:     if l == 0:
....:         yield []
....:     else:
....:         for word in words(alphabet, l-1):
....:             for l in alphabet:
....:                 yield word + [l]
sage: [ w for w in words(['a','b'], 3) ]
[['a', 'a', 'a'], ['a', 'a', 'b'], ['a', 'b', 'a'],
 ['a', 'b', 'b'], ['b', 'a', 'a'], ['b', 'a', 'b'],
 ['b', 'b', 'a'], ['b', 'b', 'b']]
```

These words can then be counted by:

```
sage: sum(1 for w in words(['a','b','c','d'], 10))
1048576
```

Counting the words one by one is clearly not an efficient method in this case, since the formula n^ℓ is also available; note, though, that this is not the stupidest possible approach — it does, at least, avoid constructing the entire list in memory.

We now consider Dyck words, which are well-parenthesized words in the letters "(" and ")". The function below generates all the Dyck words of a given length (where the length is the number of pairs of parentheses), using the recursive definition which says that a Dyck word is either empty or of the form $(w_1)w_2$ where w_1 and w_2 are Dyck words:

```
sage: def dyck_words(l):
....:     if l==0:
....:         yield ''
....:     else:
....:         for k in range(l):
....:             for w1 in dyck_words(k):
....:                 for w2 in dyck_words(l-k-1):
....:                     yield '('+w1+')'+w2
```

Here are all the Dyck words of length 4:

```
sage: list(dyck_words(4))
['()()()()', '()()(())', '()(())()', '()(()())', '()((()))',
 '(())()()', '(())(())', '(()())()', '((()))()', '(()()())',
 '(()(()))', '((())())', '(((())))']
```

Counting them, we recover a well-known sequence:

```
sage: [ sum(1 for w in dyck_words(l)) for l in range(10) ]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

Exercise: complete binary tree iterator

Construct an iterator on the set C_n of complete binary trees with n leaves (see *Enumeration of trees using generating functions*).
Hint: Sage 4.8.2 does not yet have a native data structure to represent complete binary trees. One simple way to represent them is to define a formal variable `Leaf` for the leaves and a formal 2-ary function `Node`:

```
sage: var('Leaf')
Leaf
sage: function('Node', nargs=2)
Node
```

The second tree in *Figure: The five complete binary trees with four leaves* can be represented by the expression:

```
sage: tr = Node(Node(Leaf, Node(Leaf, Leaf)), Leaf)
```

Constructions

We will now see how to construct new sets starting from these building blocks. In fact, we have already begun to do this with the construction of $\mathcal{P}(\mathcal{P}(\mathcal{P}(\{1,2,3,4\})))$ in the previous section, and to construct the example of sets of cards in *Initial examples*.

Consider a large Cartesian product:

```
sage: C = cartesian_product([Compositions(8), Permutations(20)]); C
The Cartesian product of (Compositions of 8, Standard permutations of 20)
sage: C.cardinality()
311411457046609920000
```

Clearly, it is impractical to construct the list of all the elements of this Cartesian product! And, in the following example, H is equipped with the usual combinatorial operations and also its structure as a product group:

```
sage: G = DihedralGroup(4)
sage: H = cartesian_product([G,G])
sage: H in Groups()
True
sage: t = H.an_element()
sage: t
((1,2,3,4), (1,2,3,4))
sage: t*t
((1,3)(2,4), (1,3)(2,4))
```

We now construct the union of two existing disjoint sets:

```
sage: C = DisjointUnionEnumeratedSets(
....:       [ Compositions(4), Permutations(3)] )
sage: C
Disjoint union of Family (Compositions of 4,
Standard permutations of 3)
sage: C.cardinality()
14
sage: C.list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2],
[3, 1], [4], [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1],
[3, 1, 2], [3, 2, 1]]
```

It is also possible to take the union of more than two disjoint sets, or even an infinite number of them. We will now construct the set of all permutations, viewed as the union of the sets P_n of permutations of size n. We begin by constructing the infinite family $F = (P_n)_{n \in N}$:

```
sage: F = Family(NonNegativeIntegers(), Permutations); F
Lazy family (<class 'sage.combinat.permutation.Permutations'>(i))_{i in Non negative integers}
sage: F.keys()
Non negative integers
sage: F[1000]
Standard permutations of 1000
```

Now we can construct the disjoint union $\bigcup_{n \in \mathbf{N}} P_n$:

```
sage: U = DisjointUnionEnumeratedSets(F); U
Disjoint union of
Lazy family (<class 'sage.combinat.permutation.Permutations'>(i))_{i in Non negative integers}
```

It is an infinite set:

```
sage: U.cardinality()
+Infinity
```

which doesn't prohibit iteration through its elements, though it will be necessary to interrupt it at some point:

```
sage: for p in U:                # not tested
....:     print p
[]
[1]
[1, 2]
[2, 1]
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
...
```

Note: the above set could also have been constructed directly with:

```
sage: U = Permutations(); U
Standard permutations
```

Summary

Sage provides a library of common enumerated sets, which can be combined by standard constructions, giving a toolbox that is flexible (but which could still be expanded). It is also possible to add new building blocks to Sage with a few lines (see the code in `FiniteEnumeratedSets().example()`). This is made possible by the uniformity of the interfaces and the fact that Sage is based on an object-oriented language. Also, very large or even infinite sets can be manipulated thanks to lazy evaluation strategies (iterators, etc.).

There is no magic to any of this: under the hood, Sage applies the usual rules (for example, that the cardinality of $E \times E$ is $|E|^2$); the added value comes from the capacity to manipulate complicated constructions. The situation is comparable to Sage's implementation of differential calculus: Sage applies the usual rules for differentiation of functions and their compositions, where the added value comes from the possibility of manipulating complicated formulas. In this sense, Sage implements a *calculus* of finite enumerated sets.

Generic algorithms

Lexicographic generation of lists of integers

Among the classic enumerated sets, especially in algebraic combinatorics, a certain number are composed of lists of integers of fixed sum, such as partitions, compositions, or integer vectors. These examples can also have supplementary constraints added to them. Here are some examples. We start with the integer vectors with sum 10 and length 3, with parts bounded below by 2, 4 and 2 respectively:

```
sage: IntegerVectors(10, 3, min_part=2, max_part=5,
....:                inner=[2, 4, 2]).list()
[[4, 4, 2], [3, 5, 2], [3, 4, 3], [2, 5, 3], [2, 4, 4]]
```

The compositions of 5 with each part at most 3, and with length 2 or 3:

```
sage: Compositions(5, max_part=3,
....:              min_length=2, max_length=3).list()
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2], [1, 3, 1],
 [1, 2, 2], [1, 1, 3]]
```

The strictly decreasing partitions of 5:

```
sage: Partitions(5, max_slope=-1).list()
[[5], [4, 1], [3, 2]]
```

These sets share the same underlying algorithmic structure, implemented in the more general – and slightly more cumbersome – class `IntegerListsLex`. This class models sets of vectors $(\ell_0, \ldots, \ell_k)$ of non-negative integers, with constraints on the sum and the length, and bounds on the parts and on the consecutive differences between the parts. Here are some more examples:

```
sage: IntegerListsLex(10, length=3,
....:                 min_part=2, max_part=5,
....:                 floor=[2, 4, 2]).list()
[[4, 4, 2], [3, 5, 2], [3, 4, 3], [2, 5, 3], [2, 4, 4]]

sage: IntegerListsLex(5, min_part=1, max_part=3,
....:                 min_length=2, max_length=3).list()
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2],
 [1, 3, 1], [1, 2, 2], [1, 1, 3]]

sage: IntegerListsLex(5, min_part=1, max_slope=-1).list()
[[5], [4, 1], [3, 2]]
```

```
sage: list(Compositions(5, max_length=2))
[[5], [4, 1], [3, 2], [2, 3], [1, 4]]

sage: list(IntegerListsLex(5, max_length=2, min_part=1))
[[5], [4, 1], [3, 2], [2, 3], [1, 4]]
```

The point of the model of `IntegerListsLex` is in the compromise between generality and efficiency. The main algorithm permits iteration through the elements of such a set S in reverse lexicographic order with a good complexity in most practical use cases. Roughly speaking, the time needed to iterate through all the elements of S is proportional to the number of elements, where the proportion factor is controlled by the length l of the longest element of S. In addition, the memory usage is also controlled by l, which is to say negligible in practice.

This algorithm is based on a very general principle for traversing a decision tree, called *branch and bound*: at the top level, we run through all the possible choices for ℓ_0; for each of these choices, we run through all the possible choices for ℓ_1, and so on. Mathematically speaking, we have put the structure of a prefix tree on the elements of S: a node of the tree at depth k corresponds to a prefix $\ell_0, \ldots, \ell_k$ of one (or more) elements of S (see *Figure: The prefix tree of the partitions of 5.*).

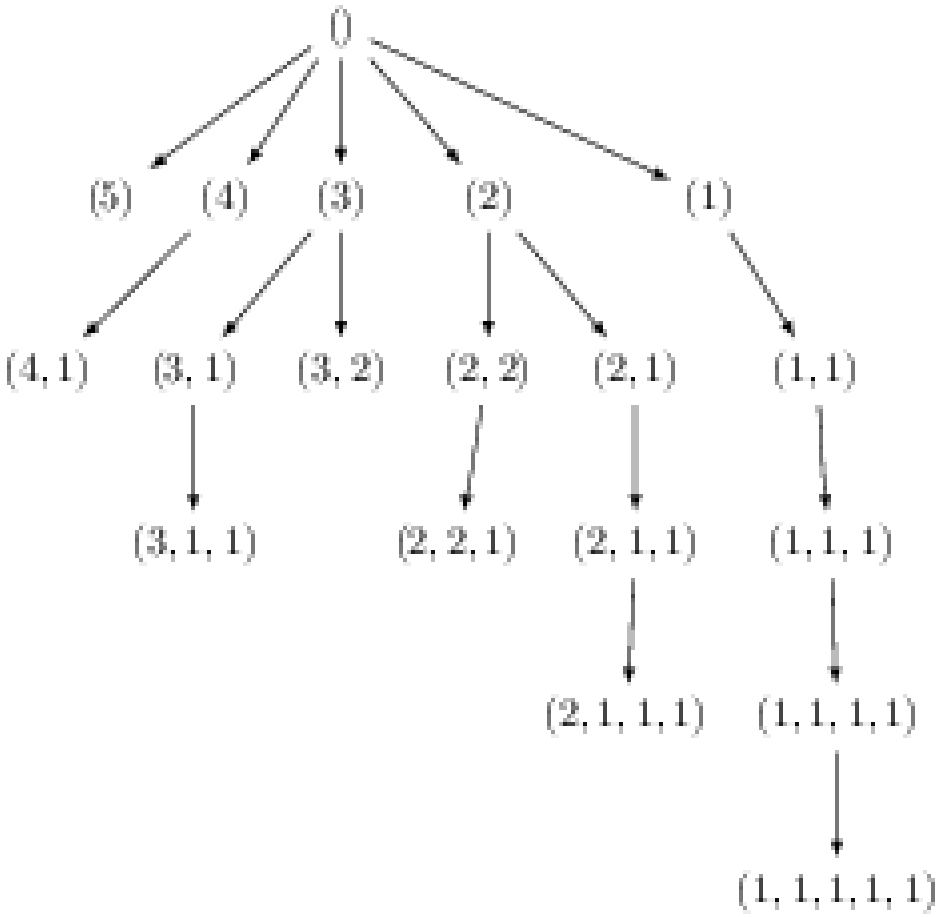


Figure 5.2: Figure: The prefix tree of the partitions of 5.

The usual problem with this type of approach is to avoid bad decisions which lead to leaving the prefix tree and exploring dead branches; this is particularly problematic because the growth of the number of elements is usually exponential in the depth. It turns out that the constraints listed above are simple enough to be able to reasonably

predict when a sequence $\ell_0, \ldots, \ell_k$ is a prefix of some element S. Hence, most dead branches can be pruned.

Integer points in polytopes

Although the algorithm for iteration in `IntegerListsLex` is efficient, its counting algorithm is naive: it just iterates over all the elements.

There is an alternative approach to treating this problem: modelling the desired lists of integers as the set of integer points of a polytope, that is to say, the set of solutions with integer coordinates of a system of linear inequalities. This is a very general context in which there exist advanced counting algorithms (e.g. Barvinok), which are implemented in libraries like `LattE`. Iteration does not pose a hard problem in principle. However, there are two limitations that justify the existence of `IntegerListsLex`. The first is theoretical: lattice points in a polytope only allow modelling of problems of a fixed dimension (length). The second is practical: at the moment only the library `PALP` has a `Sage` interface, and though it offers multiple capabilities for the study of polytopes, in the present application it only produces a list of lattice points, without providing either an iterator or non-naive counting:

```
sage: A = random_matrix(ZZ, 6, 3, x=7)
sage: L = LatticePolytope(A.rows())
sage: L.points()                                    # random
M(4, 1, 0),
M(0, 3, 5),
M(2, 2, 3),
M(6, 1, 3),
M(1, 3, 6),
M(6, 2, 3),
M(3, 2, 4),
M(3, 2, 3),
M(4, 2, 4),
M(4, 2, 3),
M(5, 2, 3)
in 3-d lattice M
sage: L.npoints()                                     # random
11
```

This polytope can be visualized in 3D with `L.plot3d()` (see *Figure: The polytope and its integer points, in cross-eyed stereographic perspective.*).

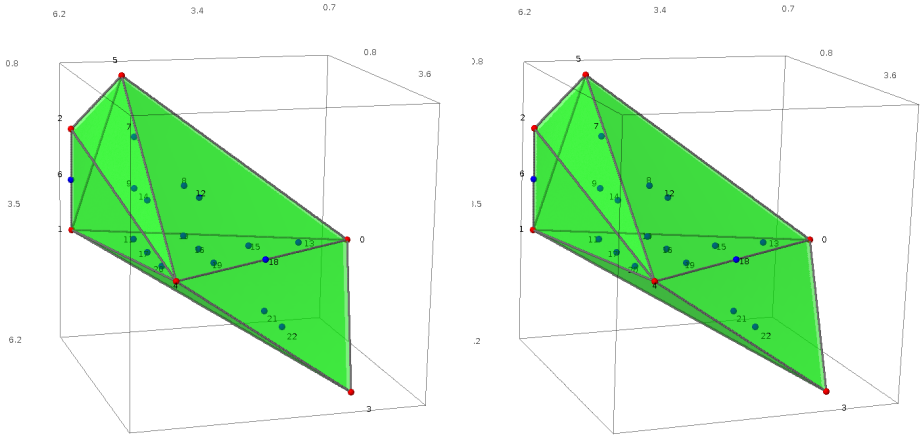


Figure 5.3: Figure: The polytope L and its integer points, in cross-eyed stereographic perspective.

Species, decomposable combinatorial classes

In *Enumeration of trees using generating functions*, we showed how to use the recursive definition of binary trees to count them efficiently using generating functions. The techniques we used there are very general, and apply whenever the sets involved can be defined recursively (depending on who you ask, such a set is called a *decomposable combinatorial class* or, roughly speaking, a *combinatorial species*). This includes all the types of trees, but also permutations, compositions, functional graphs, etc.

Here, we illustrate just a few examples using the `Sage` library on combinatorial species:

```
sage: from sage.combinat.species.library import *
sage: o = var('o')
```

We begin by redefining the complete binary trees; to do so, we stipulate the recurrence relation directly on the sets:

```
sage: BT = CombinatorialSpecies()
sage: Leaf =  SingletonSpecies()
sage: BT.define( Leaf + (BT*BT) )
```

Now we can construct the set of trees with five nodes, list them, count them...:

```
sage: BT5 = BT.isotypes([o]*5)
sage: BT5.cardinality()
14
sage: BT5.list()
[o*(o*(o*(o*o))), o*(o*((o*o)*o)), o*((o*o)*(o*o)),
 o*((o*(o*o))*o), o*(((o*o)*o)*o), (o*o)*(o*(o*o)),
 (o*o)*((o*o)*o), (o*(o*o))*(o*o), ((o*o)*o)*(o*o),
 (o*(o*(o*o)))*o, (o*((o*o)*o))*o, ((o*o)*(o*o))*o,
 ((o*(o*o))*o)*o, (((o*o)*o)*o)*o]
```

The trees are constructed using a generic recursive structure; the display is therefore not wonderful. To do better, it would be necessary to provide `Sage` with a more specialized data structure with the desired display capabilities.

We recover the generating function for the Catalan numbers:

```
sage: g = BT.isotype_generating_series(); g
x + x^2 + 2*x^3 + 5*x^4 + 14*x^5 + O(x^6)
```

which is returned in the form of a lazy power series:

```
sage: g[100]
227508830794229349661819540395688853956041682601541047340
```

We finish with the Fibonacci words, which are binary words without two consecutive "1"s. They admit a natural recursive definition:

```
sage: Eps =  EmptySetSpecies()
sage: Z0  =  SingletonSpecies()
sage: Z1  =  Eps*SingletonSpecies()
sage: FW  = CombinatorialSpecies()
sage: FW.define(Eps + Z0*FW  +  Z1*Eps + Z1*Z0*FW)
```

The Fibonacci sequence is easily recognized here, hence the name:

```
sage: L = FW.isotype_generating_series().coefficients(15); L
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

```
sage: oeis(L)                                             # optional -- internet
0: A000045: Fibonacci numbers: F(n) = F(n-1) + F(n-2) with F(0) = 0 and F(1) = 1.
1: A212804: Expansion of (1-x)/(1-x-x^2).
2: A132636: Fib(n) mod n^3.
```

This is an immediate consequence of the recurrence relation. One can also generate immediately all the Fibonacci words of a given length, with the same limitations resulting from the generic display.

```
sage: FW3 = FW.isotypes([o]*3)
sage: FW3.list()
[o*(o*(o*{})), o*(o*(({}*o)*{})), o*((({}*o)*o)*{}),
 (({}*o)*o)*(o*{}), (({}*o)*o)*(({}*o)*{})]
```

Graphs up to isomorphism

We saw in *Some other finite enumerated sets* that Sage could generate graphs and partial orders up to isomorphism. We will now describe the underlying algorithm, which is the same in both cases, and covers a substantially wider class of problems.

We begin by recalling some notions. A graph $G = (V, E)$ is a set V of vertices and a set E of edges connecting these vertices; an edge is described by a pair $\{u, v\}$ of distinct vertices of V. Such a graph is called labelled; its vertices are typically numbered by considering $V = \{1, 2, 3, 4, 5\}$.

In many problems, the labels on the vertices play no role. Typically a chemist wants to study all the possible molecules with a given composition, for example the alkanes with $n = 8$ atoms of carbon and $2n + 2 = 18$ atoms of hydrogen. He therefore wants to find all the graphs consisting of 8 vertices with 4 neighbours, and 18 vertices with a single neighbour. The different carbon atoms, however, are all considered to be identical, and the same for the hydrogen atoms. The problem of our chemist is not imaginary; this type of application is actually at the origin of an important part of the research in graph theory on isomorphism problems.

Working by hand on a small graph it is possible, as in the example of *Some other finite enumerated sets*, to make a drawing, erase the labels, and "forget" the geometrical information about the location of the vertices in the plane. However, to represent a graph in a computer program, it is necessary to introduce labels on the vertices so as to be able to describe how the edges connect them together. To compensate for the extra information which we have introduced, we then say that two labelled graphs g_1 and g_2 are *isomorphic* if there is a bijection from the vertices of g_1 to those of g_2, which maps bijectively the edges of g_1 to those of g_2; an *unlabelled graph* is then an equivalence class of labelled graphs.

In general, testing if two labelled graphs are isomorphic is expensive. However, the number of graphs, even unlabelled, grows very rapidly. Nonetheless, it is possible to list unlabelled graphs very efficiently considering their number. For example, the program Nauty can list the 12005168 simple graphs with 10 vertices in 20 seconds.

As in *Lexicographic generation of lists of integers*, the general principle of the algorithm is to organize the objects to be enumerated into a tree that one traverses.

For this, in each equivalence class of labelled graphs (that is to say, for each unlabelled graph) one fixes a convenient canonical representative. The following are the fundamental operations:

- Testing whether a labelled graph is canonical
- Calculating the canonical representative of a labelled graph

These unavoidable operations remain expensive; one therefore tries to minimize the number of calls to them.

The canonical representatives are chosen in such a way that, for each canonical labelled graph G, there is a canonical choice of an edge whose removal produces a canonical graph again, which is called the father of G. This property implies that it is possible to organize the set of canonical representatives as a tree: at the root, the graph with no edges; below it, its unique child, the graph with one edge; then the graphs with two edges, and so on. The set of children of

a graph G can be constructed by *augmentation*, adding an edge in all the possible ways to G, and then selecting, from among those graphs, the ones that are still canonical [7]. Recursively, one obtains all the canonical graphs.

In what sense is this algorithm generic? Consider for example planar graphs (graphs which can be drawn in the plane without edges crossing): by removing an edge from a planar graph, one obtains another planar graph; so planar graphs form a subtree of the previous tree. To generate them, exactly the same algorithm can be used, selecting only the children which are planar:

```
sage: [len(list(graphs(n, property = lambda G: G.is_planar())))
....:  for n in range(7)]
[1, 1, 2, 4, 11, 33, 142]
```

In a similar fashion, one can generate any family of graphs closed under deletion of an edge, and in particular any family characterized by a forbidden subgraph. This includes for example forests (graphs without cycles), bipartite graphs (graphs without odd cycles), etc. This can be applied to generate:

- partial orders, via the bijection with Hasse diagrams which are oriented graphs without cycles and without edges implied by the transitivity of the order relation;
- lattices (not implemented in `Sage`), via the bijection with the meet semi-lattice obtained by deleting the maximal vertex; in this case an augmentation by vertices rather than by edges is used.

REFERENCES:

5.1.317 Vector Partitions

AUTHORS:

- Amritanshu Prasad (2013): Initial version

`sage.combinat.vector_partition.`**`IntegerVectorsIterator`**(*vect*, *min=None*)

Return an iterator over the list of integer vectors which are componentwise less than or equal to `vect`, and lexicographically greater than or equal to `min`.

INPUT:

- `vect` – A list of non-negative integers
- `min` – A list of non-negative integers dominated elementwise by `vect`

OUTPUT:

A list in lexicograohic order of all integer vectors (as lists) which are dominated elementwise by `vect` and are greater than or equal to `min` in lexicographic order.

EXAMPLES:

```
sage: from sage.combinat.vector_partition import IntegerVectorsIterator
sage: list(IntegerVectorsIterator([1, 1]))
[[0, 0], [0, 1], [1, 0], [1, 1]]

sage: list(IntegerVectorsIterator([1, 1], min = [1, 0]))
[[1, 0], [1, 1]]
```

class `sage.combinat.vector_partition.`**`VectorPartition`**(*parent*, *vecpar*)

Bases: `sage.combinat.combinat.CombinatorialElement`

[7] In practice, an efficient implementation would exploit the symmetries of G, i.e., its automorphism group, to reduce the number of children to explore, and to reduce the cost of each test of canonicity.

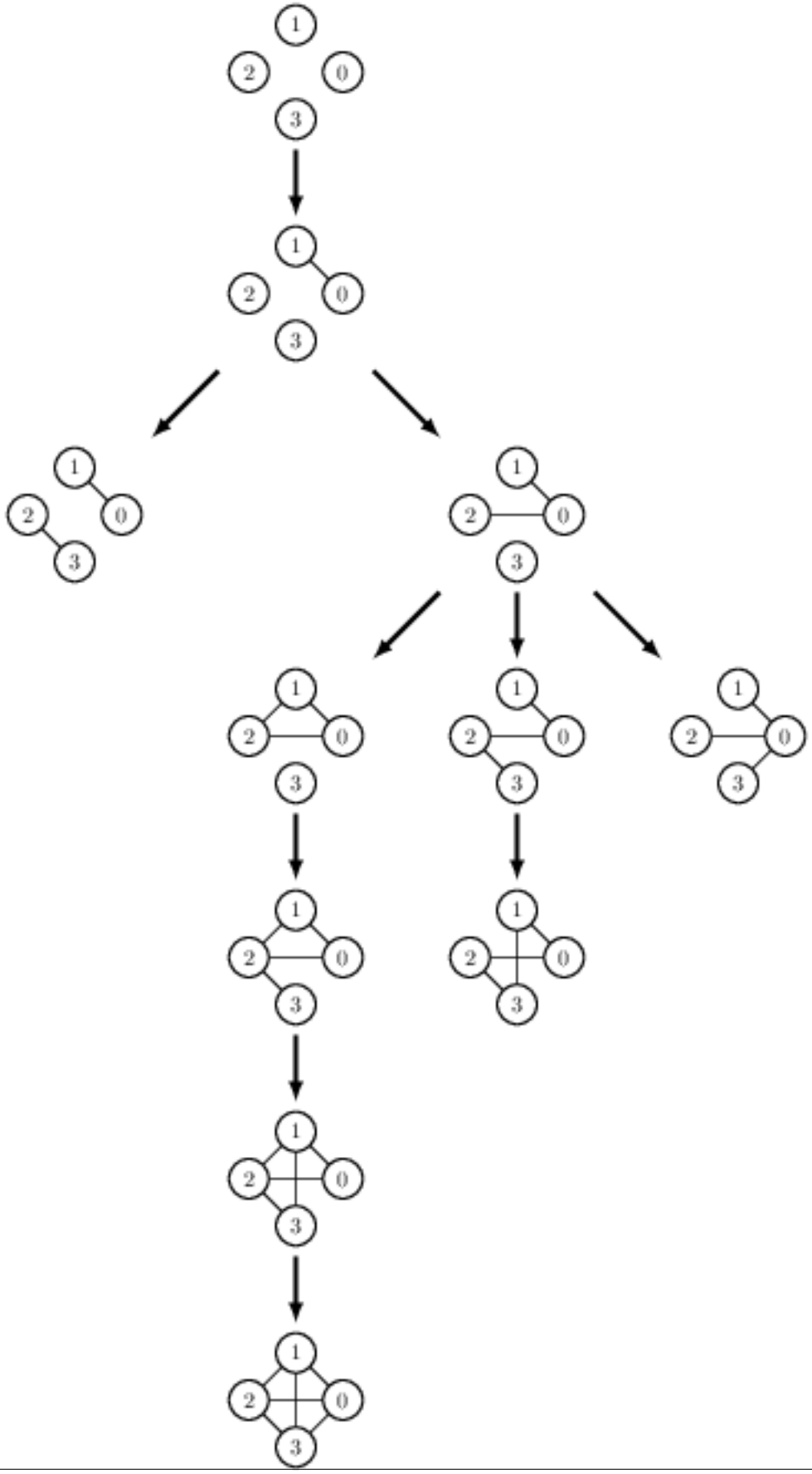


Figure 5.4: Figure: The generation tree of simple graphs with 4 vertices.

A vector partition is a multiset of integer vectors.

partition_at_vertex(*i*)
Return the partition obtained by sorting the `i`-th elements of the vectors in the vector partition.

EXAMPLES:
```
sage: V = VectorPartition([[1, 2, 1], [2, 4, 1]])
sage: V.partition_at_vertex(1)
[4, 2]
```

sum()
Return the sum vector as a list.

EXAMPLES:
```
sage: V = VectorPartition([[3, 2, 1], [2, 2, 1]])
sage: V.sum()
[5, 4, 2]
```

class sage.combinat.vector_partition.**VectorPartitions**(*vec*, *min*)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Class of all vector partitions of `vec` with all parts greater than or equal to `min` in lexicographic order.

A vector partition of `vec` is a list of vectors with non-negative integer entries whose sum is `vec`.

INPUT:

- `vec` – a list of non-negative integers.

EXAMPLES:

If `min` is not specified, then the class of all vector partitions of `vec` is created:
```
sage: VP = VectorPartitions([2, 2])
sage: for vecpar in VP:
....:     print vecpar
[[0, 1], [0, 1], [1, 0], [1, 0]]
[[0, 1], [0, 1], [2, 0]]
[[0, 1], [1, 0], [1, 1]]
[[0, 1], [2, 1]]
[[0, 2], [1, 0], [1, 0]]
[[0, 2], [2, 0]]
[[1, 0], [1, 2]]
[[1, 1], [1, 1]]
[[2, 2]]
```

If `min` is specified, then the class consists of only those vector partitions whose parts are all greater than or equal to `min` in lexicographic order:
```
sage: VP = VectorPartitions([2, 2], min = [1, 0])
sage: for vecpar in VP:
....:     print vecpar
[[1, 0], [1, 2]]
[[1, 1], [1, 1]]
[[2, 2]]
```

Element
alias of VectorPartition

sage.combinat.vector_partition.**find_min**(*vect*)
Return a string of 0's with one 1 at the location where the list vect has its last entry which is not equal to 0.

INPUT:

- vec – A list of integers

OUTPUT:

A list of the same length with 0's everywhere, except for a 1 at the last position where vec has an entry not equal to 0.

EXAMPLES:

```
sage: from sage.combinat.vector_partition import find_min
sage: find_min([2, 1])
[0, 1]
sage: find_min([2, 1, 0])
[0, 1, 0]
```

5.1.318 Words

Todo

- Check that those are the important entry points
- Add links from the module documentation to the important classes/functions

---

- sage.combinat.words.alphabet.build_alphabet()
- sage.combinat.words.word.Word()
- sage.combinat.words.words.Words()
- *A collection of constructors of common words*
- sage.combinat.words.shuffle_product.ShuffleProduct, sage.combinat.words.shuffle_product
  ???
- *Suffix Tries and Suffix Trees*
- *Word morphisms/substitutions*
- *Word paths*

5.1.319 Abstract word (finite or infinite)

This module gathers functions that works for both finite and infinite words.

AUTHORS:

- Sebastien Labbe
- Franco Saliola

EXAMPLES:

```
sage: a = 0.618
sage: g = words.CodingOfRotationWord(alpha=a, beta=1-a, x=a)
sage: f = words.FibonacciWord()
sage: p = f.longest_common_prefix(g, length='finite')
sage: p
word: 0100101001001010010100100101001001010010...
sage: p.length()
231
```

class sage.combinat.words.abstract_word.**Word_class**

Bases: sage.structure.sage_object.SageObject

apply_morphism(*morphism*)

Returns the word obtained by applying the morphism to self.

INPUT:

- morphism - Can be an instance of WordMorphism, or anything that can be used to construct one.

EXAMPLES:

```
sage: w = Word("ab")
sage: d = {'a':'ab', 'b':'ba'}
sage: w.apply_morphism(d)
word: abba
sage: w.apply_morphism(WordMorphism(d))
word: abba

sage: w = Word('ababa')
sage: d = dict(a='ab', b='ba')
sage: d
{'a': 'ab', 'b': 'ba'}
sage: w.apply_morphism(d)
word: abbaabbaab
```

For infinite words:

```
sage: t = words.ThueMorseWord([0,1]); t
word: 0110100110010110100101100110100110010110...
sage: t.apply_morphism({0:8,1:9})
word: 89989889988989989889989889988998898998898998...
```

complete_return_words_iterator(*fact*)

Returns an iterator over all the complete return words of fact in self (without unicity).

A complete return words u of a factor v is a factor starting by the given factor v and ending just after the next occurrence of this factor v. See for instance [1].

INPUT:

- fact - a non empty finite word

OUTPUT:

iterator

EXAMPLES:

```
sage: TM = words.ThueMorseWord()
sage: fact = Word([0,1,1,0,1])
sage: it = TM.complete_return_words_iterator(fact)
sage: next(it)
word: 01101001100101101
```

```
sage: next(it)
word: 01101001011001101
sage: next(it)
word: 011010011001011001101
sage: next(it)
word: 0110100101101
sage: next(it)
word: 01101001100101101
sage: next(it)
word: 01101001011001101
```

REFERENCES:

- [1] J. Justin, L. Vuillon, Return words in Sturmian and episturmian words, Theor. Inform. Appl. 34 (2000) 343–356.

delta()

Returns the image of self under the delta morphism. This is the word composed of the length of consecutive runs of the same letter in a given word.

OUTPUT:

Word over integers

EXAMPLES:

For finite words:

```
sage: W = Words('0123456789')
sage: W('22112122').delta()
word: 22112
sage: W('555008').delta()
word: 321
sage: W().delta()
word:
sage: Word('aabbabaa').delta()
word: 22112
```

For infinite words:

```
sage: t = words.ThueMorseWord()
sage: t.delta()
word: 12112221121121122211222112221122112112112221...
```

factor_occurrences_iterator(*fact*)

Returns an iterator over all occurrences (including overlapping ones) of fact in self in their order of appearance.

INPUT:

- `fact` - a non empty finite word

OUTPUT:

iterator

EXAMPLES:

```
sage: TM = words.ThueMorseWord()
sage: fact = Word([0,1,1,0,1])
sage: it = TM.factor_occurrences_iterator(fact)
sage: next(it)
0
```

```
sage: next(it)
12
sage: next(it)
24
```

finite_differences(*mod=None*)

Returns the word obtained by the diffences of consecutive letters of self.

INPUT:

- `self` - A word over the integers.
- **mod - (default: None) It can be one of the following:**
  - None or 0 : result is over the integers
  - integer : result is over the integers modulo `mod`.

EXAMPLES:

```
sage: w = Word([x^2 for x in range(10)])
sage: w.finite_differences()
word: 1,3,5,7,9,11,13,15,17
sage: w.finite_differences(mod=4)
word: 131313131
sage: w.finite_differences(mod=0)
word: 1,3,5,7,9,11,13,15,17
```

TESTS:

```
sage: w = Word([2,3,6])
sage: w.finite_differences()
word: 13
sage: w = Word([2,6])
sage: w.finite_differences()
word: 4
sage: w = Word([2])
sage: w.finite_differences()
word:
sage: w = Word()
sage: w.finite_differences()
word:
```

If the word is infinite, so is the result:

```
sage: w = Word(lambda n:n)
sage: u = w.finite_differences()
sage: u
word: 11111111111111111111111111111111111111...
sage: type(u)
<class 'sage.combinat.words.word.InfiniteWord_iter_with_caching'>
```

is_empty()

Returns True if the length of self is zero, and False otherwise.

EXAMPLES:

```
sage: it = iter([])
sage: Word(it).is_empty()
True
sage: it = iter([1,2,3])
sage: Word(it).is_empty()
```

```
False
sage: from itertools import count
sage: Word(count()).is_empty()
False
```

is_finite()

Returns whether this word is known to be finite.

Warning: A word defined by an iterator such that its end has never been reached will returns False.

EXAMPLES:

```
sage: Word([]).is_finite()
True
sage: Word('a').is_finite()
True
sage: TM = words.ThueMorseWord()
sage: TM.is_finite()
False

sage: w = Word(iter('a'*100))
sage: w.is_finite()
False
```

iterated_right_palindromic_closure(*f=None*, *algorithm='recursive'*)

Returns the iterated (f-)palindromic closure of self.

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be callable on letters as well as words (e.g. WordMorphism).
- `algorithm` - string (default: `'recursive'`) specifying which algorithm to be used when computing the iterated palindromic closure. It must be one of the two following values:
  - `'definition'` - computed using the definition
  - `'recursive'` - computation based on an efficient formula that recursively computes the iterated right palindromic closure without having to recompute the longest f-palindromic suffix at each iteration [2].

OUTPUT:

word – the iterated (f-)palindromic closure of self

EXAMPLES:

```
sage: Word('123').iterated_right_palindromic_closure()
word: 1213121

sage: w = Word('abc')
sage: w.iterated_right_palindromic_closure()
word: abacaba

sage: w = Word('aaa')
sage: w.iterated_right_palindromic_closure()
word: aaa

sage: w = Word('abbab')
sage: w.iterated_right_palindromic_closure()
word: ababaababababaababa
```

A right f-palindromic closure:

```
sage: f = WordMorphism('a->b,b->a')
sage: w = Word('abbab')
sage: w.iterated_right_palindromic_closure(f=f)
word: abbaabbaababbaabbaabbaababbaabbaab
```

An infinite word:

```
sage: t = words.ThueMorseWord('ab')
sage: t.iterated_right_palindromic_closure()
word: ababaababababaababaababababaababababaababab...
```

There are two implementations computing the iterated right f-palindromic closure, the latter being much more efficient:

```
sage: w = Word('abaab')
sage: u = w.iterated_right_palindromic_closure(algorithm='definition')
sage: v = w.iterated_right_palindromic_closure(algorithm='recursive')
sage: u
word: abaabaababaabaaba
sage: u == v
True
sage: w = words.RandomWord(8)
sage: u = w.iterated_right_palindromic_closure(algorithm='definition')
sage: v = w.iterated_right_palindromic_closure(algorithm='recursive')
sage: u == v
True
```

TESTS:

The empty word:

```
sage: w = Word()
sage: w.iterated_right_palindromic_closure()
word:
```

The length-1 word:

```
sage: Word('1').iterated_right_palindromic_closure()
word: 1
```

If the word is finite, so is the result:

```
sage: w = Word([0,1]*7)
sage: c = w.iterated_right_palindromic_closure()
sage: type(c)
<class 'sage.combinat.words.word.FiniteWord_iter_with_caching'>
```

REFERENCES:

- [1] A. de Luca, A. De Luca, Pseudopalindrome closure operators in free monoids, Theoret. Comput. Sci. 362 (2006) 282–300.
- [2] J. Justin, Episturmian morphisms and a Galois theorem on continued fractions, RAIRO Theoret. Informatics Appl. 39 (2005) 207-215.

length()

Returns the length of self.

TESTS:

```
sage: from sage.combinat.words.word import Word_class
sage: w = Word(iter('abba'*100), length="unknown")
sage: w.length() is None
True
sage: w = Word(iter('abba'), length="finite")
sage: w.length()
4
sage: w = Word(iter([0,1,1,0,1,0,0,1]*100), length="unknown")
sage: w.length() is None
True
sage: w = Word(iter([0,1,1,0,1,0,0,1]), length="finite")
sage: w.length()
8
```

lex_greater(*other*)

Returns True if self is lexicographically greater than other.

EXAMPLES:

```
sage: w = Word([1,2,3])
sage: u = Word([1,3,2])
sage: v = Word([3,2,1])
sage: w.lex_greater(u)
False
sage: v.lex_greater(w)
True
sage: a = Word("abba")
sage: b = Word("abbb")
sage: a.lex_greater(b)
False
sage: b.lex_greater(a)
True
```

For infinite words:

```
sage: t = words.ThueMorseWord()
sage: t[:10].lex_greater(t)
False
sage: t.lex_greater(t[:10])
True
```

lex_less(*other*)

Returns True if self is lexicographically less than other.

EXAMPLES:

```
sage: w = Word([1,2,3])
sage: u = Word([1,3,2])
sage: v = Word([3,2,1])
sage: w.lex_less(u)
True
sage: v.lex_less(w)
False
sage: a = Word("abba")
sage: b = Word("abbb")
sage: a.lex_less(b)
True
sage: b.lex_less(a)
False
```

For infinite words:

```
sage: t = words.ThueMorseWord()
sage: t.lex_less(t[:10])
False
sage: t[:10].lex_less(t)
True
```

longest_common_prefix(*other*, *length='unknown'*)

Returns the longest common prefix of self and other.

INPUT:

- `other` - word
- `length` - string (optional, default: `'unknown'`) the length type of the resulting word if known. It may be one of the following:
  - `'unknown'`
  - `'finite'`
  - `'infinite'`

EXAMPLES:

```
sage: f = lambda n : add(Integer(n).digits(2)) % 2
sage: t = Word(f)
sage: u = t[:10]
sage: t.longest_common_prefix(u)
word: 0110100110
```

The longest common prefix of two equal infinite words:

```
sage: t1 = Word(f)
sage: t2 = Word(f)
sage: t1.longest_common_prefix(t2)
word: 01101001100101101001011001101001100100110...
```

Useful to study the approximation of an infinite word:

```
sage: a = 0.618
sage: g = words.CodingOfRotationWord(alpha=a, beta=1-a, x=a)
sage: f = words.FibonacciWord()
sage: p = f.longest_common_prefix(g, length='finite')
sage: p.length()
231
```

TESTS:

```
sage: w = Word('12345')
sage: y = Word('1236777')
sage: w.longest_common_prefix(y)
word: 123
sage: w.longest_common_prefix(w)
word: 12345
sage: y.longest_common_prefix(w)
word: 123
sage: y.longest_common_prefix(y)
word: 1236777
sage: Word().longest_common_prefix(w)
word:
sage: w.longest_common_prefix(Word())
```

```
word:
sage: w.longest_common_prefix(w[:3])
word: 123
sage: Word("11").longest_common_prefix(Word("1"))
word: 1
sage: Word("1").longest_common_prefix(Word("11"))
word: 1
```

With infinite words:

```
sage: t = words.ThueMorseWord('ab')
sage: u = t[:10]
sage: u.longest_common_prefix(t)
word: abbabaabba
sage: u.longest_common_prefix(u)
word: abbabaabba
```

Check length:

```
sage: w1 = Word(iter('ab'*200))
sage: w2 = Word(iter('bcd'*200))
sage: w1.longest_common_prefix(w2, length=19)
Traceback (most recent call last):
...
ValueError: invalid argument length (=19)
```

longest_periodic_prefix(*period=1*)

Returns the longest prefix of self having the given period.

INPUT:

- `period` - positive integer (optional, default 1)

OUTPUT:

word

EXAMPLES:

```
sage: Word([]).longest_periodic_prefix()
word:
sage: Word([1]).longest_periodic_prefix()
word: 1
sage: Word([1,2]).longest_periodic_prefix()
word: 1
sage: Word([1,1,2]).longest_periodic_prefix()
word: 11
sage: Word([1,2,1,2,1,3]).longest_periodic_prefix(2)
word: 12121
sage: type(_)
<class 'sage.combinat.words.word.FiniteWord_iter_with_caching'>
sage: Word(lambda n:0).longest_periodic_prefix()
word: 00000000000000000000000000000000000000000...
```

palindrome_prefixes_iterator(*max_length=None*)

Returns an iterator over the palindrome prefixes of self.

INPUT:

- `max_length` - non negative integer or None (optional, default: None) the maximum length of the prefixes

OUTPUT:

iterator

EXAMPLES:

```
sage: w = Word('abaaba')
sage: for pp in w.palindrome_prefixes_iterator(): pp
word:
word: a
word: aba
word: abaaba
sage: for pp in w.palindrome_prefixes_iterator(max_length=4): pp
word:
word: a
word: aba
```

You can iterate over the palindrome prefixes of an infinite word:

```
sage: f = words.FibonacciWord()
sage: for pp in f.palindrome_prefixes_iterator(max_length=20): pp
word:
word: 0
word: 010
word: 010010
word: 01001010010
word: 0100101001001010010
```

parent()

Returns the parent of self.

TESTS:

```
sage: Word(iter([1,2,3]), length="unknown").parent()
Finite words over Set of Python objects of type 'object'
sage: Word(range(12)).parent()
Finite words over Set of Python objects of type 'object'
sage: Word(range(4), alphabet=range(6)).parent()
Finite words over {0, 1, 2, 3, 4, 5}
sage: Word(iter('abac'), alphabet='abc').parent()
Finite words over {'a', 'b', 'c'}
```

partial_sums(*start*, *mod=None*)

Returns the word defined by the partial sums of its prefixes.

INPUT:

- `self` - A word over the integers.
- `start` - integer, the first letter of the resulting word.
- **mod - (default: None) It can be one of the following:**
  - None or 0 : result is over the integers
  - integer : result is over the integers modulo `mod`.

EXAMPLES:

```
sage: w = Word(range(10))
sage: w.partial_sums(0)
word: 0,0,1,3,6,10,15,21,28,36,45
sage: w.partial_sums(1)
word: 1,1,2,4,7,11,16,22,29,37,46
```

```
sage: w = Word([1,2,3,1,2,3,2,2,2,2])
sage: w.partial_sums(0, mod=None)
word: 0,1,3,6,7,9,12,14,16,18,20
sage: w.partial_sums(0, mod=0)
word: 0,1,3,6,7,9,12,14,16,18,20
sage: w.partial_sums(0, mod=8)
word: 01367146024
sage: w.partial_sums(0, mod=4)
word: 01323102020
sage: w.partial_sums(0, mod=2)
word: 01101100000
sage: w.partial_sums(0, mod=1)
word: 00000000000
```

TESTS:

If the word is infinite, so is the result:

```
sage: w = Word(lambda n:1)
sage: u = w.partial_sums(0)
sage: type(u)
<class 'sage.combinat.words.word.InfiniteWord_iter_with_caching'>
```

prefixes_iterator(*max_length=None*)

Returns an iterator over the prefixes of self.

INPUT:

- `max_length` - non negative integer or None (optional, default: None) the maximum length of the prefixes

OUTPUT:

iterator

EXAMPLES:

```
sage: w = Word('abaaba')
sage: for p in w.prefixes_iterator(): p
word:
word: a
word: ab
word: aba
word: abaa
word: abaab
word: abaaba
sage: for p in w.prefixes_iterator(max_length=3): p
word:
word: a
word: ab
word: aba
```

You can iterate over the prefixes of an infinite word:

```
sage: f = words.FibonacciWord()
sage: for p in f.prefixes_iterator(max_length=8): p
word:
word: 0
word: 01
word: 010
word: 0100
```

```
word: 01001
word: 010010
word: 0100101
word: 01001010
```

TESTS:

```
sage: list(f.prefixes_iterator(max_length=0))
[word: ]
```

return_words_iterator (*fact*)

Returns an iterator over all the return words of fact in self (without unicity).

INPUT:

- `fact` - a non empty finite word

OUTPUT:

iterator

EXAMPLES:

```
sage: w = Word('baccabccbacbca')
sage: b = Word('b')
sage: list(w.return_words_iterator(b))
[word: bacca, word: bcc, word: bac]

sage: TM = words.ThueMorseWord()
sage: fact = Word([0,1,1,0,1])
sage: it = TM.return_words_iterator(fact)
sage: next(it)
word: 011010011001
sage: next(it)
word: 011010010110
sage: next(it)
word: 0110100110010110
sage: next(it)
word: 01101001
sage: next(it)
word: 011010011001
sage: next(it)
word: 011010010110
```

string_rep()

Returns the (truncated) raw sequence of letters as a string.

EXAMPLES:

```
sage: Word('abbabaab').string_rep()
'abbabaab'
sage: Word([0, 1, 0, 0, 1]).string_rep()
'01001'
sage: Word([0,1,10,101]).string_rep()
'0,1,10,101'
sage: WordOptions(letter_separator='-')
sage: Word([0,1,10,101]).string_rep()
'0-1-10-101'
sage: WordOptions(letter_separator=',')
```

TESTS:

Insertion in a str:

```
sage: from itertools import count
sage: w = Word((i % 5 for i in count()), length='unknown')
sage: "w = %s in this string." % w
'w = 012340123401234012340123401234012340123401234... in this string.'
```

Using LatexExpr:

```
sage: from sage.misc.latex import LatexExpr
sage: LatexExpr(w)
012340123401234012340123401234012340123401234...
```

With the print statement:

```
sage: print w
012340123401234012340123401234012340123401234...
```

Truncation is done for possibily infinite words:

```
sage: print w
012340123401234012340123401234012340123401234...
```

sum_digits(*base=2*, *mod=None*)

Return the sequence of the sum modulo `mod` of the digits written in base `base` of `self`.

INPUT:

- `self` - word over natural numbers
- `base` - integer (default : 2), greater or equal to 2
- `mod` - modulo (default: `None`), can take the following values:
  - integer - the modulo
  - `None` - the value `base` is considered for the modulo.

EXAMPLES:

The Thue-Morse word:

```
sage: from itertools import count
sage: Word(count()).sum_digits()
word: 01101001100101101001011001101001100101 10...
```

Sum of digits modulo 2 of the prime numbers written in base 2:

```
sage: Word(primes(1000)).sum_digits()
word: 10011101001110101110110010111011100110 11...
```

Sum of digits modulo 3 of the prime numbers written in base 3:

```
sage: Word(primes(1000)).sum_digits(base=3)
word: 21000020200022212221210222210221221110 22...
sage: Word(primes(1000)).sum_digits(base=3, mod=3)
word: 21000020200022212221210222210221221110 22...
```

Sum of digits modulo 2 of the prime numbers written in base 3:

```
sage: Word(primes(1000)).sum_digits(base=3, mod=2)
word: 01111111111111111111111111111111111111 11...
```

Sum of digits modulo 7 of the prime numbers written in base 10:

```
sage: Word(primes(1000)).sum_digits(base=10, mod=7)
word: 23502413544350410061324322413535460063 04...
```

Negative entries:

```
sage: w = Word([-1,0,1,2,3,4,5])
sage: w.sum_digits()
Traceback (most recent call last):
...
NotImplementedError: nth digit of Thue-Morse word is not implemented for negative value of n
```

TESTS:

The Thue-Morse word:

```
sage: from itertools import count
sage: w = Word(count()).sum_digits()
sage: t = words.ThueMorseWord()
sage: w[:100]  == t[:100]
True

sage: type(Word(range(10)).sum_digits())
<class 'sage.combinat.words.word.FiniteWord_iter_with_caching'>
```

to_integer_word()

Returns a word over the integers whose letters are those output by self._to_integer_iterator()

EXAMPLES:

```
sage: from itertools import count
sage: w = Word(count()); w
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
sage: w.to_integer_word()
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
sage: w = Word(iter("abbacabba"), length="finite"); w
word: abbacabba
sage: w.to_integer_word()
word: 011020110
sage: w = Word(iter("abbacabba"), length="unknown"); w
word: abbacabba
sage: w.to_integer_word()
word: 011020110
```

5.1.320 Word features that are imported by default in the interpreter namespace

5.1.321 Alphabet

AUTHORS:

- Franco Saliola (2008-12-17) : merged into sage
- Vincent Delecroix and Stepan Starosta (2012): remove classes for alphabet and use other Sage classes otherwise (TotallyOrderFiniteSet, FiniteEnumeratedSet, ...). More shortcut to standard alphabets.

EXAMPLES:

```
sage: build_alphabet("ab")
{'a', 'b'}
sage: build_alphabet([0,1,2])
```

```
{0, 1, 2}
sage: build_alphabet(name="PP")
Positive integers
sage: build_alphabet(name="NN")
Non negative integers
sage: build_alphabet(name="lower")
{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
```

sage.combinat.words.alphabet.**Alphabet**(*data=None*, *names=None*, *name=None*)

Return an object representing an ordered alphabet.

INPUT:

- data – the letters of the alphabet; it can be:
  - a list/tuple/iterable of letters; the iterable may be infinite
  - an integer n to represent $\{1, \ldots, n\}$, or infinity to represent $\mathbf{N}$
- names – (optional) a list for the letters (i.e. variable names) or a string for prefix for all letters; if given a list, it must have the same cardinality as the set represented by data
- name – (optional) if given, then return a named set and can be equal to : 'lower', 'upper', 'space', 'underscore', 'punctuation', 'printable', 'binary', 'octal', 'decimal', 'hexadecimal', 'radix64'.

  You can use many of them at once, separated by spaces : 'lower punctuation' represents the union of the two alphabets 'lower' and 'punctuation'.

  Alternatively, name can be set to "positive integers" (or "PP") or "natural numbers" (or "NN").

  name cannot be combined with data.

EXAMPLES:

If the argument is a Set, it just returns it:

```
sage: build_alphabet(ZZ) is ZZ
True
sage: F = FiniteEnumeratedSet('abc')
sage: build_alphabet(F) is F
True
```

If a list, tuple or string is provided, then it builds a proper Sage class (TotallyOrderedFiniteSet):

```
sage: build_alphabet([0,1,2])
{0, 1, 2}
sage: F = build_alphabet('abc'); F
{'a', 'b', 'c'}
sage: print type(F).__name__
TotallyOrderedFiniteSet_with_category
```

If an integer and a set is given, then it constructs a TotallyOrderedFiniteSet:

```
sage: build_alphabet(3, ['a','b','c'])
{'a', 'b', 'c'}
```

If an integer and a string is given, then it considers that string as a prefix:

```
sage: build_alphabet(3, 'x')
{'x0', 'x1', 'x2'}
```

If no data is provided, `name` may be a string which describe an alphabet. The available names decompose into two families. The first one are 'positive integers', 'PP', 'natural numbers' or 'NN' which refer to standard set of numbers:

```
sage: build_alphabet(name="positive integers")
Positive integers
sage: build_alphabet(name="PP")
Positive integers
sage: build_alphabet(name="natural numbers")
Non negative integers
sage: build_alphabet(name="NN")
Non negative integers
```

The other families for the option `name` are among 'lower', 'upper', 'space', 'underscore', 'punctuation', 'printable', 'binary', 'octal', 'decimal', 'hexadecimal', 'radix64' which refer to standard set of charaters. Theses names may be combined by separating them by a space:

```
sage: build_alphabet(name="lower")
{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's',
sage: build_alphabet(name="hexadecimal")
{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'}
sage: build_alphabet(name="decimal punctuation")
{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ' ', ',', '.', ';', ':', '!', '?'}
```

In the case the alphabet is built from a list or a tuple, the order on the alphabet is given by the elements themselves:

```
sage: A = build_alphabet([0,2,1])
sage: A(0) < A(2)
True
sage: A(2) < A(1)
False
```

If a different order is needed, you may use `TotallyOrderedFiniteSet` and set the option `facade` to `False`. That way, the comparison fits the order of the input:

```
sage: A = TotallyOrderedFiniteSet([4,2,6,1], facade=False)
sage: A(4) < A(2)
True
sage: A(1) < A(6)
False
```

Be careful, the element of the set in the last example are no more integers and do not compare equal with integers:

```
sage: type(A.an_element())
<class 'sage.sets.totally_ordered_finite_set.TotallyOrderedFiniteSet_with_category.element_class
sage: A(1) == 1
False
sage: 1 == A(1)
False
```

We give an example of an infinite alphabet indexed by the positive integers and the prime numbers:

```
sage: build_alphabet(oo, 'x')
Lazy family (x(i))_{i in Non negative integers}
sage: build_alphabet(Primes(), 'y')
Lazy family (y(i))_{i in Set of all prime numbers: 2, 3, 5, 7, ...}
```

TESTS:

```
sage: Alphabet(3, name="punctuation")
Traceback (most recent call last):
...
ValueError: name cannot be specified with any other argument
sage: Alphabet(8, ['e']*10)
Traceback (most recent call last):
...
ValueError: invalid value for names
sage: Alphabet(8, x)
Traceback (most recent call last):
...
ValueError: invalid value for names
sage: Alphabet(name=x, names="punctuation")
Traceback (most recent call last):
...
ValueError: name cannot be specified with any other argument
sage: Alphabet(x)
Traceback (most recent call last):
...
ValueError: unable to construct an alphabet from the given parameters
```

class sage.combinat.words.alphabet.**OrderedAlphabet**

Bases: object

Warning: Not to be used! (backward compatibility)

Returns a finite or infinite ordered alphabet.

EXAMPLES:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet
sage: A = OrderedAlphabet('ab'); A
doctest:...: DeprecationWarning: OrderedAlphabet is deprecated; use Alphabet instead.
See http://trac.sagemath.org/8920 for details.
{'a', 'b'}
sage: type(A)
<class 'sage.sets.totally_ordered_finite_set.TotallyOrderedFiniteSet_with_category'>
```

sage.combinat.words.alphabet.**OrderedAlphabet_Finite**

alias of OrderedAlphabet

class sage.combinat.words.alphabet.**OrderedAlphabet_backward_compatibility**(*elements*, *facade=True*)

Bases: sage.sets.totally_ordered_finite_set.TotallyOrderedFiniteSet

Warning: Not to be used! (backward compatibility)

Version prior to trac ticket #8920 uses classes Alphabet with an argument ._alphabet instead of ._elements used in TotallyOrderedFiniteSet. This class is dedicated to handle this problem which occurs when unpickling OrderedAlphabet.

sage.combinat.words.alphabet.**build_alphabet**(*data=None*, *names=None*, *name=None*)

Return an object representing an ordered alphabet.

INPUT:

- data – the letters of the alphabet; it can be:

–a list/tuple/iterable of letters; the iterable may be infinite

–an integer n to represent $\{1, \ldots, n\}$, or infinity to represent $\mathbf{N}$

- `names` – (optional) a list for the letters (i.e. variable names) or a string for prefix for all letters; if given a list, it must have the same cardinality as the set represented by `data`
- `name` – (optional) if given, then return a named set and can be equal to : `'lower'`, `'upper'`, `'space'`, `'underscore'`, `'punctuation'`, `'printable'`, `'binary'`, `'octal'`, `'decimal'`, `'hexadecimal'`, `'radix64'`.

  You can use many of them at once, separated by spaces : `'lower punctuation'` represents the union of the two alphabets `'lower'` and `'punctuation'`.

  Alternatively, `name` can be set to `"positive integers"` (or `"PP"`) or `"natural numbers"` (or `"NN"`).

  `name` cannot be combined with `data`.

EXAMPLES:

If the argument is a Set, it just returns it:

```
sage: build_alphabet(ZZ) is ZZ
True
sage: F = FiniteEnumeratedSet('abc')
sage: build_alphabet(F) is F
True
```

If a list, tuple or string is provided, then it builds a proper Sage class (`TotallyOrderedFiniteSet`):

```
sage: build_alphabet([0,1,2])
{0, 1, 2}
sage: F = build_alphabet('abc'); F
{'a', 'b', 'c'}
sage: print type(F).__name__
TotallyOrderedFiniteSet_with_category
```

If an integer and a set is given, then it constructs a `TotallyOrderedFiniteSet`:

```
sage: build_alphabet(3, ['a','b','c'])
{'a', 'b', 'c'}
```

If an integer and a string is given, then it considers that string as a prefix:

```
sage: build_alphabet(3, 'x')
{'x0', 'x1', 'x2'}
```

If no data is provided, `name` may be a string which describe an alphabet. The available names decompose into two families. The first one are 'positive integers', 'PP', 'natural numbers' or 'NN' which refer to standard set of numbers:

```
sage: build_alphabet(name="positive integers")
Positive integers
sage: build_alphabet(name="PP")
Positive integers
sage: build_alphabet(name="natural numbers")
Non negative integers
sage: build_alphabet(name="NN")
Non negative integers
```

The other families for the option `name` are among 'lower', 'upper', 'space', 'underscore', 'punctuation', 'printable', 'binary', 'octal', 'decimal', 'hexadecimal', 'radix64' which refer to standard set of charaters. Theses

names may be combined by separating them by a space:

```
sage: build_alphabet(name="lower")
{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's',
sage: build_alphabet(name="hexadecimal")
{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'}
sage: build_alphabet(name="decimal punctuation")
{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ' ', ',', '.', ';', ':', '!', '?'}
```

In the case the alphabet is built from a list or a tuple, the order on the alphabet is given by the elements themselves:

```
sage: A = build_alphabet([0,2,1])
sage: A(0) < A(2)
True
sage: A(2) < A(1)
False
```

If a different order is needed, you may use `TotallyOrderedFiniteSet` and set the option `facade` to `False`. That way, the comparison fits the order of the input:

```
sage: A = TotallyOrderedFiniteSet([4,2,6,1], facade=False)
sage: A(4) < A(2)
True
sage: A(1) < A(6)
False
```

Be careful, the element of the set in the last example are no more integers and do not compare equal with integers:

```
sage: type(A.an_element())
<class 'sage.sets.totally_ordered_finite_set.TotallyOrderedFiniteSet_with_category.element_class
sage: A(1) == 1
False
sage: 1 == A(1)
False
```

We give an example of an infinite alphabet indexed by the positive integers and the prime numbers:

```
sage: build_alphabet(oo, 'x')
Lazy family (x(i))_{i in Non negative integers}
sage: build_alphabet(Primes(), 'y')
Lazy family (y(i))_{i in Set of all prime numbers: 2, 3, 5, 7, ...}
```

TESTS:

```
sage: Alphabet(3, name="punctuation")
Traceback (most recent call last):
...
ValueError: name cannot be specified with any other argument
sage: Alphabet(8, ['e']*10)
Traceback (most recent call last):
...
ValueError: invalid value for names
sage: Alphabet(8, x)
Traceback (most recent call last):
...
ValueError: invalid value for names
sage: Alphabet(name=x, names="punctuation")
Traceback (most recent call last):
...
```

```
ValueError: name cannot be specified with any other argument
sage: Alphabet(x)
Traceback (most recent call last):
...
ValueError: unable to construct an alphabet from the given parameters
```

5.1.322 Finite word

AUTHORS:

- Arnaud Bergeron
- Amy Glen
- Sébastien Labbé
- Franco Saliola
- Julien Leroy (March 2010): reduced_rauzy_graph

EXAMPLES:

Creation of a finite word

Finite words from python strings, lists and tuples:

```
sage: Word("abbabaab")
word: abbabaab
sage: Word([0, 1, 1, 0, 1, 0, 0, 1])
word: 01101001
sage: Word( ('a', 0, 5, 7, 'b', 9, 8) )
word: a057b98
```

Finite words from functions:

```
sage: f = lambda n : n%3
sage: Word(f, length=13)
word: 0120120120120
```

Finite words from iterators:

```
sage: from itertools import count
sage: Word(count(), length=10)
word: 0123456789

sage: Word( iter('abbccdef') )
word: abbccdef
```

Finite words from words via concatenation:

```
sage: u = Word("abcccabba")
sage: v = Word([0, 4, 8, 8, 3])
sage: u * v
word: abcccabba04883
sage: v * u
word: 04883abcccabba
sage: u + v
```

```
word: abcccabba04883
sage: u^3 * v^(8/5)
word: abcccabbaabcccabbaabcccabba04883048
```

Finite words from infinite words:

```
sage: vv = v^Infinity
sage: vv[10000:10015]
word: 048830488304883
```

Finite words in a specific combinatorial class:

```
sage: W = Words("ab")
sage: W
Finite and infinite words over {'a', 'b'}
sage: W("abbabaab")
word: abbabaab
sage: W(["a","b","b","a","b","a","a","b"])
word: abbabaab
sage: W( iter('ababab') )
word: ababab
```

Finite word as the image under a morphism:

```
sage: m = WordMorphism({0:[4,4,5,0],5:[0,5,5],4:[4,0,0,0]})
sage: m(0)
word: 4450
sage: m(0, order=2)
word: 400040000554450
sage: m(0, order=3)
word: 40004450445044504000445044504450445044500550...
```

Functions and algorithms

There are more than 100 functions defined on a finite word. Here are some of them:

```
sage: w = Word('abaabbba'); w
word: abaabbba
sage: w.is_palindrome()
False
sage: w.is_lyndon()
False
sage: w.number_of_factors()
28
sage: w.critical_exponent()
3
```

```
sage: print w.lyndon_factorization()
(ab, aabbb, a)
sage: print w.crochemore_factorization()
(a, b, a, ab, bb, a)
```

```
sage: st = w.suffix_tree()
sage: st
Implicit Suffix Tree of the word: abaabbba
sage: st.show(word_labels=True)
```

```
sage: T = words.FibonacciWord('ab')
sage: T.longest_common_prefix(Word('abaababababbbbbb'))
word: abaababa
```

As matrix and many other sage objects, words have a parent:

```
sage: u = Word('xyxxyxyyy')
sage: u.parent()
Finite words over Set of Python objects of type 'object'

sage: v = Word('xyxxyxyyy', alphabet='xy')
sage: v.parent()
Finite words over {'x', 'y'}
```

Factors and Rauzy Graphs

Enumeration of factors, the successive values returned by next(it) can appear in a different order depending on hardware. Therefore we mark the three first results of the test random. The important test is that the iteration stops properly on the fourth call:

```
sage: w = Word([4,5,6])^7
sage: it = w.factor_iterator(4)
sage: next(it) # random
word: 6456
sage: next(it) # random
word: 5645
sage: next(it) # random
word: 4564
sage: next(it)
Traceback (most recent call last):
...
StopIteration
```

The set of factors:

```
sage: sorted(w.factor_set(3))
[word: 456, word: 564, word: 645]
sage: sorted(w.factor_set(4))
[word: 4564, word: 5645, word: 6456]
sage: w.factor_set().cardinality()
61
```

Rauzy graphs:

```
sage: f = words.FibonacciWord()[:30]
sage: f.rauzy_graph(4)
Looped digraph on 5 vertices
sage: f.reduced_rauzy_graph(4)
Looped multi-digraph on 2 vertices
```

Left-special and bispecial factors:

```
sage: f.number_of_left_special_factors(7)
1
sage: f.bispecial_factors()
[word: , word: 0, word: 010, word: 010010, word: 01001010010]
```

class sage.combinat.words.finite_word.**CallableFromListOfWords**
Bases: tuple

A class to create a callable from a list of words. The concatenation of a list of words is obtained by creating a word from this callable.

class sage.combinat.words.finite_word.**Factorization**
Bases: list

A list subclass having a nicer representation for factorization of words.

TESTS:

```
sage: f = sage.combinat.words.finite_word.Factorization()
sage: f == loads(dumps(f))
True
```

class sage.combinat.words.finite_word.**FiniteWord_class**
Bases: sage.combinat.words.abstract_word.Word_class

BWT()
Returns the Burrows-Wheeler Transform (BWT) of self.

The *Burrows-Wheeler transform* of a finite word w is obtained from w by first listing the conjugates of w in lexicographic order and then concatenating the final letters of the conjugates in this order. See [1].

EXAMPLES:

```
sage: Word('abaccaaba').BWT()
word: cbaabaaca
sage: Word('abaab').BWT()
word: bbaaa
sage: Word('bbabbaca').BWT()
word: cbbbbaaa
sage: Word('aabaab').BWT()
word: bbaaaa
sage: Word().BWT()
word:
sage: Word('a').BWT()
word: a
```

REFERENCES:

- [1] M. Burrows, D.J. Wheeler, "A block-sorting lossless data compression algorithm", HP Lab Technical Report, 1994, available at http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html

abelian_complexity(*n*)
Return the number of abelian vectors of factors of length n of self.

EXAMPLES:

```
sage: w = words.FibonacciWord()[:100]
sage: [w.abelian_complexity(i) for i in range(20)]
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]

sage: w = words.ThueMorseWord()[:100]
sage: [w.abelian_complexity(i) for i in range(20)]
[1, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2]
```

abelian_vector(*alphabet=None*)
Return the abelian vector of self counting the occurrences of each letter.

The vector is defined w.r.t the order of the alphabet of the parent. See also evaluation_dict().

INPUT:

- `self` – word having a parent on a finite alphabet
- `alphabet` – DEPRECATED

OUTPUT:

list

EXAMPLES:

```
sage: W = Words('ab')
sage: W('aaabbbbb').abelian_vector()
[3, 5]
sage: W('a').abelian_vector()
[1, 0]
sage: W().abelian_vector()
[0, 0]
```

The argument alphabet is deprecated:

```
sage: Word('aabaa').abelian_vector('abc')
doctest:...: DeprecationWarning: The argument alphabet of
methods abelian_vector and parikh_vector is deprecated and will
be removed in a future version of Sage. In order to fix this,
you must define your word on a parent with a finite alphabet.
See http://trac.sagemath.org/17058 for details.
[4, 1, 0]
```

You may fix the above deprecated use of alphabet argument this way:

```
sage: W = Words('abc')
sage: W('aabaa').abelian_vector()
[4, 1, 0]
```

TESTS:

```
sage: W = Words()
sage: W('aabaa').abelian_vector()
Traceback (most recent call last):
...
TypeError: The alphabet of the parent is infinite; define the
word with a parent on a finite alphabet or use
evaluation_dict() instead
```

abelian_vectors(*n*)

Return the abelian vectors of factors of length `n` of self.

The vectors are defined w.r.t the order of the alphabet of the parent.

OUTPUT:

Set of tuples

EXAMPLES:

```
sage: W = Words([0,1,2])
sage: w = W([0,1,1,0,1,2,0,2,0,2])
sage: w.abelian_vectors(3)
{(1, 0, 2), (1, 1, 1), (1, 2, 0), (2, 0, 1)}
sage: w[:5].abelian_vectors(3)
{(1, 2, 0)}
```

```
sage: w[5:].abelian_vectors(3)
{(1, 0, 2), (2, 0, 1)}

sage: w = words.FibonacciWord()[:100]
sage: sorted(w.abelian_vectors(0))
[(0, 0)]
sage: sorted(w.abelian_vectors(1))
[(0, 1), (1, 0)]
sage: sorted(w.abelian_vectors(7))
[(4, 3), (5, 2)]
```

The word must be defined with a parent on a finite alphabet:

```
sage: from itertools import count
sage: w = Word(count(), alphabet=NN)
sage: w[:2].abelian_vectors(2)
Traceback (most recent call last):
...
TypeError: The alphabet of the parent is infinite; define the
word with a parent on a finite alphabet
```

TESTS:

```
sage: W = Words([0, 1])
sage: w = W([0,0,0])
sage: sorted(w.abelian_vectors(3))
[(3, 0)]
sage: w = W([0,0,0,1])
sage: sorted(w.abelian_vectors(3))
[(2, 1), (3, 0)]
sage: w = W([0,0,0,1,1])
sage: sorted(w.abelian_vectors(3))
[(1, 2), (2, 1), (3, 0)]
sage: w = W([0,0,0,1,1,1])
sage: sorted(w.abelian_vectors(3))
[(0, 3), (1, 2), (2, 1), (3, 0)]

sage: w = Word([0,1,0], alphabet=[0,1])
sage: w.abelian_complexity(3)
1
sage: w.abelian_complexity(4)
0
```

apply_permutation_to_letters(*permutation*)

Return the word obtained by applying the permutation `permutation` of the alphabet of `self` to each letter of `self`.

EXAMPLES:

```
sage: w = Words('abcd')('abcd')
sage: p = [2,1,4,3]
sage: w.apply_permutation_to_letters(p)
word: badc
sage: u = Words('dabc')('abcd')
sage: u.apply_permutation_to_letters(p)
word: dcba
sage: w.apply_permutation_to_letters(Permutation(p))
word: badc
sage: w.apply_permutation_to_letters(PermutationGroupElement(p))
word: badc
```

apply_permutation_to_positions(*permutation*)

Return the word obtained by permuting the positions of the letters in `self` according to the permutation `permutation`.

EXAMPLES:

```
sage: w = Words('abcd')('abcd')
sage: w.apply_permutation_to_positions([2,1,4,3])
word: badc
sage: u = Words('dabc')('abcd')
sage: u.apply_permutation_to_positions([2,1,4,3])
word: badc
sage: w.apply_permutation_to_positions(Permutation([2,1,4,3]))
word: badc
sage: w.apply_permutation_to_positions(PermutationGroupElement([2,1,4,3]))
word: badc
sage: Word([1,2,3,4]).apply_permutation_to_positions([3,4,2,1])
word: 3421
```

balance()

Returns the balance of self.

The balance of a word is the smallest number q such that self is q-balanced [1].

A finite or infinite word w is said to be *q-balanced* if for any two factors u, v of w of the same length, the difference between the number of x's in each of u and v is at most q for all letters x in the alphabet of w. A 1-balanced word is simply said to be balanced. See Chapter 2 of [2].

OUTPUT:

integer

EXAMPLES:

```
sage: Word('1111111').balance()
0
sage: Word('001010101011').balance()
2
sage: Word('0101010101').balance()
1

sage: w = Word('11112222')
sage: w.is_balanced(2)
False
sage: w.is_balanced(3)
False
sage: w.is_balanced(4)
True
sage: w.is_balanced(5)
True
sage: w.balance()
4
```

TESTS:

```
sage: Word('1111122222').balance()
5
sage: Word('').balance()
0
sage: Word('1').balance()
```

```
0
sage: Word('12').balance()
1
sage: Word('1112').balance()
1
```

REFERENCES:

- [1] I. Fagnot, L. Vuillon, Generalized balances in Sturmian words, Discrete Applied Mathematics 121 (2002), 83–101.
- [2] M. Lothaire, Algebraic Combinatorics On Words, vol. 90 of Encyclopedia of Mathematics and its Applications, Cambridge University Press, U.K., 2002.

bispecial_factors(*n=None*)

Returns the bispecial factors (of length n).

A factor u of a word w is *bispecial* if it is right special and left special.

INPUT:

- n - integer (optional, default: None). If None, it returns all bispecial factors.

OUTPUT:

A list of words.

EXAMPLES:

```
sage: w = words.FibonacciWord()[:30]
sage: w.bispecial_factors()
[word: , word: 0, word: 010, word: 010010, word: 01001010010]

sage: w = words.ThueMorseWord()[:30]
sage: for i in range(10): print i, sorted(w.bispecial_factors(i))
0 [word: ]
1 [word: 0, word: 1]
2 [word: 01, word: 10]
3 [word: 010, word: 101]
4 [word: 0110, word: 1001]
5 []
6 [word: 011001, word: 100110]
7 []
8 [word: 10010110]
9 []
```

bispecial_factors_iterator(*n=None*)

Returns an iterator over the bispecial factors (of length n).

A factor u of a word w is *bispecial* if it is right special and left special.

INPUT:

- n - integer (optional, default: None). If None, it returns an iterator over all bispecial factors.

EXAMPLES:

```
sage: w = words.ThueMorseWord()[:30]
sage: for i in range(10):
...       for u in sorted(w.bispecial_factors_iterator(i)):
...           print i,u
0
1 0
```

```
1 1
2 01
2 10
3 010
3 101
4 0110
4 1001
6 011001
6 100110
8 10010110

sage: key = lambda u : (len(u), u)
sage: for u in sorted(w.bispecial_factors_iterator(), key=key): u
word:
word: 0
word: 1
word: 01
word: 10
word: 010
word: 101
word: 0110
word: 1001
word: 011001
word: 100110
word: 10010110
```

border()

Returns the longest word that is both a proper prefix and a proper suffix of self.

EXAMPLES:

```
sage: Word('121212').border()
word: 1212
sage: Word('12321').border()
word: 1
sage: Word().border() is None
True
```

charge(*check=True*)

Returns the charge of self. This is defined as follows.

If w is a permutation of length n, (in other words, the evaluation of w is $(1, 1, \ldots, 1)$), the statistic charge(w) is given by $\sum_{i=1}^{n} c_i(w)$ where $c_1(w) = 0$ and $c_i(w)$ is defined recursively by setting p_i equal to 1 if i appears to the right of $i-1$ in w and 0 otherwise. Then we set $c_i(w) = c_{i-1}(w) + p_i$.

EXAMPLES:

```
sage: Word([1, 2, 3]).charge()
3
sage: Word([3, 5, 1, 4, 2]).charge() == 0 + 1 + 1 + 2 + 2
True
```

If w is not a permutation, but the evaluation of w is a partition, the charge of w is defined to be the sum of its charge subwords (each of which will be a permutation). The first charge subword is found by starting at the end of w and moving left until the first 1 is found. This is marked, and we continue to move to the left until the first 2 is found, wrapping around from the beginning of the word back to the end, if necessary. We mark this 2, and continue on until we have marked the largest letter in w. The marked letters, with relative order preserved, form the first charge subword of w. This subword is removed, and the next charge subword is found in the same manner from the remaining letters. In the following example, $w1, w2, w3$

are the charge subwords of w.

EXAMPLE:

```
sage: w = Word([5,2,3,4,4,1,1,1,2,2,3])
sage: w1 = Word([5, 2, 4, 1, 3])
sage: w2 = Word([3, 4, 1, 2])
sage: w3 = Word([1, 2])
sage: w.charge() == w1.charge() + w2.charge() + w3.charge()
True
```

Finally, if w does not have partition content, we apply the Lascoux-Schutzenberger standardization operators s_i in such a manner as to obtain a word with partition content. (The word we obtain is independent of the choice of operators.) The charge is then defined to be the charge of this word:

```
sage: Word([3,3,2,1,1]).charge()
0
sage: Word([1,2,3,1,2]).charge()
2
```

Note that this differs from the definition of charge given in Macdonald's book. The difference amounts to a choice of reading a word from left-to-right or right-to-left. The choice in Sage was made to agree with the definition of a reading word of a tableau in Sage, and seems to be the more common convention in the literature.

REFERENCES:

[1] Ian Macdonald, *Symmetric Functions and Hall Polynomials* second edition, 1995, Oxford University Press

[2] A. Lascoux, L. Lapointe, and J. Morse. *Tableau atoms and a new Macdonald positivity conjecture.* Duke Math Journal, **116 (1)**, 2003. Available at: [http://arxiv.org/abs/math/0008073]

[3] A. Lascoux, B. Leclerc, and J.Y. Thibon. *The Plactic Monoid.* Survey article available at [http://www-igm.univ-mlv.fr/~jyt/ARTICLES/plactic.ps]

TESTS:

```
sage: Word([1,1,2,2,3]).charge()
4
sage: Word([3,1,1,2,2]).charge()
3
sage: Word([2,1,1,2,3]).charge()
2
sage: Word([2,1,1,3,2]).charge()
2
sage: Word([3,2,1,1,2]).charge()
1
sage: Word([2,2,1,1,3]).charge()
1
sage: Word([3,2,2,1,1]).charge()
0
sage: Word([]).charge()
0
```

cocharge()

Returns the cocharge of self. For a word w, this can be defined as $n_{ev} - ch(w)$, where $ch(w)$ is the charge of w and ev is the evaluation of w, and n_{ev} is $\sum_{i<j} min(ev_i, ev_j)$.

EXAMPLES:

```
sage: Word([1,2,3]).cocharge()
0
sage: Word([3,2,1]).cocharge()
3
sage: Word([1,1,2]).cocharge()
0
sage: Word([2,1,2]).cocharge()
1
```

TESTS:

```
sage: Word([]).cocharge()
0
```

coerce(*other*)

Tries to return a pair of words with a common parent; raises an exception if this is not possible.

This function begins by checking if both words have the same parent. If this is the case, then no work is done and both words are returned as-is.

Otherwise it will attempt to convert other to the domain of self. If that fails, it will attempt to convert self to the domain of other. If both attempts fail, it raises a TypeError to signal failure.

EXAMPLES:

```
sage: W1 = Words('abc'); W2 = Words('ab')
sage: w1 = W1('abc'); w2 = W2('abba'); w3 = W1('baab')
sage: w1.parent() is w2.parent()
False
sage: a, b = w1.coerce(w2)
sage: a.parent() is b.parent()
True
sage: w1.parent() is w2.parent()
False
```

colored_vector(*x=0*, *y=0*, *width='default'*, *height=1*, *cmap='hsv'*, *thickness=1*, *label=None*)

Returns a vector (Graphics object) illustrating self. Each letter is represented by a coloured rectangle.

If the parent of self is a class of words over a finite alphabet, then each letter in the alphabet is assigned a unique colour, and this colour will be the same every time this method is called. This is especially useful when plotting and comparing words defined on the same alphabet.

If the alphabet is infinite, then the letters appearing in the word are used as the alphabet.

INPUT:

- `x` - (default: 0) bottom left x-coordinate of the vector
- `y` - (default: 0) bottom left y-coordinate of the vector
- `width` - (default: 'default') width of the vector. By default, the width is the length of self.
- `height` - (default: 1) height of the vector
- `thickness` - (default: 1) thickness of the contour
- **`cmap` - (default: 'hsv') color map; for available color map names** type: `import matplotlib.cm; matplotlib.cm.datad.keys()`
- `label` - str (default: None) a label to add on the colored vector.

OUTPUT:

Graphics

EXAMPLES:

```
sage: Word(range(20)).colored_vector()
Graphics object consisting of 21 graphics primitives
sage: Word(range(100)).colored_vector(0,0,10,1)
Graphics object consisting of 101 graphics primitives
sage: Words(range(100))(range(10)).colored_vector()
Graphics object consisting of 11 graphics primitives
sage: w = Word('abbabaab')
sage: w.colored_vector()
Graphics object consisting of 9 graphics primitives
sage: w.colored_vector(cmap='autumn')
Graphics object consisting of 9 graphics primitives
sage: Word(range(20)).colored_vector(label='Rainbow')
Graphics object consisting of 23 graphics primitives
```

When two words are defined under the same parent, same letters are mapped to same colors:

```
sage: W = Words(range(20))
sage: w = W(range(20))
sage: y = W(range(10,20))
sage: y.colored_vector(y=1, x=10) + w.colored_vector()
Graphics object consisting of 32 graphics primitives
```

TESTS:

The empty word:

```
sage: Word().colored_vector()
Graphics object consisting of 1 graphics primitive
sage: Word().colored_vector(label='empty')
Graphics object consisting of 3 graphics primitives
```

Unknown cmap:

```
sage: Word(range(100)).colored_vector(cmap='jolies')
Traceback (most recent call last):
...
RuntimeError: Color map jolies not known
sage: Word(range(100)).colored_vector(cmap='__doc__')
Traceback (most recent call last):
...
RuntimeError: Color map __doc__ not known
```

commutes_with(*other*)

Returns True if self commutes with other, and False otherwise.

EXAMPLES:

```
sage: Word('12').commutes_with(Word('12'))
True
sage: Word('12').commutes_with(Word('11'))
False
sage: Word().commutes_with(Word('21'))
True
```

complete_return_words(*fact*)

Returns the set of complete return words of fact in self.

This is the set of all factors starting by the given factor and ending just after the next occurrence of this factor. See for instance [1].

INPUT:

- `fact` - a non empty finite word

OUTPUT:

Python set of finite words

EXAMPLES:

```
sage: s = Word('21331233213231').complete_return_words(Word('2'))
sage: sorted(s)
[word: 2132, word: 213312, word: 2332]
sage: Word('').complete_return_words(Word('213'))
set()
sage: Word('121212').complete_return_words(Word('1212'))
{word: 121212}
```

REFERENCES:

- [1] J. Justin, L. Vuillon, Return words in Sturmian and episturmian words, Theor. Inform. Appl. 34 (2000) 343–356.

concatenate(*other*)

Returns the concatenation of self and other.

INPUT:

- `other` - a word over the same alphabet as self

EXAMPLES:

Concatenation may be made using + or * operations:

```
sage: w = Word('abadafd')
sage: y = Word([5,3,5,8,7])
sage: w * y
word: abadafd53587
sage: w + y
word: abadafd53587
sage: w.concatenate(y)
word: abadafd53587
```

Both words must be defined over the same alphabet:

```
sage: z = Word('12223', alphabet = '123')
sage: z + y
Traceback (most recent call last):
...
ValueError: 5 not in alphabet!
```

Eventually, it should work:

```
sage: z = Word('12223', alphabet = '123')
sage: z + y                    #todo: not implemented
word: 1222353587
```

TESTS:

The empty word is not considered by concatenation:

```
sage: type(Word([]) * Word('abcd'))
<class 'sage.combinat.words.word.FiniteWord_str'>
sage: type(Word('abcd') * Word())
<class 'sage.combinat.words.word.FiniteWord_str'>
```

```
sage: type(Word('abcd') * Word([]))
<class 'sage.combinat.words.word.FiniteWord_str'>
sage: type(Word('abcd') * Word(()))
<class 'sage.combinat.words.word.FiniteWord_str'>
sage: type(Word([1,2,3]) * Word(''))
<class 'sage.combinat.words.word.FiniteWord_list'>
```

Concatenation of finite words with infinite words works as expected:

```
sage: from itertools import repeat
sage: W = Words('ab')
sage: w1 = W('aba')
sage: w2 = W(repeat('b'), length='infinite')
sage: w1*w2
word: abaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb...
sage: _.parent()
Infinite words over {'a', 'b'}
```

conjugate(*pos*)

Returns the conjugate at pos of self.

pos can be any integer, the distance used is the modulo by the length of self.

EXAMPLES:

```
sage: Word('12112').conjugate(1)
word: 21121
sage: Word().conjugate(2)
word:
sage: Word('12112').conjugate(8)
word: 12121
sage: Word('12112').conjugate(-1)
word: 21211
```

conjugate_position(*other*)

Returns the position where self is conjugate with other. Returns None if there is no such position.

EXAMPLES:

```
sage: Word('12113').conjugate_position(Word('31211'))
1
sage: Word('12131').conjugate_position(Word('12113')) is None
True
sage: Word().conjugate_position(Word('123')) is None
True
```

TESTS:

We check that trac ticket #11128 is fixed:

```
sage: w = Word([0,0,1,0,2,1])
sage: [w.conjugate(i).conjugate_position(w) for i in range(w.length())]
[0, 1, 2, 3, 4, 5]
```

conjugates()

Returns the list of unique conjugates of self.

EXAMPLES:

```
sage: Word(range(6)).conjugates()
[word: 012345,
 word: 123450,
```

```
 word: 234501,
 word: 345012,
 word: 450123,
 word: 501234]
sage: Word('cbbca').conjugates()
[word: cbbca, word: bbcac, word: bcacb, word: cacbb, word: acbbc]
```

The result contains each conjugate only once:

```
sage: Word('abcabc').conjugates()
[word: abcabc, word: bcabca, word: cabcab]
```

TESTS:

```
sage: Word().conjugates()
[word: ]
sage: Word('a').conjugates()
[word: a]
```

conjugates_iterator()

Returns an iterator over the conjugates of self.

EXAMPLES:

```
sage: it = Word(range(4)).conjugates_iterator()
sage: for w in it: w
word: 0123
word: 1230
word: 2301
word: 3012
```

count(*letter*)

Counts the number of occurrences of letter in self.

EXAMPLES:

```
sage: Word('abbabaab').count('a')
4
```

critical_exponent()

Returns the critical exponent of self.

The *critical exponent* of a word is the supremum of the order of all its (finite) factors. See [1].

Note: The implementation here uses the suffix tree to enumerate all the factors. It should be improved.

EXAMPLES:

```
sage: Word('aaba').critical_exponent()
2
sage: Word('aabaa').critical_exponent()
2
sage: Word('aabaaba').critical_exponent()
7/3
sage: Word('ab').critical_exponent()
1
sage: Word('aba').critical_exponent()
3/2
sage: words.ThueMorseWord()[:20].critical_exponent()
2
```

REFERENCES:

- [1] F. Dejean. Sur un théorème de Thue. J. Combinatorial Theory Ser. A 13:90–99, 1972.

crochemore_factorization()

Returns the Crochemore factorization of self as an ordered list of factors.

The *Crochemore factorization* of a finite word w is the unique factorization: $(x_1, x_2, \ldots, x_n)$ of w with each x_i satisfying either: C1. x_i is a letter that does not appear in $u = x_1 \ldots x_{i-1}$; C2. x_i is the longest prefix of $v = x_i \ldots x_n$ that also has an occurrence beginning within $u = x_1 \ldots x_{i-1}$. See [1].

Note: This is not a very good implementation, and should be improved.

EXAMPLES:

```
sage: x = Word('abababb')
sage: x.crochemore_factorization()
(a, b, abab, b)
sage: mul(x.crochemore_factorization()) == x
True
sage: y = Word('abaababacabba')
sage: y.crochemore_factorization()
(a, b, a, aba, ba, c, ab, ba)
sage: mul(y.crochemore_factorization()) == y
True
sage: x = Word([0,1,0,1,0,1,1])
sage: x.crochemore_factorization()
(0, 1, 0101, 1)
sage: mul(x.crochemore_factorization()) == x
True
```

REFERENCES:

- [1] M. Crochemore, Recherche linéaire d'un carré dans un mot, C. R. Acad. Sci. Paris Sér. I Math. 296 (1983) 14 781–784.

defect(*f=None*)

Returns the defect of self.

The *defect* of a finite word w is given by the difference between the maximum number of possible palindromic factors in a word of length $|w|$ and the actual number of palindromic factors contained in w. It is well known that the maximum number of palindromic factors in w is $|w| + 1$ (see [DJP01]).

An optional involution on letters `f` can be given. In that case, the *f-palindromic defect* (or *pseudopalindromic defect*, or *theta-palindromic defect*) of w is returned. It is a generalization of defect to f-palindromes. More precisely, the defect is $D(w) = |w| + 1 - g_f(w) - |PAL_f(w)|$, where $PAL_f(w)$ denotes the set of f-palindromic factors of w (including the empty word) and $g_f(w)$ is the number of pairs $\{a, f(a)\}$ such that a is a letter, a is not equal to $f(a)$, and a or $f(a)$ occurs in w. In the case of usual palindromes (i.e., for `f` not given or equal to the identity), $g_f(w) = 0$ for all w. See [BHNR04] for usual palindromes and [Sta11] for f-palindromes.

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be callable on letters as well as words (e.g. WordMorphism). The default value corresponds to usual palindromes, i.e., f equal to the identity.

OUTPUT:

integer – If f is None, the palindromic defect of self; otherwise, the f-palindromic defect of self.

EXAMPLES:

```
sage: Word('ara').defect()
0
sage: Word('abcacba').defect()
1
```

It is known that Sturmian words (see [DJP01]) have zero defect:

```
sage: words.FibonacciWord()[:100].defect()
0

sage: sa = WordMorphism('a->ab,b->b')
sage: sb = WordMorphism('a->a,b->ba')
sage: w = (sa*sb*sb*sa*sa*sa*sb).fixed_point('a')
sage: w[:30].defect()
0
sage: w[110:140].defect()
0
```

It is even conjectured that the defect of an aperiodic word which is a fixed point of a primitive morphism is either 0 or infinite (see [BBGL08]):

```
sage: w = words.ThueMorseWord()
sage: w[:50].defect()
12
sage: w[:100].defect()
16
sage: w[:300].defect()
52
```

For generalized defect with an involution different from the identity, there is always a letter which is not a palindrome! This is the reason for the modification of the definition:

```
sage: f = WordMorphism('a->b,b->a')
sage: Word('a').defect(f)
0
sage: Word('ab').defect(f)
0
sage: Word('aa').defect(f)
1
sage: Word('abbabaabbaababba').defect(f)
3

sage: f = WordMorphism('a->b,b->a,c->c')
sage: Word('cabc').defect(f)
0
sage: Word('abcaab').defect(f)
2
```

Other examples:

```
sage: Word('000000000000').defect()
0
sage: Word('011010011001').defect()
2
sage: Word('0101001010001').defect()
0
sage: Word().defect()
0
sage: Word('abbabaabbaababba').defect()
```

```
2
```

REFERENCES:

deg_inv_lex_less(*other*, *weights=None*)

Returns True if the word self is degree inverse lexicographically less than other.

EXAMPLES:

```
sage: Word([1,2,4]).deg_inv_lex_less(Word([1,3,2]))
False
sage: Word([3,2,1]).deg_inv_lex_less(Word([1,2,3]))
True
```

deg_lex_less(*other*, *weights=None*)

Returns True if self is degree lexicographically less than other, and False otherwise. The weight of each letter in the ordered alphabet is given by weights, which defaults to [1, 2, 3, ...].

EXAMPLES:

```
sage: Word([1,2,3]).deg_lex_less(Word([1,3,2]))
True
sage: Word([3,2,1]).deg_lex_less(Word([1,2,3]))
False
sage: W = Words(range(5))
sage: W([1,2,4]).deg_lex_less(W([1,3,2]))
False
sage: Word("abba").deg_lex_less(Word("abbb"), dict(a=1,b=2))
True
sage: Word("abba").deg_lex_less(Word("baba"), dict(a=1,b=2))
True
sage: Word("abba").deg_lex_less(Word("aaba"), dict(a=1,b=2))
False
sage: Word("abba").deg_lex_less(Word("aaba"), dict(a=1,b=0))
True
```

deg_rev_lex_less(*other*, *weights=None*)

Returns True if self is degree reverse lexicographically less than other.

EXAMPLES:

```
sage: Word([3,2,1]).deg_rev_lex_less(Word([1,2,3]))
False
sage: Word([1,2,4]).deg_rev_lex_less(Word([1,3,2]))
False
sage: Word([1,2,3]).deg_rev_lex_less(Word([1,2,4]))
True
```

degree(*weights=None*)

Returns the weighted degree of self, where the weighted degree of each letter in the ordered alphabet is given by weights, which defaults to [1, 2, 3, ...].

INPUT:

- `weights` - a list or tuple, or a dictionary keyed by the letters occurring in self.

EXAMPLES:

```
sage: Word([1,2,3]).degree()
6
sage: Word([3,2,1]).degree()
6
```

```
sage: Words("ab")("abba").degree()
6
sage: Words("ab")("abba").degree([0,2])
4
sage: Words("ab")("abba").degree([-1,-1])
-4
sage: Words("ab")("aabba").degree([1,1])
5
sage: Words([1,2,4])([1,2,4]).degree()
6
sage: Word([1,2,4]).degree()
7
sage: Word("aabba").degree({'a':1,'b':2})
7
sage: Word([0,1,0]).degree({0:17,1:0})
34
```

delta()

Returns the image of self under the delta morphism. This is the word composed of the length of consecutive runs of the same letter in a given word.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('22112122').delta()
word: 22112
sage: W('555008').delta()
word: 321
sage: W().delta()
word:
sage: Word('aabbabaa').delta()
word: 22112
```

delta_derivate(*W=None*)

Returns the derivative under delta for self.

EXAMPLES:

```
sage: W = Words('12')
sage: W('12211').delta_derivate()
word: 22
sage: W('1').delta_derivate(Words([1]))
word: 1
sage: W('2112').delta_derivate()
word: 2
sage: W('2211').delta_derivate()
word: 22
sage: W('112').delta_derivate()
word: 2
sage: W('11222').delta_derivate(Words([1, 2, 3]))
word: 3
```

delta_derivate_left(*W=None*)

Returns the derivative under delta for self.

EXAMPLES:

```
sage: W = Words('12')
sage: W('12211').delta_derivate_left()
word: 22
```

```
sage: W('1').delta_derivate_left(Words([1]))
word: 1
sage: W('2112').delta_derivate_left()
word: 21
sage: W('2211').delta_derivate_left()
word: 22
sage: W('112').delta_derivate_left()
word: 21
sage: W('11222').delta_derivate_left(Words([1, 2, 3]))
word: 3
```

delta_derivate_right(*W=None*)

Returns the right derivative under delta for self.

EXAMPLES:

```
sage: W = Words('12')
sage: W('12211').delta_derivate_right()
word: 122
sage: W('1').delta_derivate_right(Words([1]))
word: 1
sage: W('2112').delta_derivate_right()
word: 12
sage: W('2211').delta_derivate_right()
word: 22
sage: W('112').delta_derivate_right()
word: 2
sage: W('11222').delta_derivate_right(Words([1, 2, 3]))
word: 23
```

delta_inv(*W=None*, *s=None*)

Lifts self via the delta operator to obtain a word containing the letters in alphabet (default is [0, 1]). The letters used in the construction start with s (default is alphabet[0]) and cycle through alphabet.

INPUT:

- `alphabet` - an iterable
- `s` - an object in the iterable

EXAMPLES:

```
sage: W = Words([1, 2])
sage: W([2, 2, 1, 1]).delta_inv()
word: 112212
sage: W([1, 1, 1, 1]).delta_inv(Words('123'))
word: 1231
sage: W([2, 2, 1, 1, 2]).delta_inv(s=2)
word: 22112122
```

evaluation(*alphabet=None*)

Return the abelian vector of self counting the occurrences of each letter.

The vector is defined w.r.t the order of the alphabet of the parent. See also `evaluation_dict()`.

INPUT:

- `self` – word having a parent on a finite alphabet
- `alphabet` – DEPRECATED

OUTPUT:

list

EXAMPLES:

```
sage: W = Words('ab')
sage: W('aaabbbbb').abelian_vector()
[3, 5]
sage: W('a').abelian_vector()
[1, 0]
sage: W().abelian_vector()
[0, 0]
```

The argument alphabet is deprecated:

```
sage: Word('aabaa').abelian_vector('abc')
doctest:...: DeprecationWarning: The argument alphabet of
methods abelian_vector and parikh_vector is deprecated and will
be removed in a future version of Sage. In order to fix this,
you must define your word on a parent with a finite alphabet.
See http://trac.sagemath.org/17058 for details.
[4, 1, 0]
```

You may fix the above deprecated use of alphabet argument this way:

```
sage: W = Words('abc')
sage: W('aabaa').abelian_vector()
[4, 1, 0]
```

TESTS:

```
sage: W = Words()
sage: W('aabaa').abelian_vector()
Traceback (most recent call last):
...
TypeError: The alphabet of the parent is infinite; define the
word with a parent on a finite alphabet or use
evaluation_dict() instead
```

evaluation_dict()

Returns a dictionary keyed by the letters occurring in self with values the number of occurrences of the letter.

EXAMPLES:

```
sage: Word([2,1,4,2,3,4,2]).evaluation_dict()
{1: 1, 2: 3, 3: 1, 4: 2}
sage: Word('badbcdb').evaluation_dict()
{'a': 1, 'b': 3, 'c': 1, 'd': 2}
sage: Word().evaluation_dict()
{}

sage: f = Word('1213121').evaluation_dict() # keys appear in random order
{'1': 4, '2': 2, '3': 1}
```

TESTS:

```
sage: f = Word('1213121').evaluation_dict()
sage: f['1'] == 4
True
sage: f['2'] == 2
True
sage: f['3'] == 1
```

```
True
```

evaluation_partition()

Returns the evaluation of the word w as a partition.

EXAMPLES:

```
sage: Word("acdabda").evaluation_partition()
[3, 2, 1, 1]
sage: Word([2,1,4,2,3,4,2]).evaluation_partition()
[3, 2, 1, 1]
```

evaluation_sparse()

Returns a list representing the evaluation of self. The entries of the list are two-element lists [a, n], where a is a letter occurring in self and n is the number of occurrences of a in self.

EXAMPLES:

```
sage: Word([4,4,2,5,2,1,4,1]).evaluation_sparse()
[(1, 2), (2, 2), (4, 3), (5, 1)]
sage: Word("abcaccab").evaluation_sparse()
[('a', 3), ('c', 3), ('b', 2)]
```

exponent()

Returns the exponent of self.

OUTPUT:

integer – the exponent

EXAMPLES:

```
sage: Word('1231').exponent()
1
sage: Word('121212').exponent()
3
sage: Word().exponent()
0
```

factor_iterator(*n=None*)

Generates distinct factors of `self`.

INPUT:

- `n` - an integer, or `None`.

OUTPUT:

If `n` is an integer, returns an iterator over all distinct factors of length `n`. If `n` is `None`, returns an iterator generating all distinct factors.

EXAMPLES:

```
sage: w = Word('1213121')
sage: sorted( w.factor_iterator(0) )
[word: ]
sage: sorted( w.factor_iterator(10) )
[]
sage: sorted( w.factor_iterator(1) )
[word: 1, word: 2, word: 3]
sage: sorted( w.factor_iterator(4) )
[word: 1213, word: 1312, word: 2131, word: 3121]
```

```
sage: sorted( w.factor_iterator() )
[word: , word: 1, word: 12, word: 121, word: 1213, word: 12131, word: 121312, word: 1213121,

sage: u = Word([1,2,1,2,3])
sage: sorted( u.factor_iterator(0) )
[word: ]
sage: sorted( u.factor_iterator(10) )
[]
sage: sorted( u.factor_iterator(1) )
[word: 1, word: 2, word: 3]
sage: sorted( u.factor_iterator(5) )
[word: 12123]
sage: sorted( u.factor_iterator() )
[word: , word: 1, word: 12, word: 121, word: 1212, word: 12123, word: 123, word: 2, word: 21

sage: xxx = Word("xxx")
sage: sorted( xxx.factor_iterator(0) )
[word: ]
sage: sorted( xxx.factor_iterator(4) )
[]
sage: sorted( xxx.factor_iterator(2) )
[word: xx]
sage: sorted( xxx.factor_iterator() )
[word: , word: x, word: xx, word: xxx]

sage: e = Word()
sage: sorted( e.factor_iterator(0) )
[word: ]
sage: sorted( e.factor_iterator(17) )
[]
sage: sorted( e.factor_iterator() )
[word: ]
```

TESTS:

```
sage: type( Word('cacao').factor_iterator() )
<type 'generator'>
```

factor_occurrences_in(*other*)

Returns an iterator over all occurrences (including overlapping ones) of self in other in their order of appearance.

EXAMPLES:

```
sage: u = Word('121')
sage: w = Word('1212132112l3')
sage: list(u.factor_occurrences_in(w))
[0, 2, 8]
```

factor_set(*n=None*, *algorithm='suffix tree'*)

Returns the set of factors (of length n) of self.

INPUT:

- `n` - an integer or `None` (default: None).
- `algorithm` - string (default: `'suffix tree'`), takes the following values:
  - `'suffix tree'` – construct and use the suffix tree of the word
  - `'naive'` – algorithm uses a sliding window

OUTPUT:

If n is an integer, returns the set of all distinct factors of length n. If n is None, returns the set of all distinct factors.

EXAMPLES:

```
sage: w = Word('121')
sage: sorted(w.factor_set())
[word: , word: 1, word: 12, word: 121, word: 2, word: 21]
sage: sorted(w.factor_set(algorithm='naive'))
[word: , word: 1, word: 12, word: 121, word: 2, word: 21]

sage: w = Word('1213121')
sage: for i in range(w.length()): sorted(w.factor_set(i))
[word: ]
[word: 1, word: 2, word: 3]
[word: 12, word: 13, word: 21, word: 31]
[word: 121, word: 131, word: 213, word: 312]
[word: 1213, word: 1312, word: 2131, word: 3121]
[word: 12131, word: 13121, word: 21312]
[word: 121312, word: 213121]

sage: w = Word([1,2,1,2,3])
sage: s = w.factor_set()
sage: sorted(s)
[word: , word: 1, word: 12, word: 121, word: 1212, word: 12123, word: 123, word: 2, word: 21
```

TESTS:

```
sage: w = Word("xx")
sage: s = w.factor_set()
sage: sorted(s)
[word: , word: x, word: xx]

sage: Set(Word().factor_set())
{word: }

sage: w = Word(range(10), alphabet=range(10))
sage: S1 = w.factor_set(3, algorithm='suffix tree')
sage: S2 = w.factor_set(3, algorithm='naive')
sage: S1 == S2
True
```

find(*sub*, *start=0*, *end=None*)

Returns the index of the first occurrence of sub in self, such that sub is contained within self[start:end]. Returns -1 on failure.

INPUT:

- sub - string, list, tuple or word to search for.
- start - non negative integer (default: 0) specifying the position from which to start the search.
- end - non negative integer (default: None) specifying the position at which the search must stop. If None, then the search is performed up to the end of the string.

OUTPUT:

non negative integer or -1

EXAMPLES:

```
sage: w = Word([0,1,0,0,1])
sage: w.find(Word([1,0]))
1
```

The `sub` argument can also be a tuple or a list:

```
sage: w.find([1,0])
1
sage: w.find((1,0))
1
```

Examples using `start` and `end`:

```
sage: w.find(Word([0,1]), start=1)
3
sage: w.find(Word([0,1]), start=1, end=5)
3
sage: w.find(Word([0,1]), start=1, end=4) == -1
True
sage: w.find(Word([1,1])) == -1
True
sage: w.find("aa")
-1
```

Instances of Word_str handle string inputs as well:

```
sage: w = Word('abac')
sage: w.find('a')
0
sage: w.find('ba')
1
```

TESTS:

Check that trac ticket #12804 is fixed:

```
sage: w = Word(iter("ababab"), length="finite")
sage: w.find("ab")
0
sage: w.find("ab", start=1)
2
sage: w.find("aa")
-1
sage: w.find("abc")
-1
sage: w = Words('ab')(tuple('babaabaaab'))
sage: w.find('abc')
-1
```

first_pos_in(*other*)

Returns the position of the first occurrence of self in other, or None if self is not a factor of other.

EXAMPLES:

```
sage: Word('12').first_pos_in(Word('131231'))
2
sage: Word('32').first_pos_in(Word('131231')) is None
True
```

foata_bijection()

Return word `self` under the Foata bijection.

The Foata bijection ϕ is a bijection on the set of words of given content (by a slight generalization of Section 2 in [FoSc78]). It can be defined by induction on the size of the word: Given a word $w_1 w_2 \cdots w_n$, start with $\phi(w_1) = w_1$. At the i-th step, if $\phi(w_1 w_2 \cdots w_i) = v_1 v_2 \cdots v_i$, we define $\phi(w_1 w_2 \cdots w_i w_{i+1})$ by placing w_{i+1} on the end of the word $v_1 v_2 \cdots v_i$ and breaking the word up into blocks as follows. If $w_{i+1} \geq v_i$, place a vertical line to the right of each v_k for which $w_{i+1} \geq v_k$. Otherwise, if $w_{i+1} < v_i$, place a vertical line to the right of each v_k for which $w_{i+1} < v_k$. In either case, place a vertical line at the start of the word as well. Now, within each block between vertical lines, cyclically shift the entries one place to the right.

For instance, to compute $\phi([4, 1, 5, 4, 2, 2, 3])$, the sequence of words is

- 4,
- $|4|1 \to 41$,
- $|4|1|5 \to 415$,
- $|415|4 \to 5414$,
- $|5|4|14|2 \to 54412$,
- $|5441|2|2 \to 154422$,
- $|1|5442|2|3 \to 1254423$.

So $\phi([4, 1, 5, 4, 2, 2, 3]) = [1, 2, 5, 4, 4, 2, 3]$.

See also:

`Foata bijection on Permutations`.

EXAMPLES:

```
sage: w = Word([2,2,2,1,1,1])
sage: w.foata_bijection()
word: 112221
sage: w = Word([2,2,1,2,2,2,1,1,2,1])
sage: w.foata_bijection()
word: 2122212211
sage: w = Word([4,1,5,4,2,2,3])
sage: w.foata_bijection()
word: 1254423
```

TESTS:

```
sage: w = Word('121314')
sage: w.foata_bijection()
word: 231114
sage: w = Word('1133a1')
sage: w.foata_bijection()
word: 3113a1
```

good_suffix_table()

Returns a table of the maximum skip you can do in order not to miss a possible occurrence of self in a word.

This is a part of the Boyer-Moore algorithm to find factors. See [1].

EXAMPLES:

```
sage: Word('121321').good_suffix_table()
[5, 5, 5, 5, 3, 3, 1]
sage: Word('12412').good_suffix_table()
[3, 3, 3, 3, 3, 1]
```

REFERENCES:

- [1] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Communications of the ACM 20 (1977) 762–772.

has_period(*p*)

Returns True if self has the period p, False otherwise.

Note: By convention, integers greater than the length of self are periods of self.

INPUT:

- p - an integer to check if it is a period of self.

EXAMPLES:

```
sage: w = Word('ababa')
sage: w.has_period(2)
True
sage: w.has_period(3)
False
sage: w.has_period(4)
True
sage: w.has_period(-1)
False
sage: w.has_period(5)
True
sage: w.has_period(6)
True
```

has_prefix(*other*)

Test whether self has other as a prefix.

INPUT:

- other - a word, or data describing a word

OUTPUT:

- boolean

EXAMPLES:

```
sage: w = Word("abbabaababababa")
sage: u = Word("abbab")
sage: w.has_prefix(u)
True
sage: u.has_prefix(w)
False
sage: u.has_prefix("abbab")
True

sage: w = Word([0,1,1,0,1,0,0,1,0,1,0,1,0])
sage: u = Word([0,1,1,0,1])
sage: w.has_prefix(u)
True
sage: u.has_prefix(w)
False
sage: u.has_prefix([0,1,1,0,1])
True
```

has_suffix(*other*)
Test whether `self` has `other` as a suffix.

Note: Some word datatype classes, like `WordDatatype_str`, override this method.

INPUT:

•`other` - a word, or data describing a word

OUTPUT:

•boolean

EXAMPLES:

```
sage: w = Word("abbabaababababa")
sage: u = Word("ababa")
sage: w.has_suffix(u)
True
sage: u.has_suffix(w)
False
sage: u.has_suffix("ababa")
True

sage: w = Word([0,1,1,0,1,0,0,1,0,1,0,1,0])
sage: u = Word([0,1,0,1,0])
sage: w.has_suffix(u)
True
sage: u.has_suffix(w)
False
sage: u.has_suffix([0,1,0,1,0])
True
```

implicit_suffix_tree()
Returns the implicit suffix tree of self.

The *suffix tree* of a word w is a compactification of the suffix trie for w. The compactification removes all nodes that have exactly one incoming edge and exactly one outgoing edge. It consists of two components: a tree and a word. Thus, instead of labelling the edges by factors of w, we can labelled them by indices of the occurrence of the factors in w.

See sage.combinat.words.suffix_trees.ImplicitSuffixTree? for more information.

EXAMPLES:

```
sage: w = Word("cacao")
sage: w.implicit_suffix_tree()
Implicit Suffix Tree of the word: cacao

sage: w = Word([0,1,0,1,1])
sage: w.implicit_suffix_tree()
Implicit Suffix Tree of the word: 01011
```

inv_lex_less(*other*)
Returns True if self is inverse lexicographically less than other.

EXAMPLES:

```
sage: Word([1,2,4]).inv_lex_less(Word([1,3,2]))
False
sage: Word([3,2,1]).inv_lex_less(Word([1,2,3]))
True
```

inversions()

Returns a list of the inversions of self. An inversion is a pair (i,j) of non-negative integers i < j such that self[i] > self[j].

EXAMPLES:

```
sage: Word([1,2,3,2,2,1]).inversions()
[[1, 5], [2, 3], [2, 4], [2, 5], [3, 5], [4, 5]]
sage: Words([3,2,1])([1,2,3,2,2,1]).inversions()
[[0, 1], [0, 2], [0, 3], [0, 4], [1, 2]]
sage: Word('abbaba').inversions()
[[1, 3], [1, 5], [2, 3], [2, 5], [4, 5]]
sage: Words('ba')('abbaba').inversions()
[[0, 1], [0, 2], [0, 4], [3, 4]]
```

is_balanced(*q=1*)

Returns True if self is q-balanced, and False otherwise.

A finite or infinite word w is said to be *q-balanced* if for any two factors u, v of w of the same length, the difference between the number of x's in each of u and v is at most q for all letters x in the alphabet of w. A 1-balanced word is simply said to be balanced. See for instance [1] and Chapter 2 of [2].

INPUT:

- q - integer (default 1), the balance level

OUTPUT:

boolean – the result

EXAMPLES:

```
sage: Word('1213121').is_balanced()
True
sage: Word('1122').is_balanced()
False
sage: Word('121333121').is_balanced()
False
sage: Word('121333121').is_balanced(2)
False
sage: Word('121333121').is_balanced(3)
True
sage: Word('121122121').is_balanced()
False
sage: Word('121122121').is_balanced(2)
True
```

TESTS:

```
sage: Word('121122121').is_balanced(-1)
Traceback (most recent call last):
...
TypeError: the balance level must be a positive integer
sage: Word('121122121').is_balanced(0)
Traceback (most recent call last):
...
TypeError: the balance level must be a positive integer
sage: Word('121122121').is_balanced('a')
Traceback (most recent call last):
...
TypeError: the balance level must be a positive integer
```

REFERENCES:

- [1] J. Cassaigne, S. Ferenczi, L.Q. Zamboni, Imbalances in Arnoux-Rauzy sequences, Ann. Inst. Fourier (Grenoble) 50 (2000) 1265–1276.
- [2] M. Lothaire, Algebraic Combinatorics On Words, vol. 90 of Encyclopedia of Mathematics and its Applications, Cambridge University Press, U.K., 2002.

is_cadence(*seq*)

Returns True if seq is a cadence of self, and False otherwise.

A *cadence* is an increasing sequence of indexes that all map to the same letter.

EXAMPLES:

```
sage: Word('121132123').is_cadence([0, 2, 6])
True
sage: Word('121132123').is_cadence([0, 1, 2])
False
sage: Word('121132123').is_cadence([])
True
```

is_christoffel()

Returns True if self is a Christoffel word, and False otherwise.

The *Christoffel word* of slope p/q is obtained from the Cayley graph of $\mathbf{Z}/(p+q)\mathbf{Z}$ with generator q as follows. If $u \to v$ is an edge in the Cayley graph, then, $v = u + p \mod p+q$. Let a,'b' be the alphabet of w. Label the edge $u \to v$ by a if u < v and b otherwise. The Christoffel word is the word obtained by reading the edge labels along the cycle beginning from 0.

Equivalently, w is a Christoffel word iff w is a symmetric non-empty word and $w[1:n-1]$ is a palindrome.

See for instance [8] and [9].

INPUT:

`self` – word

OUTPUT:

Boolean – `True` if `self` is a Christoffel word, `False` otherwise.

EXAMPLES:

```
sage: Word('00100101').is_christoffel()
True
sage: Word('aab').is_christoffel()
True
sage: Word().is_christoffel()
False
sage: Word('123123123').is_christoffel()
False
sage: Word('00100').is_christoffel()
False
sage: Word('0').is_christoffel()
True
```

TESTS:

[8] Jean Berstel. Sturmian and episturmian words (a survey of some recent results). In S. Bozapalidis and G. Rahonis, editors, CAI 2007,volume 4728 of Lecture Notes in Computer Science, pages 23-47. Springer-Verlag, 2007.

[9] J. Berstel, A. Lauve, C. R., F. Saliola, Combinatorics on words: Christoffel words and repetitions in words, CRM Monograph Series, 27. American Mathematical Society, Providence, RI, 2009. xii+147 pp. ISBN: 978-0-8218-4480-9

```
sage: words.LowerChristoffelWord(5,4).is_christoffel()
True
sage: words.UpperChristoffelWord(5,4).is_christoffel()
True
sage: Word('aaaaaaaaa').is_christoffel()
False
```

REFERENCES:

is_conjugate_with(*other*)

Returns True if self is a conjugate of other, and False otherwise.

INPUT:

- `other` - a finite word

OUPUT

bool

EXAMPLES:

```
sage: w = Word([0..20])
sage: z = Word([7..20] + [0..6])
sage: w
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
sage: z
word: 7,8,9,10,11,12,13,14,15,16,17,18,19,20,0,1,2,3,4,5,6
sage: w.is_conjugate_with(z)
True
sage: z.is_conjugate_with(w)
True
sage: u = Word([4]*21)
sage: u.is_conjugate_with(w)
False
sage: u.is_conjugate_with(z)
False
```

Both words must be finite:

```
sage: w = Word(iter([2]*100),length='unknown')
sage: z = Word([2]*100)
sage: z.is_conjugate_with(w) #TODO: Not implemented for word of unknown length
True
sage: wf = Word(iter([2]*100),length='finite')
sage: z.is_conjugate_with(wf)
True
sage: wf.is_conjugate_with(z)
True
```

TESTS:

```
sage: Word('11213').is_conjugate_with(Word('31121'))
True
sage: Word().is_conjugate_with(Word('123'))
False
sage: Word('112131').is_conjugate_with(Word('11213'))
False
sage: Word('12131').is_conjugate_with(Word('11213'))
True
```

We make sure that trac ticket #11128 is fixed:

```
sage: Word('abaa').is_conjugate_with(Word('aaba'))
True
sage: Word('aaba').is_conjugate_with(Word('abaa'))
True
```

is_cube()

Returns True if self is a cube, and False otherwise.

EXAMPLES:

```
sage: Word('012012012').is_cube()
True
sage: Word('01010101').is_cube()
False
sage: Word().is_cube()
True
sage: Word('012012').is_cube()
False
```

is_cube_free()

Returns True if self does not contain cubes, and False otherwise.

EXAMPLES:

```
sage: Word('12312').is_cube_free()
True
sage: Word('32221').is_cube_free()
False
sage: Word().is_cube_free()
True
```

TESTS:

We make sure that trac ticket #8490 is fixed:

```
sage: Word('111').is_cube_free()
False
sage: Word('2111').is_cube_free()
False
sage: Word('32111').is_cube_free()
False
```

is_empty()

Returns True if the length of self is zero, and False otherwise.

EXAMPLES:

```
sage: Word([]).is_empty()
True
sage: Word('a').is_empty()
False
```

is_factor(*other*)

Returns True if self is a factor of other, and False otherwise.

EXAMPLES:

```
sage: u = Word('2113')
sage: w = Word('123121332131233121132123')
sage: u.is_factor(w)
True
```

```
sage: u = Word('321')
sage: w = Word('12312412313123123 12')
sage: u.is_factor(w)
False
```

The empty word is factor of another word:

```
sage: Word().is_factor(Word())
True
sage: Word().is_factor(Word('a'))
True
sage: Word().is_factor(Word([1,2,3]))
True
sage: Word().is_factor(Word(lambda n:n, length=5))
True
```

is_finite()

Returns True.

EXAMPLES:

```
sage: Word([]).is_finite()
True
sage: Word('a').is_finite()
True
```

is_full(*f=None*)

Returns True if self has defect 0, and False otherwise.

A word is *full* (or *rich*) if its defect is zero (see [1]). If `f` is given, then the f-palindromic defect is used (see [2]).

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be callable on letters as well as words (e.g. WordMorphism).

OUTPUT:

boolean – If f is None, whether self is full; otherwise, whether self is full of f-palindromes.

EXAMPLES:

```
sage: words.ThueMorseWord()[:100].is_full()
False
sage: words.FibonacciWord()[:100].is_full()
True
sage: Word('000000000000000').is_full()
True
sage: Word('011010011001').is_full()
False
sage: Word('2194').is_full()
True
sage: Word().is_full()
True

sage: f = WordMorphism('a->b,b->a')
sage: Word().is_full(f)
True
sage: w = Word('ab')
sage: w.is_full()
True
```

```
sage: w.is_full(f)
True

sage: f = WordMorphism('a->b,b->a')
sage: Word('abab').is_full(f)
True
sage: Word('abba').is_full(f)
False
```

A simple example of an infinite word full of f-palindromes:

```
sage: p = WordMorphism({0:'abc',1:'ab'})
sage: f = WordMorphism('a->b,b->a,c->c')
sage: p(words.FibonacciWord()[:50]).is_full(f)
True
sage: p(words.FibonacciWord()[:150]).is_full(f)
True
```

REFERENCES:

- [1] S. Brlek, S. Hamel, M. Nivat, C. Reutenauer, On the Palindromic Complexity of Infinite Words, in J. Berstel, J. Karhumaki, D. Perrin, Eds, Combinatorics on Words with Applications, International Journal of Foundation of Computer Science, Vol. 15, No. 2 (2004) 293–306.
- [2] E. Pelantová, Š. Starosta, Infinite words rich and almost rich in generalized palindromes, in: G. Mauri, A. Leporati (Eds.), Developments in Language Theory, volume 6795 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 406–416

is_lyndon()

Return `True` if `self` is a Lyndon word, and `False` otherwise.

A *Lyndon word* is a non-empty word that is lexicographically smaller than each of its proper suffixes (for the given order on its alphabet). That is, w is a Lyndon word if w is non-empty and for each factorization $w = uv$ (with u, v both non-empty), we have $w < v$.

Equivalently, w is a Lyndon word iff w is a non-empty word that is lexicographically smaller than each of its proper conjugates for the given order on its alphabet.

See for instance [1].

EXAMPLES:

```
sage: Word('123132133').is_lyndon()
True
sage: Word().is_lyndon()
False
sage: Word('122112').is_lyndon()
False
```

TESTS:

A sanity check: `LyndonWords` generates Lyndon words, so we filter all words of length $n < 10$ on the alphabet [1,2,3] for Lyndon words, and compare with the `LyndonWords` generator:

```
sage: for n in range(1,10):
....:     lw1 = [w for w in Words([1,2,3], n) if w.is_lyndon()]
....:     lw2 = LyndonWords(3,n)
....:     if set(lw1) != set(lw2): print False
```

Filter all words of length 8 on the alphabet [c,a,b] for Lyndon words, and compare with the `LyndonWords` generator after mapping [a,b,c] to [2,3,1]:

```
sage: lw = [w for w in Words('cab', 8) if w.is_lyndon()]
sage: phi = WordMorphism({'a':2,'b':3,'c':1})
sage: set(map(phi, lw)) == set(LyndonWords(3,8))
True
```

REFERENCES:

- [1] M. Lothaire, Combinatorics On Words, vol. 17 of Encyclopedia of Mathematics and its Applications, Addison-Wesley, Reading, Massachusetts, 1983.

is_overlap()

Returns True if self is an overlap, and False otherwise.

EXAMPLES:

```
sage: Word('12121').is_overlap()
True
sage: Word('123').is_overlap()
False
sage: Word('1231').is_overlap()
False
sage: Word('123123').is_overlap()
False
sage: Word('1231231').is_overlap()
True
sage: Word().is_overlap()
False
```

is_palindrome(*f=None*)

Returns True if self is a palindrome (or a f-palindrome), and False otherwise.

Let $f : \Sigma \to \Sigma$ be an involution that extends to a morphism on Σ^*. We say that $w \in \Sigma^*$ is a *'f'-palindrome* if $w = f(\tilde{w})$ [1]. Also called *'f'-pseudo-palindrome* [2].

INPUT:

- `f` - involution (default: `None`) on the alphabet of self. It must be callable on letters as well as words (e.g. `WordMorphism`). The default value corresponds to usual palindromes, i.e., f equal to the identity.

EXAMPLES:

```
sage: Word('esope reste ici et se repose').is_palindrome()
False
sage: Word('esoperesteicietserepose').is_palindrome()
True
sage: Word('I saw I was I').is_palindrome()
True
sage: Word('abbcbba').is_palindrome()
True
sage: Word('abcbdba').is_palindrome()
False
```

Some f-palindromes:

```
sage: f = WordMorphism('a->b,b->a')
sage: Word('aababb').is_palindrome(f)
True

sage: f = WordMorphism('a->b,b->a,c->c')
sage: Word('abacbacbab').is_palindrome(f)
True
```

```
sage: f = WordMorphism({'a':'b','b':'a'})
sage: Word('aababb').is_palindrome(f)
True

sage: f = WordMorphism({0:[1],1:[0]})
sage: w = words.ThueMorseWord()[:8]; w
word: 01101001
sage: w.is_palindrome(f)
True
```

The word must be in the domain of the involution:

```
sage: f = WordMorphism('a->a')
sage: Word('aababb').is_palindrome(f)
Traceback (most recent call last):
...
KeyError: 'b'
```

TESTS:

If the given involution is not an involution:

```
sage: f = WordMorphism('a->b,b->b')
sage: Word('abab').is_palindrome(f)
Traceback (most recent call last):
...
TypeError: self (=a->b, b->b) is not an endomorphism

sage: Y = Word
sage: Y().is_palindrome()
True
sage: Y('a').is_palindrome()
True
sage: Y('ab').is_palindrome()
False
sage: Y('aba').is_palindrome()
True
sage: Y('aa').is_palindrome()
True
sage: E = WordMorphism('a->b,b->a')
sage: Y().is_palindrome(E)
True
sage: Y('a').is_palindrome(E)
False
sage: Y('ab').is_palindrome(E)
True
sage: Y('aa').is_palindrome(E)
False
sage: Y('aba').is_palindrome(E)
False
sage: Y('abab').is_palindrome(E)
True
```

REFERENCES:

- [1] S. Labbé, Propriétés combinatoires des f-palindromes, Mémoire de maîtrise en Mathématiques, Montréal, UQAM, 2008, 109 pages.
- [2] V. Anne, L.Q. Zamboni, I. Zorca, Palindromes and Pseudo- Palindromes in Episturmian and

Pseudo-Palindromic Infinite Words, in : S. Brlek, C. Reutenauer (Eds.), Words 2005, Publications du LaCIM, Vol. 36 (2005) 91–100.

is_prefix(*other*)

Returns True if self is a prefix of other, and False otherwise.

EXAMPLES:

```
sage: w = Word('0123456789')
sage: y = Word('012345')
sage: y.is_prefix(w)
True
sage: w.is_prefix(y)
False
sage: w.is_prefix(Word())
False
sage: Word().is_prefix(w)
True
sage: Word().is_prefix(Word())
True
```

is_primitive()

Returns True if self is primitive, and False otherwise.

A finite word w is *primitive* if it is not a positive integer power of a shorter word.

EXAMPLES:

```
sage: Word('1231').is_primitive()
True
sage: Word('111').is_primitive()
False
```

is_proper_prefix(*other*)

Returns True if self is a proper prefix of other, and False otherwise.

EXAMPLES:

```
sage: Word('12').is_proper_prefix(Word('123'))
True
sage: Word('12').is_proper_prefix(Word('12'))
False
sage: Word().is_proper_prefix(Word('123'))
True
sage: Word('123').is_proper_prefix(Word('12'))
False
sage: Word().is_proper_prefix(Word())
False
```

is_proper_suffix(*other*)

Returns True if self is a proper suffix of other, and False otherwise.

EXAMPLES:

```
sage: Word('23').is_proper_suffix(Word('123'))
True
sage: Word('12').is_proper_suffix(Word('12'))
False
sage: Word().is_proper_suffix(Word('123'))
True
sage: Word('123').is_proper_suffix(Word('12'))
False
```

is_quasiperiodic()

Returns True if self is quasiperiodic, and False otherwise.

A finite or infinite word w is *quasiperiodic* if it can be constructed by concatenations and superpositions of one of its proper factors u, which is called a *quasiperiod* of w. See for instance [1], [2], and [3].

EXAMPLES:

```
sage: Word('abaababaabaababaaba').is_quasiperiodic()
True
sage: Word('abacaba').is_quasiperiodic()
False
sage: Word('a').is_quasiperiodic()
False
sage: Word().is_quasiperiodic()
False
sage: Word('abaaba').is_quasiperiodic()
True
```

REFERENCES:

- [1] A. Apostolico, A. Ehrenfeucht, Efficient detection of quasiperiodicities in strings, Theoret. Comput. Sci. 119 (1993) 247–265.
- [2] S. Marcus, Quasiperiodic infinite words, Bull. Eur. Assoc. Theor. Comput. Sci. 82 (2004) 170-174.
- [3] A. Glen, F. Levé, G. Richomme, Quasiperiodic and Lyndon episturmian words, Preprint, 2008, arXiv:0805.0730.

is_rich(*f=None*)

Returns True if self has defect 0, and False otherwise.

A word is *full* (or *rich*) if its defect is zero (see [1]). If `f` is given, then the f-palindromic defect is used (see [2]).

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be callable on letters as well as words (e.g. WordMorphism).

OUTPUT:

boolean – If f is None, whether self is full; otherwise, whether self is full of f-palindromes.

EXAMPLES:

```
sage: words.ThueMorseWord()[:100].is_full()
False
sage: words.FibonacciWord()[:100].is_full()
True
sage: Word('0000000000000000').is_full()
True
sage: Word('011010011001').is_full()
False
sage: Word('2194').is_full()
True
sage: Word().is_full()
True

sage: f = WordMorphism('a->b,b->a')
sage: Word().is_full(f)
True
```

```
sage: w = Word('ab')
sage: w.is_full()
True
sage: w.is_full(f)
True

sage: f = WordMorphism('a->b,b->a')
sage: Word('abab').is_full(f)
True
sage: Word('abba').is_full(f)
False
```

A simple example of an infinite word full of f-palindromes:

```
sage: p = WordMorphism({0:'abc',1:'ab'})
sage: f = WordMorphism('a->b,b->a,c->c')
sage: p(words.FibonacciWord()[:50]).is_full(f)
True
sage: p(words.FibonacciWord()[:150]).is_full(f)
True
```

REFERENCES:

- [1] S. Brlek, S. Hamel, M. Nivat, C. Reutenauer, On the Palindromic Complexity of Infinite Words, in J. Berstel, J. Karhumaki, D. Perrin, Eds, Combinatorics on Words with Applications, International Journal of Foundation of Computer Science, Vol. 15, No. 2 (2004) 293–306.
- [2] E. Pelantová, Š. Starosta, Infinite words rich and almost rich in generalized palindromes, in: G. Mauri, A. Leporati (Eds.), Developments in Language Theory, volume 6795 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 406–416

is_smooth_prefix()

Returns True if self is the prefix of a smooth word, and False otherwise.

Let $A_k = \{1, \ldots, k\}$, $k \geq 2$. An infinite word w in A_k^ω is said to be *smooth* if and only if for all positive integers m, $\Delta^m(w)$ is in A_k^ω, where $\Delta(w)$ is the word obtained from w by composing the length of consecutive runs of the same letter in w. See for instance [1] and [2].

INPUT:

- `self` - must be a word over the integers to get something other than False

OUTPUT:

boolean – whether self is a smooth prefix or not

EXAMPLES:

```
sage: W = Words([1, 2])
sage: W([1, 1, 2, 2, 1, 2, 1, 1]).is_smooth_prefix()
True
sage: W([1, 2, 1, 2, 1, 2]).is_smooth_prefix()
False
```

REFERENCES:

- [1] S. Brlek, A. Ladouceur, A note on differentiable palindromes, Theoret. Comput. Sci. 302 (2003) 167–178.
- [2] S. Brlek, S. Dulucq, A. Ladouceur, L. Vuillon, Combinatorial properties of smooth infinite words, Theoret. Comput. Sci. 352 (2006) 306–317.

is_square()
Returns True if self is a square, and False otherwise.

EXAMPLES:

```
sage: Word([1,0,0,1]).is_square()
False
sage: Word('1212').is_square()
True
sage: Word('1213').is_square()
False
sage: Word('12123').is_square()
False
sage: Word().is_square()
True
```

is_square_free()
Returns True if self does not contain squares, and False otherwise.

EXAMPLES:

```
sage: Word('12312').is_square_free()
True
sage: Word('31212').is_square_free()
False
sage: Word().is_square_free()
True
```

TESTS:

We make sure that trac ticket #8490 is fixed:

```
sage: Word('11').is_square_free()
False
sage: Word('211').is_square_free()
False
sage: Word('3211').is_square_free()
False
```

is_sturmian_factor()
Tells whether `self` is a factor of a Sturmian word.

The finite word `self` must be defined on a two-letter alphabet.

Equivalently, tells whether `self` is balanced. The advantage over the is_balanced method is that this one runs in linear time whereas is_balanced runs in quadratic time.

OUTPUT:

- boolean – the result.

EXAMPLES:

```
sage: w = Word('01110110110111011101',alphabet='01')
sage: w.is_sturmian_factor()
True

sage: words.LowerMechanicalWord(random(),alphabet='01')[:100].is_sturmian_factor()
True
sage: words.CharacteristicSturmianWord(random())[:100].is_sturmian_factor()
True
```

```
sage: w = Word('aabb',alphabet='ab')
sage: w.is_sturmian_factor()
False

sage: s1 = WordMorphism('a->ab,b->b')
sage: s2 = WordMorphism('a->ba,b->b')
sage: s3 = WordMorphism('a->a,b->ba')
sage: s4 = WordMorphism('a->a,b->ab')
sage: W = Words('ab')
sage: w = W('ab')
sage: for i in xrange(8): w = choice([s1,s2,s3,s4])(w)
sage: w
word: abaaabaaabaabaaabaaabaabaaabaabaaabaaaba...
sage: w.is_sturmian_factor()
True
```

Famous words:

```
sage: words.FibonacciWord()[:100].is_sturmian_factor()
True
sage: words.ThueMorseWord()[:1000].is_sturmian_factor()
False
sage: words.KolakoskiWord()[:1000].is_sturmian_factor()
False
```

REFERENCES:

AUTHOR:

- Thierry Monteil

is_subword_of(*other*)

Returns True is self is a subword of other, and False otherwise.

EXAMPLES:

```
sage: Word().is_subword_of(Word('123'))
True
sage: Word('123').is_subword_of(Word('3211333213233321'))
True
sage: Word('321').is_subword_of(Word('11122212112122133111222332'))
False
```

See also:

longest_common_subword()

is_suffix(*other*)

Returns True if w is a suffix of other, and False otherwise.

EXAMPLES:

```
sage: w = Word('0123456789')
sage: y = Word('56789')
sage: y.is_suffix(w)
True
sage: w.is_suffix(y)
False
sage: Word('579').is_suffix(w)
False
sage: Word().is_suffix(y)
True
```

```
sage: w.is_suffix(Word())
False
sage: Word().is_suffix(Word())
True
```

is_symmetric(*f=None*)

Returns True if self is symmetric (or f-symmetric), and False otherwise.

A word is *symmetric* (resp. *f-symmetric*) if it is the product of two palindromes (resp. f-palindromes). See [1] and [2].

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be callable on letters as well as words (e.g. WordMorphism).

EXAMPLES:

```
sage: Word('abbabab').is_symmetric()
True
sage: Word('ababa').is_symmetric()
True
sage: Word('aababaabba').is_symmetric()
False
sage: Word('aabbbaababba').is_symmetric()
False
sage: f = WordMorphism('a->b,b->a')
sage: Word('aabbbaababba').is_symmetric(f)
True
```

REFERENCES:

- [1] S. Brlek, S. Hamel, M. Nivat, C. Reutenauer, On the Palindromic Complexity of Infinite Words, in J. Berstel, J. Karhumaki, D. Perrin, Eds, Combinatorics on Words with Applications, International Journal of Foundation of Computer Science, Vol. 15, No. 2 (2004) 293–306.
- [2] A. de Luca, A. De Luca, Pseudopalindrome closure operators in free monoids, Theoret. Comput. Sci. 362 (2006) 282–300.

is_tangent()

Tells whether `self` is a tangent word.

The finite word `self` must be defined on a two-letter alphabet.

A binary word is said to be *tangent* if it can appear in infinitely many cutting sequences of a smooth curve, where each cutting sequence is observed on a progressively smaller grid.

This class of words strictly contains the class of 1-balanced words, and is strictly contained in the class of 2-balanced words.

This method runs in linear time.

OUTPUT:

- boolean – the result.

EXAMPLES:

```
sage: w = Word('01110110110111011101',alphabet='01')
sage: w.is_tangent()
True
```

Some tangent words may not be balanced:

```
sage: Word('aabb',alphabet='ab').is_balanced()
False
sage: Word('aabb',alphabet='ab').is_tangent()
True
```

Some 2-balanced words may not be tangent:

```
sage: Word('aaabb',alphabet='ab').is_tangent()
False
sage: Word('aaabb',alphabet='ab').is_balanced(2)
True
```

Famous words:

```
sage: words.FibonacciWord()[:100].is_tangent()
True
sage: words.ThueMorseWord()[:1000].is_tangent()
True
sage: words.KolakoskiWord()[:1000].is_tangent()
False
```

REFERENCES:

AUTHOR:

- Thierry Monteil

iterated_left_palindromic_closure(*f=None*)

Returns the iterated left (f-)palindromic closure of self.

INPUT:

- `f` – involution (default: `None`) on the alphabet of `self`. It must be callable on letters as well as words (e.g. WordMorphism).

OUTPUT:

word – the left iterated f-palindromic closure of `self`.

EXAMPLES:

```
sage: Word('123').iterated_left_palindromic_closure()
word: 3231323
sage: f = WordMorphism('a->b,b->a')
sage: Word('ab').iterated_left_palindromic_closure(f=f)
word: abbaab
sage: Word('aab').iterated_left_palindromic_closure(f=f)
word: abbaabbaab
```

TESTS:

If `f` is not a involution:

```
sage: f = WordMorphism('a->b,b->b')
sage: Word('aab').iterated_left_palindromic_closure(f=f)
Traceback (most recent call last):
...
TypeError: self (=a->b, b->b) is not an endomorphism
```

REFERENCES:

- A. de Luca, A. De Luca, Pseudopalindrome closure operators in free monoids, Theoret. Comput. Sci. 362 (2006) 282–300.

lacunas (*f=None*)
Returns the list of all the lacunas of self.

A *lacuna* is a position in a word where the longest (f-)palindromic suffix is not unioccurrent (see [1]).

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be callable on letters as well as words (e.g. WordMorphism). The default value corresponds to usual palindromes, i.e., f equal to the identity.

OUTPUT:

list – list of all the lacunas of self.

EXAMPLES:

```
sage: w = Word([0,1,1,2,3,4,5,1,13,3])
sage: w.lacunas()
[7, 9]
sage: words.ThueMorseWord()[:100].lacunas()
[8, 9, 24, 25, 32, 33, 34, 35, 36, 37, 38, 39, 96, 97, 98, 99]
sage: f = WordMorphism({0:[1],1:[0]})
sage: words.ThueMorseWord()[:50].lacunas(f)
[0, 2, 4, 12, 16, 17, 18, 19, 48, 49]
```

REFERENCES:

- [1] A. Blondin-Massé, S. Brlek, S. Labbé, Palindromic lacunas of the Thue-Morse word, Proc. GASCOM 2008 (June 16-20 2008, Bibbiena, Arezzo-Italia), 53–67.

last_position_dict ()
Returns a dictionary that contains the last position of each letter in self.

EXAMPLES:

```
sage: Word('1231232').last_position_dict()
{'1': 3, '2': 6, '3': 5}
```

left_special_factors (*n=None*)
Returns the left special factors (of length n).

A factor u of a word w is *left special* if there are two distinct letters a and b such that au and bu are factors of w.

INPUT:

- `n` - integer (optional, default: `None`). If `None`, it returns all left special factors.

OUTPUT:

A list of words.

EXAMPLES:

```
sage: alpha, beta, x = 0.54, 0.294, 0.1415
sage: w = words.CodingOfRotationWord(alpha, beta, x)[:40]
sage: for i in range(5): print i, sorted(w.left_special_factors(i))
0 [word: ]
1 [word: 0]
2 [word: 00, word: 01]
3 [word: 000, word: 010]
4 [word: 0000, word: 0101]
```

left_special_factors_iterator(*n=None*)
Returns an iterator over the left special factors (of length n).

A factor u of a word w is *left special* if there are two distinct letters a and b such that au and bu are factors of w.

INPUT:

- n - integer (optional, default: None). If None, it returns an iterator over all left special factors.

EXAMPLES:

```
sage: alpha, beta, x = 0.54, 0.294, 0.1415
sage: w = words.CodingOfRotationWord(alpha, beta, x)[:40]
sage: sorted(w.left_special_factors_iterator(3))
[word: 000, word: 010]
sage: sorted(w.left_special_factors_iterator(4))
[word: 0000, word: 0101]
sage: sorted(w.left_special_factors_iterator(5))
[word: 00000, word: 01010]
```

length()
Returns the length of self.

TESTS:

```
sage: from sage.combinat.words.word import Word_class
sage: w = Word(iter('abba'*40), length="finite")
sage: w._len is None
True
sage: w.length()
160
sage: w = Word(iter('abba'), length=4)
sage: w._len
4
sage: w.length()
4
sage: def f(n):
...     return range(2,12,2)[n]
sage: w = Word(f, length=5)
sage: w.length()
5
```

length_border()
Returns the length of the border of self.

The *border* of a word is the longest word that is both a proper prefix and a proper suffix of self.

EXAMPLES:

```
sage: Word('121').length_border()
1
sage: Word('1').length_border()
0
sage: Word('1212').length_border()
2
sage: Word('111').length_border()
2
sage: Word().length_border() is None
True
```

length_maximal_palindrome(*j*, *m=None*, *f=None*)

Returns the length of the longest palindrome centered at position j.

INPUT:

- j – rational, position of the symmetry axis of the palindrome. Must return an integer when doubled. It is an integer when the center of the palindrome is a letter.
- m – integer (default: None), minimal length of palindrome, if known. The parity of m can't be the same as the parity of 2j.
- f – involution (default: None), on the alphabet. It must be callable on letters as well as words (e.g. WordMorphism).

OUTPUT:

The length of the longest f-palindrome centered at position j.

EXAMPLES:

```
sage: Word('01001010').length_maximal_palindrome(3/2)
0
sage: Word('01101001').length_maximal_palindrome(3/2)
4
sage: Word('01010').length_maximal_palindrome(j=3, f='0->1,1->0')
0
sage: Word('01010').length_maximal_palindrome(j=2.5, f='0->1,1->0')
4
sage: Word('0222220').length_maximal_palindrome(3, f='0->1,1->0,2->2')
5

sage: w = Word('abcdcbaxyzzyx')
sage: w.length_maximal_palindrome(3)
7
sage: w.length_maximal_palindrome(3, 3)
7
sage: w.length_maximal_palindrome(3.5)
0
sage: w.length_maximal_palindrome(9.5)
6
sage: w.length_maximal_palindrome(9.5, 2)
6
```

TESTS:

These are wrong inputs:

```
sage: w.length_maximal_palindrome(9.6)
Traceback (most recent call last):
...
ValueError: j must be positive, inferior to length of self
sage: w.length_maximal_palindrome(3, 2)
Traceback (most recent call last):
...
ValueError: (2*j-m-1)/2(=3/2) must be an integer, i.e., 2*j(=6) and
m(=2) can't have the same parity
sage: w.length_maximal_palindrome(9.5, 3)
Traceback (most recent call last):
...
ValueError: (2*j-m-1)/2(=15/2) must be an integer, i.e., 2*j(=19) and
m(=3) can't have the same parity
```

lengths_lps (*f=None*)

Returns the list of the length of the longest palindromic suffix (lps) for each non-empty prefix of self.

It corresponds to the function G_w defined in [1].

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be callable on letters as well as words (e.g. WordMorphism).

OUTPUT:

list – list of the length of the longest palindromic suffix (lps) for each non-empty prefix of self.

EXAMPLES:

```
sage: Word().lengths_lps()
[]
sage: Word('a').lengths_lps()
[1]
sage: Word('aaa').lengths_lps()
[1, 2, 3]
sage: Word('abbabaabbaab').lengths_lps()
[1, 1, 2, 4, 3, 3, 2, 4, 2, 4, 6, 8]

sage: f = WordMorphism('a->b,b->a')
sage: Word('abbabaabbaab').lengths_lps(f)
[0, 2, 0, 2, 2, 4, 6, 8, 4, 6, 4, 6]

sage: f = WordMorphism({5:[8],8:[5]})
sage: Word([5,8,5,5,8,8,5,5,8,8,5,8,5]).lengths_lps(f)
[0, 2, 2, 0, 2, 4, 6, 4, 6, 8, 10, 12, 4]
```

REFERENCES:

- [1] A. Blondin-Massé, S. Brlek, A. Frosini, S. Labbé, S. Rinaldi, Reconstructing words from a fixed palindromic length sequence, Proc. TCS 2008, 5th IFIP International Conference on Theoretical Computer Science (September 8-10 2008, Milano, Italia), accepted.

lengths_maximal_palindromes (*f=None*)

Returns the length of maximal palindromes centered at each position.

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be callable on letters as well as words (e.g. WordMorphism).

OUTPUT:

list – The length of the maximal palindrome (or f-palindrome) with a given symmetry axis (letter or space between two letters).

EXAMPLES:

```
sage: Word('01101001').lengths_maximal_palindromes()
[0, 1, 0, 1, 4, 1, 0, 3, 0, 3, 0, 1, 4, 1, 0, 1, 0]
sage: Word('00000').lengths_maximal_palindromes()
[0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0]
sage: Word('0').lengths_maximal_palindromes()
[0, 1, 0]
sage: Word('').lengths_maximal_palindromes()
[0]
sage: Word().lengths_maximal_palindromes()
[0]
```

```
sage: f = WordMorphism('a->b,b->a')
sage: Word('abbabaab').lengths_maximal_palindromes(f)
[0, 0, 2, 0, 0, 0, 2, 0, 8, 0, 2, 0, 0, 0, 2, 0, 0]
```

lengths_unioccurrent_lps(*f=None*)

Returns the list of the lengths of the unioccurrent longest (f)-palindromic suffixes (lps) for each non-empty prefix of self. No unioccurrent lps are indicated by None.

It corresponds to the function H_w defined in [1] and [2].

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be callable on letters as well as words (e.g. WordMorphism). The default value corresponds to usual palindromes, i.e., f equal to the identity.

OUTPUT:

list – list of the length of the unioccurrent longest palindromic suffix (lps) for each non-empty prefix of self. No unioccurrent lps are indicated by None.

EXAMPLES:

```
sage: w = Word([0,1,1,2,3,4,5,1,13,3])
sage: w.lengths_unioccurrent_lps()
[1, 1, 2, 1, 1, 1, 1, None, 1, None]
sage: f = words.FibonacciWord()[:20]
sage: f.lengths_unioccurrent_lps() == f.lengths_lps()
True
sage: t = words.ThueMorseWord()
sage: t[:20].lengths_unioccurrent_lps()
[1, 1, 2, 4, 3, 3, 2, 4, None, None, 6, 8, 10, 12, 14, 16, 6, 8, 10, 12]
sage: f = WordMorphism({1:[0],0:[1]})
sage: t[:15].lengths_unioccurrent_lps(f)
[None, 2, None, 2, None, 4, 6, 8, 4, 6, 4, 6, None, 4, 6]
```

REFERENCES:

- [1] A. Blondin-Massé, S. Brlek, S. Labbé, Palindromic lacunas of the Thue-Morse word, Proc. GASCOM 2008 (June 16-20 2008, Bibbiena, Arezzo-Italia), 53–67.
- [2] A. Blondin-Massé, S. Brlek, A. Frosini, S. Labbé, S. Rinaldi, Reconstructing words from a fixed palindromic length sequence, Proc. TCS 2008, 5th IFIP International Conference on Theoretical Computer Science (September 8-10 2008, Milano, Italia), accepted.

letters()

Return the list of letters that appear in this word, listed in the order of first appearance.

EXAMPLES:

```
sage: Word([0,1,1,0,1,0,0,1]).letters()
[0, 1]
sage: Word("cacao").letters()
['c', 'a', 'o']
```

TESTS:

```
sage: Word().letters()
[]
```

longest_common_subword(*other*)

Returns a longest subword of `self` and `other`.

A subword of a word is a subset of the word's letters, read in the order in which they appear in the word.

For more information, see Wikipedia article Longest_common_subsequence_problem.

INPUT:

- `other` – a word

ALGORITHM:

For any indices i, j, we compute the longest common subword `lcs[i,j]` of $self[:i]$ and $other[:j]$. This can be easily obtained as the longest of

- `lcs[i-1,j]`
- `lcs[i,j-1]`
- `lcs[i-1,j-1]+self[i]` if `self[i]==other[j]`

EXAMPLES:

```
sage: v1 = Word("abc")
sage: v2 = Word("ace")
sage: v1.longest_common_subword(v2)
word: ac

sage: w1 = Word("10101010101010101010101010101010101010")
sage: w2 = Word("00110011001100110011001100110011001100110011")
sage: w1.longest_common_subword(w2)
word: 00110011001100110011010101010
```

TESTS:

```
sage: Word().longest_common_subword(Word())
word:
```

See also:

`is_subword_of()`

longest_common_suffix(*other*)

Returns the longest common suffix of self and other.

EXAMPLES:

```
sage: w = Word('112345678')
sage: u = Word('1115678')
sage: w.longest_common_suffix(u)
word: 5678
sage: u.longest_common_suffix(u)
word: 1115678
sage: u.longest_common_suffix(w)
word: 5678
sage: w.longest_common_suffix(w)
word: 112345678
sage: y = Word('549332345')
sage: w.longest_common_suffix(y)
word:
```

TESTS:

With the empty word:

```
sage: w.longest_common_suffix(Word())
word:
```

```
sage: Word().longest_common_suffix(w)
word:
sage: Word().longest_common_suffix(Word())
word:
```

With an infinite word:

```
sage: t=words.ThueMorseWord('ab')
sage: w.longest_common_suffix(t)
Traceback (most recent call last):
...
TypeError: other must be a finite word
```

lps (*f=None*, *l=None*)

Returns the longest palindromic (or f-palindromic) suffix of self.

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be callable on letters as well as words (e.g. WordMorphism).
- `l` - integer (default: None) the length of the longest palindrome suffix of self[:-1], if known.

OUTPUT:

word – If f is None, the longest palindromic suffix of self; otherwise, the longest f-palindromic suffix of self.

EXAMPLES:

```
sage: Word('0111').lps()
word: 111
sage: Word('011101').lps()
word: 101
sage: Word('6667').lps()
word: 7
sage: Word('abbabaab').lps()
word: baab
sage: Word().lps()
word:
sage: f = WordMorphism('a->b,b->a')
sage: Word('abbabaab').lps(f=f)
word: abbabaab
sage: w = Word('33412321')
sage: w.lps(l=3)
word: 12321
sage: Y = Word
sage: w = Y('01101001')
sage: w.lps(l=2)
word: 1001
sage: w.lps()
word: 1001
sage: w.lps(l=None)
word: 1001
sage: Y().lps(l=2)
Traceback (most recent call last):
...
IndexError: list index out of range
sage: v = Word('abbabaab')
sage: pal = v[:0]
sage: for i in range(1, v.length()+1):
```

```
...       pal = v[:i].lps(l=pal.length())
...       pal
...
word: a
word: b
word: bb
word: abba
word: bab
word: aba
word: aa
word: baab
sage: f = WordMorphism('a->b,b->a')
sage: v = Word('abbabaab')
sage: pal = v[:0]
sage: for i in range(1, v.length()+1):
...       pal = v[:i].lps(f=f, l=pal.length())
...       pal
...
word:
word: ab
word:
word: ba
word: ab
word: baba
word: bbabaa
word: abbabaab
```

lps_lengths (*f=None*)

Returns the length of the longest palindromic suffix of each prefix.

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be callable on letters as well as words (e.g. WordMorphism).

OUTPUT:

list – The length of the longest palindromic (or f-palindromic) suffix of each prefix of self.

EXAMPLES:

```
sage: Word('01101001').lps_lengths()
[0, 1, 1, 2, 4, 3, 3, 2, 4]
sage: Word('00000').lps_lengths()
[0, 1, 2, 3, 4, 5]
sage: Word('0').lps_lengths()
[0, 1]
sage: Word('').lps_lengths()
[0]
sage: Word().lps_lengths()
[0]
sage: f = WordMorphism('a->b,b->a')
sage: Word('abbabaab').lps_lengths(f)
[0, 0, 2, 0, 2, 2, 4, 6, 8]
```

lyndon_factorization()

Return the Lyndon factorization of `self`.

The *Lyndon factorization* of a finite word w is the unique factorization of w as a non-increasing product of Lyndon words, i.e., $w = l_1 \cdots l_n$ where each l_i is a Lyndon word and $l_1 \geq \cdots \geq l_n$. See for instance [1].

OUTPUT:

the list $[l_1, \ldots, l_n]$ of factors obtained

EXAMPLES:

```
sage: Word('0100100100001000').lyndon_factorization()
(01, 001, 001, 0001, 0, 0, 0)
sage: Words('10')('0100100100001000').lyndon_factorization()
(0, 10010010001000)
sage: Word('abbababbaababba').lyndon_factorization()
(abb, ababb, aababb, a)
sage: Words('ba')('abbababbaababba').lyndon_factorization()
(a, bbababbaaba, bba)
sage: Word([1,2,1,3,1,2,1]).lyndon_factorization()
(1213, 12, 1)
```

TESTS:

```
sage: Words('01')('').lyndon_factorization()
()
sage: Word('01').lyndon_factorization()
(01)
sage: Words('10')('01').lyndon_factorization()
(0, 1)
sage: lynfac = Word('abbababbaababba').lyndon_factorization()
sage: [x.is_lyndon() for x in lynfac]
[True, True, True, True]
sage: lynfac = Words('ba')('abbababbaababba').lyndon_factorization()
sage: [x.is_lyndon() for x in lynfac]
[True, True, True]
sage: w = words.ThueMorseWord()[:1000]
sage: w == prod(w.lyndon_factorization())
True
```

REFERENCES:

- [1] J.-P. Duval, Factorizing words over an ordered alphabet, J. Algorithms 4 (1983) 363–381.
- [2] G. Melancon, Factorizing infinite words using Maple, MapleTech journal, vol. 4, no. 1, 1997, pp. 34-42.

major_index (*final_descent=False*)

Return the major index of `self`.

The major index of a word w is the sum of the descents of w.

With the `final_descent` option, the last position of a non-empty word is also considered as a descent.

See also:

`major index on Permutations`.

EXAMPLES:

```
sage: w = Word([2,1,3,3,2])
sage: w.major_index()
5
sage: w = Word([2,1,3,3,2])
sage: w.major_index(final_descent=True)
10
```

minimal_period()
Returns the period of self.

Let A be an alphabet. An integer $p \geq 1$ is a *period* of a word $w = a_1 a_2 \cdots a_n$ where $a_i \in A$ if $a_i = a_{i+p}$ for $i = 1, \ldots, n-p$. The smallest period of w is called *the* period of w. See Chapter 1 of [1].

EXAMPLES:

```
sage: Word('aba').minimal_period()
2
sage: Word('abab').minimal_period()
2
sage: Word('ababa').minimal_period()
2
sage: Word('ababaa').minimal_period()
5
sage: Word('ababac').minimal_period()
6
sage: Word('aaaaaa').minimal_period()
1
sage: Word('a').minimal_period()
1
sage: Word().minimal_period()
1
```

REFERENCES:

- [1] M. Lothaire, Algebraic Combinatorics On Words, vol. 90 of Encyclopedia of Mathematics and its Applications, Cambridge University Press, U.K., 2002.

nb_factor_occurrences_in(*other*)
Returns the number of times self appears as a factor in other.

EXAMPLES:

```
sage: Word().nb_factor_occurrences_in(Word('123'))
Traceback (most recent call last):
...
NotImplementedError: The factor must be non empty
sage: Word('123').nb_factor_occurrences_in(Word('112332312313112332121123'))
4
sage: Word('321').nb_factor_occurrences_in(Word('11233231231311233221123'))
0
```

nb_subword_occurrences_in(*other*)
Returns the number of times self appears in other as a subword.

This corresponds to the notion of $binomial coefficient$ of two finite words whose properties are presented in the chapter of Lothaire's book written by Sakarovitch and Simon [1].

INPUT:

- `other` – finite word

EXAMPLES:

```
sage: tm = words.ThueMorseWord()

sage: u = Word([0,1,0,1])
sage: u.nb_subword_occurrences_in(tm[:1000])
2604124996

sage: u = Word([0,1,0,1,1,0])
```

```
sage: u.nb_subword_occurrences_in(tm[:100])
20370432
```

Note: This code, based on [2], actually compute the number of occurrences of all prefixes of `self` as subwords in all prefixes of `other`. In particular, its complexity is bounded by `len(self) * len(other)`.

TESTS:

```
sage: Word('').nb_subword_occurrences_in(Word(''))
1
sage: parent(_)
Integer Ring
sage: v,u = Word(), Word('123')
sage: v.nb_subword_occurrences_in(u)
1
sage: v,u = Word('123'), Word('1133423111323111112')
sage: v.nb_subword_occurrences_in(u)
11
sage: v,u = Word('4321'), Word('113223111223321234223111 2')
sage: v.nb_subword_occurrences_in(u)
0
sage: v,u = Word('3'), Word('122332112321213')
sage: v.nb_subword_occurrences_in(u)
4
sage: v,u = Word([]), words.ThueMorseWord()[:1000]
sage: v.nb_subword_occurrences_in(u)
1
```

REFERENCES:

- [1] M. Lothaire, Combinatorics on Words, Cambridge University Press, (1997).
- [2] Mateescu, A., Salomaa, A., Salomaa, K. and Yu, S., A sharpening of the Parikh mapping. Theoret. Informatics Appl. 35 (2001) 551-564.

number_of_factors (*n=None*, *algorithm='suffix tree'*)

Counts the number of distinct factors of self.

INPUT:

- `n` - an integer, or None.
- `algorithm` - string (default: `'suffix tree'`), takes the following values:
  - `'suffix tree'` – construct and use the suffix tree of the word
  - `'naive'` – algorithm uses a sliding window

OUTPUT:

If n is an integer, returns the number of distinct factors of length n. If n is None, returns the total number of distinct factors.

EXAMPLES:

```
sage: w = Word([1,2,1,2,3])
sage: w.number_of_factors()
13
sage: [w.number_of_factors(i) for i in range(6)]
[1, 3, 3, 3, 2, 1]
```

```
sage: w = words.ThueMorseWord()[:100]
sage: [w.number_of_factors(i) for i in range(10)]
[1, 2, 4, 6, 10, 12, 16, 20, 22, 24]

sage: Word('1213121').number_of_factors()
22
sage: Word('1213121').number_of_factors(1)
3

sage: Word('a'*100).number_of_factors()
101
sage: Word('a'*100).number_of_factors(77)
1

sage: Word().number_of_factors()
1
sage: Word().number_of_factors(17)
0

sage: blueberry = Word("blueberry")
sage: blueberry.number_of_factors()
43
sage: [blueberry.number_of_factors(i) for i in range(10)]
[1, 6, 8, 7, 6, 5, 4, 3, 2, 1]
```

number_of_inversions()

Return the number of inversions in `self`.

An inversion of a word $w = w_1 \dots w_n$ is a pair of indices (i, j) with $i < j$ and $w_i > w_j$.

See also:

`number of inversions on Permutations`.

EXAMPLES:

```
sage: w = Word([2,1,3,3,2])
sage: w.number_of_inversions()
3
```

number_of_left_special_factors(n)

Returns the number of left special factors of length n.

A factor u of a word w is *left special* if there are two distinct letters a and b such that au and bu are factors of w.

INPUT:

- n - integer

OUTPUT:

Non negative integer

EXAMPLES:

```
sage: w = words.FibonacciWord()[:100]
sage: [w.number_of_left_special_factors(i) for i in range(10)]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

sage: w = words.ThueMorseWord()[:100]
sage: [w.number_of_left_special_factors(i) for i in range(10)]
[1, 2, 2, 4, 2, 4, 4, 2, 2, 4]
```

number_of_right_special_factors(*n*)

Returns the number of right special factors of length n.

A factor u of a word w is *right special* if there are two distinct letters a and b such that ua and ub are factors of w.

INPUT:

- n - integer

OUTPUT:

Non negative integer

EXAMPLES:

```
sage: w = words.FibonacciWord()[:100]
sage: [w.number_of_right_special_factors(i) for i in range(10)]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

sage: w = words.ThueMorseWord()[:100]
sage: [w.number_of_right_special_factors(i) for i in range(10)]
[1, 2, 2, 4, 2, 4, 4, 2, 2, 4]
```

order()

Returns the order of self.

Let $p(w)$ be the period of a word w. The positive rational number $|w|/p(w)$ is the *order* of w. See Chapter 8 of [1].

OUTPUT:

rational – the order

EXAMPLES:

```
sage: Word('abaaba').order()
2
sage: Word('ababaaba').order()
8/5
sage: Word('a').order()
1
sage: Word('aa').order()
2
sage: Word().order()
0
```

REFERENCES:

- [1] M. Lothaire, Algebraic Combinatorics On Words, vol. 90 of Encyclopedia of Mathematics and its Applications, Cambridge University Press, U.K., 2002.

overlap_partition(*other*, *delay=0*, *p=None*, *involution=None*)

Returns the partition of the alphabet induced by the overlap of self and other with the given delay.

The partition of the alphabet is given by the equivalence relation obtained from the symmetric, reflexive and transitive closure of the set of pairs of letters $R_{u,v,d} = \{(u_k, v_{k-d}) : 0 \leq k < n, 0 \leq k - d < m\}$ where $u = u_0u_1 \cdots u_{n-1}$, $v = v_0v_1 \cdots v_{m-1}$ are two words on the alphabet A and d is an integer.

The equivalence relation defined by R is inspired from [1].

INPUT:

- `other` - word on the same alphabet as self
- `delay` - integer
- `p` - disjoint sets data structure (optional, default: None), a partition of the alphabet into disjoint sets to start with. If None, each letter start in distinct equivalence classes.
- `involution` - callable (optional, default: None), an involution on the alphabet. If involution is not None, the relation $R_{u,v,d} \cup R_{involution(u),involution(v),d}$ is considered.

OUTPUT:

- disjoint set data structure

EXAMPLES:

```
sage: W = Words(list('abc')+range(6))
sage: u = W('abc')
sage: v = W(range(5))
sage: u.overlap_partition(v)
{{0, 'a'}, {1, 'b'}, {2, 'c'}, {3}, {4}, {5}}
sage: u.overlap_partition(v, 2)
{{'a'}, {'b'}, {0, 'c'}, {1}, {2}, {3}, {4}, {5}}
sage: u.overlap_partition(v, -1)
{{0}, {1, 'a'}, {2, 'b'}, {3, 'c'}, {4}, {5}}
```

You can re-use the same disjoint set and do more than one overlap:

```
sage: p = u.overlap_partition(v, 2)
sage: p
{{'a'}, {'b'}, {0, 'c'}, {1}, {2}, {3}, {4}, {5}}
sage: u.overlap_partition(v, 1, p)
{{'a'}, {0, 1, 'b', 'c'}, {2}, {3}, {4}, {5}}
```

The function `overlap_partition` can be used to study equations on words. For example, if a word w overlaps itself with delay d, then d is a period of w:

```
sage: W = Words(range(20))
sage: w = W(range(14)); w
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13
sage: d = 5
sage: p = w.overlap_partition(w, d)
sage: m = WordMorphism(p.element_to_root_dict())
sage: w2 = m(w); w2
word: 56789567895678
sage: w2.minimal_period() == d
True
```

If a word is equal to its reversal, then it is a palindrome:

```
sage: W = Words(range(20))
sage: w = W(range(17)); w
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
sage: p = w.overlap_partition(w.reversal(), 0)
sage: m = WordMorphism(p.element_to_root_dict())
sage: w2 = m(w); w2
word: 01234567876543210
sage: w2.parent()
Finite words over {0, 1, 2, 3, 4, 5, 6, 7, 8, 17, 18, 19}
sage: w2.is_palindrome()
True
```

If the reversal of a word w is factor of its square w^2, then w is symmetric, i.e. the product of two palindromes:

```
sage: W = Words(range(10))
sage: w = W(range(10)); w
word: 0123456789
sage: p = (w*w).overlap_partition(w.reversal(), 4)
sage: m = WordMorphism(p.element_to_root_dict())
sage: w2 = m(w); w2
word: 0110456654
sage: w2.is_symmetric()
True
```

If the image of the reversal of a word w under an involution f is factor of its square w^2, then w is f-symmetric:

```
sage: W = Words([-11,-9,..,11])
sage: w = W([1,3,..,11])
sage: w
word: 1,3,5,7,9,11
sage: inv = lambda x:-x
sage: f = WordMorphism(dict( (a, inv(a)) for a in W.alphabet()))
sage: p = (w*w).overlap_partition(f(w).reversal(), 2, involution=f)
sage: m = WordMorphism(p.element_to_root_dict())
sage: m(w)
word: 1,-1,5,7,-7,-5
sage: m(w).is_symmetric(f)
True
```

TESTS:

```
sage: W = Words('abcdef')
sage: w = W('abc')
sage: y = W('def')
sage: w.overlap_partition(y, -3)
{{'a'}, {'b'}, {'c'}, {'d'}, {'e'}, {'f'}}
sage: w.overlap_partition(y, -2)
{{'a', 'f'}, {'b'}, {'c'}, {'d'}, {'e'}}
sage: w.overlap_partition(y, -1)
{{'a', 'e'}, {'b', 'f'}, {'c'}, {'d'}}
sage: w.overlap_partition(y, 0)
{{'a', 'd'}, {'b', 'e'}, {'c', 'f'}}
sage: w.overlap_partition(y, 1)
{{'a'}, {'b', 'd'}, {'c', 'e'}, {'f'}}
sage: w.overlap_partition(y, 2)
{{'a'}, {'b'}, {'c', 'd'}, {'e'}, {'f'}}
sage: w.overlap_partition(y, 3)
{{'a'}, {'b'}, {'c'}, {'d'}, {'e'}, {'f'}}
sage: w.overlap_partition(y, 4)
{{'a'}, {'b'}, {'c'}, {'d'}, {'e'}, {'f'}}

sage: W = Words(range(2))
sage: w = W([0,1,0,1,0,1]); w
word: 010101
sage: w.overlap_partition(w, 0)
{{0}, {1}}
sage: w.overlap_partition(w, 1)
{{0, 1}}
```

```
sage: empty = Word()
sage: empty.overlap_partition(empty, 'yo')
Traceback (most recent call last):
...
TypeError: delay (=yo) must be an integer
sage: empty.overlap_partition(empty,2,'yo')
Traceback (most recent call last):
...
TypeError: p(=yo) is not a DisjointSet
```

The involution input can be any callable:

```
sage: w = Words([-5,..,5])([-5..5])
sage: inv = lambda x:-x
sage: w.overlap_partition(w, 2, involution=inv)
{{-4, -2, 0, 2, 4}, {-5, -3, -1, 1, 3, 5}}
```

REFERENCES:

- [1] S. Labbé, Propriétés combinatoires des f-palindromes, Mémoire de maîtrise en Mathématiques, Montréal, UQAM, 2008, 109 pages.

palindrome_prefixes()

Returns a list of all palindrome prefixes of self.

OUTPUT:

list – A list of all palindrome prefixes of self.

EXAMPLES:

```
sage: w = Word('abaaba')
sage: w.palindrome_prefixes()
[word: , word: a, word: aba, word: abaaba]
sage: w = Word('abbbbbbbbbb')
sage: w.palindrome_prefixes()
[word: , word: a]
```

palindromes(*f=None*)

Returns the set of all palindromic (or f-palindromic) factors of self.

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be callable on letters as well as words (e.g. WordMorphism).

OUTPUT:

set - If f is None, the set of all palindromic factors of self; otherwise, the set of all f-palindromic factors of self.

EXAMPLES:

```
sage: sorted(Word('01101001').palindromes())
[word: , word: 0, word: 00, word: 010, word: 0110, word: 1, word: 1001, word: 101, word: 11]
sage: sorted(Word('00000').palindromes())
[word: , word: 0, word: 00, word: 000, word: 0000, word: 00000]
sage: sorted(Word('0').palindromes())
[word: , word: 0]
sage: sorted(Word('').palindromes())
[word: ]
sage: sorted(Word().palindromes())
```

```
[word: ]
sage: f = WordMorphism('a->b,b->a')
sage: sorted(Word('abbabaab').palindromes(f))
[word: , word: ab, word: abbabaab, word: ba, word: baba, word: bbabaa]
```

palindromic_closure(*side='right', f=None*)

Return the shortest palindrome having `self` as a prefix (or as a suffix if `side` is `'left'`).

See [1].

INPUT:

- `side` – `'right'` or `'left'` (default: `'right'`) the direction of the closure
- `f` – involution (default: `None`) on the alphabet of `self`. It must be callable on letters as well as words (e.g. WordMorphism).

OUTPUT:

word – If f is `None`, the right palindromic closure of `self`; otherwise, the right `f`-palindromic closure of `self`. If side is `'left'`, the left palindromic closure.

EXAMPLES:

```
sage: Word('1233').palindromic_closure()
word: 123321
sage: Word('12332').palindromic_closure()
word: 123321
sage: Word('0110343').palindromic_closure()
word: 01103430110
sage: Word('0110343').palindromic_closure(side='left')
word: 3430110343
sage: Word('01105678').palindromic_closure(side='left')
word: 876501105678
sage: w = Word('abbaba')
sage: w.palindromic_closure()
word: abbababba

sage: f = WordMorphism('a->b,b->a')
sage: w.palindromic_closure(f=f)
word: abbabaab
sage: w.palindromic_closure(f=f, side='left')
word: babaabbaba
```

TESTS:

```
sage: f = WordMorphism('a->c,c->a')
sage: w.palindromic_closure(f=f, side='left')
Traceback (most recent call last):
...
KeyError: 'b'
```

REFERENCES:

- [1] A. de Luca, A. De Luca, Pseudopalindrome closure operators in free monoids, Theoret. Comput. Sci. 362 (2006) 282–300.

palindromic_lacunas_study(*f=None*)

Returns interesting statistics about longest (f-)palindromic suffixes and lacunas of self (see [1] and [2]).

Note that a word w has at most $|w| + 1$ different palindromic factors (see [3]). For f-palindromes (or pseudopalidromes or theta-palindromes), the maximum number of f-palindromic factors is $|w|+1-g_f(w)$,

where $g_f(w)$ is the number of pairs $\{a, f(a)\}$ such that a is a letter, a is not equal to $f(a)$, and a or $f(a)$ occurs in w, see [4].

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be callable on letters as well as words (e.g. WordMorphism). The default value corresponds to usual palindromes, i.e., f equal to the identity.

OUTPUT:

- `list` - list of the length of the longest palindromic suffix (lps) for each non-empty prefix of self;
- `list` - list of all the lacunas, i.e. positions where there is no unioccurrent lps;
- `set` - set of palindromic factors of self.

EXAMPLES:

```
sage: a,b,c = Word('abbabaabbaab').palindromic_lacunas_study()
sage: a
[1, 1, 2, 4, 3, 3, 2, 4, 2, 4, 6, 8]
sage: b
[8, 9]
sage: c           # random order
set([word: , word: b, word: bab, word: abba, word: bb, word: aa, word: baabbaab, word: baab,

sage: f = WordMorphism('a->b,b->a')
sage: a,b,c = Word('abbabaab').palindromic_lacunas_study(f=f)
sage: a
[0, 2, 0, 2, 2, 4, 6, 8]
sage: b
[0, 2, 4]
sage: c            # random order
set([word: , word: ba, word: baba, word: ab, word: bbabaa, word: abbabaab])
sage: c == set([Word(), Word('ba'), Word('baba'), Word('ab'), Word('bbabaa'), Word('abbabaab
True
```

REFERENCES:

- [1] A. Blondin-Massé, S. Brlek, S. Labbé, Palindromic lacunas of the Thue-Morse word, Proc. GASCOM 2008 (June 16-20 2008, Bibbiena, Arezzo-Italia), 53–67.
- [2] A. Blondin-Massé, S. Brlek, A. Frosini, S. Labbé, S. Rinaldi, Reconstructing words from a fixed palindromic length sequence, Proc. TCS 2008, 5th IFIP International Conference on Theoretical Computer Science (September 8-10 2008, Milano, Italia), accepted.
- [3] X. Droubay, J. Justin, G. Pirillo, Episturmian words and some constructions of de Luca and Rauzy, Theoret. Comput. Sci. 255 (2001) 539–553.
- [4] Š. Starosta, On Theta-palindromic Richness, Theoret. Comp. Sci. 412 (2011) 1111–1121

parikh_vector(**args*, ***kwds*)

Deprecated: Use `abelian_vector()` instead. See trac ticket #17058 for details.

periods(*divide_length=False*)

Returns a list containing the periods of self between 1 and $n-1$, where n is the length of self.

INPUT:

- `divide_length` - boolean (default: False). When set to True, then only periods that divide the length of self are considered.

OUTPUT:

List of positive integers

EXAMPLES:

```
sage: w = Word('ababab')
sage: w.periods()
[2, 4]
sage: w.periods(divide_length=True)
[2]
sage: w = Word('ababa')
sage: w.periods()
[2, 4]
sage: w.periods(divide_length=True)
[]
```

phi()

Applies the phi function to self and returns the result. This is the word obtained by taking the first letter of the words obtained by iterating delta on self.

OUTPUT:

word – the result of the phi function

EXAMPLES:

```
sage: W = Words([1, 2])
sage: W([2,2,1,1,2,1,2,2,1,2,2,1,1,2]).phi()
word: 222222
sage: W([2,1,2,2,1,2,2,1,2,1]).phi()
word: 212113
sage: W().phi()
word:
sage: Word([2,1,2,2,1,2,2,1,2,1]).phi()
word: 212113
sage: Word([2,3,1,1,2,1,2,3,1,2,2,3,1,2]).phi()
word: 21215
sage: Word("aabbabaabaabba").phi()
word: a22222
sage: w = Word([2,3,1,1,2,1,2,3,1,2,2,3,1,2])
```

REFERENCES:

- S. Brlek, A. Ladouceur, A note on differentiable palindromes, Theoret. Comput. Sci. 302 (2003) 167–178.
- S. Brlek, S. Dulucq, A. Ladouceur, L. Vuillon, Combinatorial properties of smooth infinite words, Theoret. Comput. Sci. 352 (2006) 306–317.

phi_inv(*W=None*)

Apply the inverse of the phi function to self.

INPUT:

- `self` - a word over the integers
- `W` - a parent object of words defined over integers

OUTPUT:

word – the inverse of the phi function

EXAMPLES:

```
sage: W = Words([1, 2])
sage: W([2, 2, 2, 2, 1, 2]).phi_inv()
word: 22112122
sage: W([2, 2, 2]).phi_inv(Words([2, 3]))
word: 2233
```

prefix_function_table()

Returns a vector containing the length of the proper prefix-suffixes for all the non-empty prefixes of self.

EXAMPLES:

```
sage: Word('121321').prefix_function_table()
[0, 0, 1, 0, 0, 1]
sage: Word('1241245').prefix_function_table()
[0, 0, 0, 1, 2, 3, 0]
sage: Word().prefix_function_table()
[]
```

primitive()

Returns the primitive of self.

EXAMPLES:

```
sage: Word('12312').primitive()
word: 12312
sage: Word('121212').primitive()
word: 12
```

primitive_length()

Returns the length of the primitive of self.

EXAMPLES:

```
sage: Word('1231').primitive_length()
4
sage: Word('121212').primitive_length()
2
```

quasiperiods()

Returns the quasiperiods of self as a list ordered from shortest to longest.

Let w be a finite or infinite word. A *quasiperiod* of w is a proper factor u of w such that the occurrences of u in w entirely cover w, i.e., every position of w falls within some occurrence of u in w. See for instance [1], [2], and [3].

EXAMPLES:

```
sage: Word('abaababaabaababababaaba').quasiperiods()
[word: aba, word: abaaba, word: abaababaaba]
sage: Word('abaaba').quasiperiods()
[word: aba]
sage: Word('abacaba').quasiperiods()
[]
```

REFERENCES:

- [1] A. Apostolico, A. Ehrenfeucht, Efficient detection of quasiperiodicities in strings, Theoret. Comput. Sci. 119 (1993) 247–265.
- [2] S. Marcus, Quasiperiodic infinite words, Bull. Eur. Assoc. Theor. Comput. Sci. 82 (2004) 170-174.

- [3] A. Glen, F. Levé, G. Richomme, Quasiperiodic and Lyndon episturmian words, Preprint, 2008, arXiv:0805.0730.

rauzy_graph(*n*)

Returns the Rauzy graph of the factors of length n of self.

The vertices are the factors of length n and there is an edge from u to v if $ua = bv$ is a factor of length $n + 1$ for some letters a and b.

INPUT:

- n - integer

EXAMPLES:

```
sage: w = Word(range(10)); w
word: 0123456789
sage: g = w.rauzy_graph(3); g
Looped digraph on 8 vertices
sage: WordOptions(identifier='')
sage: g.vertices()
[012, 123, 234, 345, 456, 567, 678, 789]
sage: g.edges()
[(012, 123, 3),
 (123, 234, 4),
 (234, 345, 5),
 (345, 456, 6),
 (456, 567, 7),
 (567, 678, 8),
 (678, 789, 9)]
sage: WordOptions(identifier='word: ')

sage: f = words.FibonacciWord()[:100]
sage: f.rauzy_graph(8)
Looped digraph on 9 vertices

sage: w = Word('1111111')
sage: g = w.rauzy_graph(3)
sage: g.edges()
[(word: 111, word: 111, word: 1)]

sage: w = Word('111')
sage: for i in range(5) : w.rauzy_graph(i)
Looped multi-digraph on 1 vertex
Looped digraph on 1 vertex
Looped digraph on 1 vertex
Looped digraph on 1 vertex
Looped digraph on 0 vertices
```

Multi-edges are allowed for the empty word:

```
sage: W = Words('abcde')
sage: w = W('abc')
sage: w.rauzy_graph(0)
Looped multi-digraph on 1 vertex
sage: _.edges()
[(word: , word: , word: a),
 (word: , word: , word: b),
 (word: , word: , word: c)]
```

reduced_rauzy_graph(*n*)

Returns the reduced Rauzy graph of order n of self.

INPUT:

- n - non negative integer. Every vertex of a reduced Rauzy graph of order n is a factor of length n of self.

OUTPUT:

Looped multi-digraph

DEFINITION:

For infinite periodic words (resp. for finite words of type $u^i u[0:j]$), the reduced Rauzy graph of order n (resp. for n smaller or equal to $(i-1)|u|+j$) is the directed graph whose unique vertex is the prefix p of length n of self and which has an only edge which is a loop on p labelled by $w[n+1:|w|]p$ where w is the unique return word to p.

In other cases, it is the directed graph defined as followed. Let G_n be the Rauzy graph of order n of self. The vertices are the vertices of G_n that are either special or not prolongable to the right or to the left. For each couple (u, v) of such vertices and each directed path in G_n from u to v that contains no other vertices that are special, there is an edge from u to v in the reduced Rauzy graph of order n whose label is the label of the path in G_n.

Note: In the case of infinite recurrent non periodic words, this definition correspond to the following one that can be found in [1] and [2] where a simple path is a path that begins with a special factor, ends with a special factor and contains no other vertices that are special:

The reduced Rauzy graph of factors of length n is obtained from G_n by replacing each simple path $P = v_1 v_2 ... v_\ell$ with an edge $v_1 v_\ell$ whose label is the concatenation of the labels of the edges of P.

EXAMPLES:

```
sage: w = Word(range(10)); w
word: 0123456789
sage: g = w.reduced_rauzy_graph(3); g
Looped multi-digraph on 2 vertices
sage: g.vertices()
[word: 012, word: 789]
sage: g.edges()
[(word: 012, word: 789, word: 3456789)]
```

For the Fibonacci word:

```
sage: f = words.FibonacciWord()[:100]
sage: g = f.reduced_rauzy_graph(8);g
Looped multi-digraph on 2 vertices
sage: g.vertices()
[word: 01001010, word: 01010010]
sage: g.edges()
[(word: 01001010, word: 01010010, word: 010), (word: 01010010, word: 01001010, word: 01010),
```

For periodic words:

```
sage: from itertools import cycle
sage: w = Word(cycle('abcd'))[:100]
sage: g = w.reduced_rauzy_graph(3)
sage: g.edges()
[(word: abc, word: abc, word: dabc)]
```

```
sage: w = Word('111')
sage: for i in range(5) : w.reduced_rauzy_graph(i)
Looped digraph on 1 vertex
Looped digraph on 1 vertex
Looped digraph on 1 vertex
Looped multi-digraph on 1 vertex
Looped multi-digraph on 0 vertices
```

For ultimately periodic words:

```
sage: sigma = WordMorphism('a->abcd,b->cd,c->cd,d->cd')
sage: w = sigma.fixed_point('a')[:100]; w
word: abcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcdcd...
sage: g = w.reduced_rauzy_graph(5)
sage: g.vertices()
[word: abcdc, word: cdcdc]
sage: g.edges()
[(word: abcdc, word: cdcdc, word: dc), (word: cdcdc, word: cdcdc, word: dc)]
```

AUTHOR:

Julien Leroy (March 2010): initial version

REFERENCES:

- [1] M. Bucci et al. A. De Luca, A. Glen, L. Q. Zamboni, A connection between palindromic and factor complexity using return words," Advances in Applied Mathematics 42 (2009) 60-74.
- [2] L'ubomira Balkova, Edita Pelantova, and Wolfgang Steiner. Sequences with constant number of return words. Monatsh. Math, 155 (2008) 251-263.

return_words (*fact*)

Returns the set of return words of fact in self.

This is the set of all factors starting by the given factor and ending just before the next occurrence of this factor. See [1] and [2].

INPUT:

- `fact` - a non empty finite word

OUTPUT:

Python set of finite words

EXAMPLES:

```
sage: Word('21331233213231').return_words(Word('2'))
{word: 213, word: 21331, word: 233}
sage: Word().return_words(Word('213'))
set()
sage: Word('121212').return_words(Word('1212'))
{word: 12}

sage: TM = words.ThueMorseWord()[:1000]
sage: sorted(TM.return_words(Word([0])))
[word: 0, word: 01, word: 011]
```

REFERENCES:

- [1] F. Durand, A characterization of substitutive sequences using return words, Discrete Math. 179 (1998) 89-101.

- [2] C. Holton, L.Q. Zamboni, Descendants of primitive substitutions, Theory Comput. Syst. 32 (1999) 133-157.

return_words_derivate (*fact*)

Returns the word generated by mapping a letter to each occurrence of the return words for the given factor dropping any dangling prefix and suffix. See for instance [1].

EXAMPLES:

```
sage: Word('12131221312313122').return_words_derivate(Word('1'))
word: 123242
```

REFERENCES:

- [1] F. Durand, A characterization of substitutive sequences using return words, Discrete Math. 179 (1998) 89–101.

rev_lex_less (*other*)

Returns True if the word self is reverse lexicographically less than other.

EXAMPLES:

```
sage: Word([1,2,4]).rev_lex_less(Word([1,3,2]))
True
sage: Word([3,2,1]).rev_lex_less(Word([1,2,3]))
False
```

reversal ()

Returns the reversal of self.

EXAMPLES:

```
sage: Word('124563').reversal()
word: 365421
```

rfind (*sub*, *start=0*, *end=None*)

Returns the index of the last occurrence of sub in self, such that sub is contained within self[start:end]. Returns -1 on failure.

INPUT:

- `sub` - string, list, tuple or word to search for.
- `start` - non negative integer (default: 0) specifying the position at which the search must stop.
- `end` - non negative integer (default: None) specifying the position from which to start the search. If None, then the search is performed up to the end of the string.

OUTPUT:

non negative integer or -1

EXAMPLES:

```
sage: w = Word([0,1,0,0,1])
sage: w.rfind(Word([0,1]))
3
```

The `sub` parameter can also be a list or a tuple:

```
sage: w.rfind([0,1])
3
sage: w.rfind((0,1))
3
```

Examples using the argument `start` and `end`:

```
sage: w.rfind(Word([0,1]), end=4)
0
sage: w.rfind(Word([0,1]), end=5)
3
sage: w.rfind(Word([0,0]), start=2, end=5)
2
sage: w.rfind(Word([0,0]), start=3, end=5)
-1
```

Instances of Word_str handle string inputs as well:

```
sage: w = Word('abac')
sage: w.rfind('a')
2
sage: w.rfind(Word('a'))
2
sage: w.rfind([0,1])
-1
```

TESTS:

Check that trac ticket #12804 is fixed:

```
sage: w = Word(iter("abab"), length="finite")
sage: w.rfind("ab")
2
sage: w.rfind("ab", end=3)
0
sage: w.rfind("aa")
-1
sage: w.rfind([0,0,0])
-1
```

right_special_factors(*n=None*)

Returns the right special factors (of length n).

A factor u of a word w is *right special* if there are two distinct letters a and b such that ua and ub are factors of w.

INPUT:

- n - integer (optional, default: None). If None, it returns all right special factors.

OUTPUT:

A list of words.

EXAMPLES:

```
sage: w = words.ThueMorseWord()[:30]
sage: for i in range(5): print i, sorted(w.right_special_factors(i))
0 [word: ]
1 [word: 0, word: 1]
2 [word: 01, word: 10]
3 [word: 001, word: 010, word: 101, word: 110]
4 [word: 0110, word: 1001]
```

right_special_factors_iterator(*n=None*)

Returns an iterator over the right special factors (of length n).

A factor u of a word w is *right special* if there are two distinct letters a and b such that ua and ub are factors of w.

INPUT:

- n - integer (optional, default: None). If None, it returns an iterator over all right special factors.

EXAMPLES:

```
sage: alpha, beta, x = 0.61, 0.54, 0.3
sage: w = words.CodingOfRotationWord(alpha, beta, x)[:40]
sage: sorted(w.right_special_factors_iterator(3))
[word: 010, word: 101]
sage: sorted(w.right_special_factors_iterator(4))
[word: 0101, word: 1010]
sage: sorted(w.right_special_factors_iterator(5))
[word: 00101, word: 11010]
```

robinson_schensted()

Return the semistandard tableau and standard tableau pair obtained by running the Robinson-Schensted algorithm on self.

This can also be done by running RSK() on self.

EXAMPLES:

```
sage: Word([1,1,3,1,2,3,1]).robinson_schensted()
[[[1, 1, 1, 1, 3], [2], [3]], [[1, 2, 3, 5, 6], [4], [7]]]
```

schuetzenberger_involution(*n=None*)

Returns the Schuetzenberger involution of the word self, which is obtained by reverting the word and then complementing all letters within the underlying ordered alphabet. If n is specified, the underlying alphabet is assumed to be $[1, 2, \ldots, n]$. If no alphabet is specified, n is the maximal letter appearing in self.

INPUT:

- self – a word
- n – an integer specifying the maximal letter in the alphabet (optional)

OUTPUT:

- a word, the Schuetzenberger involution of self

EXAMPLES:

```
sage: w = Word([9,7,4,1,6,2,3])
sage: v = w.schuetzenberger_involution(); v
word: 7849631
sage: v.parent()
Finite words over Set of Python objects of type 'object'

sage: w = Word([1,2,3],alphabet=[1,2,3,4,5])
sage: v = w.schuetzenberger_involution();v
word: 345
sage: v.parent()
Finite words over {1, 2, 3, 4, 5}

sage: w = Word([1,2,3])
sage: v = w.schuetzenberger_involution(n=5);v
word: 345
```

```
sage: v.parent()
Finite words over Set of Python objects of type 'object'

sage: w = Word([11,32,69,2,53,1,2,3,18,41])
sage: w.schuetzenberger_involution()
word: 29,52,67,68,69,17,68,1,38,59

sage: w = Word([],alphabet=[1,2,3,4,5])
sage: w.schuetzenberger_involution()
word:

sage: w = Word([])
sage: w.schuetzenberger_involution()
word:
```

shifted_shuffle(*other*, *shift=None*)

Returns the combinatorial class representing the shifted shuffle product between words self and other. This is the same as the shuffle product of self with the word obtained from other by incrementing its values (i.e. its letters) by the given shift.

INPUT:

- `other` - finite word over the integers
- `shift` - integer or None (default: None) added to each letter of other. When shift is None, it is replaced by self.length()

OUTPUT:

Combinatorial class of shifted shuffle products of self and other.

EXAMPLES:

```
sage: w = Word([0,1,1])
sage: sp = w.shifted_shuffle(w); sp
Shuffle product of word: 011 and word: 344
sage: sp = w.shifted_shuffle(w, 2); sp
Shuffle product of word: 011 and word: 233
sage: sp.cardinality()
20
sage: WordOptions(identifier='')
sage: sp.list()
[011233, 012133, 012313, 012331, 021133, 021313, 021331, 023113, 023131, 023311, 201133, 201
sage: WordOptions(identifier='word: ')
sage: y = Word('aba')
sage: y.shifted_shuffle(w,2)
Traceback (most recent call last):
...
ValueError: for shifted shuffle, words must only contain integers as letters
```

shuffle(*other*, *overlap=0*)

Returns the combinatorial class representing the shuffle product between words self and other. This consists of all words of length self.length()+other.length() that have both self and other as subwords.

If overlap is non-zero, then the combinatorial class representing the shuffle product with overlaps is returned. The calculation of the shift in each overlap is done relative to the order of the alphabet. For example, "a" shifted by "a" is "b" in the alphabet [a, b, c] and 0 shifted by 1 in [0, 1, 2, 3] is 2.

INPUT:

- `other` - finite word
- `overlap` - (default: 0) integer or True

OUTPUT:

Combinatorial class of shuffle product of self and other

EXAMPLES:

```
sage: ab = Word("ab")
sage: cd = Word("cd")
sage: sp = ab.shuffle(cd); sp
Shuffle product of word: ab and word: cd
sage: sp.cardinality()
6
sage: sp.list()
[word: abcd, word: acbd, word: acdb, word: cabd, word: cadb, word: cdab]
sage: w = Word([0,1])
sage: u = Word([2,3])
sage: w.shuffle(w)
Shuffle product of word: 01 and word: 01
sage: u.shuffle(u)
Shuffle product of word: 23 and word: 23
sage: w.shuffle(u)
Shuffle product of word: 01 and word: 23
sage: w.shuffle(u,2)
Overlapping shuffle product of word: 01 and word: 23 with 2 overlaps
```

standard_factorization()

Return the standard factorization of `self`.

The *standard factorization* of a word w of length greater than 1 is the factorization $w = uv$ where v is the longest proper suffix of w that is a Lyndon word.

Note that if w is a Lyndon word of length greater than 1 with standard factorization $w = uv$, then u and v are also Lyndon words and $u < v$.

See for instance [1], [2] and [3].

INPUT:

- `self` - finite word of length greater than 1

OUTPUT:

2-tuple (u, v)

EXAMPLES:

```
sage: Words('01')('0010110011').standard_factorization()
(word: 001011, word: 0011)
sage: Words('123')('1223312').standard_factorization()
(word: 12233, word: 12)
sage: Word([3,2,1]).standard_factorization()
(word: 32, word: 1)

sage: w = Word('0010110011',alphabet='01')
sage: w.standard_factorization()
(word: 001011, word: 0011)
sage: w = Word('0010110011',alphabet='10')
sage: w.standard_factorization()
(word: 001011001, word: 1)
sage: w = Word('1223312',alphabet='123')
```

```
sage: w.standard_factorization()
(word: 12233, word: 12)
```

TESTS:

```
sage: w = Word()
sage: w.standard_factorization()
Traceback (most recent call last):
...
ValueError: Standard factorization not defined on words of
length less than 2
sage: w = Word('a')
sage: w.standard_factorization()
Traceback (most recent call last):
...
ValueError: Standard factorization not defined on words of
length less than 2
```

REFERENCES:

- [1] K.-T. Chen, R.H. Fox, R.C. Lyndon, Free differential calculus, IV. The quotient groups of the lower central series, Ann. of Math. 68 (1958) 81–95.
- [2] J.-P. Duval, Factorizing words over an ordered alphabet, J. Algorithms 4 (1983) 363–381.
- [3] M. Lothaire, Algebraic Combinatorics On Words, vol. 90 of Encyclopedia of Mathematics and its Applications, Cambridge University Press, U.K., 2002.

standard_permutation()

Returns the standard permutation of the word self on the ordered alphabet. It is defined as the permutation with exactly the same number of inversions as w. Equivalently, it is the permutation of minimal length whose inverse sorts self.

EXAMPLES:

```
sage: w = Word([1,2,3,2,2,1]); w
word: 123221
sage: p = w.standard_permutation(); p
[1, 3, 6, 4, 5, 2]
sage: v = Word(p.inverse().action(w)); v
word: 112223
sage: filter(lambda q: q.length() <= p.length() and \
....:        q.inverse().action(w) == list(v), \
....:        Permutations(w.length()) )
[[1, 3, 6, 4, 5, 2]]

sage: w = Words([1,2,3])([1,2,3,2,2,1,2,1]); w
word: 12322121
sage: p = w.standard_permutation(); p
[1, 4, 8, 5, 6, 2, 7, 3]
sage: Word(p.inverse().action(w))
word: 11122223

sage: w = Words([3,2,1])([1,2,3,2,2,1,2,1]); w
word: 12322121
sage: p = w.standard_permutation(); p
[6, 2, 1, 3, 4, 7, 5, 8]
sage: Word(p.inverse().action(w))
word: 32222111
```

```
sage: w = Words('ab')('abbaba'); w
word: abbaba
sage: p = w.standard_permutation(); p
[1, 4, 5, 2, 6, 3]
sage: Word(p.inverse().action(w))
word: aaabbb

sage: w = Words('ba')('abbaba'); w
word: abbaba
sage: p = w.standard_permutation(); p
[4, 1, 2, 5, 3, 6]
sage: Word(p.inverse().action(w))
word: bbbaaa
```

sturmian_desubstitute_as_possible()

Sturmian desubstitutes the word `self` as much as possible.

The finite word `self` must be defined on a two-letter alphabet or use at most two-letters.

It can be Sturmian desubstituted if one letter appears isolated: the Sturmian desubstitution consists in removing one letter per run of the non-isolated letter. The accelerated Sturmian desubstitution consists in removing a run equal to the length of the shortest inner run from any run of the non-isolated letter (including possible leading and trailing runs even if they have shorter length). The (accelerated) Sturmian desubstitution is done as much as possible. A word is a factor of a Sturmian word if, and only if, the result is the empty word.

OUTPUT:

- A finite word defined on a two-letter alphabet.

EXAMPLES:

```
sage: u = Word('10111101101110111',alphabet='01') ; u
word: 10111101101110111
sage: v = u.sturmian_desubstitute_as_possible() ; v
word: 01100101
sage: v == v.sturmian_desubstitute_as_possible()
True

sage: Word('azaazaaazaaazaazaaaz', alphabet='az').sturmian_desubstitute_as_possible()
word:
```

TESTS:

```
sage: w = Word('azazaza', alphabet='aze')
sage: w.sturmian_desubstitute_as_possible()
word:
sage: Word('aze').sturmian_desubstitute_as_possible()
Traceback (most recent call last):
...
TypeError: your word must be defined on a binary alphabet or use at most two different lette
sage: Word('azaaazaazaazaaazaaza', alphabet='az').sturmian_desubstitute_as_possible()
word:
sage: Word('azaaazaazaazaaazaaaza', alphabet='az').sturmian_desubstitute_as_possible()
word: azzaa
```

Boundary effects:

```
sage: Word('', alphabet='az').sturmian_desubstitute_as_possible()
word:
sage: Word('azzzzz', alphabet='az').sturmian_desubstitute_as_possible()
```

```
word:
sage: Word('zzzza', alphabet='az').sturmian_desubstitute_as_possible()
word:
sage: Word('aaaaazaaaaaaaaa', alphabet='az').sturmian_desubstitute_as_possible()
word:
sage: Word('aaaaaaaaaaaaaa', alphabet='az').sturmian_desubstitute_as_possible()
word:
```

Boundary effects without alphabet:

```
sage: Word('').sturmian_desubstitute_as_possible()
word:
sage: Word('azzzzz').sturmian_desubstitute_as_possible()
word:
sage: Word('zzzza').sturmian_desubstitute_as_possible()
word:
sage: Word('aaaaazaaaaaaaaa').sturmian_desubstitute_as_possible()
word:
sage: Word('aaaaaaaaaaaaaa').sturmian_desubstitute_as_possible()
word:
```

Idempotence:

```
sage: r = words.RandomWord(randint(1,15)).sturmian_desubstitute_as_possible() ; r == r.sturm
True
```

AUTHOR:

- Thierry Monteil

suffix_tree()

Alias for implicit_suffix_tree().

EXAMPLES:

```
sage: Word('abbabaab').suffix_tree()
Implicit Suffix Tree of the word: abbabaab
```

suffix_trie()

Returns the suffix trie of self.

The *suffix trie* of a finite word w is a data structure representing the factors of w. It is a tree whose edges are labelled with letters of w, and whose leafs correspond to suffixes of w.

See sage.combinat.words.suffix_trees.SuffixTrie? for more information.

EXAMPLES:

```
sage: w = Word("cacao")
sage: w.suffix_trie()
Suffix Trie of the word: cacao

sage: w = Word([0,1,0,1,1])
sage: w.suffix_trie()
Suffix Trie of the word: 01011
```

swap(*i*, *j=None*)

Returns the word w with entries at positions i and j swapped. By default, j = i+1.

EXAMPLES:

```
sage: Word([1,2,3]).swap(0,2)
word: 321
sage: Word([1,2,3]).swap(1)
word: 132
sage: Word("abba").swap(1,-1)
word: aabb
```

swap_decrease(*i*)

Returns the word with positions i and i+1 exchanged if self[i] < self[i+1]. Otherwise, it returns self.

EXAMPLES:

```
sage: w = Word([1,3,2])
sage: w.swap_decrease(0)
word: 312
sage: w.swap_decrease(1)
word: 132
sage: w.swap_decrease(1) is w
True
sage: Words("ab")("abba").swap_decrease(0)
word: baba
sage: Words("ba")("abba").swap_decrease(0)
word: abba
```

swap_increase(*i*)

Returns the word with positions i and i+1 exchanged if self[i] > self[i+1]. Otherwise, it returns self.

EXAMPLES:

```
sage: w = Word([1,3,2])
sage: w.swap_increase(1)
word: 123
sage: w.swap_increase(0)
word: 132
sage: w.swap_increase(0) is w
True
sage: Words("ab")("abba").swap_increase(0)
word: abba
sage: Words("ba")("abba").swap_increase(0)
word: baba
```

to_integer_list()

Returns a list of integers from [0,1,...,self.length()-1] in the same relative order as the letters in self in the parent.

EXAMPLES:

```
sage: from itertools import count
sage: w = Word('abbabaab')
sage: w.to_integer_list()
[0, 1, 1, 0, 1, 0, 0, 1]
sage: w = Word(iter("cacao"), length="finite")
sage: w.to_integer_list()
[1, 0, 1, 0, 2]
sage: w = Words([3,2,1])([2,3,3,1])
sage: w.to_integer_list()
[1, 0, 0, 2]
```

to_integer_word()

Returns a word defined over the integers [0,1,...,self.length()-1] whose letters are in the same relative order

in the parent.

EXAMPLES:

```
sage: from itertools import count
sage: w = Word('abbabaab')
sage: w.to_integer_word()
word: 01101001
sage: w = Word(iter("cacao"), length="finite")
sage: w.to_integer_word()
word: 10102

sage: w = Words([3,2,1])([2,3,3,1])
sage: w.to_integer_word()
word: 1002
```

to_monoid_element()

Return `self` as an element the free monoid with the same alphabet as `self`.

EXAMPLES:

```
sage: w = Word('aabb')
sage: w.to_monoid_element()
a^2*b^2
sage: W = Words('abc')
sage: w = W(w)
sage: w.to_monoid_element()
a^2*b^2
```

TESTS:

Check that `w == w.to_monoid_element().to_word()`:

```
sage: all(w.to_monoid_element().to_word() == w for i in range(6) for w in Words('abc', i))
True
```

topological_entropy(n)

Return the topological entropy for the factors of length n.

The topological entropy of a sequence u is defined as the exponential growth rate of the complexity of u as the length increases: $H_{top}(u) = \lim_{n\to\infty} \frac{\log_d(p_u(n))}{n}$ where d denotes the cardinality of the alphabet and $p_u(n)$ is the complexity function, i.e. the number of factors of length n in the sequence u [1].

INPUT:

- `self` - a word defined over a finite alphabet
- `n` - positive integer

OUTPUT:

real number (a symbolic expression)

EXAMPLES:

```
sage: W = Words([0, 1])
sage: w = W([0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1])
sage: t = w.topological_entropy(3); t
1/3*log(7)/log(2)
sage: n(t)
0.935784974019201

sage: w = words.ThueMorseWord()[:100]
sage: topo = w.topological_entropy
```

```
sage: for i in range(0, 41, 5): print i, n(topo(i), digits=5)
0 1.0000
5 0.71699
10 0.48074
15 0.36396
20 0.28774
25 0.23628
30 0.20075
35 0.17270
40 0.14827
```

If no alphabet is specified, an error is raised:

```
sage: w = Word(range(20))
sage: w.topological_entropy(3)
Traceback (most recent call last):
...
TypeError: The word must be defined over a finite alphabet
```

The following is ok:

```
sage: W = Words(range(20))
sage: w = W(range(20))
sage: w.topological_entropy(3)
1/3*log(18)/log(20)
```

REFERENCES:

[1] N. Pytheas Fogg, Substitutions in Dynamics, Arithmetics, and Combinatorics, Lecture Notes in Mathematics 1794, Springer Verlag. V. Berthe, S. Ferenczi, C. Mauduit and A. Siegel, Eds. (2002).

5.1.323 Infinite word

AUTHORS:

- Sebastien Labbe
- Franco Saliola

EXAMPLES:

Creation of an infinite word

Periodic infinite words:

```
sage: v = Word([0, 4, 8, 8, 3])
sage: vv = v^Infinity
sage: vv
word: 048830488304883048830488304883048830488304883...
```

Infinite words from a function $f : \mathbb{N} \to A$ over an alphabet A:

```
sage: Word(lambda n: n%3)
word: 012012012012012012012012012012012012012012012...
```

```
sage: def t(n):
...       return add(Integer(n).digits(base=2)) % 2
sage: Word(t, alphabet = [0, 1])
word: 01101001100101101001011001101001100101 10...
```

or as a one-liner:

```
sage: Word(lambda n : add(Integer(n).digits(base=2)) % 2, alphabet = [0, 1])
word: 01101001100101101001011001101001100101 10...
```

Infinite words from iterators:

```
sage: from itertools import count,repeat
sage: Word( repeat(4) )
word: 44444444444444444444444444444444444444 44...
sage: Word( count() )
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,
```

Infinite words from morphism

For example, let $A = \{a, b\}$ and $\mu : A^* \to A^*$ be the morphism defined by $a \mapsto ab, b \mapsto ba$:

```
sage: mu = WordMorphism('a->ab,b->ba'); mu
WordMorphism: a->ab, b->ba
sage: mu.fixed_point('a')
word: abbabaabbaababbabaababbaabbabaabbaababba...
```

Infinite words in a specific combinatorial class:

```
sage: W = InfiniteWords("ab"); W
Infinite words over {'a', 'b'}
sage: f = lambda n : 'a' if n % 2 == 1 else 'b'
sage: W(f)
word: bababababababababababababababababababa...
```

class sage.combinat.words.infinite_word.**InfiniteWord_class**

Bases: sage.combinat.words.abstract_word.Word_class

length()

Returns the length of self.

EXAMPLES:

```
sage: f = lambda n : n % 6
sage: w = Word(f); w
word: 01234501234501234501234501234501234501 23...
sage: w.length()
+Infinity
```

5.1.324 Word morphisms/substitutions

This modules implements morphisms over finite and infinite words.

AUTHORS:

- Sebastien Labbe (2007-06-01): initial version
- Sebastien Labbe (2008-07-01): merged into sage-words

- Sebastien Labbe (2008-12-17): merged into sage
- Sebastien Labbe (2009-02-03): words next generation
- Sebastien Labbe (2009-11-20): allowing the choice of the datatype of the image. Doc improvements.
- Stepan Starosta (2012-11-09): growing letters

EXAMPLES:

Creation of a morphism from a dictionary or a string:

```
sage: n = WordMorphism({0:[0,2,2,1],1:[0,2],2:[2,2,1]})

sage: m = WordMorphism('x->xyxsxss,s->xyss,y->ys')

sage: n
WordMorphism: 0->0221, 1->02, 2->221
sage: m
WordMorphism: s->xyss, x->xyxsxss, y->ys
```

The codomain may be specified:

```
sage: WordMorphism({0:[0,2,2,1],1:[0,2],2:[2,2,1]}, codomain=Words([0,1,2,3,4]))
WordMorphism: 0->0221, 1->02, 2->221
```

Power of a morphism:

```
sage: n^2
WordMorphism: 0->022122122102, 1->0221221, 2->22122102
```

Image under a morphism:

```
sage: m('y')
word: ys
sage: m('xxxsy')
word: xyxsxssxyxsxssxyxsxssxyssys
```

Iterated image under a morphism:

```
sage: m('y', 3)
word: ysxyssxyxsxssysxyssxyss
```

Infinite fixed point of morphism:

```
sage: fix = m.fixed_point('x')
sage: fix
word: xyxsxssysxyxsxssxyssxyxsxssxyssxyssysxys...
sage: fix.length()
+Infinity
```

Incidence matrix:

```
sage: matrix(m)
[2 3 1]
[1 3 0]
[1 1 1]
```

Many other functionalities...:

```
sage: m.is_identity()
False
sage: m.is_endomorphism()
True
```

class sage.combinat.words.morphism.**PeriodicPointIterator**(*m*, *cycle*)

Bases: object

(Lazy) constructor of the periodic points of a word morphism.

This class is mainly used in WordMorphism.periodic_point and WordMorphism.periodic_points.

EXAMPLES:

```
sage: from sage.combinat.words.morphism import PeriodicPointIterator
sage: s = WordMorphism('a->bacca,b->cba,c->aab')
sage: p = PeriodicPointIterator(s, ['a','b','c'])
sage: p._cache[0]
lazy list ['a', 'a', 'b', ...]
sage: p._cache[1]
lazy list ['b', 'a', 'c', ...]
sage: p._cache[2]
lazy list ['c', 'b', 'a', ...]
```

get_iterator(*i*)

Internal method.

EXAMPLES:

```
sage: from sage.combinat.words.morphism import PeriodicPointIterator
sage: s = WordMorphism('a->bacca,b->cba,c->aab')
sage: p = PeriodicPointIterator(s, ['a','b','c'])
sage: p.get_iterator(0)
<generator object get_iterator at ...>
```

class sage.combinat.words.morphism.**WordMorphism**(*data*, *domain=None*, *codomain=None*)

Bases: sage.structure.sage_object.SageObject

WordMorphism class

EXAMPLES:

```
sage: n = WordMorphism({0:[0,2,2,1],1:[0,2],2:[2,2,1]})
sage: m = WordMorphism('x->xyxsxss,s->xyss,y->ys')
```

Power of a morphism:

```
sage: n^2
WordMorphism: 0->022122122102, 1->0221221, 2->22122102
```

Image under a morphism:

```
sage: m('y')
word: ys
sage: m('xxxsy')
word: xyxsxssxyxsxssxyxsxssxyssys
```

Iterated image under a morphism:

```
sage: m('y', 3)
word: ysxyssxyxsxssysxyssxyss
```

See more examples in the documentation of the call method (`m.__call__?`).

Infinite fixed point of morphism:

```
sage: fix = m.fixed_point('x')
sage: fix
word: xyxsxssysxyxsxssxyssxyxsxssxyssxyssysxys...
sage: fix.length()
+Infinity
```

Incidence matrix:

```
sage: matrix(m)
[2 3 1]
[1 3 0]
[1 1 1]
```

Many other functionalities...:

```
sage: m.is_identity()
False
sage: m.is_endomorphism()
True
```

TESTS:

```
sage: wm = WordMorphism('a->ab,b->ba')
sage: wm == loads(dumps(wm))
True
```

abelian_rotation_subspace()

Returns the subspace on which the incidence matrix of `self` acts by roots of unity.

EXAMPLES:

```
sage: WordMorphism('0->1,1->0').abelian_rotation_subspace()
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
sage: WordMorphism('0->01,1->10').abelian_rotation_subspace()
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
sage: WordMorphism('0->01,1->1').abelian_rotation_subspace()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]
sage: WordMorphism('1->122,2->211').abelian_rotation_subspace()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[ 1 -1]
sage: WordMorphism('0->1,1->102,2->3,3->4,4->2').abelian_rotation_subspace()
Vector space of degree 5 and dimension 3 over Rational Field
Basis matrix:
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

The domain needs to be equal to the codomain:

```
sage: WordMorphism('0->1,1->',codomain=Words('01')).abelian_rotation_subspace()
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

codomain()

Returns the codomain of `self`.

EXAMPLES:

```
sage: WordMorphism('a->ab,b->a').codomain()
Finite words over {'a', 'b'}
sage: WordMorphism('6->ab,y->5,0->asd').codomain()
Finite words over {'5', 'a', 'b', 'd', 's'}
```

conjugate(*pos*)

Returns the morphism where the image of the letter by `self` is conjugated of parameter `pos`.

INPUT:

- `pos` - integer

EXAMPLES:

```
sage: m = WordMorphism('a->abcde')
sage: m.conjugate(0) == m
True
sage: m.conjugate(1)
WordMorphism: a->bcdea
sage: m.conjugate(3)
WordMorphism: a->deabc
sage: WordMorphism('').conjugate(4)
WordMorphism:
sage: m = WordMorphism('a->abcde,b->xyz')
sage: m.conjugate(2)
WordMorphism: a->cdeab, b->zxy
```

domain()

Returns domain of `self`.

EXAMPLES:

```
sage: WordMorphism('a->ab,b->a').domain()
Finite words over {'a', 'b'}
sage: WordMorphism('b->ba,a->ab').domain()
Finite words over {'a', 'b'}
sage: WordMorphism('6->ab,y->5,0->asd').domain()
Finite words over {'0', '6', 'y'}
```

dual_map(*k=1*)

Return the dual map E_k^* of self (see [1]).

Note: It is acually implemented only for $k = 1$.

INPUT:

- `self` - unimodular endomorphism defined on integers `1, 2, \ldots, d`
- `k` - integer (optional, default: 1)

OUTPUT:

an instance of E1Star - the dual map

EXAMPLES:

```
sage: sigma = WordMorphism({1:[2],2:[3],3:[1,2]})
sage: sigma.dual_map()
E_1^*(1->2, 2->3, 3->12)

sage: sigma.dual_map(k=2)
Traceback (most recent call last):
...
NotImplementedError: The dual map E_k^* is implemented only for k = 1 (not 2)
```

REFERENCES:

- [1] Sano, Y., Arnoux, P. and Ito, S., Higher dimensional extensions of substitutions and their dual maps, Journal d'Analyse Mathematique 83 (2001), 183-206.

extend_by(*other*)

Returns `self` extended by `other`.

Let $\varphi_1 : A^* \to B^*$ and $\varphi_2 : C^* \to D^*$ be two morphisms. A morphism $\mu : (A \cup C)^* \to (B \cup D)^*$ corresponds to φ_1 *extended by* φ_2 if $\mu(a) = \varphi_1(a)$ if $a \in A$ and $\mu(a) = \varphi_2(a)$ otherwise.

INPUT:

- `other` - a WordMorphism.

OUTPUT:

WordMorphism

EXAMPLES:

```
sage: m = WordMorphism('a->ab,b->ba')
sage: n = WordMorphism({0:1,1:0,'a':5})
sage: m.extend_by(n)
WordMorphism: 0->1, 1->0, a->ab, b->ba
sage: n.extend_by(m)
WordMorphism: 0->1, 1->0, a->5, b->ba
sage: m.extend_by(m)
WordMorphism: a->ab, b->ba
```

TESTS:

```
sage: m.extend_by(WordMorphism({})) == m
True
sage: m.extend_by(WordMorphism('')) == m
True

sage: m.extend_by(4)
Traceback (most recent call last):
...
TypeError: other (=4) is not a WordMorphism
```

fixed_point(*letter*)

Returns the fixed point of `self` beginning by the given `letter`.

A fixed point of morphism φ is a word w such that $\varphi(w) = w$.

INPUT:

- `self` - an endomorphism, must be prolongable on `letter`

- `letter` - in the domain of `self`, the first letter of the fixed point.

OUTPUT:

- `word` - the fixed point of `self` beginning with `letter`.

EXAMPLES:

sage: W = FiniteWords('abc')

1.Infinite fixed point:

```
sage: WordMorphism('a->ab,b->ba').fixed_point(letter='a')
word: abbabaabbaababbabaababbaabbabaabbaababba...
sage: WordMorphism('a->ab,b->a').fixed_point(letter='a')
word: abaababaabaababaababaabaababaabaababaaba...
sage: WordMorphism('a->ab,b->b,c->ba', codomain=W).fixed_point(letter='a')
word: abbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb...
```

2.Infinite fixed point of an erasing morphism:

```
sage: WordMorphism('a->ab,b->,c->ba', codomain=W).fixed_point(letter='a')
word: ab
```

3.Finite fixed point:

```
sage: WordMorphism('a->ab,b->b,c->ba', codomain=W).fixed_point(letter='b')
word: b
sage: _.parent()
Finite words over {'a', 'b', 'c'}

sage: WordMorphism('a->ab,b->cc,c->', codomain=W).fixed_point(letter='a')
word: abcc
sage: _.parent()
Finite words over {'a', 'b', 'c'}

sage: m = WordMorphism('a->abc,b->,c->')
sage: fp = m.fixed_point('a'); fp
word: abc

sage: m = WordMorphism('a->ba,b->')
sage: m('ba')
word: ba
sage: m.fixed_point('a') #todo: not implemented
word: ba
```

5.Fixed point of a power of a morphism:

```
sage: m = WordMorphism('a->ba,b->ab')
sage: (m^2).fixed_point(letter='a')
word: abbabaabbaababbabaababbaabbabaabbaababba...
```

TESTS:

```
sage: WordMorphism('a->ab,b->,c->ba', codomain=W).fixed_point(letter='b')
Traceback (most recent call last):
...
TypeError: self must be prolongable on b
sage: WordMorphism('a->ab,b->,c->ba', codomain=W).fixed_point(letter='c')
Traceback (most recent call last):
...
TypeError: self must be prolongable on c
sage: WordMorphism('a->ab,b->,c->ba', codomain=W).fixed_point(letter='d')
Traceback (most recent call last):
...
TypeError: letter (=d) is not in the domain alphabet (={'a', 'b', 'c'})
sage: WordMorphism('a->aa,b->aac').fixed_point(letter='a')
Traceback (most recent call last):
...
TypeError: self (=a->aa, b->aac) is not an endomorphism
```

fixed_points()

Returns the list of all fixed points of `self`.

EXAMPLES:

```
sage: f = WordMorphism('a->ab,b->ba')
sage: for w in f.fixed_points(): print w
abbabaabbaababbabaababbaabbabaabbaababba...
baababbaabbabaababbabaabbaababbaabbabaab...

sage: f = WordMorphism('a->ab,b->c,c->a')
sage: for w in f.fixed_points(): print w
abcaababcabcaabcaababcaababcabcaababcabc...

sage: f = WordMorphism('a->ab,b->cab,c->bcc')
sage: for w in f.fixed_points(): print w
abcabbccabcabcabbccbccabcabbccabcabbccab...
```

This shows that ticket trac ticket #13668 has been resolved:

```
sage: d = {1:[1,2],2:[2,3],3:[4],4:[5],5:[6],6:[7],7:[8],8:[9],9:[10],10:[1]}
sage: s = WordMorphism(d)
sage: s7 = s^7
sage: s7.fixed_points()
[word: 12232342..., word: 2,3,4,5,6,7,8...]
sage: s7r = s7.reversal()
sage: s7r.periodic_point(2)
word: 2,1,1,10,9,8,7,6,5,4,3,2,1,10,9,8,7,6,5,4,3,2,10,9,8,7,6,5,4,3,2,9,8,7,6,5,4,3,2,8,...
```

This shows that ticket trac ticket #13668 has been resolved:

```
sage: s = "1->321331332133133,2->133321331332133133,3->213313313332133133321331332133133"
sage: s = WordMorphism(s)
sage: (s^2).fixed_points()
[]
```

growing_letters()

Returns the list of growing letters.

See `is_growing()` for more information.

EXAMPLES:

```
sage: WordMorphism('0->01,1->10').growing_letters()
['0', '1']
sage: WordMorphism('0->01,1->1').growing_letters()
['0']
sage: WordMorphism('0->01,1->0,2->1',codomain=Words('012')).growing_letters()
['0', '1', '2']
```

has_conjugate_in_classP (*f=None*)

Returns `True` if `self` has a conjugate in class f-P.

DEFINITION : Let A be an alphabet. We say that a primitive substitution S is in the *class P* if there exists a palindrome p and for each $b \in A$ a palindrome q_b such that $S(b) = pq_b$ for all $b \in A$. [1]

Let f be an involution on A. We say that a morphism φ is in class f-P if there exists an f-palindrome p and for each $\alpha \in A$ there exists an f-palindrome q_α such that $\varphi(\alpha) = pq_\alpha$. [2]

INPUT:

- `f` - involution (default: None) on the alphabet of `self`. It must be callable on letters as well as words (e.g. WordMorphism).

REFERENCES:

- [1] Hof, A., O. Knill et B. Simon, Singular continuous spectrum for palindromic Schrödinger operators, Commun. Math. Phys. 174 (1995) 149-159.
- [2] Labbe, Sebastien. Proprietes combinatoires des f-palindromes, Memoire de maitrise en Mathematiques, Montreal, UQAM, 2008, 109 pages.

EXAMPLES:

```
sage: fibo = WordMorphism('a->ab,b->a')
sage: fibo.has_conjugate_in_classP()
True
sage: (fibo^2).is_in_classP()
False
sage: (fibo^2).has_conjugate_in_classP()
True
```

has_left_conjugate()

Returns `True` if all the non empty images of `self` begins with the same letter.

EXAMPLES:

```
sage: m = WordMorphism('a->abcde,b->xyz')
sage: m.has_left_conjugate()
False
sage: WordMorphism('b->xyz').has_left_conjugate()
True
sage: WordMorphism('').has_left_conjugate()
True
sage: WordMorphism('a->,b->xyz').has_left_conjugate()
True
sage: WordMorphism('a->abbab,b->abb').has_left_conjugate()
True
sage: WordMorphism('a->abbab,b->abb,c->').has_left_conjugate()
True
```

has_right_conjugate()

Returns `True` if all the non empty images of `self` ends with the same letter.

EXAMPLES:

```
sage: m = WordMorphism('a->abcde,b->xyz')
sage: m.has_right_conjugate()
False
sage: WordMorphism('b->xyz').has_right_conjugate()
True
sage: WordMorphism('').has_right_conjugate()
True
sage: WordMorphism('a->,b->xyz').has_right_conjugate()
True
sage: WordMorphism('a->abbab,b->abb').has_right_conjugate()
True
sage: WordMorphism('a->abbab,b->abb,c->').has_right_conjugate()
True
```

image (*letter*)

Return the image of a letter

INPUT:

- `letter` - a letter in the domain alphabet

OUTPUT:

word

..NOTE:

```
The letter is assumed to be in the domain alphabet
(no check done). Hence, this method is faster
than the ``__call__`` method suitable for words input.
```

EXAMPLES:

```
sage: m = WordMorphism('a->ab,b->ac,c->a')
sage: m.image('b')
word: ac

sage: s = WordMorphism({('a', 1):[('a', 1), ('a', 2)], ('a', 2):[('a', 1)]})
sage: s.image(('a',1))
word: ('a', 1),('a', 2)

sage: s = WordMorphism({0:[1,2], 'a':(2,3,4), ():[9,8,7]})
sage: s.image(0)
word: 12
sage: s.image('a')
word: 234
sage: s.image(())
word: 987
```

images ()

Returns the list of all the images of the letters of the alphabet under `self`.

EXAMPLES:

```
sage: WordMorphism('a->ab,b->a').images()
[word: ab, word: a]
sage: WordMorphism('6->ab,y->5,0->asd').images()
[word: 5, word: asd, word: ab]
```

incidence_matrix ()

Returns the incidence matrix of the morphism. The order of the rows and column are given by the order

defined on the alphabet of the domain and the codomain.

The matrix returned is over the integers. If a different ring is desired, use either the `change_ring` function or the `matrix` function.

EXAMPLES:

```
sage: m = WordMorphism('a->abc,b->a,c->c')
sage: m.incidence_matrix()
[1 1 0]
[1 0 0]
[1 0 1]
sage: m = WordMorphism('a->abc,b->a,c->c,d->abbccccabca,e->abc')
sage: m.incidence_matrix()
[1 1 0 3 1]
[1 0 0 3 1]
[1 0 1 5 1]
```

is_empty()

Returns `True` if the cardinality of the domain is zero and `False` otherwise.

EXAMPLES:

```
sage: WordMorphism('').is_empty()
True
sage: WordMorphism('a->a').is_empty()
False
```

is_endomorphism()

Returns `True` if the codomain is a subset of the domain.

EXAMPLES:

```
sage: WordMorphism('a->ab,b->a').is_endomorphism()
True
sage: WordMorphism('6->ab,y->5,0->asd').is_endomorphism()
False
sage: WordMorphism('a->a,b->aa,c->aaa').is_endomorphism()
False
sage: Wabc = Words('abc')
sage: m = WordMorphism('a->a,b->aa,c->aaa',codomain = Wabc)
sage: m.is_endomorphism()
True
```

We check that trac ticket #8674 is fixed:

```
sage: P = WordPaths('abcd')
sage: m = WordMorphism('a->adab,b->ab,c->cbcd,d->cd', domain=P, codomain=P)
sage: m.is_endomorphism()
True
```

is_erasing()

Returns `True` if `self` is an erasing morphism, i.e. the image of a letter is the empty word.

EXAMPLES:

```
sage: WordMorphism('a->ab,b->a').is_erasing()
False
sage: WordMorphism('6->ab,y->5,0->asd').is_erasing()
False
sage: WordMorphism('6->ab,y->5,0->asd,7->').is_erasing()
True
```

```
sage: WordMorphism('').is_erasing()
False
```

is_growing(*letter=None*)

Return `True` if `letter` is a growing letter.

A letter a is *growing* for the morphism s if the length of the iterates of $|s^n(a)|$ tend to infinity as n goes to infinity.

INPUT:

- `letter` – `None` or a letter in the domain of `self`

Note: If letter is `None`, this returns `True` if `self` is everywhere growing, i.e., all letters are growing letters (see [CassNic10]), and that `self` **must** be an endomorphism.

EXAMPLES:

```
sage: WordMorphism('0->01,1->1').is_growing('0')
True
sage: WordMorphism('0->01,1->1').is_growing('1')
False
sage: WordMorphism('0->01,1->10').is_growing()
True
sage: WordMorphism('0->1,1->2,2->01').is_growing()
True
sage: WordMorphism('0->01,1->1').is_growing()
False
```

The domain needs to be equal to the codomain:

```
sage: WordMorphism('0->01,1->0,2->1',codomain=Words('012')).is_growing()
True
```

Test of erasing morphisms:

```
sage: WordMorphism('0->01,1->').is_growing('0')
False
sage: m = WordMorphism('a->bc,b->bcc,c->',codomain=Words('abc'))
sage: m.is_growing('a')
False
sage: m.is_growing('b')
False
sage: m.is_growing('c')
False
```

REFERENCES:

is_identity()

Returns `True` if `self` is the identity morphism.

EXAMPLES:

```
sage: m = WordMorphism('a->a,b->b,c->c,d->e')
sage: m.is_identity()
False
sage: WordMorphism('a->a,b->b,c->c').is_identity()
True
sage: WordMorphism('a->a,b->b,c->cb').is_identity()
False
sage: m = WordMorphism('a->b,b->c,c->a')
```

```
sage: (m^2).is_identity()
False
sage: (m^3).is_identity()
True
sage: (m^4).is_identity()
False
sage: WordMorphism('').is_identity()
True
sage: WordMorphism({0:[0],1:[1]}).is_identity()
True
```

We check that trac ticket #8618 is fixed:

```
sage: t = WordMorphism({'a1':['a2'], 'a2':['a1']})
sage: (t*t).is_identity()
True
```

is_in_classP (*f=None*)

Returns `True` if `self` is in class P (or f-P).

DEFINITION : Let A be an alphabet. We say that a primitive substitution S is in the *class P* if there exists a palindrome p and for each $b \in A$ a palindrome q_b such that $S(b) = pq_b$ for all $b \in A$. [1]

Let f be an involution on A. "We say that a morphism φ is in class f-P if there exists an f-palindrome p and for each $\alpha \in A$ there exists an f-palindrome q_α such that $\varphi(\alpha) = pq_\alpha$. [2]

INPUT:

- `f` - involution (default: None) on the alphabet of `self`. It must be callable on letters as well as words (e.g. WordMorphism).

REFERENCES:

- [1] Hof, A., O. Knill et B. Simon, Singular continuous spectrum for palindromic Schrödinger operators, Commun. Math. Phys. 174 (1995) 149-159.
- [2] Labbe, Sebastien. Proprietes combinatoires des f-palindromes, Memoire de maitrise en Mathematiques, Montreal, UQAM, 2008, 109 pages.

EXAMPLES:

```
sage: WordMorphism('a->bbaba,b->bba').is_in_classP()
True
sage: tm = WordMorphism('a->ab,b->ba')
sage: tm.is_in_classP()
False
sage: f = WordMorphism('a->b,b->a')
sage: tm.is_in_classP(f=f)
True
sage: (tm^2).is_in_classP()
True
sage: (tm^2).is_in_classP(f=f)
False
sage: fibo = WordMorphism('a->ab,b->a')
sage: fibo.is_in_classP()
True
sage: fibo.is_in_classP(f=f)
False
sage: (fibo^2).is_in_classP()
False
sage: f = WordMorphism('a->b,b->a,c->c')
```

```
sage: WordMorphism('a->acbcc,b->acbab,c->acbba').is_in_classP(f)
True
```

is_involution()

Returns `True` if `self` is an involution, i.e. its square is the identity.

INPUT:

- `self` - an endomorphism

EXAMPLES:

```
sage: WordMorphism('a->b,b->a').is_involution()
True
sage: WordMorphism('a->b,b->ba').is_involution()
False
sage: WordMorphism({0:[1],1:[0]}).is_involution()
True
```

TESTS:

```
sage: WordMorphism('').is_involution()
True
sage: WordMorphism({0:1,1:0,2:3}).is_involution()
Traceback (most recent call last):
...
TypeError: self (=0->1, 1->0, 2->3) is not an endomorphism
```

is_primitive()

Returns `True` if `self` is primitive.

A morphism φ is *primitive* if there exists an positive integer k such that for all $\alpha \in \Sigma$, $\varphi^k(\alpha)$ contains all the letters of Σ.

INPUT:

- `self` - an endomorphism

ALGORITHM:

Exercices 8.7.8, p.281 in [1] : (c) Let $y(M)$ be the least integer e such that M^e has all positive entries. Prove that, for all primitive matrices M, we have $y(M) \leq (d-1)^2 + 1$. (d) Prove that the bound $y(M) \leq (d-1)^2 + 1$ is best possible.

EXAMPLES:

```
sage: tm = WordMorphism('a->ab,b->ba')
sage: tm.is_primitive()
True
sage: fibo = WordMorphism('a->ab,b->a');
sage: fibo.is_primitive()
True
sage: m = WordMorphism('a->bb,b->aa')
sage: m.is_primitive()
False
sage: f = WordMorphism({0:[1],1:[0]})
sage: f.is_primitive()
False

sage: s = WordMorphism('a->b,b->c,c->ab')
sage: s.is_primitive()
True
```

```
sage: s = WordMorphism('a->b,b->c,c->d,d->e,e->f,f->g,g->h,h->ab')
sage: s.is_primitive()
True
```

TESTS:

```
sage: m = WordMorphism('a->bb,b->aac')
sage: m.is_primitive()
Traceback (most recent call last):
...
TypeError: self (=a->bb, b->aac) is not an endomorphism
sage: m = WordMorphism('a->,b->',codomain=Words('ab'))
sage: m.is_primitive()
False
sage: m = WordMorphism('a->,b->')
sage: m.is_primitive()
Traceback (most recent call last):
...
TypeError: self (=a->, b->) is not an endomorphism
```

REFERENCES:

- [1] Jean-Paul Allouche and Jeffrey Shallit, Automatic Sequences: Theory, Applications, Generalizations, Cambridge University Press, 2003.

is_prolongable(*letter*)

Returns `True` if `self` is prolongable on `letter`.

A morphism φ is prolongable on a letter a if a is a prefix of $\varphi(a)$.

INPUT:

- `self` - its codomain must be an instance of Words
- `letter` - a letter in the domain alphabet

OUTPUT:

Boolean

EXAMPLES:

```
sage: WordMorphism('a->ab,b->a').is_prolongable(letter='a')
True
sage: WordMorphism('a->ab,b->a').is_prolongable(letter='b')
False
sage: WordMorphism('a->ba,b->ab').is_prolongable(letter='b')
False
sage: (WordMorphism('a->ba,b->ab')^2).is_prolongable(letter='b')
True
sage: WordMorphism('a->ba,b->').is_prolongable(letter='b')
False
sage: WordMorphism('a->bb,b->aac').is_prolongable(letter='a')
False
```

We check that trac ticket #8595 is fixed:

```
sage: s = WordMorphism({('a', 1) : [('a', 1), ('a', 2)], ('a', 2) : [('a', 1)]})
sage: s.is_prolongable(('a',1))
True
```

TESTS:

```
sage: WordMorphism('a->ab,b->b,c->ba').is_prolongable(letter='d')
Traceback (most recent call last):
...
TypeError: letter (=d) is not in the domain alphabet (={'a', 'b', 'c'})

sage: n0, n1 = matrix(2,[1,1,1,0]), matrix(2,[2,1,1,0])
sage: n = {'a':n0, 'b':n1}
sage: WordMorphism(n).is_prolongable(letter='a') #todo: not implemented
Traceback (most recent call last):
...
TypeError: codomain of self must be an instance of Words
```

is_uniform(*k=None*)

Returns True if self is a k-uniform morphism.

Let k be a positive integer. A morphism ϕ is called k-uniform if for every letter α, we have $|\phi(\alpha)| = k$. In other words, all images have length k. A morphism is called uniform if it is k-uniform for some positive integer k.

INPUT:

- `k` - a positive integer or None. If set to a positive integer, then the function return True if self is k-uniform. If set to None, then the function return True if self is uniform.

EXAMPLES:

```
sage: phi = WordMorphism('a->ab,b->a')
sage: phi.is_uniform()
False
sage: phi.is_uniform(k=1)
False
sage: tau = WordMorphism('a->ab,b->ba')
sage: tau.is_uniform()
True
sage: tau.is_uniform(k=1)
False
sage: tau.is_uniform(k=2)
True
```

latex_layout(*layout=None*)

Get or set the actual latex layout (oneliner vs array).

INPUT:

- `layout` - string (default: `None`), can take one of the following values:
  - `None` - Returns the actual latex layout. By default, the layout is `'array'`
  - `'oneliner'` - Set the layout to `'oneliner'`
  - `'array'` - Set the layout to `'array'`

EXAMPLES:

```
sage: s = WordMorphism('a->ab,b->ba')
sage: s.latex_layout()
'array'
sage: s.latex_layout('oneliner')
sage: s.latex_layout()
'oneliner'
```

list_of_conjugates()

Returns the list of all the conjugate morphisms of `self`.

DEFINITION:

Recall from Lothaire [1] (Section 2.3.4) that φ is *right conjugate* of φ', noted $\varphi \triangleleft \varphi'$, if there exists $u \in \Sigma^*$ such that

$$\varphi(\alpha)u = u\varphi'(\alpha),$$

for all $\alpha \in \Sigma$, or equivalently that $\varphi(x)u = u\varphi'(x)$, for all words $x \in \Sigma^*$. Clearly, this relation is not symmetric so that we say that two morphisms φ and φ' are *conjugate*, noted $\varphi \bowtie \varphi'$, if $\varphi \triangleleft \varphi'$ or $\varphi' \triangleleft \varphi$. It is easy to see that conjugacy of morphisms is an equivalence relation.

REFERENCES:

•[1] M. Lothaire, Algebraic Combinatorics on words, Cambridge University Press, 2002.

EXAMPLES:

```
sage: m = WordMorphism('a->abbab,b->abb')
sage: m.list_of_conjugates()
[WordMorphism: a->babba, b->bab,
WordMorphism: a->abbab, b->abb,
WordMorphism: a->bbaba, b->bba,
WordMorphism: a->babab, b->bab,
WordMorphism: a->ababb, b->abb,
WordMorphism: a->babba, b->bba,
WordMorphism: a->abbab, b->bab]
sage: m = WordMorphism('a->aaa,b->aa')
sage: m.list_of_conjugates()
[WordMorphism: a->aaa, b->aa]
sage: WordMorphism('').list_of_conjugates()
[WordMorphism: ]
sage: m = WordMorphism('a->aba,b->aba')
sage: m.list_of_conjugates()
[WordMorphism: a->baa, b->baa,
WordMorphism: a->aab, b->aab,
WordMorphism: a->aba, b->aba]
sage: m = WordMorphism('a->abb,b->abbab,c->')
sage: m.list_of_conjugates()
[WordMorphism: a->bab, b->babba, c->,
WordMorphism: a->abb, b->abbab, c->,
WordMorphism: a->bba, b->bbaba, c->,
WordMorphism: a->bab, b->babab, c->,
WordMorphism: a->abb, b->ababb, c->,
WordMorphism: a->bba, b->babba, c->,
WordMorphism: a->bab, b->abbab, c->]
```

partition_of_domain_alphabet()

Returns a partition of the domain alphabet.

Let $\varphi : \Sigma^* \to \Sigma^*$ be an involution. There exists a triple of sets (A, B, C) such that

•$A \cup B \cup C = \Sigma$;

•A, B and C are mutually disjoint and

•$\varphi(A) = B$, $\varphi(B) = A$, $\varphi(C) = C$.

These sets are not unique.

INPUT:

- `self` - An involution.

OUTPUT:

A tuple of three sets

EXAMPLES:

```
sage: m = WordMorphism('a->b,b->a')
sage: m.partition_of_domain_alphabet() #random ordering
({'a'}, {'b'}, {})
sage: m = WordMorphism('a->b,b->a,c->c')
sage: m.partition_of_domain_alphabet() #random ordering
({'a'}, {'b'}, {'c'})
sage: m = WordMorphism('a->a,b->b,c->c')
sage: m.partition_of_domain_alphabet() #random ordering
({}, {}, {'a', 'c', 'b'})
sage: m = WordMorphism('A->T,T->A,C->G,G->C')
sage: m.partition_of_domain_alphabet() #random ordering
({'A', 'C'}, {'T', 'G'}, {})
sage: I = WordMorphism({0:oo,oo:0,1:-1,-1:1,2:-2,-2:2,3:-3,-3:3})
sage: I.partition_of_domain_alphabet() #random ordering
({0, -1, -3, -2}, {1, 2, 3, +Infinity}, {})
```

TESTS:

```
sage: m = WordMorphism('a->b,b->a,c->a')
sage: m.partition_of_domain_alphabet()
Traceback (most recent call last):
...
TypeError: self (=a->b, b->a, c->a) is not an endomorphism
```

periodic_point(*letter*)

Return the periodic point of self that starts with `letter`.

EXAMPLES:

```
sage: f = WordMorphism('a->bab,b->ab')
sage: f.periodic_point('a')
word: abbababbababbababbababbababbababbababbaba...
sage: f.fixed_point('a')
Traceback (most recent call last):
...
TypeError: self must be prolongable on a
```

periodic_points()

Return the periodic points of `f` as a list of tuples where each tuple is a periodic orbit of `f`.

EXAMPLES:

```
sage: f = WordMorphism('a->aba,b->baa')
sage: for p in f.periodic_points():
...       print len(p), ',', p[0]
1 , ababaaababaaabaabaababaaababaaabaabaabab...
1 , baaabaabaababaaabaababaaabaababaaababaaa...

sage: f = WordMorphism('a->bab,b->aa')
sage: for p in f.periodic_points():
...       print len(p), ',', p[0]
2 , aababaaaababaabababbabaababaababbabaababaa...
sage: f.fixed_points()
[]
```

This shows that ticket trac ticket #13668 has been resolved:

```
sage: d = {1:[1,2],2:[2,3],3:[4],4:[5],5:[6],6:[7],7:[8],8:[9],9:[10],10:[1]}
sage: s = WordMorphism(d)
sage: s7 = s^7
sage: s7r = s7.reversal()
sage: for p in s7r.periodic_points(): p
[word: 1,10,9,8,7,6,5,4,3,2,10,9,8,7,6,5,4,3,2,...,
 word: 876543276543265432543243232176543265432...,
 word: 5,4,3,2,4,3,2,3,2,2,1,4,3,2,3,2,2,1,3,2,...,
 word: 2,1,1,10,9,8,7,6,5,4,3,2,1,10,9,8,7,6,5,...,
 word: 98765432876543276543265432543243232221876...,
 word: 65432543243232215432432321432321322121...,
 word: 3,2,2,1,2,1,1,10,9,8,7,6,5,4,3,2,2,1,1,1...,
 word: 10,9,8,7,6,5,4,3,2,9,8,7,6,5,4,3,2,8,7,6...,
 word: 765432654325432432321654325432432322154...,
 word: 4,3,2,3,2,2,1,3,2,2,1,2,1,1,10,9,8,7,6,5...]
```

pisot_eigenvector_left()

Returns the left eigenvector of the incidence matrix associated to the largest eigenvalue (in absolute value).

Unicity of the result is guaranteed when the multiplicity of the largest eigenvalue is one, for example when self is a Pisot irreductible substitution.

A substitution is Pisot irreducible if the characteristic polynomial of its incidence matrix is irreducible over **Q** and has all roots, except one, of modulus strictly smaller than 1.

INPUT:

- `self` - a Pisot irreducible substitution.

EXAMPLES:

```
sage: m = WordMorphism('a->aaaabbc,b->aaabbc,c->aabc')
sage: matrix(m)
[4 3 2]
[2 2 1]
[1 1 1]
sage: m.pisot_eigenvector_left()
(1, 0.8392867552141611?, 0.5436890126920763?)
```

pisot_eigenvector_right()

Returns the right eigenvector of the incidence matrix associated to the largest eigenvalue (in absolute value).

Unicity of the result is guaranteed when the multiplicity of the largest eigenvalue is one, for example when self is a Pisot irreductible substitution.

A substitution is Pisot irreducible if the characteristic polynomial of its incidence matrix is irreducible over **Q** and has all roots, except one, of modulus strictly smaller than 1.

INPUT:

- `self` - a Pisot irreducible substitution.

EXAMPLES:

```
sage: m = WordMorphism('a->aaaabbc,b->aaabbc,c->aabc')
sage: matrix(m)
[4 3 2]
[2 2 1]
```

```
[1 1 1]
sage: m.pisot_eigenvector_right()
(1, 0.5436890126920763?, 0.2955977425220848?)
```

rauzy_fractal_plot(*n=None*, *exchange=False*, *eig=None*, *translate=None*, *prec=53*, *colormap='hsv'*, *opacity=None*, *plot_origin=None*, *plot_basis=False*, *point_size=None*)

Returns a plot of the Rauzy fractal associated with a substitution.

The substitution does not have to be irreducible. The usual definition of a Rauzy fractal requires that its dominant eigenvalue is a Pisot number but the present method doesn't require this, allowing to plot some interesting pictures in the non-Pisot case (see the examples below).

For more details about the definition of the fractal and the projection which is used, see Section 3.1 of [1].

Plots with less than 100,000 points take a few seconds, and several millions of points can be plotted in reasonable time.

Other ways to draw Rauzy fractals (and more generally projections of paths) can be found in `sage.combinat.words.paths.FiniteWordPath_all.plot_projection()` or in `sage.combinat.e_one_star()`.

OUTPUT:

A Graphics object.

INPUT:

- `n` - integer (default: `None`) The number of points used to plot the fractal. Default values: `1000` for a 1D fractal, `50000` for a 2D fractal, `10000` for a 3D fractal.
- `exchange` - boolean (default: `False`). Plot the Rauzy fractal with domain exchange.
- `eig` - a real element of `QQbar` of degree >= 2 (default: `None`). The eigenvalue used to plot the fractal. It must be an eigenvalue of `self.incidence_matrix()`. The one used by default the maximal eigenvalue of `self.incidence_matrix()` (usually a Pisot number), but for substitutions with more than 3 letters other interesting choices are sometimes possible.
- `translate` - a list of vectors of `RR^size_alphabet`, or a dictionary from the alphabet to lists of vectors (default: `None`). Plot translated copies of the fractal. This option allows to plot tilings easily. The projection used for these vectors is the same as the projection used for the canonical basis to plot the fractal. If the input is a list, all the pieces will be translated and plotted. If the input is a dictionary, each piece will be translated and plotted accordingly to the vectors associated with each letter in the dictionary. Note: by default, the Rauzy fractal placed at the origin is not plotted with the `translate` option; the vector `(0,0,...,0)` has to be added manually.
- `prec` - integer (default: `53`). The number of bits used in the floating point representations of the points of the fractal.
- `colormap` - color map or dictionary (default: `'hsv'`). It can be one of the following :
  - `string` - a coloring map. For available coloring map names type: `sorted(colormaps)`
  - `dict` - a dictionary of the alphabet mapped to colors.
- `opacity` - a dictionary from the alphabet to the real interval [0,1] (default: `None`). If none is specified, all letters are plotted with opacity `1`.
- `plot_origin` - a couple `(k,c)` (default: `None`). If specified, mark the origin by a point of size `k` and color `c`.
- `plot_basis` - boolean (default: `False`). Plot the projection of the canonical basis with the fractal.

•`point_size` - float (default: `None`). The size of the points used to plot the fractal.

EXAMPLES:

1.The Rauzy fractal of the Tribonacci substitution:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: s.rauzy_fractal_plot()      # long time
Graphics object consisting of 3 graphics primitives
```

2.The "Hokkaido" fractal. We tweak the plot using the plotting options to get a nice reusable picture, in which we mark the origin by a black dot:

```
sage: s = WordMorphism('a->ab,b->c,c->d,d->e,e->a')
sage: G = s.rauzy_fractal_plot(n=100000, point_size=3, plot_origin=(50,"black"))   # not
sage: G.show(figsize=10, axes=false) # not tested
```

3.Another "Hokkaido" fractal and its domain exchange:

```
sage: s = WordMorphism({1:[2], 2:[4,3], 3:[4], 4:[5,3], 5:[6], 6:[1]})
sage: s.rauzy_fractal_plot()                    # not tested takes > 1 second
sage: s.rauzy_fractal_plot(exchange=True)       # not tested takes > 1 second
```

4.A three-dimensional Rauzy fractal:

```
sage: s = WordMorphism('1->12,2->13,3->14,4->1')
sage: s.rauzy_fractal_plot()      # not tested takes > 1 second
```

5.A one-dimensional Rauzy fractal (very scattered):

```
sage: s = WordMorphism('1->2122,2->1')
sage: s.rauzy_fractal_plot().show(figsize=20)      # not tested takes > 1 second
```

6.A high resolution plot of a complicated fractal:

```
sage: s = WordMorphism('1->23,2->123,3->1122233')
sage: G = s.rauzy_fractal_plot(n=300000)  # not tested takes > 1 second
sage: G.show(axes=false, figsize=20)      # not tested takes > 1 second
```

7.A nice colorful animation of a domain exchange:

```
sage: s = WordMorphism('1->21,2->3,3->4,4->25,5->6,6->7,7->1')
sage: L = [s.rauzy_fractal_plot(), s.rauzy_fractal_plot(exchange=True)]      # not tested
sage: animate(L, axes=false).show(delay=100)      # not tested takes > 1 second
```

8.Plotting with only one color:

```
sage: s = WordMorphism('1->12,2->31,3->1')
sage: s.rauzy_fractal_plot(colormap={'1':'black', '2':'black', '3':'black'})      # not t
```

9. Different fractals can be obtained by choosing another (non-Pisot) eigenvalue:

```
sage: s = WordMorphism('1->12,2->3,3->45,4->5,5->6,6->7,7->8,8->1')
sage: E = s.incidence_matrix().eigenvalues()
sage: x = [x for x in E if -0.8 < x < -0.7][0]
sage: s.rauzy_fractal_plot()          # not tested takes > 1 second
sage: s.rauzy_fractal_plot(eig=x)     # not tested takes > 1 second
```

10. A Pisot reducible substitution with seemingly overlapping tiles:

```
sage: s = WordMorphism({1:[1,2], 2:[2,3], 3:[4], 4:[5], 5:[6], 6:[7], 7:[8], 8:[9], 9:[1
sage: s.rauzy_fractal_plot()     # not tested takes > 1 second
```

11. A non-Pisot reducible substitution with a strange Rauzy fractal:

```
sage: s = WordMorphism({1:[3,2], 2:[3,3], 3:[4], 4:[1]})
sage: s.rauzy_fractal_plot()     # not tested takes > 1 second
```

12. A substitution with overlapping tiles. We use the options `colormap` and `opacity` to study how the tiles overlap:

```
sage: s = WordMorphism('1->213,2->4,3->5,4->1,5->21')
sage: s.rauzy_fractal_plot()                                    # not tested takes > 1 se
sage: s.rauzy_fractal_plot(colormap={'1':'red', '4':'purple'})     # not tested takes >
sage: s.rauzy_fractal_plot(opacity={'1':0.1,'2':1,'3':0.1,'4':0.1,'5':0.1}, n=150000)
```

13. Funny experiments by playing with the precision of the float numbers used to plot the fractal:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: s.rauzy_fractal_plot(prec=6)      # not tested
sage: s.rauzy_fractal_plot(prec=9)      # not tested
sage: s.rauzy_fractal_plot(prec=15)     # not tested
sage: s.rauzy_fractal_plot(prec=19)     # not tested
sage: s.rauzy_fractal_plot(prec=25)     # not tested
```

14. Using the `translate` option to plot periodic tilings:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: s.rauzy_fractal_plot(n=10000, translate=[(0,0,0),(-1,0,1),(0,-1,1),(1,-1,0),(1,0,-
```

```
sage: t = WordMorphism("a->aC,b->d,C->de,d->a,e->ab")   # substitution found by Julien B
sage: V = [vector((0,0,1,0,-1)), vector((0,0,1,-1,0))]
sage: S = set(map(tuple, [i*V[0] + j*V[1] for i in [-1,0,1] for j in [-1,0,1]]))
sage: t.rauzy_fractal_plot(n=10000, translate=S, exchange=true)     # not tested takes >
```

15. Using the `translate` option to plot arbitrary tilings with the fractal pieces. This can be used for example to plot the self-replicating tiling of the Rauzy fractal:

```
sage: s = WordMorphism({1:[1,2], 2:[3], 3:[4,3], 4:[5], 5:[6], 6:[1]})
sage: s.rauzy_fractal_plot()      # not tested takes > 1 second
sage: D = {1:[(0,0,0,0,0,0), (0,1,0,0,0,0)], 3:[(0,0,0,0,0,0), (0,1,0,0,0,0)], 6:[(0,1,0
sage: s.rauzy_fractal_plot(n=30000, translate=D)      # not tested takes > 1 second
```

16. Plot the projection of the canonical basis with the fractal:

```
sage: s = WordMorphism({1:[2,1], 2:[3], 3:[6,4], 4:[5,1], 5:[6], 6:[7], 7:[8], 8:[9], 9:
sage: s.rauzy_fractal_plot(plot_basis=True)      # not tested takes > 1 second
```

TESTS:

```
sage: s = WordMorphism('a->ab,b->c,c->d,d->e,e->a')
sage: s.rauzy_fractal_plot(n=1000, colormap='Set1', opacity={'a':0.5,'b':1,'c':0.7,'d':0,'e'
Graphics object consisting of 10 graphics primitives
```

REFERENCES:

- [1] Valerie Berthe and Anne Siegel, Tilings associated with beta-numeration and substitutions, Integers 5 (3), 2005. http://www.integers-ejcnt.org/vol5-3.html

AUTHOR:

Timo Jolivet (2012-06-16)

rauzy_fractal_points(*n=None*, *exchange=False*, *eig=None*, *translate=None*, *prec=53*)

Returns a dictionary of list of points associated with the pieces of the Rauzy fractal of `self`.

INPUT:

See the method `rauzy_fractal_plot()` for a description of the options and more examples.

OUTPUT:

dictionary of list of points

EXAMPLES:

The Rauzy fractal of the Tribonacci substitution and the number of points in the piece of the fractal associated with `'1'`, `'2'` and `'3'` are respectively:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: D = s.rauzy_fractal_points(n=100)
sage: len(D['1'])
54
sage: len(D['2'])
30
sage: len(D['3'])
16
```

TESTS:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: D = s.rauzy_fractal_points(n=100, exchange=True, translate=[(3,1,-2), (5,-33,8)], prec
sage: len(D['1'])
108
```

AUTHOR:

Timo Jolivet (2012-06-16)

rauzy_fractal_projection(*eig=None*, *prec=53*)

Returns a dictionary giving the projection of the canonical basis.

See the method `rauzy_fractal_plot()` for more details about the projection.

INPUT:

- `eig` - a real element of `QQbar` of degree >= 2 (default: `None`). The eigenvalue used for the projection. It must be an eigenvalue of `self.incidence_matrix()`. The one used by default is the maximal eigenvalue of `self.incidence_matrix()` (usually a Pisot number), but for substitutions with more than 3 letters other interesting choices are sometimes possible.
- `prec` - integer (default: `53`). The number of bits used in the floating point representations of the coordinates.

OUTPUT:

dictionary, letter -> vector, giving the projection

EXAMPLES:

The projection for the Rauzy fractal of the Tribonacci substitution is:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: s.rauzy_fractal_projection()
{'1': (1.00000000000000, 0.000000000000000),
 '2': (-1.41964337760708, -0.606290729207199),
 '3': (-0.771844506346038, 1.11514250803994)}
```

TESTS:

```
sage: t = WordMorphism('1->12,2->3,3->45,4->5,5->6,6->7,7->8,8->1')
sage: E = t.incidence_matrix().eigenvalues()
sage: x = [x for x in E if -0.8 < x < -0.7][0]
sage: t.rauzy_fractal_projection(prec=10)
{'1': (1.0, 0.00),
 '2': (-1.7, -0.56),
 '3': (0.79, 1.3),
 '4': (1.9, -0.74),
 '5': (-1.7, -0.56),
 '6': (0.79, 1.3),
 '7': (0.21, -1.3),
 '8': (-0.88, 0.74)}
sage: t.rauzy_fractal_projection(eig=x, prec=10)
{'1': (1.0, 0.00),
 '2': (-0.12, -0.74),
 '3': (-0.66, -0.56),
 '4': (-0.46, -0.18),
 '5': (-0.54, 0.18),
 '6': (-0.34, 0.56),
 '7': (0.12, 0.74),
 '8': (0.66, 0.56)}
```

AUTHOR:

Timo Jolivet (2012-06-16)

restrict_domain(*alphabet*)

Returns a restriction of `self` to the given alphabet.

INPUT:

- `alphabet` - an iterable

OUTPUT:

WordMorphism

EXAMPLES:

```
sage: m = WordMorphism('a->b,b->a')
sage: m.restrict_domain('a')
WordMorphism: a->b
sage: m.restrict_domain('')
WordMorphism:
sage: m.restrict_domain('A')
WordMorphism:
sage: m.restrict_domain('Aa')
WordMorphism: a->b
```

The input alphabet must be iterable:

```
sage: m.restrict_domain(66)
Traceback (most recent call last):
...
TypeError: 'sage.rings.integer.Integer' object is not iterable
```

reversal()

Returns the reversal of `self`.

EXAMPLES:

```
sage: WordMorphism('6->ab,y->5,0->asd').reversal()
WordMorphism: 0->dsa, 6->ba, y->5
sage: WordMorphism('a->ab,b->a').reversal()
WordMorphism: a->ba, b->a
```

sage.combinat.words.morphism.**get_cycles**(*f*, *domain=None*)

Return the cycle of the function `f` on the finite set domain. It is assumed that f is an endomorphism.

INPUT:

- `f` - function.
- `domain` - set (default: None) - the domain of `f`. If none, then tries to use `f.domain()`.

EXAMPLES:

```
sage: from sage.combinat.words.morphism import get_cycles
sage: get_cycles(lambda i: (i+1)%3, domain=[0,1,2])
[(0, 1, 2)]
sage: get_cycles(lambda i: [0,0,0][i], domain=[0,1,2])
[(0,)]
sage: get_cycles(lambda i: [1,1,1][i], domain=[0,1,2])
[(1,)]
```

5.1.325 Word paths

This module implements word paths, which is an application of Combinatorics on Words to Discrete Geometry. A word path is the representation of a word as a discrete path in a vector space using a one-to-one correspondence between the alphabet and a set of vectors called steps. Many problems surrounding 2d lattice polygons (such as questions of self-intersection, area, inertia moment, etc.) can be solved in linear time (linear in the length of the perimeter) using theory from Combinatorics on Words.

On the square grid, the encoding of a path using a four-letter alphabet (for East, North, West and South directions) is also known as the Freeman chain code [1,2] (see [3] for further reading).

AUTHORS:

- Arnaud Bergeron (2008) : Initial version, path on the square grid
- Sebastien Labbe (2009-01-14) : New classes and hierarchy, doc and functions.

EXAMPLES:

The combinatorial class of all paths defined over three given steps:

```
sage: P = WordPaths('abc', steps=[(1,2), (-3,4), (0,-3)]); P
Word Paths over 3 steps
```

This defines a one-to-one correspondence between alphabet and steps:

```
sage: d = P.letters_to_steps()
sage: sorted(d.items())
[('a', (1, 2)), ('b', (-3, 4)), ('c', (0, -3))]
```

Creation of a path from the combinatorial class P defined above:

```
sage: p = P('abaccba'); p
Path: abaccba
```

Many functions can be used on p: the coordinates of its trajectory, ask whether p is a closed path, plot it and many other:

```
sage: list(p.points())
[(0, 0), (1, 2), (-2, 6), (-1, 8), (-1, 5), (-1, 2), (-4, 6), (-3, 8)]
sage: p.is_closed()
False
sage: p.plot()
Graphics object consisting of 3 graphics primitives
```

To obtain a list of all the available word path specific functions, use `help(p)`:

```
sage: help(p)
Help on FiniteWordPath_2d_str in module sage.combinat.words.paths object:
...
Methods inherited from FiniteWordPath_2d:
...
Methods inherited from FiniteWordPath_all:
...
This only works on Python classes that derive from SageObject.
...
See http://trac.sagemath.org/2536 for details.
```

Since p is a finite word, many functions from the word library are available:

```
sage: p.crochemore_factorization()
(a, b, a, c, c, ba)
sage: p.is_palindrome()
False
sage: p[:3]
Path: aba
sage: len(p)
7
```

P also herits many functions from Words:

```
sage: P = WordPaths('rs', steps=[(1,2), (-1,4)]); P
Word Paths over 2 steps
sage: P.alphabet()
{'r', 's'}
sage: list(P.iterate_by_length(3))
[Path: rrr,
 Path: rrs,
 Path: rsr,
 Path: rss,
 Path: srr,
 Path: srs,
 Path: ssr,
 Path: sss]
```

When the number of given steps is half the size of alphabet, the opposite of vectors are used:

```
sage: P = WordPaths('abcd', [(1,0), (0,1)])
sage: sorted(P.letters_to_steps().items())
[('a', (1, 0)), ('b', (0, 1)), ('c', (-1, 0)), ('d', (0, -1))]
```

Some built-in combinatorial classes of paths:

```
sage: P = WordPaths('abAB', steps='square_grid'); P
Word Paths on the square grid
```

```
sage: D = WordPaths('()', steps='dyck'); D
Finite Dyck paths
sage: d = D('()()()(())'); d
Path: ()()()(())
sage: d.plot()
Graphics object consisting of 3 graphics primitives
```

```
sage: P = WordPaths('abcdef', steps='triangle_grid')
sage: p = P('babaddefadabcadefaadfafabacdefa')
sage: p.plot()
Graphics object consisting of 3 graphics primitives
```

Vector steps may be in more than 2 dimensions:

```
sage: d = [(1,0,0), (0,1,0), (0,0,1)]
sage: P = WordPaths(alphabet='abc', steps=d); P
Word Paths over 3 steps
sage: p = P('abcabcabcabcaabacabcababcacbabacacabcaccbcac')
sage: p.plot()
Graphics3d Object
```

```
sage: d = [(1,3,5,1), (-5,1,-6,0), (0,0,1,9), (4,2,-1,0)]
sage: P = WordPaths(alphabet='rstu', steps=d); P
Word Paths over 4 steps
sage: p = P('rtusuusususuturrsust'); p
Path: rtusuusususuturrsust
sage: p.end_point()
(5, 31, -26, 30)
```

```
sage: CubePaths = WordPaths('abcABC', steps='cube_grid'); CubePaths
Word Paths on the cube grid
sage: CubePaths('abcabaabcabAAAAA').plot()
Graphics3d Object
```

The input data may be a str, a list, a tuple, a callable or a finite iterator:

```
sage: P = WordPaths([0, 1, 2, 3])
sage: P([0,1,2,3,2,1,2,3,2])
Path: 012321232
sage: P((0,1,2,3,2,1,2,3,2))
Path: 012321232
sage: P(lambda n:n%4, length=10)
Path: 0123012301
sage: P(iter([0,3,2,1]), length='finite')
Path: 0321
```

REFERENCES:

- [1] Freeman, H.: On the encoding of arbitrary geometric configurations. IRE Trans. Electronic Computer 10 (1961) 260-268.
- [2] Freeman, H.: Boundary encoding and processing. In Lipkin, B., Rosenfeld, A., eds.: Picture Processing and Psychopictorics, Academic Press, New York (1970) 241-266.
- [3] Braquelaire, J.P., Vialard, A.: Euclidean paths: A new representation of boundary of discrete regions. Graphical Models and Image Processing 61 (1999) 16-43.
- [4] http://en.wikipedia.org/wiki/Regular_tiling
- [5] http://en.wikipedia.org/wiki/Dyck_word

class sage.combinat.words.paths.**FiniteWordPath_2d**

Bases: sage.combinat.words.paths.FiniteWordPath_all

animate()

Returns an animation object illustrating the path growing step by step.

EXAMPLES:

```
sage: P = WordPaths('abAB')
sage: p = P('aaababbb')
sage: a = p.animate(); a     # optional -- ImageMagick
Animation with 9 frames
sage: show(a)                # optional -- ImageMagick
sage: a.gif(delay=35, iterations=3)     # optional -- ImageMagick
doctest:...: DeprecationWarning: use tmp_filename instead
See http://trac.sagemath.org/17234 for details.

sage: P = WordPaths('abcdef',steps='triangle')
sage: p =  P('abcdef')
sage: p.animate()            # optional -- ImageMagick
Animation with 8 frames
```

If the path is closed, the plain polygon is added at the end of the animation:

```
sage: P = WordPaths('abAB')
sage: p = P('ababAbABABaB')
sage: a = p.animate(); a     # optional -- ImageMagick
Animation with 14 frames
```

Another example illustrating a Fibonacci tile:

```
sage: w = words.fibonacci_tile(2)
sage: show(w.animate())  # optional -- ImageMagick
```

The first 4 Fibonacci tiles in an animation:

```
sage: a = words.fibonacci_tile(0).animate()
sage: b = words.fibonacci_tile(1).animate()
sage: c = words.fibonacci_tile(2).animate()
sage: d = words.fibonacci_tile(3).animate()
sage: (a*b*c*d).show()  # optional -- ImageMagick
```

Note: If ImageMagick is not installed, you will get an error message like this:

```
/usr/local/share/sage/local/bin/sage-native-execute: 8: convert:
not found

Error: ImageMagick does not appear to be installed. Saving an
animation to a GIF file or displaying an animation requires
ImageMagick, so please install it and try again.
```

See www.imagemagick.org, for example.

---

area()

Returns the area of a closed path.

INPUT:

- `self` - a closed path

EXAMPLES:

```
sage: P = WordPaths('abcd',steps=[(1,1),(-1,1),(-1,-1),(1,-1)])
sage: p = P('abcd')
sage: p.area()          #todo: not implemented
2
```

height()

Returns the height of self.

The height of a $2d$-path is merely the difference between the highest and the lowest y-coordinate of each points traced by it.

OUTPUT:

non negative real number

EXAMPLES:

```
sage: Freeman = WordPaths('abAB')
sage: Freeman('aababaabbbAA').height()
5
```

The function is well-defined if self is not simple or close:

```
sage: Freeman('aabAAB').height()
1
sage: Freeman('abbABa').height()
2
```

This works for any $2d$-paths:

```
sage: Paths = WordPaths('ab', steps=[(1,0),(1,1)])
sage: p = Paths('abbaa')
sage: p.height()
2
sage: DyckPaths = WordPaths('ab', steps='dyck')
sage: p = DyckPaths('abaabb')
sage: p.height()
2
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.height()
2.59807621135332
```

plot (*pathoptions={'rgbcolor': 'red', 'thickness': 3}*, *fill=True*, *filloptions={'alpha': 0.2, 'rgbcolor': 'red'}*, *startpoint=True*, *startoptions={'pointsize': 100, 'rgbcolor': 'red'}*, *endarrow=True*, *arrowoptions={'width': 3, 'rgbcolor': 'red', 'arrowsize': 20}*, *gridlines=False*, *gridoptions={}*)

Returns a 2d Graphics illustrating the path.

INPUT:

- `pathoptions` - (dict, default:dict(rgbcolor='red',thickness=3)), options for the path drawing
- `fill` - (boolean, default: True), if fill is True and if the path is closed, the inside is colored
- `filloptions` - (dict, default:dict(rgbcolor='red',alpha=0.2)), ptions for the inside filling
- `startpoint` - (boolean, default: True), draw the start point?
- `startoptions` - (dict, default:dict(rgbcolor='red',pointsize=100)) options for the start point drawing
- `endarrow` - (boolean, default: True), draw an arrow end at the end?
- `arrowoptions` - (dict, default:dict(rgbcolor='red',arrowsize=20, width=3)) options for the end point arrow
- `gridlines`- (boolean, default: False), show gridlines?
- `gridoptions` - (dict, default: {}), options for the gridlines

EXAMPLES:

A non closed path on the square grid:

```
sage: P = WordPaths('abAB')
sage: P('abababAABAB').plot()
Graphics object consisting of 3 graphics primitives
```

A closed path on the square grid:

```
sage: P('abababAABABB').plot()
Graphics object consisting of 4 graphics primitives
```

A Dyck path:

```
sage: P = WordPaths('()', steps='dyck')
sage: P('()()()((()))').plot()
Graphics object consisting of 3 graphics primitives
```

A path in the triangle grid:

```
sage: P = WordPaths('abcdef', steps='triangle_grid')
sage: P('abcdedededefab').plot()
Graphics object consisting of 3 graphics primitives
```

A polygon of length 220 that tiles the plane in two ways:

```
sage: P = WordPaths('abAB')
sage: P('aBababAbababaBaBABaBabaBaBABAbABABaBabaBaBABaBababAbababaBaBABaBabaBaBABAbABABaBABAbAba
Graphics object consisting of 4 graphics primitives
```

With gridlines:

```
sage: P('abababababab').plot(gridlines=True)
```

TESTS:

```
sage: P = WordPaths('abAB')
sage: P().plot()
Graphics object consisting of 3 graphics primitives
sage: sum(map(plot,map(P,['a','A','b','B'])))
Graphics object consisting of 12 graphics primitives
```

plot_directive_vector(*options={'rgbcolor': 'blue'}*)

Returns an arrow 2d graphics that goes from the start of the path to the end.

INPUT:

- `options` - dictionary, default: {'rgbcolor': 'blue'} graphic options for the arrow

If the start is the same as the end, a single point is returned.

EXAMPLES:

```
sage: P = WordPaths('abcd'); P
Word Paths on the square grid
sage: p = P('aaaccaccacacacaccccccbbdd'); p
Path: aaaccaccacacacaccccccbbdd
sage: R = p.plot() + p.plot_directive_vector()
sage: R.axes(False)
sage: R.set_aspect_ratio(1)
sage: R.plot()
Graphics object consisting of 4 graphics primitives
```

TESTS:

A closed path:

```
sage: P('acbd').plot_directive_vector()
Graphics object consisting of 1 graphics primitive
```

width()

Returns the width of self.

The height of a $2d$-path is merely the difference between the rightmost and the leftmost x-coordinate of each points traced by it.

OUTPUT:

non negative real number

EXAMPLES:

```
sage: Freeman = WordPaths('abAB')
sage: Freeman('aababaabbbAA').width()
5
```

The function is well-defined if self is not simple or close:

```
sage: Freeman('aabAAB').width()
2
sage: Freeman('abbABa').width()
1
```

This works for any $2d$-paths:

```
sage: Paths = WordPaths('ab', steps=[(1,0),(1,1)])
sage: p = Paths('abbaa')
sage: p.width()
5
sage: DyckPaths = WordPaths('ab', steps='dyck')
sage: p = DyckPaths('abaabb')
sage: p.width()
6
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.width()
4.500000000000000
```

xmax()

Returns the maximum of the x-coordinates of the path.

EXAMPLES:

```
sage: P = WordPaths('0123')
sage: p = P('0101013332')
sage: p.xmax()
3
```

This works for any $2d$-paths:

```
sage: Paths = WordPaths('ab', steps=[(1,-1),(-1,1)])
sage: p = Paths('ababa')
sage: p.xmax()
1
sage: DyckPaths = WordPaths('ab', steps='dyck')
sage: p = DyckPaths('abaabb')
sage: p.xmax()
6
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.xmax()
4.500000000000000
```

xmin()

Returns the minimum of the x-coordinates of the path.

EXAMPLES:

```
sage: P = WordPaths('0123')
sage: p = P('0101013332')
sage: p.xmin()
0
```

This works for any $2d$-paths:

```
sage: Paths = WordPaths('ab', steps=[(1,0),(-1,1)])
sage: p = Paths('abbba')
sage: p.xmin()
-2
sage: DyckPaths = WordPaths('ab', steps='dyck')
sage: p = DyckPaths('abaabb')
```

```
sage: p.xmin()
0
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.xmin()
0.000000000000000
```

ymax()

Returns the maximum of the y-coordinates of the path.

EXAMPLES:

```
sage: P = WordPaths('0123')
sage: p = P('0101013332')
sage: p.ymax()
3
```

This works for any $2d$-paths:

```
sage: Paths = WordPaths('ab', steps=[(1,-1),(-1,1)])
sage: p = Paths('ababa')
sage: p.ymax()
0
sage: DyckPaths = WordPaths('ab', steps='dyck')
sage: p = DyckPaths('abaabb')
sage: p.ymax()
2
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.ymax()
2.59807621135332
```

ymin()

Returns the minimum of the y-coordinates of the path.

EXAMPLES:

```
sage: P = WordPaths('0123')
sage: p = P('0101013332')
sage: p.ymin()
0
```

This works for any $2d$-paths:

```
sage: Paths = WordPaths('ab', steps=[(1,-1),(-1,1)])
sage: p = Paths('ababa')
sage: p.ymin()
-1
sage: DyckPaths = WordPaths('ab', steps='dyck')
sage: p = DyckPaths('abaabb')
sage: p.ymin()
0
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.ymin()
0.000000000000000
```

class sage.combinat.words.paths.**FiniteWordPath_2d_callable**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable, sage.combinat.words.paths.FiniteWordPath_2d, sage.combinat.words.finite_word.FiniteWord_

INPUT:

- parent - a parent
- callable - a callable defined on range(stop=length)
- length - (default: None) nonnegative integer or None

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=False); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable'>
sage: w.length()
9

sage: w = Word(f, caching=False); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

TESTS:

```
sage: from sage.combinat.words.word_infinite_datatypes import WordDatatype_callable
sage: WordDatatype_callable(Words(),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
sage: WordDatatype_callable(Words([0,1,2]),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
```

class sage.combinat.words.paths.**FiniteWordPath_2d_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable_with_caching, sage.combinat.words.paths.FiniteWordPath_2d, sage.combinat.words.finite_word.FiniteWord_

INPUT:

- parent - a parent
- callable - a callable defined on range(stop=length)
- length - (default: None) nonnegative integer or None

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=True); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable_with_caching'>
sage: w.length()
9

sage: w = Word(f, caching=True); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable_with_caching'>
sage: w.length() is None
False
```

```
sage: w.length()
+Infinity
```

class sage.combinat.words.paths.**FiniteWordPath_2d_iter**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter, sage.combinat.words.paths.FiniteWordPath_2d, sage.combinat.words.finite_word.FiniteWord_

INPUT:

- `parent` - a parent
- `iter` - an iterator
- `length` - (default: `None`) the length of the word

EXAMPLES:

```
sage: w = Word(iter("abbabaab"), length="unknown", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length() is None
False
sage: w.length()
8
sage: s = "abbabaabbaabab babaabab baabbabaabbaabab baabbabaabab"
sage: w = Word(iter(s), length="unknown", caching=False); w
word: abbabaabbaababbabaababbaabbabaabbaababba...
sage: w.length() is None
True

sage: w = Word(iter("abbabaab"), length="finite", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8, caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
```

class sage.combinat.words.paths.**FiniteWordPath_2d_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter_with_caching, sage.combinat.words.paths.FiniteWordPath_2d, sage.combinat.words.finite_word.FiniteWord_

INPUT:

- `parent` - a parent
- `iter` - an iterator
- `length` - (default: `None`) the length of the word

EXAMPLES:

```
sage: import itertools
sage: Word(itertools.cycle("abbabaab"))
word: abbabaababbabaababbabaababbabaababbabaab...
```

```
sage: w = Word(iter("abbabaab"), length="finite"); w
word: abbabaab
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length="unknown"); w
word: abbabaab
sage: w.length()
8
sage: list(w)
['a', 'b', 'b', 'a', 'b', 'a', 'a', 'b']
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8)
sage: w._len
8
```

class sage.combinat.words.paths.**FiniteWordPath_2d_list**

Bases: sage.combinat.words.word_datatypes.WordDatatype_list, sage.combinat.words.paths.FiniteWordPath_2d, sage.combinat.words.finite_word.FiniteWord_

TESTS:

```
sage: P = WordPaths(['a','b'],[(1,2),(3,4)])
sage: p = P(['a','b','a']);p
Path: aba
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_2d_list'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_2d_str**

Bases: sage.combinat.words.word_datatypes.WordDatatype_str, sage.combinat.words.paths.FiniteWordPath_2d, sage.combinat.words.finite_word.FiniteWord_

TESTS:

```
sage: P = WordPaths(['a','b'],[(1,2),(3,4)])
sage: p = P('aba'); p
Path: aba
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_2d_str'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_2d_tuple**

Bases: sage.combinat.words.word_datatypes.WordDatatype_tuple, sage.combinat.words.paths.FiniteWordPath_2d, sage.combinat.words.finite_word.FiniteWord_

TESTS:

```
sage: P = WordPaths(['a','b'],[(1,2),(3,4)])
sage: p = P(('a','b','a'));p
Path: aba
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_2d_tuple'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_3d**

Bases: sage.combinat.words.paths.FiniteWordPath_all

plot(*pathoptions={'arrow_head': True, 'rgbcolor': 'red', 'thickness': 3}, startpoint=True, startoptions={'rgbcolor': 'red', 'size': 10}*)

INPUT:

- `pathoptions` - (dict, default:dict(rgbcolor='red',arrow_head=True, thickness=3)), options for the path drawing
- `startpoint` - (boolean, default: True), draw the start point?
- **`startoptions` - (dict, default:dict(rgbcolor='red',size=10))** options for the start point drawing

EXAMPLES:

```
sage: d = ( vector((1,3,2)), vector((2,-4,5)) )
sage: P = WordPaths(alphabet='ab', steps=d); P
Word Paths over 2 steps
sage: p = P('ababab'); p
Path: ababab
sage: p.plot()
Graphics3d Object

sage: P = WordPaths('abcABC', steps='cube_grid')
sage: p = P('abcabcAABBC')
sage: p.plot()
Graphics3d Object
```

class sage.combinat.words.paths.**FiniteWordPath_3d_callable**(*parent, callable, length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable, sage.combinat.words.paths.FiniteWordPath_3d, sage.combinat.words.finite_word.FiniteWord_

INPUT:

- `parent` - a parent
- `callable` - a callable defined on `range(stop=length)`
- `length` - (default: `None`) nonnegative integer or `None`

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=False); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable'>
sage: w.length()
9

sage: w = Word(f, caching=False); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

TESTS:

```
sage: from sage.combinat.words.word_infinite_datatypes import WordDatatype_callable
sage: WordDatatype_callable(Words(),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
```

```
sage: WordDatatype_callable(Words([0,1,2]),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
```

class sage.combinat.words.paths.**FiniteWordPath_3d_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable_with_caching, sage.combinat.words.paths.FiniteWordPath_3d, sage.combinat.words.finite_word.FiniteWord_

INPUT:

- parent - a parent
- callable - a callable defined on range(stop=length)
- length - (default: None) nonnegative integer or None

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=True); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable_with_caching'>
sage: w.length()
9

sage: w = Word(f, caching=True); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable_with_caching'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

class sage.combinat.words.paths.**FiniteWordPath_3d_iter**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter, sage.combinat.words.paths.FiniteWordPath_3d, sage.combinat.words.finite_word.FiniteWord_

INPUT:

- parent - a parent
- iter - an iterator
- length - (default: None) the length of the word

EXAMPLES:

```
sage: w = Word(iter("abbabaab"), length="unknown", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length() is None
False
sage: w.length()
8
sage: s = "abbabaabbaabbabbabaabbabbaabbabaabbaababbaabbabaabab"
sage: w = Word(iter(s), length="unknown", caching=False); w
word: abbabaabbaabbabbabaabbabbaabbabaabbaababba...
```

```
sage: w.length() is None
True

sage: w = Word(iter("abbabaab"), length="finite", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8, caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
```

class sage.combinat.words.paths.**FiniteWordPath_3d_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter_with_caching, sage.combinat.words.paths.FiniteWordPath_3d, sage.combinat.words.finite_word.FiniteWord_

INPUT:

- `parent` - a parent
- `iter` - an iterator
- `length` - (default: `None`) the length of the word

EXAMPLES:

```
sage: import itertools
sage: Word(itertools.cycle("abbabaab"))
word: abbabaababbabaababbabaababbabaababbabaab...
sage: w = Word(iter("abbabaab"), length="finite"); w
word: abbabaab
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length="unknown"); w
word: abbabaab
sage: w.length()
8
sage: list(w)
['a', 'b', 'b', 'a', 'b', 'a', 'a', 'b']
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8)
sage: w._len
8
```

class sage.combinat.words.paths.**FiniteWordPath_3d_list**

Bases: sage.combinat.words.word_datatypes.WordDatatype_list, sage.combinat.words.paths.FiniteWordPath_3d, sage.combinat.words.finite_word.FiniteWord_

TESTS:

```
sage: P = WordPaths(['a','b'],[(1,2,0),(3,4,0)])
sage: p = P(['a','b','a']);p
Path: aba
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_3d_list'>
```

```
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_3d_str**

Bases: sage.combinat.words.word_datatypes.WordDatatype_str, sage.combinat.words.paths.FiniteWordPath_3d, sage.combinat.words.finite_word.FiniteWord_

TESTS:

```
sage: P = WordPaths(['a','b'],[(1,2,0),(3,4,0)])
sage: p = P('aba'); p
Path: aba
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_3d_str'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_3d_tuple**

Bases: sage.combinat.words.word_datatypes.WordDatatype_tuple, sage.combinat.words.paths.FiniteWordPath_3d, sage.combinat.words.finite_word.FiniteWord_

TESTS:

```
sage: P = WordPaths(['a','b'],[(1,2,0),(3,4,0)])
sage: p = P(('a','b','a'));p
Path: aba
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_3d_tuple'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_all**

Bases: sage.structure.sage_object.SageObject

directive_vector()

Returns the directive vector of self.

The directive vector is the vector starting at the start point and ending at the end point of the path self.

EXAMPLES:

```
sage: WordPaths('abcdef')('abababab').directive_vector()
(6, 2*sqrt3)
sage: WordPaths('abAB')('abababab').directive_vector()
(4, 4)
sage: P = WordPaths('abcABC', steps='cube_grid')
sage: P('abababCC').directive_vector()
(4, 4, -2)
sage: WordPaths('abcdef')('abcdef').directive_vector()
(0, 0)
sage: P = WordPaths('abc', steps=[(1,3,7,9),(-4,1,0,0),(0,32,1,8)])
sage: P('abcababababacaacccbbcac').directive_vector()
(-16, 254, 63, 128)
```

end_point()

Returns the end point of the path.

EXAMPLES:

```
sage: WordPaths('abcdef')('abababab').end_point()
(6, 2*sqrt3)
```

```
sage: WordPaths('abAB')('abababab').end_point()
(4, 4)
sage: P = WordPaths('abcABC', steps='cube_grid')
sage: P('abababCC').end_point()
(4, 4, -2)
sage: WordPaths('abcdef')('abcdef').end_point()
(0, 0)
sage: P = WordPaths('abc', steps=[(1,3,7,9),(-4,1,0,0),(0,32,1,8)])
sage: P('abcababababacaacccbbcac').end_point()
(-16, 254, 63, 128)
```

is_closed()

Returns True if the path is closed, i.e. if the origin and the end of the path are equal.

EXAMPLES:

```
sage: P = WordPaths('abcd', steps=[(1,0),(0,1),(-1,0),(0,-1)])
sage: P('abcd').is_closed()
True
sage: P('abc').is_closed()
False
sage: P().is_closed()
True
sage: P('aacacc').is_closed()
True
```

is_simple()

Returns True if the path is simple, i.e. if all its points are distincts.

If the path is closed, the last point is not considered.

EXAMPLES:

```
sage: P = WordPaths('abcdef',steps='triangle_grid');P
Word Paths on the triangle grid
sage: P('abc').is_simple()
True
sage: P('abcde').is_simple()
True
sage: P('abcdef').is_simple()
True
sage: P('ad').is_simple()
True
sage: P('aabdee').is_simple()
False
```

is_tangent()

The is_tangent() method, which is implemented for words, has an extended meaning for word paths, which is not implemented yet.

TESTS:

```
sage: WordPaths('ab')('abbab').is_tangent()
Traceback (most recent call last):
...
NotImplementedError
```

AUTHOR:

- Thierry Monteil

plot_projection(*v=None*, *letters=None*, *color=None*, *ring=None*, *size=12*, *kind='right'*)

Return an image of the projection of the successive points of the path into the space orthogonal to the given vector.

INPUT:

- `self` - a word path in a 3 or 4 dimension vector space
- `v` - vector (optional, default: None) If None, the directive vector (i.e. the end point minus starting point) of the path is considered.
- `letters` - iterable (optional, default: None) of the letters to be projected. If None, then all the letters are considered.
- `color` - dictionary (optional, default: None) of the letters mapped to colors. If None, automatic colors are chosen.
- `ring` - ring (optional, default: None) where to do the computations. If None, RealField(53) is used.
- `size` - number (optional, default: `12`) size of the points.
- `kind` - string (optional, default `'right'`) either `'right'` or `'left'`. The color of a letter is given to the projected prefix to the right or the left of the letter.

OUTPUT:

2d or 3d Graphic object.

EXAMPLES:

The Rauzy fractal:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: D = s.fixed_point('1')
sage: v = s.pisot_eigenvector_right()
sage: P = WordPaths('123',[(1,0,0),(0,1,0),(0,0,1)])
sage: w = P(D[:200])
sage: w.plot_projection(v)  # long time (2s)
Graphics object consisting of 200 graphics primitives
```

In this case, the abelianized vector doesn't give a good projection:

```
sage: w.plot_projection()  # long time (2s)
Graphics object consisting of 200 graphics primitives
```

You can project only the letters you want:

```
sage: w.plot_projection(v, letters='12')  # long time (2s)
Graphics object consisting of 168 graphics primitives
```

You can increase or decrease the precision of the computations by changing the ring of the projection matrix:

```
sage: w.plot_projection(v, ring=RealField(20))  # long time (2s)
Graphics object consisting of 200 graphics primitives
```

You can change the size of the points:

```
sage: w.plot_projection(v, size=30)  # long time (2s)
Graphics object consisting of 200 graphics primitives
```

You can assign the color of a letter to the projected prefix to the right or the left of the letter:

```
sage: w.plot_projection(v, kind='left')  # long time (2s)
Graphics object consisting of 200 graphics primitives
```

To remove the axis, do like this:

```
sage: r = w.plot_projection(v)
sage: r.axes(False)
sage: r                # long time (2s)
Graphics object consisting of 200 graphics primitives
```

You can assign different colors to each letter:

```
sage: color = {'1':'purple', '2':(.2,.3,.4), '3': 'magenta'}
sage: w.plot_projection(v, color=color)  # long time (2s)
Graphics object consisting of 200 graphics primitives
```

The 3d-Rauzy fractal:

```
sage: s = WordMorphism('1->12,2->13,3->14,4->1')
sage: D = s.fixed_point('1')
sage: v = s.pisot_eigenvector_right()
sage: P = WordPaths('1234',[(1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1)])
sage: w = P(D[:200])
sage: w.plot_projection(v)
Graphics3d Object
```

The dimension of vector space of the parent must be 3 or 4:

```
sage: P = WordPaths('ab', [(1, 0), (0, 1)])
sage: p = P('aabbabbab')
sage: p.plot_projection()
Traceback (most recent call last):
...
TypeError: The dimension of the vector space (=2) must be 3 or 4
```

points(*include_last=True*)

Returns an iterator yielding a list of points used to draw the path represented by this word.

INPUT:

- `include_last` - bool (default: True) whether to include the last point

EXAMPLES:

A simple closed square:

```
sage: P = WordPaths('abAB')
sage: list(P('abAB').points())
[(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)]
```

A simple closed square without the last point:

```
sage: list(P('abAB').points(include_last=False))
[(0, 0), (1, 0), (1, 1), (0, 1)]

sage: list(P('abaB').points())
[(0, 0), (1, 0), (1, 1), (2, 1), (2, 0)]
```

projected_path(*v=None*, *ring=None*)

Return the path projected into the space orthogonal to the given vector.

INPUT:

- `v` - vector (optional, default: None) If None, the directive vector (i.e. the end point minus starting point) of the path is considered.
- `ring` - ring (optional, default: None) where to do the computations. If None, RealField(53) is used.

OUTPUT:

word path

EXAMPLES:

The projected path of the tribonacci word:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: D = s.fixed_point('1')
sage: v = s.pisot_eigenvector_right()
sage: P = WordPaths('123',[(1,0,0),(0,1,0),(0,0,1)])
sage: w = P(D[:1000])
sage: p = w.projected_path(v)
sage: p
Path: 12131211213121213121112131213121121312121...
sage: p[:20].plot()
Graphics object consisting of 3 graphics primitives
```

The `ring` argument allows to change the precision of the projected steps:

```
sage: p = w.projected_path(v, RealField(10))
sage: p
Path: 12131211213121213121112131213121121312121...
sage: p.parent().letters_to_steps()
{'1': (-0.53, 0.00), '2': (0.75, -0.48), '3': (0.41, 0.88)}
```

projected_point_iterator(*v=None*, *ring=None*)

Return an iterator of the projection of the orbit points of the path into the space orthogonal to the given vector.

INPUT:

- `v` - vector (optional, default: None) If None, the directive vector (i.e. the end point minus starting point) of the path is considered.
- `ring` - ring (optional, default: None) where to do the computations. If None, RealField(53) is used.

OUTPUT:

iterator of points

EXAMPLES:

Projected points of the Rauzy fractal:

```
sage: s = WordMorphism('1->12,2->13,3->1')
sage: D = s.fixed_point('1')
sage: v = s.pisot_eigenvector_right()
sage: P = WordPaths('123',[(1,0,0),(0,1,0),(0,0,1)])
sage: w = P(D[:200])
sage: it = w.projected_point_iterator(v)
sage: for i in range(6): next(it)
(0.000000000000000, 0.000000000000000)
(-0.526233343362516, 0.000000000000000)
(0.220830337618112, -0.477656250512816)
(-0.305403005744404, -0.477656250512816)
(0.100767309386062, 0.400890564600664)
(-0.425466033976454, 0.400890564600664)
```

Projected points of a 2d path:

```
sage: P = WordPaths('ab','ne')
sage: p = P('aabbabbab')
sage: it = p.projected_point_iterator(ring=RealField(20))
sage: for i in range(8): next(it)
(0.00000)
(0.78087)
(1.5617)
(0.93704)
(0.31235)
(1.0932)
(0.46852)
(-0.15617)
```

start_point()

Return the starting point of self.

OUTPUT:

vector

EXAMPLES:

```
sage: WordPaths('abcdef')('abcdef').start_point()
(0, 0)
sage: WordPaths('abcdef', steps='cube_grid')('abcdef').start_point()
(0, 0, 0)
sage: P = WordPaths('ab', steps=[(1,0,0,0),(0,1,0,0)])
sage: P('abbba').start_point()
(0, 0, 0, 0)
```

tikz_trajectory()

Returns the trajectory of self as a tikz str.

EXAMPLES:

```
sage: P = WordPaths('abcdef')
sage: p = P('abcde')
sage: p.tikz_trajectory()
'(0.000, 0.000) -- (1.00, 0.000) -- (1.50, 0.866) -- (1.00, 1.73) -- (0.000, 1.73) -- (-0.50
```

class sage.combinat.words.paths.**FiniteWordPath_all_callable**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable, sage.combinat.words.paths.FiniteWordPath_all, sage.combinat.words.finite_word.FiniteWord

INPUT:

- parent - a parent
- callable - a callable defined on range(stop=length)
- length - (default: None) nonnegative integer or None

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=False); w
word: xyxyxyxyx
sage: type(w)
```

```
<class 'sage.combinat.words.word.FiniteWord_callable'>
sage: w.length()
9

sage: w = Word(f, caching=False); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

TESTS:

```
sage: from sage.combinat.words.word_infinite_datatypes import WordDatatype_callable
sage: WordDatatype_callable(Words(),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
sage: WordDatatype_callable(Words([0,1,2]),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
```

class sage.combinat.words.paths.**FiniteWordPath_all_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable_with_caching, sage.combinat.words.paths.FiniteWordPath_all, sage.combinat.words.finite_word.FiniteWord

INPUT:

- parent - a parent
- callable - a callable defined on range(stop=length)
- length - (default: None) nonnegative integer or None

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=True); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable_with_caching'>
sage: w.length()
9

sage: w = Word(f, caching=True); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable_with_caching'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

class sage.combinat.words.paths.**FiniteWordPath_all_iter**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter, sage.combinat.words.paths.FiniteWordPath_all, sage.combinat.words.finite_word.FiniteWord

INPUT:

- parent - a parent

- iter - an iterator
- length - (default: None) the length of the word

EXAMPLES:

```
sage: w = Word(iter("abbabaab"), length="unknown", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length() is None
False
sage: w.length()
8
sage: s = "abbabaabbaababbabaababbaabbabaabbaababbaabbabaabab"
sage: w = Word(iter(s), length="unknown", caching=False); w
word: abbabaabbaababbabaababbaabbabaabbaababba...
sage: w.length() is None
True

sage: w = Word(iter("abbabaab"), length="finite", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8, caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
```

class sage.combinat.words.paths.**FiniteWordPath_all_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter_with_caching, sage.combinat.words.paths.FiniteWordPath_all, sage.combinat.words.finite_word.FiniteWord

INPUT:

- parent - a parent
- iter - an iterator
- length - (default: None) the length of the word

EXAMPLES:

```
sage: import itertools
sage: Word(itertools.cycle("abbabaab"))
word: abbabaababbabaababbabaababbabaababbabaab...
sage: w = Word(iter("abbabaab"), length="finite"); w
word: abbabaab
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length="unknown"); w
word: abbabaab
sage: w.length()
8
sage: list(w)
['a', 'b', 'b', 'a', 'b', 'a', 'a', 'b']
```

```
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8)
sage: w._len
8
```

class sage.combinat.words.paths.**FiniteWordPath_all_list**

Bases: sage.combinat.words.word_datatypes.WordDatatype_list, sage.combinat.words.paths.FiniteWordPath_all, sage.combinat.words.finite_word.FiniteWord

TESTS:

```
sage: P = WordPaths(['a','b'],[(1,2,0,0),(3,4,0,0)])
sage: p = P(['a','b','a']);p
Path: aba
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_all_list'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_all_str**

Bases: sage.combinat.words.word_datatypes.WordDatatype_str, sage.combinat.words.paths.FiniteWordPath_all, sage.combinat.words.finite_word.FiniteWord

TESTS:

```
sage: P = WordPaths('ab',[(1,2,0,0),(3,4,0,0)])
sage: p = P('aabbb'); p
Path: aabbb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_all_str'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_all_tuple**

Bases: sage.combinat.words.word_datatypes.WordDatatype_tuple, sage.combinat.words.paths.FiniteWordPath_all, sage.combinat.words.finite_word.FiniteWord

TESTS:

```
sage: P = WordPaths('ab',[(1,2,0,0),(3,4,0,0)])
sage: p = P( ('a','b','b') ); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_all_tuple'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_cube_grid**

Bases: sage.combinat.words.paths.FiniteWordPath_3d

class sage.combinat.words.paths.**FiniteWordPath_cube_grid_callable**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable, sage.combinat.words.paths.FiniteWordPath_cube_grid, sage.combinat.words.finite_word.Fini

INPUT:

- `parent` - a parent

- `callable` - a callable defined on `range(stop=length)`
- `length` - (default: `None`) nonnegative integer or `None`

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=False); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable'>
sage: w.length()
9

sage: w = Word(f, caching=False); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

TESTS:

```
sage: from sage.combinat.words.word_infinite_datatypes import WordDatatype_callable
sage: WordDatatype_callable(Words(),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
sage: WordDatatype_callable(Words([0,1,2]),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
```

class sage.combinat.words.paths.**FiniteWordPath_cube_grid_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable_with_caching, sage.combinat.words.paths.FiniteWordPath_cube_grid, sage.combinat.words.finite_word.Fini

INPUT:

- `parent` - a parent
- `callable` - a callable defined on `range(stop=length)`
- `length` - (default: `None`) nonnegative integer or `None`

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=True); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable_with_caching'>
sage: w.length()
9

sage: w = Word(f, caching=True); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable_with_caching'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

class sage.combinat.words.paths.**FiniteWordPath_cube_grid_iter**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter, sage.combinat.words.paths.FiniteWordPath_cube_grid, sage.combinat.words.finite_word.Fin

INPUT:

- parent - a parent
- iter - an iterator
- length - (default: None) the length of the word

EXAMPLES:

```
sage: w = Word(iter("abbabaab"), length="unknown", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length() is None
False
sage: w.length()
8
sage: s = "abbabaabbaababbabaababbaabbabaabbaababbaabbabaabab"
sage: w = Word(iter(s), length="unknown", caching=False); w
word: abbabaabbaababbabaababbaabbabaabbaababba...
sage: w.length() is None
True

sage: w = Word(iter("abbabaab"), length="finite", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8, caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
```

class sage.combinat.words.paths.**FiniteWordPath_cube_grid_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter_with_caching, sage.combinat.words.paths.FiniteWordPath_cube_grid, sage.combinat.words.finite_word.Fin

INPUT:

- parent - a parent
- iter - an iterator
- length - (default: None) the length of the word

EXAMPLES:

```
sage: import itertools
sage: Word(itertools.cycle("abbabaab"))
word: abbabaababbabaababbabaababbabaababbabaab...
```

```
sage: w = Word(iter("abbabaab"), length="finite"); w
word: abbabaab
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length="unknown"); w
word: abbabaab
sage: w.length()
8
sage: list(w)
['a', 'b', 'b', 'a', 'b', 'a', 'a', 'b']
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8)
sage: w._len
8
```

class sage.combinat.words.paths.**FiniteWordPath_cube_grid_list**

Bases: sage.combinat.words.word_datatypes.WordDatatype_list, sage.combinat.words.paths.FiniteWordPath_cube_grid, sage.combinat.words.finite_word.Fini

TESTS:

```
sage: P = WordPaths('abcdef', steps='cube')
sage: p = P(['a','b','b']); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_cube_grid_list'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_cube_grid_str**

Bases: sage.combinat.words.word_datatypes.WordDatatype_str, sage.combinat.words.paths.FiniteWordPath_cube_grid, sage.combinat.words.finite_word.Fini

TESTS:

```
sage: P = WordPaths('abcdef', steps='cube')
sage: p = P('abb'); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_cube_grid_str'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_cube_grid_tuple**

Bases: sage.combinat.words.word_datatypes.WordDatatype_tuple, sage.combinat.words.paths.FiniteWordPath_cube_grid, sage.combinat.words.finite_word.Fini

TESTS:

```
sage: P = WordPaths('abcdef', steps='cube')
sage: p = P(('a','b','b')); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_cube_grid_tuple'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_dyck**

Bases: sage.combinat.words.paths.FiniteWordPath_2d

class sage.combinat.words.paths.**FiniteWordPath_dyck_callable**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable, sage.combinat.words.paths.FiniteWordPath_dyck, sage.combinat.words.finite_word.FiniteWo

INPUT:

- parent - a parent
- callable - a callable defined on range(stop=length)
- length - (default: None) nonnegative integer or None

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=False); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable'>
sage: w.length()
9

sage: w = Word(f, caching=False); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

TESTS:

```
sage: from sage.combinat.words.word_infinite_datatypes import WordDatatype_callable
sage: WordDatatype_callable(Words(),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
sage: WordDatatype_callable(Words([0,1,2]),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
```

class sage.combinat.words.paths.**FiniteWordPath_dyck_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable_with_caching, sage.combinat.words.paths.FiniteWordPath_dyck, sage.combinat.words.finite_word.FiniteWo

INPUT:

- parent - a parent
- callable - a callable defined on range(stop=length)
- length - (default: None) nonnegative integer or None

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=True); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable_with_caching'>
sage: w.length()
9
```

```
sage: w = Word(f, caching=True); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable_with_caching'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

class sage.combinat.words.paths.**FiniteWordPath_dyck_iter**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter, sage.combinat.words.paths.FiniteWordPath_dyck, sage.combinat.words.finite_word.FiniteWo

INPUT:

- parent - a parent
- iter - an iterator
- length - (default: None) the length of the word

EXAMPLES:

```
sage: w = Word(iter("abbabaab"), length="unknown", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length() is None
False
sage: w.length()
8
sage: s = "abbabaabbaababbabaababbaabbabaabbaababbaabbabaabab"
sage: w = Word(iter(s), length="unknown", caching=False); w
word: abbabaabbaababbabaababbaabbabaabbaababba...
sage: w.length() is None
True

sage: w = Word(iter("abbabaab"), length="finite", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8, caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
```

class sage.combinat.words.paths.**FiniteWordPath_dyck_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter_with_caching, sage.combinat.words.paths.FiniteWordPath_dyck, sage.combinat.words.finite_word.FiniteWo

INPUT:

- parent - a parent
- iter - an iterator

- `length` - (default: `None`) the length of the word

EXAMPLES:

```
sage: import itertools
sage: Word(itertools.cycle("abbabaab"))
word: abbabaababbabaababbabaababbabaababbabaab...
sage: w = Word(iter("abbabaab"), length="finite"); w
word: abbabaab
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length="unknown"); w
word: abbabaab
sage: w.length()
8
sage: list(w)
['a', 'b', 'b', 'a', 'b', 'a', 'a', 'b']
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8)
sage: w._len
8
```

class sage.combinat.words.paths.**FiniteWordPath_dyck_list**

Bases: sage.combinat.words.word_datatypes.WordDatatype_list, sage.combinat.words.paths.FiniteWordPath_dyck, sage.combinat.words.finite_word.FiniteWo

TESTS:

```
sage: P = WordPaths('ab', steps='dyck')
sage: p = P(['a','b','b']); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_dyck_list'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_dyck_str**

Bases: sage.combinat.words.word_datatypes.WordDatatype_str, sage.combinat.words.paths.FiniteWordPath_dyck, sage.combinat.words.finite_word.FiniteWo

TESTS:

```
sage: P = WordPaths('ab', steps='dyck')
sage: p = P('abb'); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_dyck_str'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_dyck_tuple**

Bases: sage.combinat.words.word_datatypes.WordDatatype_tuple, sage.combinat.words.paths.FiniteWordPath_dyck, sage.combinat.words.finite_word.FiniteWo

TESTS:

```
sage: P = WordPaths('ab', steps='dyck')
sage: p = P(('a','b','b')); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_dyck_tuple'>
```

```
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid**(*parent*, **args*, ***kwds*)

Bases: sage.combinat.words.paths.FiniteWordPath_triangle_grid

INPUT:

- `parent` - a parent object inheriting from Words_all that has the alphabet attribute defined
- `*args, **kwds` - arguments accepted by AbstractWord

EXAMPLES:

```
sage: F = WordPaths('abcdef', steps='hexagon'); F
Word Paths on the hexagonal grid
sage: f = F('aaabbbccddef'); f
Path: aaabbbccddef

sage: f == loads(dumps(f))
True
```

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid_callable**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable, sage.combinat.words.paths.FiniteWordPath_hexagonal_grid, sage.combinat.words.finite_word.FiniteWord_class

INPUT:

- `parent` - a parent
- `callable` - a callable defined on `range(stop=length)`
- `length` - (default: `None`) nonnegative integer or `None`

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=False); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable'>
sage: w.length()
9

sage: w = Word(f, caching=False); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

TESTS:

```
sage: from sage.combinat.words.word_infinite_datatypes import WordDatatype_callable
sage: WordDatatype_callable(Words(),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
```

```
sage: WordDatatype_callable(Words([0,1,2]),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
```

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable_with_caching, sage.combinat.words.paths.FiniteWordPath_hexagonal_grid, sage.combinat.words.finite_word.FiniteWord_class

INPUT:

- `parent` - a parent
- `callable` - a callable defined on `range(stop=length)`
- `length` - (default: `None`) nonnegative integer or `None`

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=True); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable_with_caching'>
sage: w.length()
9

sage: w = Word(f, caching=True); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable_with_caching'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid_iter**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter, sage.combinat.words.paths.FiniteWordPath_hexagonal_grid, sage.combinat.words.finite_word.FiniteWord_class

INPUT:

- `parent` - a parent
- `iter` - an iterator
- `length` - (default: `None`) the length of the word

EXAMPLES:

```
sage: w = Word(iter("abbabaab"), length="unknown", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length() is None
False
sage: w.length()
8
sage: s = "abbabaabbaababbabaababbaabbabaabbaababbaabbabaabab"
```

```
sage: w = Word(iter(s), length="unknown", caching=False); w
word: abbabaabbaababbabaababbaabbabaabbaababba...
sage: w.length() is None
True

sage: w = Word(iter("abbabaab"), length="finite", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8, caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
```

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter_with_caching, sage.combinat.words.paths.FiniteWordPath_hexagonal_grid, sage.combinat.words.finite_word.FiniteWord_class

INPUT:

- `parent` - a parent
- `iter` - an iterator
- `length` - (default: `None`) the length of the word

EXAMPLES:

```
sage: import itertools
sage: Word(itertools.cycle("abbabaab"))
word: abbabaababbabaababbabaababbabaababbabaab...
sage: w = Word(iter("abbabaab"), length="finite"); w
word: abbabaab
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length="unknown"); w
word: abbabaab
sage: w.length()
8
sage: list(w)
['a', 'b', 'b', 'a', 'b', 'a', 'a', 'b']
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8)
sage: w._len
8
```

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid_list**

Bases: sage.combinat.words.word_datatypes.WordDatatype_list, sage.combinat.words.paths.FiniteWordPath_hexagonal_grid, sage.combinat.words.finite_word.FiniteWord_class

TESTS:

```
sage: P = WordPaths('abcdef', steps='hexagon')
sage: p = P(['a','b','b']); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_hexagonal_grid_list'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid_str**

Bases: sage.combinat.words.word_datatypes.WordDatatype_str, sage.combinat.words.paths.FiniteWordPath_hexagonal_grid, sage.combinat.words.finite_word.FiniteWord_class

TESTS:

```
sage: P = WordPaths('abcdef', steps='hexagon')
sage: p = P('abb'); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_hexagonal_grid_str'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_hexagonal_grid_tuple**

Bases: sage.combinat.words.word_datatypes.WordDatatype_tuple, sage.combinat.words.paths.FiniteWordPath_hexagonal_grid, sage.combinat.words.finite_word.FiniteWord_class

TESTS:

```
sage: P = WordPaths('abcdef', steps='hexagon')
sage: p = P(('a','b','b')); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_hexagonal_grid_tuple'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_north_east**

Bases: sage.combinat.words.paths.FiniteWordPath_2d

class sage.combinat.words.paths.**FiniteWordPath_north_east_callable**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable, sage.combinat.words.paths.FiniteWordPath_north_east, sage.combinat.words.finite_word.Fi

INPUT:

- `parent` - a parent
- `callable` - a callable defined on `range(stop=length)`
- `length` - (default: `None`) nonnegative integer or `None`

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=False); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable'>
```

```
sage: w.length()
9

sage: w = Word(f, caching=False); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

TESTS:

```
sage: from sage.combinat.words.word_infinite_datatypes import WordDatatype_callable
sage: WordDatatype_callable(Words(),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
sage: WordDatatype_callable(Words([0,1,2]),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
```

class sage.combinat.words.paths.**FiniteWordPath_north_east_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable_with_caching, sage.combinat.words.paths.FiniteWordPath_north_east, sage.combinat.words.finite_word.Fi

INPUT:

- `parent` - a parent
- `callable` - a callable defined on `range(stop=length)`
- `length` - (default: `None`) nonnegative integer or `None`

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=True); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable_with_caching'>
sage: w.length()
9

sage: w = Word(f, caching=True); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable_with_caching'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

class sage.combinat.words.paths.**FiniteWordPath_north_east_iter**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter, sage.combinat.words.paths.FiniteWordPath_north_east, sage.combinat.words.finite_word.Fi

INPUT:

- `parent` - a parent

- iter - an iterator
- length - (default: None) the length of the word

EXAMPLES:

```
sage: w = Word(iter("abbabaab"), length="unknown", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length() is None
False
sage: w.length()
8
sage: s = "abbabaabbaababbabaababbaabbabaabbaababbaabbabaabab"
sage: w = Word(iter(s), length="unknown", caching=False); w
word: abbabaabbaababbabaababbaabbabaabbaababba...
sage: w.length() is None
True

sage: w = Word(iter("abbabaab"), length="finite", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8, caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
```

class sage.combinat.words.paths.**FiniteWordPath_north_east_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter_with_caching, sage.combinat.words.paths.FiniteWordPath_north_east, sage.combinat.words.finite_word.Fi

INPUT:

- parent - a parent
- iter - an iterator
- length - (default: None) the length of the word

EXAMPLES:

```
sage: import itertools
sage: Word(itertools.cycle("abbabaab"))
word: abbabaababbabaababbabaababbabaababbabaab...
sage: w = Word(iter("abbabaab"), length="finite"); w
word: abbabaab
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length="unknown"); w
word: abbabaab
sage: w.length()
8
sage: list(w)
['a', 'b', 'b', 'a', 'b', 'a', 'a', 'b']
```

```
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8)
sage: w._len
8
```

class sage.combinat.words.paths.**FiniteWordPath_north_east_list**

Bases: sage.combinat.words.word_datatypes.WordDatatype_list, sage.combinat.words.paths.FiniteWordPath_north_east, sage.combinat.words.finite_word.Fi

TESTS:

```
sage: P = WordPaths('ab', steps='ne')
sage: p = P(['a','b','b']); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_north_east_list'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_north_east_str**

Bases: sage.combinat.words.word_datatypes.WordDatatype_str, sage.combinat.words.paths.FiniteWordPath_north_east, sage.combinat.words.finite_word.Fi

TESTS:

```
sage: P = WordPaths('ab', steps='ne')
sage: p = P('abb'); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_north_east_str'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_north_east_tuple**

Bases: sage.combinat.words.word_datatypes.WordDatatype_tuple, sage.combinat.words.paths.FiniteWordPath_north_east, sage.combinat.words.finite_word.Fi

TESTS:

```
sage: P = WordPaths('ab', steps='ne')
sage: p = P(('a','b','b')); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_north_east_tuple'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_square_grid**

Bases: sage.combinat.words.paths.FiniteWordPath_2d

area()

Returns the area of a closed path.

INPUT:

- `self` - a closed path

EXAMPLES:

```
sage: P = WordPaths('abAB', steps='square_grid')
sage: P('abAB').area()
1
sage: P('aabbAABB').area()
4
sage: P('aabbABAB').area()
3
```

The area of the Fibonacci tiles:

```
sage: [words.fibonacci_tile(i).area() for i in range(6)]
[1, 5, 29, 169, 985, 5741]
sage: [words.dual_fibonacci_tile(i).area() for i in range(6)]
[1, 5, 29, 169, 985, 5741]
sage: oeis(_)[0]                              # optional -- internet
A001653: Numbers n such that 2*n^2 - 1 is a square.
sage: _.first_terms()                         # optional -- internet
(1,
 5,
 29,
 169,
 985,
 5741,
 33461,
 195025,
 1136689,
 6625109,
 38613965,
 225058681,
 1311738121,
 7645370045,
 44560482149,
 259717522849,
 1513744654945,
 8822750406821,
 51422757785981,
 299713796309065,
 1746860020068409,
 10181446324101389,
 59341817924539925)
```

TESTS:

```
sage: P = WordPaths('abAB', steps='square_grid')
sage: P('a').area()
Traceback (most recent call last):
...
TypeError: the path must be closed to compute its area
```

is_closed()

Returns True if self represents a closed path and False otherwise.

EXAMPLES:

```
sage: P = WordPaths('abAB', steps='square_grid')
sage: P('aA').is_closed()
True
sage: P('abAB').is_closed()
True
sage: P('ababAABB').is_closed()
```

```
True
sage: P('aaabbbAABB').is_closed()
False
sage: P('ab').is_closed()
False
```

is_simple()

Returns True if the path is simple, i.e. if all its points are distincts.

If the path is closed, the last point is not considered.

Note: The linear algorithm described in the thesis of Xavier Provençal should be implemented here.

EXAMPLES:

```
sage: P = WordPaths('abAB', steps='square_grid')
sage: P('abab').is_simple()
True
sage: P('abAB').is_simple()
True
sage: P('abA').is_simple()
True
sage: P('aabABB').is_simple()
False
sage: P().is_simple()
True
sage: P('A').is_simple()
True
sage: P('aA').is_simple()
True
sage: P('aaA').is_simple()
False
```

REFERENCES:

- Provençal, X., Combinatoires des mots, geometrie discrete et pavages, These de doctorat en Mathematiques, Montreal, UQAM, septembre 2008, 115 pages.

tikz_trajectory()

Returns the trajectory of self as a tikz str.

EXAMPLES:

```
sage: f = words.fibonacci_tile(1)
sage: f.tikz_trajectory()
'(0, 0) -- (0, -1) -- (-1, -1) -- (-1, -2) -- (0, -2) -- (0, -3) -- (1, -3) -- (1, -2) -- (2
```

class sage.combinat.words.paths.**FiniteWordPath_square_grid_callable**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable, sage.combinat.words.paths.FiniteWordPath_square_grid, sage.combinat.words.finite_word.FiniteWord_class

INPUT:

- `parent` - a parent
- `callable` - a callable defined on `range(stop=length)`
- `length` - (default: `None`) nonnegative integer or `None`

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=False); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable'>
sage: w.length()
9

sage: w = Word(f, caching=False); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

TESTS:

```
sage: from sage.combinat.words.word_infinite_datatypes import WordDatatype_callable
sage: WordDatatype_callable(Words(),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
sage: WordDatatype_callable(Words([0,1,2]),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
```

class sage.combinat.words.paths.**FiniteWordPath_square_grid_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable_with_caching, sage.combinat.words.paths.FiniteWordPath_square_grid, sage.combinat.words.finite_word.FiniteWord_class

INPUT:

- parent - a parent
- callable - a callable defined on range(stop=length)
- length - (default: None) nonnegative integer or None

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=True); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable_with_caching'>
sage: w.length()
9

sage: w = Word(f, caching=True); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable_with_caching'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

class sage.combinat.words.paths.**FiniteWordPath_square_grid_iter**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter, sage.combinat.words.paths.FiniteWordPath_square_grid, sage.combinat.words.finite_word.FiniteWord_class

INPUT:

- parent - a parent
- iter - an iterator
- length - (default: None) the length of the word

EXAMPLES:

```
sage: w = Word(iter("abbabaab"), length="unknown", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length() is None
False
sage: w.length()
8
sage: s = "abbabaabbaababbabaababbaabbabaabbaababbaabbabaabab"
sage: w = Word(iter(s), length="unknown", caching=False); w
word: abbabaabbaababbabaababbaabbabaabbaababba...
sage: w.length() is None
True

sage: w = Word(iter("abbabaab"), length="finite", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8, caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
```

class sage.combinat.words.paths.**FiniteWordPath_square_grid_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter_with_caching, sage.combinat.words.paths.FiniteWordPath_square_grid, sage.combinat.words.finite_word.FiniteWord_class

INPUT:

- parent - a parent
- iter - an iterator
- length - (default: None) the length of the word

EXAMPLES:

```
sage: import itertools
sage: Word(itertools.cycle("abbabaab"))
word: abbabaababbabaababbabaababbabaababbabaab...
```

```
sage: w = Word(iter("abbabaab"), length="finite"); w
word: abbabaab
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length="unknown"); w
word: abbabaab
sage: w.length()
8
sage: list(w)
['a', 'b', 'b', 'a', 'b', 'a', 'a', 'b']
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8)
sage: w._len
8
```

class sage.combinat.words.paths.**FiniteWordPath_square_grid_list**

Bases: sage.combinat.words.word_datatypes.WordDatatype_list, sage.combinat.words.paths.FiniteWordPath_square_grid, sage.combinat.words.finite_word.FiniteWord_class

TESTS:

```
sage: P = WordPaths('abcd', steps='square')
sage: p = P(['a','b','b']); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_square_grid_list'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_square_grid_str**

Bases: sage.combinat.words.word_datatypes.WordDatatype_str, sage.combinat.words.paths.FiniteWordPath_square_grid, sage.combinat.words.finite_word.FiniteWord_class

TESTS:

```
sage: P = WordPaths('abcd', steps='square')
sage: p = P('abccc'); p
Path: abccc
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_square_grid_str'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_square_grid_tuple**

Bases: sage.combinat.words.word_datatypes.WordDatatype_tuple, sage.combinat.words.paths.FiniteWordPath_square_grid, sage.combinat.words.finite_word.FiniteWord_class

TESTS:

```
sage: P = WordPaths('abcd', steps='square')
sage: p = P(('a','b','b')); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_square_grid_tuple'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid**

Bases: sage.combinat.words.paths.FiniteWordPath_2d

xmax()

Returns the maximum of the x-coordinates of the path.

EXAMPLES:

```
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.xmax()
4.500000000000000
sage: w = WordPaths('abcABC', steps='triangle')('ABAcacacababababcbcbAC')
sage: w.xmax()
4.000000000000000
```

xmin()

Returns the minimum of the x-coordinates of the path.

EXAMPLES:

```
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.xmin()
0.000000000000000
sage: w = WordPaths('abcABC', steps='triangle')('ABAcacacababababcbcbAC')
sage: w.xmin()
-3.000000000000000
```

ymax()

Returns the maximum of the y-coordinates of the path.

EXAMPLES:

```
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.ymax()
2.59807621135332
sage: w = WordPaths('abcABC', steps='triangle')('ABAcacacababababcbcbAC')
sage: w.ymax()
8.66025403784439
```

ymin()

Returns the minimum of the y-coordinates of the path.

EXAMPLES:

```
sage: w = WordPaths('abcABC', steps='triangle')('ababcaaBC')
sage: w.ymin()
0.000000000000000
sage: w = WordPaths('abcABC', steps='triangle')('ABAcacacababababcbcbAC')
sage: w.ymin()
-0.866025403784439
```

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid_callable**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable, sage.combinat.words.paths.FiniteWordPath_triangle_grid, sage.combinat.words.finite_word.FiniteWord_class

INPUT:

- parent - a parent

- `callable` - a callable defined on `range(stop=length)`
- `length` - (default: `None`) nonnegative integer or `None`

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=False); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable'>
sage: w.length()
9

sage: w = Word(f, caching=False); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable'>
sage: w.length() is None
False
sage: w.length()
+Infinity
```

TESTS:

```
sage: from sage.combinat.words.word_infinite_datatypes import WordDatatype_callable
sage: WordDatatype_callable(Words(),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
sage: WordDatatype_callable(Words([0,1,2]),lambda n:n%3)
<sage.combinat.words.word_infinite_datatypes.WordDatatype_callable object at ...>
```

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: `sage.combinat.words.word_infinite_datatypes.WordDatatype_callable_with_caching`, `sage.combinat.words.paths.FiniteWordPath_triangle_grid`, `sage.combinat.words.finite_word.FiniteWord_class`

INPUT:

- `parent` - a parent
- `callable` - a callable defined on `range(stop=length)`
- `length` - (default: `None`) nonnegative integer or `None`

EXAMPLES:

```
sage: f = lambda n : 'x' if n % 2 == 0 else 'y'
sage: w = Word(f, length=9, caching=True); w
word: xyxyxyxyx
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable_with_caching'>
sage: w.length()
9

sage: w = Word(f, caching=True); w
word: xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy...
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable_with_caching'>
sage: w.length() is None
False
```

```
sage: w.length()
+Infinity
```

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid_iter**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter, sage.combinat.words.paths.FiniteWordPath_triangle_grid, sage.combinat.words.finite_word.FiniteWord_class

INPUT:

- parent - a parent
- iter - an iterator
- length - (default: None) the length of the word

EXAMPLES:

```
sage: w = Word(iter("abbabaab"), length="unknown", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length() is None
False
sage: w.length()
8
sage: s = "abbabaabbaababbabaababbaabbabaabbaababbaabbabaabab"
sage: w = Word(iter(s), length="unknown", caching=False); w
word: abbabaabbaababbabaababbaabbabaabbaababba...
sage: w.length() is None
True

sage: w = Word(iter("abbabaab"), length="finite", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8, caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
```

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter_with_caching, sage.combinat.words.paths.FiniteWordPath_triangle_grid, sage.combinat.words.finite_word.FiniteWord_class

INPUT:

- parent - a parent
- iter - an iterator
- length - (default: None) the length of the word

EXAMPLES:

```
sage: import itertools
sage: Word(itertools.cycle("abbabaab"))
word: abbabaababbabaababbabaababbabaababbabaab...
sage: w = Word(iter("abbabaab"), length="finite"); w
word: abbabaab
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length="unknown"); w
word: abbabaab
sage: w.length()
8
sage: list(w)
['a', 'b', 'b', 'a', 'b', 'a', 'a', 'b']
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8)
sage: w._len
8
```

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid_list**

Bases: sage.combinat.words.word_datatypes.WordDatatype_list, sage.combinat.words.paths.FiniteWordPath_triangle_grid, sage.combinat.words.finite_word.FiniteWord_class

TESTS:

```
sage: P = WordPaths('abcdef', steps='triangle')
sage: p = P(['a','b','b']); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_triangle_grid_list'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid_str**

Bases: sage.combinat.words.word_datatypes.WordDatatype_str, sage.combinat.words.paths.FiniteWordPath_triangle_grid, sage.combinat.words.finite_word.FiniteWord_class

TESTS:

```
sage: P = WordPaths('abcdef', steps='triangle')
sage: p = P('abb'); p
Path: abb
sage: type(p)
<class 'sage.combinat.words.paths.FiniteWordPath_triangle_grid_str'>
sage: p == loads(dumps(p))
True
```

class sage.combinat.words.paths.**FiniteWordPath_triangle_grid_tuple**

Bases: sage.combinat.words.word_datatypes.WordDatatype_tuple, sage.combinat.words.paths.FiniteWordPath_triangle_grid, sage.combinat.words.finite_word.FiniteWord_class

TESTS:

```
sage: P = WordPaths('abcdef', steps='triangle')
sage: p = P(('a','b','b')); p
Path: abb
sage: type(p)
```

```
<class 'sage.combinat.words.paths.FiniteWordPath_triangle_grid_tuple'>
sage: p == loads(dumps(p))
True
```

sage.combinat.words.paths.**WordPaths**(*alphabet*, *steps=None*)

Returns the combinatorial class of paths of the given type of steps.

INPUT:

- `alphabet` - ordered alphabet
- `steps` - (default is None). It can be one of the following:
  - an iterable ordered container of as many vectors as there are letters in the alphabet. The vectors are associated to the letters according to their order in steps. The vectors can be a tuple or anything that can be passed to vector function.
  - an iterable ordered container of k vectors where k is half the size of alphabet. The vectors and their opposites are associated to the letters according to their order in steps (given vectors first, opposite vectors after).
  - `None`: In this case, the type of steps are guessed from the length of alphabet.
  - 'square_grid' or 'square' : (default when size of alphabet is 4) The order is : East, North, West, South.
  - 'triangle_grid' or 'triangle':
  - 'hexagonal_grid' or 'hexagon' :(default when size of alphabet is 6)
  - 'cube_grid' or 'cube' :
  - 'north_east', 'ne' or 'NE' : (the default when size of alphabet is 2)
  - 'dyck':

OUTPUT:

- The combinatorial class of all paths of the given type.

EXAMPLES:

The steps can be given explicitely:

```
sage: WordPaths('abc', steps=[(1,2), (-1,4), (0,-3)])
Word Paths over 3 steps
```

Different type of input alphabet:

```
sage: WordPaths(range(3), steps=[(1,2), (-1,4), (0,-3)])
Word Paths over 3 steps
sage: WordPaths(['cric','crac','croc'], steps=[(1,2), (1,4), (0,3)])
Word Paths over 3 steps
```

Directions can be in three dimensions as well:

```
sage: WordPaths('ab', steps=[(1,2,2),(-1,4,2)])
Word Paths over 2 steps
```

When the number of given steps is half the size of alphabet, the opposite of vectors are used:

```
sage: P = WordPaths('abcd', [(1,0), (0,1)])
sage: P
Word Paths over 4 steps
sage: sorted(P.letters_to_steps().items())
[('a', (1, 0)), ('b', (0, 1)), ('c', (-1, 0)), ('d', (0, -1))]
```

When no steps are given, default classes are returned:

```
sage: WordPaths('ab')
Word Paths in North and East steps
sage: WordPaths(range(4))
Word Paths on the square grid
sage: WordPaths(range(6))
Word Paths on the hexagonal grid
```

There are many type of built-in steps...

On a two letters alphabet:

```
sage: WordPaths('ab', steps='north_east')
Word Paths in North and East steps
sage: WordPaths('()', steps='dyck')
Finite Dyck paths
```

On a four letters alphabet:

```
sage: WordPaths('ruld', steps='square_grid')
Word Paths on the square grid
```

On a six letters alphabet:

```
sage: WordPaths('abcdef', steps='hexagonal_grid')
Word Paths on the hexagonal grid
sage: WordPaths('abcdef', steps='triangle_grid')
Word Paths on the triangle grid
sage: WordPaths('abcdef', steps='cube_grid')
Word Paths on the cube grid
```

TESTS:

```
sage: WordPaths(range(5))
Traceback (most recent call last):
...
TypeError: Unable to make a class WordPaths from {0, 1, 2, 3, 4}
sage: WordPaths('abAB', steps='square_gridd')
Traceback (most recent call last):
...
TypeError: Unknown type of steps : square_gridd
```

class sage.combinat.words.paths.**WordPaths_all**(*alphabet*, *steps*)

Bases: sage.combinat.words.words.FiniteWords

The combinatorial class of all paths, i.e of all words over an alphabet where each letter is mapped to a step (a vector).

letters_to_steps()

Returns the dictionary mapping letters to vectors (steps).

EXAMPLES:

```
sage: d = WordPaths('ab').letters_to_steps()
sage: sorted(d.items())
[('a', (0, 1)), ('b', (1, 0))]
sage: d = WordPaths('abcd').letters_to_steps()
sage: sorted(d.items())
[('a', (1, 0)), ('b', (0, 1)), ('c', (-1, 0)), ('d', (0, -1))]
```

```
sage: d = WordPaths('abcdef').letters_to_steps()
sage: sorted(d.items())
[('a', (1, 0)),
 ('b', (1/2, 1/2*sqrt3)),
 ('c', (-1/2, 1/2*sqrt3)),
 ('d', (-1, 0)),
 ('e', (-1/2, -1/2*sqrt3)),
 ('f', (1/2, -1/2*sqrt3))]
```

vector_space()

Return the vector space over which the steps of the paths are defined.

EXAMPLES:

```
sage: WordPaths('ab',steps='dyck').vector_space()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
sage: WordPaths('ab',steps='north_east').vector_space()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
sage: WordPaths('abcd',steps='square_grid').vector_space()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
sage: WordPaths('abcdef',steps='hexagonal_grid').vector_space()
Vector space of dimension 2 over Number Field in sqrt3 with defining polynomial x^2 - 3
sage: WordPaths('abcdef',steps='cube_grid').vector_space()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: WordPaths('abcdef',steps='triangle_grid').vector_space()
Vector space of dimension 2 over Number Field in sqrt3 with defining polynomial x^2 - 3
```

class sage.combinat.words.paths.**WordPaths_cube_grid**(*alphabet*)

Bases: sage.combinat.words.paths.WordPaths_all

The combinatorial class of all paths on the cube grid.

class sage.combinat.words.paths.**WordPaths_dyck**(*alphabet*)

Bases: sage.combinat.words.paths.WordPaths_all

The combinatorial class of all Dyck paths.

class sage.combinat.words.paths.**WordPaths_hexagonal_grid**(*alphabet*)

Bases: sage.combinat.words.paths.WordPaths_triangle_grid

The combinatorial class of all paths on the hexagonal grid.

class sage.combinat.words.paths.**WordPaths_north_east**(*alphabet*)

Bases: sage.combinat.words.paths.WordPaths_all

The combinatorial class of all paths using North and East directions.

class sage.combinat.words.paths.**WordPaths_square_grid**(*alphabet*)

Bases: sage.combinat.words.paths.WordPaths_all

The combinatorial class of all paths on the square grid.

class sage.combinat.words.paths.**WordPaths_triangle_grid**(*alphabet*)

Bases: sage.combinat.words.paths.WordPaths_all

The combinatorial class of all paths on the triangle grid.

5.1.326 Shuffle product of words

See also:

The module sage.combinat.shuffle contains a more general implementation of shuffle product.

class sage.combinat.words.shuffle_product.**ShuffleProduct_overlapping**(*w1*, *w2*)

Bases: sage.combinat.combinat.CombinatorialClass

The overlapping shuffle product of the two words w1 and w2.

If u and v are two words whose letters belong to an additive monoid or to another kind of alphabet on which addition is well-defined, then the *overlapping shuffle product* of u and v is a certain multiset of words defined as follows: Let a and b be the lengths of u and v, respectively. Let A be the set $\{(0,1),(0,2),\cdots,(0,a)\}$, and let B be the set $\{(1,1),(1,2),\cdots,(1,b)\}$. Notice that the sets A and B are disjoint. We can make A and B into posets by setting $(k,i) \leq (k,j)$ for all $k \in \{0,1\}$ and $i \leq j$. Then, $A \cup B$ becomes a poset by disjoint union (we don't set $(0,i) \leq (1,i)$). Let p be the map from $A \cup B$ to the set of all letters which sends every $(0,i)$ to the i-th letter of u, and every $(1,j)$ to the j-th letter of v. For every nonnegative integer c and every surjective map $f : A \cup B \to \{1,2,\cdots,c\}$ for which both restrictions $f\mid_A$ and $f\mid_B$ are strictly increasing, let $w(f)$ be the length-c word such that for every $1 \leq k \leq c$, the k-th letter of $w(f)$ equals $\sum_{j \in f^{-1}(k)} p(j)$ (this sum always has either one or two addends). The overlapping shuffle product of u and v is then the multiset of all $w(f)$ with c ranging over all nonnegative integers and f ranging over the surjective maps $f : A \cup B \to \{1,2,\cdots,c\}$ for which both restrictions $f\mid_A$ and $f\mid_B$ are strictly increasing.

If one restricts c to a particular fixed nonnegative integer, then the multiset is instead called the *overlapping shuffle product with precisely 'a + b - c' overlaps*. This is nonempty only if $\max\{a,b\} \leq c \leq a+b$.

If $c = a+b$, then the overlapping shuffle product with precisely $a+b-c$ overlaps is plainly the shuffle product (ShuffleProduct_w1w2).

EXAMPLES:

```
sage: from sage.combinat.words.shuffle_product import ShuffleProduct_overlapping
sage: w, u = map(Words(range(20)), [[2, 9], [9, 1]])
sage: S = ShuffleProduct_overlapping(w,u)
sage: sorted([list(i) for i in list(S)])
[[2, 9, 1, 9],
 [2, 9, 9, 1],
 [2, 9, 9, 1],
 [2, 9, 10],
 [2, 18, 1],
 [9, 1, 2, 9],
 [9, 2, 1, 9],
 [9, 2, 9, 1],
 [9, 2, 10],
 [9, 3, 9],
 [11, 1, 9],
 [11, 9, 1],
 [11, 10]]
sage: S == loads(dumps(S))
True
```

class sage.combinat.words.shuffle_product.**ShuffleProduct_overlapping_r**(*w1*, *w2*, *r*)

Bases: sage.combinat.combinat.CombinatorialClass

The overlapping shuffle product of the two words w1 and w2 with precisely r overlaps.

See ShuffleProduct_overlapping for a definition.

EXAMPLES:

```
sage: from sage.combinat.words.shuffle_product import ShuffleProduct_overlapping_r
sage: w, u = map(Words(range(20)), [[2, 9], [9, 1]])
sage: S = ShuffleProduct_overlapping_r(w,u,1)
```

```
sage: S == loads(dumps(S))
True
```

class sage.combinat.words.shuffle_product.**ShuffleProduct_shifted**(*w1*, *w2*)
Bases: sage.combinat.words.shuffle_product.ShuffleProduct_w1w2

Shifted shuffle product of `w1` with `w2`.

This is the shuffle product of `w1` with the word obtained by adding the length of `w1` to every letter of `w2`.

Note that this class is meant to be used for words; it misbehaves when `w1` is a permutation or composition.

EXAMPLES:

```
sage: from sage.combinat.words.shuffle_product import ShuffleProduct_shifted
sage: w, u = Word([1,2]), Word([3,4])
sage: S = ShuffleProduct_shifted(w,u)
sage: S == loads(dumps(S))
True
```

class sage.combinat.words.shuffle_product.**ShuffleProduct_w1w2**(*w1*, *w2*)
Bases: sage.combinat.combinat.CombinatorialClass

The shuffle product of the two words `w1` and `w2`.

If u and v are two words, then the *shuffle product* of u and v is a certain multiset of words defined as follows: Let a and b be the lengths of u and v, respectively. For every a-element subset I of $\{1, 2, \cdots, a+b\}$, let $w(I)$ be the length-$a+b$ word such that:

- for every $1 \leq k \leq a$, the i_k-th letter of $w(I)$ is the k-th letter of u, where i_k is the k-th smallest element of I;
- for every $1 \leq l \leq b$, the j_l-th letter of $w(I)$ is the l-th letter of v, where j_l is the l-th smallest element of $\{1, 2, \cdots, a+b\} \setminus I$.

The shuffle product of u and v is then the multiset of all $w(I)$ with I ranging over the a-element subsets of $\{1, 2, \cdots, a+b\}$.

EXAMPLES:

```
sage: from sage.combinat.words.shuffle_product import ShuffleProduct_w1w2
sage: W = Words([1,2,3,4])
sage: s = ShuffleProduct_w1w2(W([1,2]),W([3,4]))
sage: sorted(list(s))
[word: 1234, word: 1324, word: 1342, word: 3124, word: 3142, word: 3412]
sage: s == loads(dumps(s))
True

sage: s = ShuffleProduct_w1w2(W([1,4,3]),W([2]))
sage: sorted(list(s))
[word: 1243, word: 1423, word: 1432, word: 2143]

sage: s = ShuffleProduct_w1w2(W([1,4,3]),W([]))
sage: sorted(list(s))
[word: 143]
```

cardinality()
Return the number of words in the shuffle product of `w1` and `w2`.

This is understood as a multiset cardinality, not as a set cardinality; it does not count the distinct words only.

It is given by $\binom{l_1+l_2}{l_1}$, where l_1 is the length of `w1` and where l_2 is the length of `w2`.

EXAMPLES:

```
sage: from sage.combinat.words.shuffle_product import ShuffleProduct_w1w2
sage: w, u = map(Words("abcd"), ["ab", "cd"])
sage: S = ShuffleProduct_w1w2(w,u)
sage: S.cardinality()
6

sage: w, u = map(Words("ab"), ["ab", "ab"])
sage: S = ShuffleProduct_w1w2(w,u)
sage: S.cardinality()
6
```

5.1.327 Suffix Tries and Suffix Trees

class `sage.combinat.words.suffix_trees.`**`ImplicitSuffixTree`**(*word*)

Bases: `sage.structure.sage_object.SageObject`

Construct the implicit suffix tree of a word w.

The suffix tree of a word w is a compactification of the suffix trie for w. The compactification removes all nodes that have exactly one incoming edge and exactly one outgoing edge. It consists of two components: a tree and a word. Thus, instead of labelling the edges by factors of w, we can labelled them by indices of the occurrence of the factors in w.

The following is a straightforward implementation of Ukkonen's on-line algorithm for constructing the implicit suffix tree [1]. It constructs the suffix tree for w[:i] from that of w[:i-1].

GENERAL IDEA. The suffix tree of w[:i+1] can be obtained from that of w[:i] by visiting each node corresponding to a suffix of w[:i] and modifying the tree by applying one of two rules (either append a new node to the tree, or split an edge into two). The "active state" is the node where the algorithm begins and the "suffix link" carries us to the next node that needs to be dealt with.

TREE. The tree is modelled as an automaton, which is stored as a dictionary of dictionaries: it is keyed by the nodes of the tree, and the corresponding dictionary is keyed by pairs (i, j) of integers representing the word w[i-1:j]. This makes it faster to look up a particular transition beginning at a specific node.

STATES/NODES. The states will always be -1, 0, 1, ..., n. The state -1 is special and is only used for the purposes of the algorithm. All transitions map -1 to 0, so this information is not explicitly stored in the transition function.

EXPLICIT/IMPLICIT NODES. By definition, some of the nodes will not be states, but merely locations along an edge; these are called implicit nodes. A node r (implicit or explicit) is referenced as a pair (s,(k,p)) where s is an ancestor of r and w[k-1:p] is the word read by transitioning from s to r in the suffix trie. A reference pair is canonical if s is the closest ancestor of r.

SUFFIX LINK. The algorithm makes use of a map from (some) nodes to other nodes, called the suffix link. This is stored as a dictionary.

ACTIVE STATE. We store as ._active_state the active state of the tree, the state where the algorithm will begin when processing the next letter.

RUNNING TIME. The running time and storage space of the algorithm is linear in the length of the word w (whereas for a suffix tree it is quadratic).

REFERENCES:

- [1] E. Ukkonen, "On-line construction of suffix trees", Algorithmica, 1995, volume 14, number 3, pages 249–260.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: w = Words("aco")("cacao")
sage: t = ImplicitSuffixTree(w); t
Implicit Suffix Tree of the word: cacao
sage: ababb = Words([0,1])([0,1,0,1,1])
sage: s = ImplicitSuffixTree(ababb); s
Implicit Suffix Tree of the word: 01011
```

TESTS:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: w = Words("cao")("cacao")
sage: s = ImplicitSuffixTree(w)
sage: loads(dumps(s))
Implicit Suffix Tree of the word: cacao
```

active_state()

Returns the active state of the suffix tree.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words([0,1,2])
sage: t = ImplicitSuffixTree(W([0,1,0,1,2]))
sage: t.active_state()
(0, (6, 6))
```

edge_iterator()

Returns an iterator over the edges of the suffix tree. The edge from u to v labelled by (i, j) is returned as the tuple $(u, v, (i, j))$.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: sorted( ImplicitSuffixTree(Word("aaaaa")).edge_iterator() )
[(0, 1, (0, None))]
sage: sorted( ImplicitSuffixTree(Word([0,1,0,1])).edge_iterator() )
[(0, 1, (0, None)), (0, 2, (1, None))]
sage: sorted( ImplicitSuffixTree(Word()).edge_iterator() )
[]
```

factor_iterator(*n=None*)

Generate distinct factors of `self`.

INPUT:

- `n` - an integer, or `None`.

OUTPUT:

- If `n` is an integer, returns an iterator over all distinct factors of length `n`. If `n` is `None`, returns an iterator generating all distinct factors.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: sorted( ImplicitSuffixTree(Word("cacao")).factor_iterator() )
[word: , word: a, word: ac, word: aca, word: acao, word: ao, word: c, word: ca, word: cac, w
sage: sorted( ImplicitSuffixTree(Word("cacao")).factor_iterator(1) )
[word: a, word: c, word: o]
sage: sorted( ImplicitSuffixTree(Word("cacao")).factor_iterator(2) )
```

```
[word: ac, word: ao, word: ca]
sage: sorted( ImplicitSuffixTree(Word([0,0,0])).factor_iterator() )
[word: , word: 0, word: 00, word: 000]
sage: sorted( ImplicitSuffixTree(Word([0,0,0])).factor_iterator(2) )
[word: 00]
sage: sorted( ImplicitSuffixTree(Word([0,0,0])).factor_iterator(0) )
[word: ]
sage: sorted( ImplicitSuffixTree(Word()).factor_iterator() )
[word: ]
sage: sorted( ImplicitSuffixTree(Word()).factor_iterator(2) )
[]
```

number_of_factors(*n=None*)

Count the number of distinct factors of `self.word()`.

INPUT:

- `n` - an integer, or `None`.

OUTPUT:

- If `n` is an integer, returns the number of distinct factors of length `n`. If `n` is `None`, returns the total number of distinct factors.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: t = ImplicitSuffixTree(Word([1,2,1,3,1,2,1]))
sage: t.number_of_factors()
22
sage: t.number_of_factors(1)
3
sage: t.number_of_factors(9)
0
sage: t.number_of_factors(0)
1

sage: t = ImplicitSuffixTree(Word("cacao"))
sage: t.number_of_factors()
13
sage: map(t.number_of_factors, range(10))
[1, 3, 3, 3, 2, 1, 0, 0, 0, 0]

sage: t = ImplicitSuffixTree(Word("c"*1000))
sage: t.number_of_factors()
1001
sage: t.number_of_factors(17)
1
sage: t.number_of_factors(0)
1

sage: ImplicitSuffixTree(Word()).number_of_factors()
1

sage: blueberry = ImplicitSuffixTree(Word("blueberry"))
sage: blueberry.number_of_factors()
43
sage: map(blueberry.number_of_factors, range(10))
[1, 6, 8, 7, 6, 5, 4, 3, 2, 1]
```

plot(*word_labels=False*, *layout='tree'*, *tree_root=0*, *tree_orientation='up'*, *vertex_colors=None*, *edge_labels=True*, **args*, ***kwds*)

Returns a Graphics object corresponding to the transition graph of the suffix tree.

INPUT:

- `word_labels` - boolean (defaut: `False`) if `False`, labels the edges by pairs (i, j); if `True`, labels the edges by `word[i:j]`.
- `layout` - (defaut: `'tree'`)
- `tree_root` - (defaut: 0)
- `tree_orientation` - (defaut: `'up'`)
- `vertex_colors` - (defaut: `None`)
- `edge_labels` - (defaut: `True`)

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: ImplicitSuffixTree(Word('cacao')).plot(word_labels=True)
Graphics object consisting of 23 graphics primitives
sage: ImplicitSuffixTree(Word('cacao')).plot(word_labels=False)
Graphics object consisting of 23 graphics primitives
```

TESTS:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: type(ImplicitSuffixTree(Word('cacao')).plot(word_labels=True))
<class 'sage.plot.graphics.Graphics'>
sage: type(ImplicitSuffixTree(Word('cacao')).plot(word_labels=False))
<class 'sage.plot.graphics.Graphics'>
```

process_letter(*letter*)

Modifies the current implicit suffix tree producing the implicit suffix tree for `self.word() + letter`.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: w = Words("aco")("cacao")
sage: t = ImplicitSuffixTree(w[:-1]); t
Implicit Suffix Tree of the word: caca
sage: t.process_letter(w[-1]); t
Implicit Suffix Tree of the word: cacao

sage: W = Words([0,1])
sage: s = ImplicitSuffixTree(W([0,1,0,1])); s
Implicit Suffix Tree of the word: 0101
sage: s.process_letter(W([1])[0]); s
Implicit Suffix Tree of the word: 01011
```

show(*word_labels=None*, **args*, ***kwds*)

Displays the output of `self.plot()`.

INPUT:

- `word_labels` - (default: `None`) if `False`, labels the edges by pairs (i, j); if `True`, labels the edges by `word[i:j]`.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: w = Words("cao")("cacao")
sage: t = ImplicitSuffixTree(w)
sage: t.show(word_labels=True)
sage: t.show(word_labels=False)
```

states()

Returns the states (explicit nodes) of the suffix tree.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words([0,1,2])
sage: t = ImplicitSuffixTree(W([0,1,0,1,2]))
sage: t.states()
[0, 1, 2, 3, 4, 5, 6, 7]
```

suffix_link(*state*)

Evaluates the suffix link map of the implicit suffix tree on `state`. Note that the suffix link is not defined for all states.

The suffix link of a state x' that corresponds to the suffix x is defined to be -1 is x' is the root (0) and y' otherwise, where y' is the state corresponding to the suffix `x[1:]`.

INPUT:

- `state` - a state

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words([0,1,2])
sage: t = ImplicitSuffixTree(W([0,1,0,1,2]))
sage: t.suffix_link(3)
5
sage: t.suffix_link(5)
0
sage: t.suffix_link(0)
-1
sage: t.suffix_link(-1)
Traceback (most recent call last):
    ...
TypeError: there is no suffix link from -1
```

to_digraph(*word_labels=False*)

Returns a `DiGraph` object of the transition graph of the suffix tree.

INPUT:

- `word_labels` - boolean (defaut: `False`) if `False`, labels the edges by pairs (i, j); if `True`, labels the edges by `word[i:j]`.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words([0,1,2])
sage: t = ImplicitSuffixTree(W([0,1,0,1,2]))
sage: t.to_digraph()
Digraph on 8 vertices
```

to_explicit_suffix_tree()

Converts self to an explicit suffix tree. It is obtained by processing an end of string letter as if it were a regular letter, except that no new leaf nodes are created (thus, the only thing that happens is that some implicit nodes become explicit).

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: w = Words("aco")("cacao")
sage: t = ImplicitSuffixTree(w)
sage: t.to_explicit_suffix_tree()

sage: W = Words([0,1])
sage: s = ImplicitSuffixTree(W([0,1,0,1,1]))
sage: s.to_explicit_suffix_tree()
```

transition_function(*word*, *node=0*)

Returns the node obtained by starting from `node` and following the edges labelled by the letters of `word`. Returns `("explicit", end_node)` if we end at `end_node`, or `("implicit", (edge, d))` if we end d spots along an edge.

INPUT:

- `word` - a word
- `node` - (default: 0) starting node

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words([0,1,2])
sage: t = ImplicitSuffixTree(W([0,1,0,1,2]))
sage: t.transition_function(W([0,1,0]))
('implicit', (3, 1), 1)
sage: t.transition_function(W([0,1,2]))
('explicit', 4)
sage: t.transition_function(W([0,1,2]), 5)
('explicit', 2)
sage: t.transition_function(W([0,1]), 5)
('implicit', (5, 2), 2)
```

transition_function_dictionary()

Returns the transition function as a dictionary of dictionaries. The format is consistent with the input format for `DiGraph`.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words("aco")
sage: t = ImplicitSuffixTree(W("cac"))
sage: t.transition_function_dictionary()
{0: {1: (0, None), 2: (1, None)}}

sage: W = Words([0,1])
sage: t = ImplicitSuffixTree(W([0,1,0]))
sage: t.transition_function_dictionary()
{0: {1: (0, None), 2: (1, None)}}
```

trie_type_dict()

Returns a dictionary in a format compatible with that of the suffix trie transition function.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree, SuffixTrie
sage: W = Words("ab")
sage: t = ImplicitSuffixTree(W("aba"))
sage: d = t.trie_type_dict()
sage: len(d)
5
sage: d                       # random
{(4, word: b): 5, (0, word: a): 4, (0, word: b): 3, (5, word: a): 1, (3, word: a): 2}
```

uncompactify()

Returns the tree obtained from self by splitting edges so that they are labelled by exactly one letter. The resulting tree is isomorphic to the suffix trie.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree, SuffixTrie
sage: abbab = Words("ab")("abbab")
sage: s = SuffixTrie(abbab)
sage: t = ImplicitSuffixTree(abbab)
sage: t.uncompactify().is_isomorphic(s.to_digraph())
True
```

word()

Returns the word whose implicit suffix tree this is.

TESTS:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: ImplicitSuffixTree(Word([0,1,0,1,0])).word() == Word([0,1,0,1,0])
True
```

class sage.combinat.words.suffix_trees.**SuffixTrie**(*word*)

Bases: sage.structure.sage_object.SageObject

Construct the suffix trie of the word w.

The suffix trie of a finite word w is a data structure representing the factors of w. It is a tree whose edges are labelled with letters of w, and whose leafs correspond to suffixes of w.

This is a straightforward implementation of Algorithm 1 from [1]. It constructs the suffix trie of w[:i] from that of w[:i-1].

A suffix trie is modelled as a deterministic finite-state automaton together with the suffix_link map. The set of states corresponds to factors of the word (below we write x' for the state corresponding to x); these are always 0, 1, ... The state 0 is the initial state, and it corresponds to the empty word. For the purposes of the algorithm, there is also an auxiliary state -1. The transition function t is defined as:

```
t(-1,a) = 0 for all letters a; and
t(x',a) = y' for all x',y' \in Q such that y = xa,
```

and the suffix link function is defined as:

```
suffix_link(0) = -1;
suffix_link(x') = y', if x = ay for some letter a.
```

REFERENCES:

- [1] E. Ukkonen, "On-line construction of suffix trees", Algorithmica, 1995, volume 14, number 3, pages 249–260.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w); t
Suffix Trie of the word: cacao

sage: e = Words("ab")()
sage: t = SuffixTrie(e); t
Suffix Trie of the word:
sage: t.process_letter("a"); t
Suffix Trie of the word: a
sage: t.process_letter("b"); t
Suffix Trie of the word: ab
sage: t.process_letter("a"); t
Suffix Trie of the word: aba
```

TESTS:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: s = SuffixTrie(w)
sage: loads(dumps(s))
Suffix Trie of the word: cacao
```

active_state()

Returns the active state of the suffix trie. This is the state corresponding to the word as a suffix of itself.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w)
sage: t.active_state()
8

sage: u = Words([0,1])([0,1,1,0,1,0,0,1])
sage: s = SuffixTrie(u)
sage: s.active_state()
22
```

final_states()

Returns the set of final states of the suffix trie. These are the states corresponding to the suffixes of `self.word()`. They are obtained be repeatedly following the suffix link from the active state until we reach 0.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w)
sage: t.final_states() == Set([8, 9, 10, 11, 12, 0])
True
```

has_suffix(*word*)

Return `True` if and only if `word` is a suffix of `self.word()`.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w)
sage: [t.has_suffix(w[i:]) for i in range(w.length()+1)]
```

```
[True, True, True, True, True, True]
sage: [t.has_suffix(w[:i]) for i in range(w.length()+1)]
[True, False, False, False, False, True]
```

node_to_word(*state=0*)

Returns the word obtained by reading the edge labels from 0 to `state`.

INPUT:

- `state` - (default: 0) a state

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("abc")("abcba")
sage: t = SuffixTrie(w)
sage: t.node_to_word(10)
word: abcba
sage: t.node_to_word(7)
word: abcb
```

plot(*layout='tree'*, *tree_root=0*, *tree_orientation='up'*, *vertex_colors=None*, *edge_labels=True*, **args*, ***kwds*)

Returns a Graphics object corresponding to the transition graph of the suffix trie.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: SuffixTrie(Word("cacao")).plot()
Graphics object consisting of 38 graphics primitives
```

TESTS:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: type(SuffixTrie(Word("cacao")).plot())
<class 'sage.plot.graphics.Graphics'>
```

process_letter(*letter*)

Modify `self` to produce the suffix trie for `self.word() + letter`.

Note: `letter` must occur within the alphabet of the word.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("ab")("ababba")
sage: t = SuffixTrie(w); t
Suffix Trie of the word: ababba
sage: t.process_letter("a"); t
Suffix Trie of the word: ababbaa
```

TESTS:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w); t
Suffix Trie of the word: cacao
sage: t.process_letter("d")
Traceback (most recent call last):
...
ValueError: d not in alphabet!
```

show(**args*, ***kwds*)

Displays the output of `self.plot()`.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cac")
sage: t = SuffixTrie(w)
sage: t.show()
```

states()

Returns the states of the automaton defined by the suffix trie.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words([0,1])([0,1,1])
sage: t = SuffixTrie(w)
sage: t.states()
[0, 1, 2, 3, 4]

sage: u = Words("aco")("cacao")
sage: s = SuffixTrie(u)
sage: s.states()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

suffix_link(*state*)

Evaluates the suffix link map of the suffix trie on `state`. Note that the suffix link map is not defined on -1.

INPUT:

- `state` - a state

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w)
sage: map(t.suffix_link, range(13))
[-1, 0, 3, 0, 5, 1, 7, 2, 9, 10, 11, 12, 0]
sage: t.suffix_link(0)
-1
```

TESTS:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w)
sage: t.suffix_link([1])
Traceback (most recent call last):
...
TypeError: [1] is not an integer
sage: t.suffix_link(-1)
Traceback (most recent call last):
...
TypeError: suffix link is not defined for -1
sage: t.suffix_link(17)
Traceback (most recent call last):
```

```
...
TypeError: 17 is not a state
```

to_digraph()

Returns a DiGraph object of the transition graph of the suffix trie.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cac")
sage: t = SuffixTrie(w)
sage: d = t.to_digraph(); d
Digraph on 6 vertices
sage: d.adjacency_matrix()
[0 1 0 1 0 0]
[0 0 1 0 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
[0 0 0 0 0 0]
[0 0 0 0 0 0]
```

transition_function(*node*, *word*)

Returns the state reached by beginning at node and following the arrows in the transition graph labelled by the letters of word.

INPUT:

- node - a node
- word - a word

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words([0,1])([0,1,0,1,1])
sage: t = SuffixTrie(w)
sage: [t.transition_function(u,letter) == v \
        for ((u,letter),v) in t._transition_function.iteritems()] \
        == [True] * len(t._transition_function)
True
```

word()

Returns the word whose suffix tree this is.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("abc")("abcba")
sage: t = SuffixTrie(w)
sage: t.word()
word: abcba
sage: t.word() == w
True
```

5.1.328 Word classes

AUTHORS:

- Arnaud Bergeron

- Amy Glen
- Sébastien Labbé
- Franco Saliola

class sage.combinat.words.word.**FiniteWord_callable**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable, sage.combinat.words.finite_word.FiniteWord_class

Finite word represented by a callable.

For such word w, type `w.` and hit TAB key to see the list of functions defined on w.

EXAMPLES:

```
sage: f = lambda n : 3 if n > 8 else 6
sage: w = Word(f, length=30, caching=False)
sage: w
word: 666666666333333333333333333333
sage: w.is_symmetric()
True
```

TESTS:

```
sage: w = Word(lambda n:n, length=10, caching=False)
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable'>
sage: z = loads(dumps(w))
sage: w == z
True
sage: type(z)
<class 'sage.combinat.words.word.FiniteWord_callable'>
```

class sage.combinat.words.word.**FiniteWord_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable_with_caching, sage.combinat.words.finite_word.FiniteWord_class

Finite word represented by a callable (with caching).

For such word w, type `w.` and hit TAB key to see the list of functions defined on w.

EXAMPLES:

```
sage: f = lambda n : n % 3
sage: w = Word(f, length=32)
sage: w
word: 01201201201201201201201201201201
sage: w.border()
word: 012012012012012012012012012012
```

TESTS:

```
sage: w = Word(lambda n:n, length=10)
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable_with_caching'>
sage: z = loads(dumps(w))
sage: w == z
True
sage: type(z)
<class 'sage.combinat.words.word.FiniteWord_callable_with_caching'>
```

Pickle also works for concatenation of words:

```
sage: w = Word(range(10)) * Word('abcdef')
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable_with_caching'>
sage: z = loads(dumps(w))
sage: w == z
True
sage: type(z)
<class 'sage.combinat.words.word.FiniteWord_list'>
```

Pickle also works for power of words:

```
sage: w = Word(range(10)) ^ 2
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_callable_with_caching'>
sage: z = loads(dumps(w))
sage: w == z
True
sage: type(z)
<class 'sage.combinat.words.word.FiniteWord_list'>
```

class sage.combinat.words.word.**FiniteWord_char**

Bases: sage.combinat.words.word_char.WordDatatype_char, sage.combinat.words.finite_word.FiniteWord_class

Finite word represented by an array `unsigned char *` (i.e. integers between 0 and 255).

For any word `w`, type `w.<TAB>` to see the functions that can be applied to `w`.

EXAMPLES:

```
sage: W = Words(range(20))

sage: w = W(range(1,10)*2)
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_char'>
sage: w
word: 123456789123456789

sage: w.is_palindrome()
False
sage: (w*w[::-1]).is_palindrome()
True
sage: (w[:-1:]*w[::-1]).is_palindrome()
True

sage: w.is_lyndon()
False
sage: W(range(10)+[10,10]).is_lyndon()
True

sage: w.is_square_free()
False
sage: w[:-1].is_square_free()
True

sage: u = W([randint(0,10) for i in range(10)])
sage: (u*u).is_square()
True
sage: (u*u*u).is_cube()
```

```
True

sage: len(w.factor_set())
127
sage: w.rauzy_graph(5)
Looped digraph on 9 vertices

sage: u = W([1,2,3])
sage: u.first_pos_in(w)
0
sage: u.first_pos_in(w[1:])
8
```

TESTS:

```
sage: W = Words([0,1,2])
sage: w = W([0,1,1,0])
sage: w == loads(dumps(w))
True
```

class sage.combinat.words.word.**FiniteWord_iter**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter, sage.combinat.words.finite_word.FiniteWord_class

Finite word represented by an iterator.

For such word w, type `w.` and hit TAB key to see the list of functions defined on w.

EXAMPLES:

```
sage: w = Word(iter(range(10)), caching=False)
sage: w
word: 0123456789
sage: w.finite_differences()
word: 111111111
```

TESTS:

```
sage: w = Word(iter(range(10)), caching=False)
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_iter'>
sage: z = loads(dumps(w))
sage: w == z
True
sage: type(z)
<class 'sage.combinat.words.word.FiniteWord_list'>
```

class sage.combinat.words.word.**FiniteWord_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter_with_caching, sage.combinat.words.finite_word.FiniteWord_class

Finite word represented by an iterator (with caching).

For such word w, type `w.` and hit TAB key to see the list of functions defined on w.

EXAMPLES:

```
sage: w = Word(iter('abcdef'))
sage: w.conjugate(2)
word: cdefab
```

TESTS:

```
sage: w = Word(iter(range(10)))
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_iter_with_caching'>
sage: z = loads(dumps(w))
sage: w == z
True
sage: type(z)
<class 'sage.combinat.words.word.FiniteWord_list'>
```

class sage.combinat.words.word.**FiniteWord_list**

Bases: sage.combinat.words.word_datatypes.WordDatatype_list, sage.combinat.words.finite_word.FiniteWord_class

Finite word represented by a Python list.

For any word w, type `w.` and hit TAB key to see the list of functions defined on w.

EXAMPLES:

```
sage: w = Word(range(10))
sage: w.iterated_right_palindromic_closure()
word: 0102010301020104010201030102010501020103...
```

TESTS:

```
sage: w = Word([0,1,1,0])
sage: w == loads(dumps(w))
True
```

class sage.combinat.words.word.**FiniteWord_str**

Bases: sage.combinat.words.word_datatypes.WordDatatype_str, sage.combinat.words.finite_word.FiniteWord_class

Finite word represented by a Python str.

For such word w, type `w.` and hit TAB key to see the list of functions defined on w.

EXAMPLES:

```
sage: w = Word('abcdef')
sage: w.is_square()
False
```

TESTS:

```
sage: w = Word('abba')
sage: w == loads(dumps(w))
True
```

class sage.combinat.words.word.**FiniteWord_tuple**

Bases: sage.combinat.words.word_datatypes.WordDatatype_tuple, sage.combinat.words.finite_word.FiniteWord_class

Finite word represented by a Python tuple.

For such word w, type `w.` and hit TAB key to see the list of functions defined on w.

EXAMPLES:

```
sage: w = Word(())
sage: w.is_empty()
True
```

TESTS:

```
sage: w = Word((0,1,1,0))
sage: w == loads(dumps(w))
True
```

class sage.combinat.words.word.**InfiniteWord_callable**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable, sage.combinat.words.infinite_word.InfiniteWord_class

Infinite word represented by a callable.

For such word w, type `w.` and hit TAB key to see the list of functions defined on w.

Infinite words behave like a Python list : they can be sliced using square braquets to define for example a prefix or a factor.

EXAMPLES:

```
sage: w = Word(lambda n:n, caching=False)
sage: w
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,3
sage: w.iterated_right_palindromic_closure()
word: 0102010301020104010201030102010501020103...
```

TESTS:

```
sage: w = Word(lambda n:n, caching=False)
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable'>
sage: z = loads(dumps(w))
sage: z
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,3
sage: type(z)
<class 'sage.combinat.words.word.InfiniteWord_callable'>
```

class sage.combinat.words.word.**InfiniteWord_callable_with_caching**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable_with_caching, sage.combinat.words.infinite_word.InfiniteWord_class

Infinite word represented by a callable (with caching).

For such word w, type `w.` and hit TAB key to see the list of functions defined on w.

Infinite words behave like a Python list : they can be sliced using square braquets to define for example a prefix or a factor.

EXAMPLES:

```
sage: w = Word(lambda n:n)
sage: factor = w[4:13]
sage: factor
word: 4,5,6,7,8,9,10,11,12
```

TESTS:

```
sage: w = Word(lambda n:n)
sage: type(w)
<class 'sage.combinat.words.word.InfiniteWord_callable_with_caching'>
sage: z = loads(dumps(w))
sage: z
```

```
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,3
sage: type(z)
<class 'sage.combinat.words.word.InfiniteWord_callable_with_caching'>
```

class sage.combinat.words.word.**InfiniteWord_iter**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter, sage.combinat.words.infinite_word.InfiniteWord_class

Infinite word represented by an iterable.

For such word w, type `w.` and hit TAB key to see the list of functions defined on w.

Infinite words behave like a Python list : they can be sliced using square braquets to define for example a prefix or a factor.

EXAMPLES:

```
sage: from itertools import chain, cycle
sage: w = Word(chain('letsgo', cycle('forever')), caching=False)
sage: w
word: letsgoforeverforeverforeverforeverforeve...
sage: prefix = w[:100]
sage: prefix
word: letsgoforeverforeverforeverforeverforeve...
sage: prefix.is_lyndon()
False
```

TESTS:

```
sage: from itertools import count
sage: w = Word(count(), caching=False)
sage: type(w)
<class 'sage.combinat.words.word.Word_iter'>
```

Pickle is not supported for infinite word defined by an iterator:

```
sage: dumps(w)
Traceback (most recent call last):
...
PicklingError: Can't pickle <type 'generator'>: attribute lookup __builtin__.generator failed
```

class sage.combinat.words.word.**InfiniteWord_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter_with_caching, sage.combinat.words.infinite_word.InfiniteWord_class

Infinite word represented by an iterable (with caching).

For such word w, type `w.` and hit TAB key to see the list of functions defined on w.

Infinite words behave like a Python list : they can be sliced using square braquets to define for example a prefix or a factor.

EXAMPLES:

```
sage: from itertools import cycle
sage: w = Word(cycle([9,8,4]))
sage: w
word: 98498498498498498498498498498498498498498...
sage: prefix = w[:23]
sage: prefix
word: 98498498498498498498498
```

```
sage: prefix.minimal_period()
3
```

TESTS:

```
sage: from itertools import count
sage: w = Word(count())
sage: type(w)
<class 'sage.combinat.words.word.Word_iter_with_caching'>
```

Pickle is not supported for infinite word defined by an iterator:

```
sage: dumps(w)
Traceback (most recent call last):
...
PicklingError: Can't pickle <type 'generator'>: attribute lookup __builtin__.generator failed
```

sage.combinat.words.word.**Word**(*data=None*, *alphabet=None*, *length=None*, *datatype=None*, *caching=True*, *RSK_data=None*)

Construct a word.

INPUT:

- `data` – (default: `None`) list, string, tuple, iterator, free monoid element, `None` (shorthand for `[]`), or a callable defined on `[0,1,...,length]`.
- `alphabet` – any argument accepted by Words
- `length` – (default: `None`) This is dependent on the type of data. It is ignored for words defined by lists, strings, tuples, etc., because they have a naturally defined length. For callables, this defines the domain of definition, which is assumed to be `[0, 1, 2, ..., length-1]`. For iterators: Infinity if you know the iterator will not terminate (default); `"unknown"` if you do not know whether the iterator terminates; `"finite"` if you know that the iterator terminates, but do know know the length.
- `datatype` – (default: `None`) `None`, `"list"`, `"str"`, `"tuple"`, `"iter"`, `"callable"`. If `None`, then the function tries to guess this from the data.
- `caching` – (default: `True`) `True` or `False`. Whether to keep a cache of the letters computed by an iterator or callable.
- `RSK_data` – (Optional. Default: `None`) A semistandard and a standard Young tableau to run the inverse RSK bijection on.

Note: Be careful when defining words using callables and iterators. It appears that islice does not pickle correctly causing various errors when reloading. Also, most iterators do not support copying and should not support pickling by extension.

EXAMPLES:

Empty word:

```
sage: Word()
word:
```

Word with string:

```
sage: Word("abbabaab")
word: abbabaab
```

Word with string constructed from other types:

```
sage: Word([0,1,1,0,1,0,0,1], datatype="str")
word: 01101001
sage: Word((0,1,1,0,1,0,0,1), datatype="str")
word: 01101001
```

Word with list:

```
sage: Word([0,1,1,0,1,0,0,1])
word: 01101001
```

Word with list constructed from other types:

```
sage: Word("01101001", datatype="list")
word: 01101001
sage: Word((0,1,1,0,1,0,0,1), datatype="list")
word: 01101001
```

Word with tuple:

```
sage: Word((0,1,1,0,1,0,0,1))
word: 01101001
```

Word with tuple constructed from other types:

```
sage: Word([0,1,1,0,1,0,0,1], datatype="tuple")
word: 01101001
sage: Word("01101001", datatype="str")
word: 01101001
```

Word with iterator:

```
sage: from itertools import count
sage: Word(count())
word: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,3
sage: Word(iter("abbabaab")) # iterators default to infinite words
word: abbabaab
sage: Word(iter("abbabaab"), length="unknown")
word: abbabaab
sage: Word(iter("abbabaab"), length="finite")
word: abbabaab
```

Word with function (a 'callable'):

```
sage: f = lambda n : add(Integer(n).digits(2)) % 2
sage: Word(f)
word: 0110100110010110100101100110100110010110...
sage: Word(f, length=8)
word: 01101001
```

Word over a string with a parent:

```
sage: w = Word("abbabaab", alphabet="abc"); w
word: abbabaab
sage: w.parent()
Finite words over {'a', 'b', 'c'}
```

Word from a free monoid element:

```
sage: M.<x,y,z> = FreeMonoid(3)
sage: Word(x^3*y*x*z^2*x)
word: xxxyxzzx
```

The default parent is the combinatorial class of all words:

```
sage: w = Word("abbabaab"); w
word: abbabaab
sage: w.parent()
Finite words over Set of Python objects of type 'object'
```

We can also input a semistandard tableau and a standard tableau to obtain a word from the inverse RSK algorithm using the `RSK_data` option:

```
sage: p = Tableau([[1,2,2],[3]]); q = Tableau([[1,2,4],[3]])
sage: Word(RSK_data=[p, q])
word: 1322
```

TESTS:

```
sage: Word(5)
Traceback (most recent call last):
...
ValueError: Cannot guess a datatype from data (=5); please specify one

sage: W = Words()
sage: w  = W('abc')
sage: w is W(w)
True
sage: w is Word(w, alphabet='abc')
False
```

class `sage.combinat.words.word.`**`Word_iter`**(*parent*, *iter*, *length=None*)

Bases: `sage.combinat.words.word_infinite_datatypes.WordDatatype_iter`, `sage.combinat.words.abstract_word.Word_class`

Word of unknown length (finite or infinite) represented by an iterable.

For such word w, type `w.` and hit TAB key to see the list of functions defined on w.

Words behave like a Python list : they can be sliced using square braquets to define for example a prefix or a factor.

EXAMPLES:

```
sage: w = Word(iter([1,1,4,9]*1000), length='unknown', caching=False)
sage: w
word: 11491149114911491149114911491149114911491149...
sage: w.delta()
word: 21121121121121121121121121121121121121121121...
```

TESTS:

```
sage: w = Word(iter('abcd'*100), length='unknown', caching=False)
sage: type(w)
<class 'sage.combinat.words.word.Word_iter'>
sage: w
word: abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcd...
```

Pickle is not supported for word of unknown length defined by an iterator:

```
sage: dumps(w)
Traceback (most recent call last):
...
PicklingError: Can't pickle <type 'generator'>: attribute lookup __builtin__.generator failed
```

class sage.combinat.words.word.**Word_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter_with_caching, sage.combinat.words.abstract_word.Word_class

Word of unknown length (finite or infinite) represented by an iterable (with caching).

For such word w, type `w.` and hit TAB key to see the list of functions defined on w.

Words behave like a Python list : they can be sliced using square braquets to define for example a prefix or a factor.

EXAMPLES:

```
sage: w = Word(iter([1,2,3]*1000), length='unknown')
sage: w
word: 12312312312312312312312312312312312312312...
sage: w.finite_differences(mod=2)
word: 11011011011011011011011011011011011011011101...
```

TESTS:

```
sage: w = Word(iter('abcd'*100), length='unknown')
sage: type(w)
<class 'sage.combinat.words.word.Word_iter_with_caching'>
sage: w
word: abcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcd...
```

Pickle is not supported for word of unknown length defined by an iterator:

```
sage: dumps(w)
Traceback (most recent call last):
...
PicklingError: Can't pickle <type 'generator'>: attribute lookup __builtin__.generator failed
```

5.1.329 Fast word datatype using an array of unsigned char.

class sage.combinat.words.word_char.**WordDatatype_char**

Bases: sage.combinat.words.word_datatypes.WordDatatype

A Fast class for words represented by an array `unsigned char *`.

Currently, only handles letters in [0,255].

has_prefix(*other*)

Test whether `other` is a prefix of `self`.

INPUT:

- `other` – a word or a sequence (e.g. tuple, list)

EXAMPLES:

```
sage: W = Words([0,1,2])
sage: w = W([0,1,1,0,1,2,0])
sage: w.has_prefix([0,1,1])
True
sage: w.has_prefix([0,1,2])
False
sage: w.has_prefix(w)
True
sage: w.has_prefix(w[:-1])
True
```

```
sage: w.has_prefix(w[1:])
False
```

TESTS:

trac ticket #19322:

```
sage: W = Words([0,1,2])
sage: w = W([0,1,0,2])
sage: w.has_prefix(words.FibonacciWord())
False

sage: w.has_prefix([0,1,0,2,0])
False
sage: w.has_prefix([0,1,0,2])
True
sage: w.has_prefix([0,1,0])
True
```

is_empty()

Return whether the word is empty.

EXAMPLES:

```
sage: W = Words([0,1,2])
sage: W([0,1,2,2]).is_empty()
False
sage: W([]).is_empty()
True
```

is_square()

Returns True if self is a square, and False otherwise.

EXAMPLES:

```
sage: w = Word([n % 4 for n in range(48)], alphabet=[0,1,2,3])
sage: w.is_square()
True

sage: w = Word([n % 4 for n in range(49)], alphabet=[0,1,2,3])
sage: w.is_square()
False
sage: (w*w).is_square()
True
```

TESTS:

The above tests correspond to the present class (char):

```
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_char'>

sage: Word([], alphabet=[0,1]).is_square()
True
sage: Word([0], alphabet=[0,1]).is_square()
False
sage: Word([0,0], alphabet=[0,1]).is_square()
True
```

length()

Return the length of the word as a Sage integer.

EXAMPLES:

```
sage: W = Words([0,1,2,3,4])
sage: w = W([0,1,2,0,3,2,1])
sage: w.length()
7
sage: type(w.length())
<type 'sage.rings.integer.Integer'>
sage: type(len(w))
<type 'int'>
```

letters()

Return the list of letters that appear in this word, listed in the order of first appearance.

EXAMPLES:

```
sage: W = Words(5)
sage: W([1,3,1,2,2,3,1]).letters()
[1, 3, 2]
```

longest_common_prefix(*other*)

Return the longest common prefix of this word and `other`.

EXAMPLES:

```
sage: W = Words([0,1,2])
sage: W([0,1,0,2]).longest_common_prefix([0,1])
word: 01
sage: u = W([0,1,0,0,1])
sage: v = W([0,1,0,2])
sage: u.longest_common_prefix(v)
word: 010
sage: v.longest_common_prefix(u)
word: 010
```

Using infinite words is also possible (and the return type is also a of the same type as `self`):

```
sage: W([0,1,0,0]).longest_common_prefix(words.FibonacciWord())
word: 0100
sage: type(_)
<class 'sage.combinat.words.word.FiniteWord_char'>
```

An example of an intensive usage:

```
sage: W = Words([0,1])
sage: w = words.FibonacciWord()
sage: w = W(list(w[:5000]))
sage: L = [[len(w[n:].longest_common_prefix(w[n+fibonacci(i):]))
....:       for i in range(5,15)] for n in range(1,1000)]
sage: for n,l in enumerate(L):
....:     if l.count(0) > 4: print n+1,l
375 [0, 13, 0, 34, 0, 89, 0, 233, 0, 233]
376 [0, 12, 0, 33, 0, 88, 0, 232, 0, 232]
608 [8, 0, 21, 0, 55, 0, 144, 0, 377, 0]
609 [7, 0, 20, 0, 54, 0, 143, 0, 376, 0]
985 [0, 13, 0, 34, 0, 89, 0, 233, 0, 610]
986 [0, 12, 0, 33, 0, 88, 0, 232, 0, 609]
```

TESTS:

```
sage: W = Words([0,1,2])
sage: w = W([0,2,1,0,0,1])
sage: w.longest_common_prefix(0)
Traceback (most recent call last):
...
TypeError: unsupported input 0
```

longest_common_suffix(*other*)

Return the longest common suffix between this word and `other`.

EXAMPLES:

```
sage: W = Words([0,1,2])
sage: W([0,1,0,2]).longest_common_suffix([2,0,2])
word: 02
sage: u = W([0,1,0,0,1])
sage: v = W([1,2,0,0,1])
sage: u.longest_common_suffix(v)
word: 001
sage: v.longest_common_suffix(u)
word: 001
```

TESTS:

```
sage: W = Words([0,1,2])
sage: w = W([0,2,1,0,0,1])
sage: w.longest_common_suffix(0)
Traceback (most recent call last):
...
TypeError: unsupported input 0
```

sage.combinat.words.word_char.**reversed_word_iterator**(*w*)

This function exists only because it is not possible to use yield in the special method `__reversed__`.

EXAMPLES:

```
sage: W = Words([0,1,2])
sage: w = W([0,1,0,0,1,2])
sage: list(reversed(w)) # indirect doctest
[2, 1, 0, 0, 1, 0]
```

5.1.330 Datatypes for finite words

class sage.combinat.words.word_datatypes.**WordDatatype**

Bases: `object`

The generic WordDatatype class.

Any word datatype must contain two attributes (at least):

- `_parent`
- `_hash`

They are automatically defined here and it's not necessary (and forbidden) to define them anywhere else.

TESTS:

```
sage: w = Word([0,1,1,0,0,1])
sage: isinstance(w, sage.combinat.words.word_datatypes.WordDatatype)
True
```

class sage.combinat.words.word_datatypes.**WordDatatype_list**
Bases: sage.combinat.words.word_datatypes.WordDatatype

Datatype class for words defined by lists.

count(*a*)
Returns the number of occurrences of the letter `a` in the word `self`.

INPUT:

- `a` - a letter

OUTPUT:

- integer

EXAMPLES:

```
sage: w = Word([0,1,1,0,1])
sage: w.count(0)
2
sage: w.count(1)
3
sage: w.count(2)
0
```

length()
Return the length of the word.

EXAMPLES:

```
sage: w = Word([0,1,1,0])
sage: w.length()
4
```

class sage.combinat.words.word_datatypes.**WordDatatype_str**
Bases: sage.combinat.words.word_datatypes.WordDatatype

Datatype for words defined by strings.

count(*letter*)
Count the number of occurrences of `letter`.

INPUT:

- `letter` - a letter

OUTPUT:

- integer

EXAMPLES:

```
sage: w = Word("abbabaababababa")
sage: w.count('a')
7
sage: w.count('b')
6
sage: w.count('c')
0
```

find(*sub*, *start=0*, *end=None*)

Returns the index of the first occurrence of sub in self, such that sub is contained within self[start:end]. Returns -1 on failure.

INPUT:

- `sub` - string or word to search for.
- `start` - non negative integer (default: 0) specifying the position from which to start the search.
- `end` - non negative integer (default: None) specifying the position at which the search must stop. If None, then the search is performed up to the end of the string.

OUTPUT:

non negative integer or -1

EXAMPLES:

```
sage: w = Word("abbabaababababa")
sage: w.find("a")
0
sage: w.find("a", 4)
5
sage: w.find("a", 4, 5)
-1
```

has_prefix(*other*)

Test whether `self` has `other` as a prefix.

INPUT:

- `other` - a word (an instance of `Word_class`) or a `str`.

OUTPUT:

- boolean

EXAMPLES:

```
sage: w = Word("abbabaababababa")
sage: u = Word("abbab")
sage: w.has_prefix(u)
True
sage: u.has_prefix(w)
False
sage: u.has_prefix("abbab")
True
```

TESTS:

```
sage: ab = Word('ab')
sage: abba = Word(['a','b','b','a'])
sage: ab.has_prefix(abba)
False
sage: abba.has_prefix(ab)
True
```

has_suffix(*other*)

Test whether `self` has `other` as a suffix.

INPUT:

- `other` - a word (an instance of `Word_class`) or a `str`.

OUTPUT:

- boolean

EXAMPLES:

```
sage: w = Word("abbabaababababa")
sage: u = Word("ababa")
sage: w.has_suffix(u)
True
sage: u.has_suffix(w)
False
sage: u.has_suffix("ababa")
True
```

is_prefix(*other*)

Test whether `self` is a prefix of `other`.

INPUT:

- `other` - a word (an instance of `Word_class`) or a `str`.

OUTPUT:

- boolean

EXAMPLES:

```
sage: w = Word("abbabaababababa")
sage: u = Word("abbab")
sage: w.is_prefix(u)
False
sage: u.is_prefix(w)
True
sage: u.is_prefix("abbabaababababa")
True
```

TESTS:

```
sage: ab = Word('ab')
sage: abba = Word(['a','b','b','a'])
sage: ab.is_prefix(abba)
True
sage: abba.is_prefix(ab)
False
```

is_suffix(*other*)

Test whether `self` is a suffix of `other`.

INPUT:

- `other` - a word (an instance of `Word_class`) or a `str`.

OUTPUT:

- boolean

EXAMPLES:

```
sage: w = Word("abbabaababababa")
sage: u = Word("ababa")
sage: w.is_suffix(u)
False
sage: u.is_suffix(w)
True
```

```
sage: u.is_suffix("abbabaababababa")
True
```

TESTS:

```
sage: w = Word("abbabaababababa")
sage: u = Word(['a','b','a','b','a'])
sage: w.is_suffix(u)
False
sage: u.is_suffix(w)
True
```

length()

Return the length of the word.

EXAMPLES:

```
sage: w = Word("abbabaababababa")
sage: w.length()
13
```

partition(*sep*)

Search for the separator sep in S, and return the part before it, the separator itself, and the part after it. The concatenation of the terms in the list gives back the initial word.

See also the split method.

Note: This just wraps Python's builtin `str::partition()` for `str`.

INPUT:

- `sep` - string or word

EXAMPLES:

```
sage: w = Word("MyTailorIsPoor")
sage: w.partition("Tailor")
[word: My, word: Tailor, word: IsPoor]

sage: w = Word("32303010303232123230323212101212321210 10")
sage: l = w.partition("323")
sage: print l
[word: , word: 323, word: 03010303232123230323212101212321210 10]
sage: sum(l, Word('')) == w
True
```

If the separator is not a string an error is raised:

```
sage: w = Word("le papa du papa du papa etait un petit pioupiou")
sage: w.partition(Word(['p','a','p','a']))
Traceback (most recent call last):
...
ValueError: the separator must be a string.
```

rfind(*sub*, *start=0*, *end=None*)

Returns the index of the last occurrence of sub in self, such that sub is contained within self[start:end]. Returns -1 on failure.

INPUT:

- `sub` - string or word to search for.

- `start` - non negative integer (default: 0) specifying the position at which the search must stop.
- `end` - non negative integer (default: None) specifying the position from which to start the search. If None, then the search is performed up to the end of the string.

OUTPUT:

non negative integer or -1

EXAMPLES:

```
sage: w = Word("abbabaababababa")
sage: w.rfind("a")
12
sage: w.rfind("a", 4, 8)
6
sage: w.rfind("a", 4, 5)
-1
```

split(*sep=None*, *maxsplit=None*)

Returns a list of words, using sep as a delimiter string. If maxsplit is given, at most maxsplit splits are done.

See also the partition method.

Note: This just wraps Python's builtin `str::split()` for `str`.

INPUT:

- `sep` - string or word (optional, default: None)
- `maxsplit` - positive integer (optional, default: None)

OUTPUT:

- a list of words

EXAMPLES:

You can split along white space to find words in a sentence:

```
sage: w = Word("My tailor is poor")
sage: w.split(" ")
[word: My, word: tailor, word: is, word: poor]
```

The python behavior is kept when no argument is given:

```
sage: w.split()
[word: My, word: tailor, word: is, word: poor]
```

You can split in two words letters to get the length of blocks in the other letter:

```
sage: w = Word("ababbabaaba")
sage: w.split('a')
[word: , word: b, word: bb, word: b, word: , word: b, word: ]
sage: w.split('b')
[word: a, word: a, word: , word: a, word: aa, word: a]
```

You can split along words:

```
sage: w = Word("32303010303232123230323 21")
sage: w.split("32")
[word: , word: 30301030, word: , word: 12, word: 30, word: , word: 1]
```

If the separator is not a string a ValueError is raised:

```
sage: w = Word("le papa du papa du papa etait un petit pioupiou")
sage: w.split(Word(['p','a','p','a']))
Traceback (most recent call last):
...
ValueError: the separator must be a string.
```

class sage.combinat.words.word_datatypes.**WordDatatype_tuple**

Bases: sage.combinat.words.word_datatypes.WordDatatype

Datatype class for words defined by tuples.

length()

Return the length of the word.

EXAMPLES:

```
sage: w = Word((0,1,1,0))
sage: w.length()
4
```

5.1.331 A collection of constructors of common words

AUTHORS:

- Franco Saliola (2008-12-17): merged into sage
- Sebastien Labbe (2008-12-17): merged into sage
- Arnaud Bergeron (2008-12-17): merged into sage
- Amy Glen (2008-12-17): merged into sage
- Sebastien Labbe (2009-12-19): Added S-adic words (trac ticket #7543)

USE:

To see a list of all word constructors, type `words.` and then press the tab key. The documentation for each constructor includes information about each word, which provides a useful reference.

REFERENCES:

EXAMPLES:

```
sage: t = words.ThueMorseWord(); t
word: 01101001100101101001011001101001100101 10...
```

class sage.combinat.words.word_generators.**LowerChristoffelWord**(*p*, *q*, *alphabet=(0, 1)*, *algorithm='cf'*)

Bases: sage.combinat.words.word.FiniteWord_list

Returns the lower Christoffel word of slope p/q, where p and q are relatively prime non-negative integers, over the given two-letter alphabet.

The *Christoffel word of slope 'p/q'* is obtained from the Cayley graph of $\mathbf{Z}/(p+q)\mathbf{Z}$ with generator q as follows. If $u \to v$ is an edge in the Cayley graph, then $v = u + p \mod p+q$. Label the edge $u \to v$ by `alphabet[1]` if $u < v$ and `alphabet[0]` otherwise. The Christoffel word is the word obtained by reading the edge labels along the cycle beginning from 0.

EXAMPLES:

```
sage: words.LowerChristoffelWord(4,7)
word: 00100100101

sage: words.LowerChristoffelWord(4,7,alphabet='ab')
word: aabaabaabab
```

TESTS:

```
sage: words.LowerChristoffelWord(1,0)
word: 1
sage: words.LowerChristoffelWord(0,1,'xy')
word: x
sage: words.LowerChristoffelWord(1,1)
word: 01
```

markoff_number()

Returns the Markoff number associated to the Christoffel word self.

The *Markoff number* of a Christoffel word w is $trace(M(w))/3$, where $M(w)$ is the 2×2 matrix obtained by applying the morphism: 0 -> matrix(2,[2,1,1,1]) 1 -> matrix(2,[5,2,2,1])

EXAMPLES:

```
sage: w0 = words.LowerChristoffelWord(4,7)
sage: w1, w2 = w0.standard_factorization()
sage: (m0,m1,m2) = (w.markoff_number() for w in (w0,w1,w2))
sage: (m0,m1,m2)
(294685, 13, 7561)
sage: m0**2 + m1**2 + m2**2 == 3*m0*m1*m2
True
```

standard_factorization()

Returns the standard factorization of the Christoffel word `self`.

The *standard factorization* of a Christoffel word w is the unique factorization of w into two Christoffel words.

EXAMPLES:

```
sage: w = words.LowerChristoffelWord(5,9)
sage: w
word: 00100100100101
sage: w1, w2 = w.standard_factorization()
sage: w1
word: 001
sage: w2
word: 00100100101

sage: w = words.LowerChristoffelWord(51,37)
sage: w1, w2 = w.standard_factorization()
sage: w1
word: 0101011010101101011
sage: w2
word: 01010110101011010110101011010110101101101...
sage: w1 * w2 == w
True
```

class sage.combinat.words.word_generators.**WordGenerator**

Bases: object

Constructor of several famous words.

EXAMPLES:

```
sage: words.ThueMorseWord()
word: 0110100110010110100101100110100110010110...

sage: words.FibonacciWord()
word: 0100101001001010010100100101001001010010...

sage: words.ChristoffelWord(5, 8)
word: 0010010100101

sage: words.RandomWord(10, 4)     # not tested random
word: 1311131221

sage: words.CodingOfRotationWord(alpha=0.618, beta=0.618)
word: 1010110101101101011010110110101101101011...

sage: tm = WordMorphism('a->ab,b->ba')
sage: fib = WordMorphism('a->ab,b->a')
sage: tmword = words.ThueMorseWord([0, 1])
sage: from itertools import repeat
sage: words.s_adic(tmword, repeat('a'), {0:tm, 1:fib})
word: abbaababbaabbaabbaababbaabababbaabbaababba...
```

Note: To see a list of all word constructors, type `words.` and then hit the TAB key. The documentation for each constructor includes information about each word, which provides a useful reference.

TESTS:

```
sage: from sage.combinat.words.word_generators import WordGenerator
sage: words2 = WordGenerator()
sage: type(loads(dumps(words2)))
<class 'sage.combinat.words.word_generators.WordGenerator'>
```

CharacteristicSturmianWord(*slope*, *alphabet=(0, 1)*, *bits=None*)

Returns the characteristic Sturmian word (also called standard Sturmian word) of given slope.

Over a binary alphabet $\{a, b\}$, the characteristic Sturmian word c_α of irrational slope α is the infinite word satisfying $s_{\alpha,0} = ac_\alpha$ and $s'_{\alpha,0} = bc_\alpha$, where $s_{\alpha,0}$ and $s'_{\alpha,0}$ are respectively the lower and upper mechanical words with slope α and intercept 0. Equivalently, for irrationnal α, $c_\alpha = s_{\alpha,\alpha} = s'_{\alpha,\alpha}$.

Let $\alpha = [0, d_1 + 1, d_2, d_3, \ldots]$ be the continued fraction expansion of α. It has been shown that the characteristic Sturmian word of slope α is also the limit of the sequence: $s_0 = b, s_1 = a, \ldots, s_{n+1} = s_n^{d_n} s_{n-1}$ for $n > 0$.

See Section 2.1 of [Loth02] for more details.

INPUT:

- `slope` - the slope of the word. It can be one of the following :
  - –real number in $]0, 1[$
  - –iterable over the continued fraction expansion of a real number in $]0, 1[$
- `alphabet` - any container of length two that is suitable to build an instance of OrderedAlphabet (list, tuple, str, ...)
- `bits` - integer (optional and considered only if `slope` is a real number) the number of bits to consider when computing the continued fraction.

OUTPUT:

word

ALGORITHM:

Let $[0, d_1+1, d_2, d_3, \ldots]$ be the continued fraction expansion of α. Then, the characteristic Sturmian word of slope α is the limit of the sequence: $s_0 = b$, $s_1 = a$ and $s_{n+1} = s_n^{d_n} s_{n-1}$ for $n > 0$.

EXAMPLES:

From real slope:

```
sage: words.CharacteristicSturmianWord(1/golden_ratio^2)
word: 0100101001001010010100100101001001010010...
sage: words.CharacteristicSturmianWord(4/5)
word: 11110
sage: words.CharacteristicSturmianWord(5/14)
word: 01001001001001
sage: words.CharacteristicSturmianWord(pi-3)
word: 0000001000000100000010000001000000100000...
```

From an iterator of the continued fraction expansion of a real:

```
sage: def cf():
...     yield 0
...     yield 2
...     while True: yield 1
sage: F = words.CharacteristicSturmianWord(cf()); F
word: 0100101001001010010100100101001001010010...
sage: Fib = words.FibonacciWord(); Fib
word: 0100101001001010010100100101001001010010...
sage: F[:10000] == Fib[:10000]
True
```

The alphabet may be specified:

```
sage: words.CharacteristicSturmianWord(cf(), 'rs')
word: rsrrsrsrrsrrsrsrrsrsrrsrrsrsrrsrrsrsrrsr...
```

The characteristic sturmian word of slope $(\sqrt{3}-1)/2$:

```
sage: words.CharacteristicSturmianWord((sqrt(3)-1)/2)
word: 0100100101001001001010010010010100100101...
```

The same word defined from the continued fraction expansion of $(\sqrt{3}-1)/2$:

```
sage: from itertools import cycle, chain
sage: it = chain([0], cycle([2, 1]))
sage: words.CharacteristicSturmianWord(it)
word: 0100100101001001001010010010010100100101...
```

The first terms of the standard sequence of the characteristic sturmian word of slope $(\sqrt{3}-1)/2$:

```
sage: words.CharacteristicSturmianWord([0,2])
word: 01
sage: words.CharacteristicSturmianWord([0,2,1])
word: 010
sage: words.CharacteristicSturmianWord([0,2,1,2])
word: 01001001
sage: words.CharacteristicSturmianWord([0,2,1,2,1])
word: 01001001010
sage: words.CharacteristicSturmianWord([0,2,1,2,1,2])
word: 0100100101001001001010010010010100100101
```

```
sage: words.CharacteristicSturmianWord([0,2,1,2,1,2,1])
word: 0100100101001001001010010010010100100101...
```

TESTS:

```
sage: words.CharacteristicSturmianWord([1,1,1],'xyz')
Traceback (most recent call last):
...
TypeError: alphabet does not contain two distinct elements

sage: words.CharacteristicSturmianWord(5/4)
Traceback (most recent call last):
...
ValueError: The argument slope (=5/4) must be in ]0,1[.

sage: words.CharacteristicSturmianWord(1/golden_ratio^2, bits=30)
doctest:...: DeprecationWarning: the argument 'bits' is deprecated
See http://trac.sagemath.org/14567 for details.
word: 0100101001001010010100100101001001010010...
sage: _.length()
+Infinity

sage: a = words.LowerMechanicalWord(1/pi)[1:]
sage: b = words.UpperMechanicalWord(1/pi)[1:]
sage: c = words.CharacteristicSturmianWord(1/pi)
sage: n = 500; a[:n] == b[:n] == c[:n]
True

sage: alpha = random()
sage: c = words.CharacteristicSturmianWord(alpha)
sage: l = words.LowerMechanicalWord(alpha)[1:]
sage: u = words.UpperMechanicalWord(alpha)[1:]
sage: i = 10000; j = i + 500; c[i:j] == l[i:j] == u[i:j]
True

sage: a, b = 207, 232
sage: u = words.ChristoffelWord(a, b)
sage: v = words.CharacteristicSturmianWord(a/(a+b))
sage: v.length()
439
sage: u[1:-1] == v[:-2]
True
```

ChristoffelWord

alias of LowerChristoffelWord

CodingOfRotationWord(*alpha*, *beta*, *x=0*, *alphabet=(0, 1)*)

Returns the infinite word obtained from the coding of rotation of parameters (α, β, x) over the given two-letter alphabet.

The *coding of rotation* corresponding to the parameters (α, β, x) is the symbolic sequence $u = (u_n)_{n\geq 0}$ defined over the binary alphabet $\{0, 1\}$ by $u_n = 1$ if $x + n\alpha \in [0, \beta[$ and $u_n = 0$ otherwise. See [AC03].

EXAMPLES:

```
sage: alpha = 0.45
sage: beta = 0.48
sage: words.CodingOfRotationWord(0.45, 0.48)
word: 1101010101001010101011010101010010101010...
```

```
sage: words.CodingOfRotationWord(0.45, 0.48, alphabet='xy')
word: yyxyxyxyxyxxyxyxyxyxyyxyxyxyxyxxyxyxyxyx...
```

TESTS:

```
sage: words.CodingOfRotationWord(0.51,0.43,alphabet=[1,0,2])
Traceback (most recent call last):
...
TypeError: alphabet does not contain two distinct elements
```

FibonacciWord(*alphabet=(0, 1)*, *construction_method='recursive'*)

Returns the Fibonacci word on the given two-letter alphabet.

INPUT:

- `alphabet` – any container of length two that is suitable to build an instance of OrderedAlphabet (list, tuple, str, ...)
- `construction_method` – can be any of the following: "recursive", "fixed point", "function" (see below for definitions).

Recursive construction: the Fibonacci word is the limit of the following sequence of words: $S_0 = 0$, $S_1 = 01$, $S_n = S_{n-1}S_{n-2}$ for $n \geq 2$.

Fixed point construction: the Fibonacci word is the fixed point of the morphism: $0 \mapsto 01$ and $1 \mapsto 0$. Hence, it can be constructed by the following read-write process:

1. beginning at the first letter of 01,
2. if the next letter is 0, append 01 to the word;
3. if the next letter is 1, append 1 to the word;
4. move to the next letter of the word.

Function: Over the alphabet $\{1, 2\}$, the n-th letter of the Fibonacci word is $\lfloor (n+2)\varphi \rfloor - \lfloor (n+1)\varphi \rfloor$ where $\varphi = (1 + \sqrt{5})/2$ is the golden ratio.

EXAMPLES:

```
sage: w = words.FibonacciWord(construction_method="recursive"); w
word: 01001010010010100101001001010010100101001010010...

sage: v = words.FibonacciWord(construction_method="recursive", alphabet='ab'); v
word: abaababaabaababaababaabaababaabaababaababa...

sage: u = words.FibonacciWord(construction_method="fixed point"); u
word: 01001010010010100101001001010010100101001010010...

sage: words.FibonacciWord(construction_method="fixed point", alphabet=[4, 1])
word: 41441414414414144144144144144144144144144144...

sage: words.FibonacciWord([0,1], 'function')
word: 01001010010010100101001001010010100101001010010...
sage: words.FibonacciWord('ab', 'function')
word: abaababaabaababaababaabaababaabaababaababa...
```

TESTS:

```
sage: from math import floor, sqrt
sage: golden_ratio = (1 + sqrt(5))/2.0
sage: a = golden_ratio / (1  + 2*golden_ratio)
sage: wn = lambda n : int(floor(a*(n+2)) - floor(a*(n+1)))
```

```
sage: f = Words([0,1])(wn); f
word: 0100101001001010010100100101001001010010...
sage: f[:10000] == w[:10000]
True
sage: f[:10000] == u[:10000] #long time
True
sage: words.FibonacciWord("abc")
Traceback (most recent call last):
...
TypeError: alphabet does not contain two distinct elements
```

FixedPointOfMorphism(*morphism*, *first_letter*)

Returns the fixed point of the morphism beginning with `first_letter`.

A *fixed point* of a morphism φ is a word w such that $\varphi(w) = w$.

INPUT:

- `morphism` – endomorphism prolongable on `first_letter`. It must be something that WordMorphism's constructor understands (dict, str, ...).
- `first_letter` – the first letter of the fixed point

OUTPUT:

The fixed point of the morphism beginning with `first_letter`

EXAMPLES:

```
sage: mu = {0:[0,1], 1:[1,0]}
sage: tm = words.FixedPointOfMorphism(mu,0); tm
word: 0110100110010110100101100110100110010110...
sage: TM = words.ThueMorseWord()
sage: tm[:1000] == TM[:1000]
True

sage: mu = {0:[0,1], 1:[0]}
sage: f = words.FixedPointOfMorphism(mu,0); f
word: 0100101001001010010100100101001001010010...
sage: F = words.FibonacciWord(); F
word: 0100101001001010010100100101001001010010...
sage: f[:1000] == F[:1000]
True

sage: fp = words.FixedPointOfMorphism('a->abc,b->,c->','a'); fp
word: abc
```

KolakoskiWord(*alphabet=(1, 2)*)

Returns the Kolakoski word over the given alphabet and starting with the first letter of the alphabet.

Let $A = \{a, b\}$ be an alphabet, where a and b are two distinct positive integers. The Kolakoski word $K_{a,b}$ over A and starting with a is the unique infinite word w such that $w = \Delta(w)$, where $\Delta(w)$ is the word encoding the runs of w (see `delta()` method on words for more details).

Note that $K_{a,b} \neq K_{b,a}$. On the other hand, the words $K_{a,b}$ and $K_{b,a}$ are the unique two words over A that are fixed by Δ.

INPUT:

- `alphabet` - (default: (1,2)) an iterable of two positive integers

OUTPUT:

infinite word

EXAMPLES:

The usual Kolakoski word:

```
sage: w = words.KolakoskiWord()
sage: w
word: 122112122122112112212112122112112122122 1...
sage: w.delta()
word: 122112122122112112212112122112112122122 1...
```

The other Kolakoski word on the same alphabet:

```
sage: w = words.KolakoskiWord(alphabet = (2,1))
sage: w
word: 221121221221121122121121221121121221221 1...
sage: w.delta()
word: 221121221221121122121121221121121221221 1...
```

It is naturally generalized to any two integers alphabet:

```
sage: w = words.KolakoskiWord(alphabet = (2,5))
sage: w
word: 225522222555552255225522555552222255555 2...
sage: w.delta()
word: 225522222555552255225522555552222255555 2...
```

TESTS:

```
sage: for i in range(1,10):
...       for j in range(1,10):
...           if i != j:
...               w = words.KolakoskiWord(alphabet=(i,j))
...               assert w[:50] == w.delta()[:50]

sage: words.KolakoskiWord((0, 2))
Traceback (most recent call last):
...
ValueError: The alphabet (=(0, 2)) must consist of two distinct positive integers
```

REFERENCES:

LowerChristoffelWord
alias of LowerChristoffelWord

LowerMechanicalWord(*alpha*, *rho=0*, *alphabet=None*)
Returns the lower mechanical word with slope α and intercept ρ

The lower mechanical word $s_{\alpha,\rho}$ with slope α and intercept ρ is defined by $s_{\alpha,\rho}(n) = \lfloor \alpha(n+1) + \rho \rfloor - \lfloor \alpha n + \rho \rfloor$. [Loth02]

INPUT:

- `alpha` – real number such that $0 \leq \alpha \leq 1$
- `rho` – real number (optional, default: 0)
- `alphabet` – iterable of two elements or `None` (optional, default: `None`)

OUTPUT:

infinite word

EXAMPLES:

```
sage: words.LowerMechanicalWord(1/golden_ratio^2)
word: 0010010100100101001010010010100100101001...
sage: words.LowerMechanicalWord(1/5)
word: 0000100001000010000100001000010000100001...
sage: words.LowerMechanicalWord(1/pi)
word: 0001001001001001001001000100100100100100...
```

TESTS:

```
sage: m = words.LowerMechanicalWord(1/golden_ratio^2)[1:]
sage: s = words.CharacteristicSturmianWord(1/golden_ratio^2)
sage: m[:500] == s[:500]
True
```

Check that this returns a word in an alphabet (trac ticket #10054):

```
sage: words.UpperMechanicalWord(1/golden_ratio^2).parent()
Infinite words over {0, 1}
```

MinimalSmoothPrefix(*n*)

This function finds and returns the minimal smooth prefix of length n.

See [BMP07] for a definition.

INPUT:

- n – the desired length of the prefix

OUTPUT:

word – the prefix

Note: Be patient, this function can take a really long time if asked for a large prefix.

EXAMPLES:

```
sage: words.MinimalSmoothPrefix(10)
word: 1212212112
```

REFERENCES:

PalindromicDefectWord(*k=1*, *alphabet='ab'*)

Returns the finite word $w = ab^k ab^{k-1} aab^{k-1} ab^k a$.

As described by Brlek, Hamel, Nivat and Reuteunaer in [BHNR04], this finite word w is such that the infinite periodic word w^ω have palindromic defect k.

INPUT:

- k – positive integer (optional, default: 1)
- alphabet – iterable (optional, default: 'ab') of size two

OUTPUT:

finite word

EXAMPLES:

```
sage: words.PalindromicDefectWord(10)
word: abbbbbbbbbbabbbbbbbbbaabbbbbbbbbbabbbbbbb...
```

```
sage: w = words.PalindromicDefectWord(3)
sage: w
word: abbbabbaabbabbba
sage: w.defect()
0
sage: (w^2).defect()
3
sage: (w^3).defect()
3
```

On other alphabets:

```
sage: words.PalindromicDefectWord(3, alphabet='cd')
word: cdddcddccddcdddc
sage: words.PalindromicDefectWord(3, alphabet=['c', 3])
word: c333c33cc33c333c
```

TESTS:

```
sage: k = 25
sage: (words.PalindromicDefectWord(k)^2).defect()
25
```

If k is negative or zero, then we get the same word:

```
sage: words.PalindromicDefectWord(0)
word: aaaaaa
sage: words.PalindromicDefectWord(-3)
word: aaaaaa
```

RandomWord(*n*, *m=2*, *alphabet=None*)

Returns a random word of length n over the given m-letter alphabet.

INPUT:

- `n` - integer, the length of the word
- `m` - integer (default 2), the size of the output alphabet
- `alphabet` - (default is $\{0, 1, ..., m-1\}$) any container of length m that is suitable to build an instance of OrderedAlphabet (list, tuple, str, ...)

EXAMPLES:

```
sage: words.RandomWord(10)          # random results
word: 0110100101
sage: words.RandomWord(10, 4)       # random results
word: 0322313320
sage: words.RandomWord(100, 7)      # random results
word: 26306440236425164426500256113000034413310...
sage: words.RandomWord(100, 7, range(-3,4))  # random results
word: 1,3,-1,-1,3,2,2,0,1,-2,1,-1,-3,-2,2,0,3,0,-3,0,3,0,-2,-2,2,0,1,-3,2,-2,-2,2,0,2,1,-2,-
sage: words.RandomWord(100, 5, "abcde") # random results
word: acebeaaccdbedbbbdeadeebbdeeebeaaacbadaac...
sage: words.RandomWord(17, 5, "abcde")      # random results
word: dcacbbecbddebaadd
```

TESTS:

```
sage: words.RandomWord(2,3,"abcd")
Traceback (most recent call last):
```

```
...
TypeError: alphabet does not contain 3 distinct elements
```

StandardEpisturmianWord(*directive_word*)

Returns the standard episturmian word (or epistandard word) directed by directive_word. Over a 2-letter alphabet, this function gives characteristic Sturmian words.

An infinite word w over a finite alphabet A is said to be *standard episturmian* (or *epistandard*) iff there exists an infinite word $x_1x_2x_3\cdots$ over A (called the *directive word* of w) such that w is the limit as n goes to infinity of $Pal(x_1\cdots x_n)$, where Pal is the iterated palindromic closure function.

Note that an infinite word is *episturmian* if it has the same set of factors as some epistandard word.

See for instance [DJP01], [JP02], and [GJ07].

INPUT:

- `directive_word` - an infinite word or a period of a periodic infinite word

EXAMPLES:

```
sage: Fibonacci = words.StandardEpisturmianWord(Words('ab')('ab')); Fibonacci
word: abaababaabaababaababaabaababaabaababaaba...
sage: Tribonacci = words.StandardEpisturmianWord(Words('abc')('abc')); Tribonacci
word: abacabaabacababacabaabacabacabaabacababa...
sage: S = words.StandardEpisturmianWord(Words('abcd')('aabcabada')); S
word: aabaacaabaaabaacaabaabaacaabaaabaacaabaa...
sage: S = words.StandardEpisturmianWord(Fibonacci); S
word: abaabaababaabaabaababaabaababaabaabaabab...
sage: S[:25]
word: abaabaababaabaabaababaaba
sage: S = words.StandardEpisturmianWord(Tribonacci); S
word: abaabacabaabaabacabaababaabacabaabaabaca...
sage: words.StandardEpisturmianWord(123)
Traceback (most recent call last):
...
TypeError: directive_word is not a word, so it cannot be used to build an episturmian word
sage: words.StandardEpisturmianWord(Words('ab'))
Traceback (most recent call last):
...
TypeError: directive_word is not a word, so it cannot be used to build an episturmian word
```

REFERENCES:

ThueMorseWord(*alphabet=(0, 1)*, *base=2*)

Returns the (Generalized) Thue-Morse word over the given alphabet.

There are several ways to define the Thue-Morse word t. We use the following definition: $t[n]$ is the sum modulo m of the digits in the given base expansion of n.

See [BmBGL07], [Brlek89], and [MH38].

INPUT:

- `alphabet` - (default: (0, 1)) any container that is suitable to build an instance of OrderedAlphabet (list, tuple, str, ...)
- `base` - an integer (default : 2) greater or equal to 2

EXAMPLES:

Thue-Morse word:

```
sage: t = words.ThueMorseWord(); t
word: 0110100110010110100101100110100110010110...
```

Thue-Morse word on other alphabets:

```
sage: t = words.ThueMorseWord('ab'); t
word: abbabaabbaababbabaababbaabbabaabbaababba...

sage: t = words.ThueMorseWord(['L1', 'L2'])
sage: t[:8]
word: L1,L2,L2,L1,L2,L1,L1,L2
```

Generalized Thue Morse word:

```
sage: words.ThueMorseWord(alphabet=(0,1,2), base=2)
word: 0112122012202001122020012001011212202001...
sage: t = words.ThueMorseWord(alphabet=(0,1,2), base=5); t
word: 0120112012201200120112012120122012001201...
sage: t[100:130].critical_exponent()
10/3
```

TESTS:

```
sage: words.ThueMorseWord(alphabet='ab', base=1)
Traceback (most recent call last):
...
ValueError: base (=1) and len(alphabet) (=2) must be at least 2
```

REFERENCES:

UpperChristoffelWord(*p*, *q*, *alphabet=(0, 1)*)

Returns the upper Christoffel word of slope p/q, where p and q are relatively prime non-negative integers, over the given alphabet.

The *upper Christoffel word of slope 'p/q'* is equal to the reversal of the lower Christoffel word of slope p/q. Equivalently, if xuy is the lower Christoffel word of slope p/q, where x and y are letters, then yux is the upper Christoffel word of slope p/q (because u is a palindrome).

INPUT:

- `alphabet` - any container of length two that is suitable to build an instance of OrderedAlphabet (list, tuple, str, ...)

EXAMPLES:

```
sage: words.UpperChristoffelWord(1,0)
word: 1

sage: words.UpperChristoffelWord(0,1)
word: 0

sage: words.UpperChristoffelWord(1,1)
word: 10

sage: words.UpperChristoffelWord(4,7)
word: 10100100100
```

TESTS:

```
sage: words.UpperChristoffelWord(51,43,"abc")
Traceback (most recent call last):
```

```
...
ValueError: alphabet must contain exactly two distinct elements
```

UpperMechanicalWord(*alpha*, *rho=0*, *alphabet=None*)

Returns the upper mechanical word with slope α and intercept ρ

The upper mechanical word $s'_{\alpha,\rho}$ with slope α and intercept ρ is defined by $s'_{\alpha,\rho}(n) = \lceil \alpha(n+1) + \rho \rceil - \lceil \alpha n + \rho \rceil$. [Loth02]

INPUT:

- `alpha` – real number such that $0 \leq \alpha \leq 1$
- `rho` – real number (optional, default: 0)
- `alphabet` – iterable of two elements or `None` (optional, default: `None`)

OUTPUT:

infinite word

EXAMPLES:

```
sage: words.UpperMechanicalWord(1/golden_ratio^2)
word: 10100101001001010010100100101001001010001...
sage: words.UpperMechanicalWord(1/5)
word: 10000100001000010000100001000010000100000...
sage: words.UpperMechanicalWord(1/pi)
word: 10010010010010010010010001001001001001001...
```

TESTS:

```
sage: m = words.UpperMechanicalWord(1/golden_ratio^2)[1:]
sage: s = words.CharacteristicSturmianWord(1/golden_ratio^2)
sage: m[:500] == s[:500]
True
```

Check that this returns a word in an alphabet (trac ticket #10054):

```
sage: words.UpperMechanicalWord(1/golden_ratio^2).parent()
Infinite words over {0, 1}
```

dual_fibonacci_tile(*n*)

Returns the n-th dual Fibonacci Tile [BmBGL09].

EXAMPLES:

```
sage: for i in range(4): words.dual_fibonacci_tile(i)
Path: 3210
Path: 32123032301030121012
Path: 32123032301032303212321012321230321232103...
Path: 32123032301032303212321012321230321232103...
```

fibonacci_tile(*n*)

Returns the n-th Fibonacci Tile [BmBGL09].

EXAMPLES:

```
sage: for i in range(3): words.fibonacci_tile(i)
Path: 3210
Path: 323030101212
Path: 32303010303232123230323212101212321210103...
```

s_adic(*sequence*, *letters*, *morphisms=None*)

Returns the s-adic infinite word obtained from a sequence of morphisms applied on a letter.

DEFINITION (from [Fogg]):

Let w be a infinite word over an alphabet $A = A_0$. A standard representation of w is obtained from a sequence of substitutions $\sigma_k : A_{k+1} \to A_k$ and a sequence of letters $a_k \in A_k$ such that:

$$\lim_{k\to\infty} \sigma_0 \circ \sigma_1 \circ \cdots \sigma_k(a_k).$$

Given a set of substitutions S, we say that the representation is S-adic standard if the subtitutions are chosen in S.

INPUT:

- `sequence` - An iterable sequence of indices or of morphisms. It may be finite or infinite. If `sequence` is infinite, the image of the $(i + 1)$-th letter under the $(i + 1)$-th morphism must start with the i-th letter.
- `letters` - A letter or a sequence of letters.
- `morphisms` - dict, list, callable or `None` (optional, default `None`) an object that maps indices to morphisms. If `None`, then `sequence` must consist of morphisms.

OUTPUT:

A word.

EXAMPLES:

Let's define three morphisms and compute the first nested succesive prefixes of the s-adic word:

```
sage: m1 = WordMorphism('e->gh,f->hg')
sage: m2 = WordMorphism('c->ef,d->e')
sage: m3 = WordMorphism('a->cd,b->dc')
sage: words.s_adic([m1],'e')
word: gh
sage: words.s_adic([m1,m2],'ec')
word: ghhg
sage: words.s_adic([m1,m2,m3],'eca')
word: ghhggh
```

When the given sequence of morphism is finite, one may simply give the last letter, i.e. `'a'`, instead of giving all of them, i.e. `'eca'`:

```
sage: words.s_adic([m1,m2,m3],'a')
word: ghhggh
sage: words.s_adic([m1,m2,m3],'b')
word: ghghhg
```

If the letters don't satisfy the hypothesis of the algorithm (nested prefixes), an error is raised:

```
sage: words.s_adic([m1,m2,m3],'ecb')
Traceback (most recent call last):
...
ValueError: The hypothesis of the algorithm used is not satisfied: the image of the 3-th let
```

Let's define the Thue-Morse morphism and the Fibonacci morphism which will be used below to illustrate more examples and let's import the `repeat` tool from the `itertools`:

```
sage: tm = WordMorphism('a->ab,b->ba')
sage: fib = WordMorphism('a->ab,b->a')
sage: from itertools import repeat
```

Two trivial examples of infinite s-adic words:

```
sage: words.s_adic(repeat(tm),repeat('a'))
word: abbabaabbaababbabaababbaabbabaabbaababba...

sage: words.s_adic(repeat(fib),repeat('a'))
word: abaababaabaababaababaabaababaabaababaaba...
```

A less trivial infinite s-adic word:

```
sage: t = words.ThueMorseWord([tm,fib])
sage: words.s_adic(t, repeat('a'))
word: abbaababbaabbaabbaababbaababbaabbaababba...
```

The same thing using a sequence of indices:

```
sage: tmword = words.ThueMorseWord([0,1])
sage: words.s_adic(tmword, repeat('a'), [tm,fib])
word: abbaababbaabbaabbaababbaababbaabbaababba...
```

The correspondance of the indices may be given as a dict:

```
sage: words.s_adic(tmword, repeat('a'), {0:tm,1:fib})
word: abbaababbaabbaabbaababbaababbaabbaababba...
```

because dict are more versatile for indices:

```
sage: tmwordTF = words.ThueMorseWord('TF')
sage: words.s_adic(tmwordTF, repeat('a'), {'T':tm,'F':fib})
word: abbaababbaabbaabbaababbaababbaabbaababba...
```

or by a callable:

```
sage: f = lambda n: tm if n == 0 else fib
sage: words.s_adic(words.ThueMorseWord(), repeat('a'), f)
word: abbaababbaabbaabbaababbaababbaabbaababba...
```

Random infinite s-adic words:

```
sage: from sage.misc.prandom import randint
sage: def it():
...       while True: yield randint(0,1)
sage: words.s_adic(it(), repeat('a'), [tm,fib])
word: abbaababababbaababbaabbaababbaababababbaabba...
sage: words.s_adic(it(), repeat('a'), [tm,fib])
word: abbaababbaabbaababbaababbaabbaababbaababbaabba...
sage: words.s_adic(it(), repeat('a'), [tm,fib])
word: abaaababaabaabaaababaabaaababaabaaababaabaa...
```

An example where the sequences cycle on two morphisms and two letters:

```
sage: G = WordMorphism('a->cd,b->dc')
sage: H = WordMorphism('c->ab,d->ba')
sage: from itertools import cycle
sage: words.s_adic([G,H],'ac')
word: cddc
sage: words.s_adic(cycle([G,H]),cycle('ac'))
word: cddcdccddccdcddcdccdcddccddcdccddccdcddc...
```

The morphism $\sigma : a \mapsto ba, b \mapsto b$ can't satisfy the hypothesis of the nested prefixes, but one may compute arbitrarily long finite words having the limit $\sigma^\omega(a)$:

```
sage: sigma = WordMorphism('a->ba,b->b')
sage: words.s_adic(repeat(sigma),repeat('a'))
Traceback (most recent call last):
...
ValueError: The hypothesis of the algorithm used is not satisfied: the image of the 2-th let
sage: words.s_adic([sigma],'a')
word: ba
sage: words.s_adic([sigma,sigma],'a')
word: bba
sage: words.s_adic([sigma]*3,'a')
word: bbba
sage: words.s_adic([sigma]*4,'a')
word: bbbba
sage: words.s_adic([sigma]*5,'a')
word: bbbbba
sage: words.s_adic([sigma]*6,'a')
word: bbbbbba
sage: words.s_adic([sigma]*7,'a')
word: bbbbbbba
```

The following examples illustrates an S-adic word defined over an infinite set S of morphisms x_h:

```
sage: x = lambda h:WordMorphism({1:[2],2:[3]+[1]*(h+1),3:[3]+[1]*h})
sage: for h in [0,1,2,3]: print h, x(h)
0 1->2, 2->31, 3->3
1 1->2, 2->311, 3->31
2 1->2, 2->3111, 3->311
3 1->2, 2->31111, 3->3111
sage: w = Word(lambda n : valuation(n+1, 2) ); w
word: 0102010301020104010201030102010501020103...
sage: s = words.s_adic(w, repeat(3), x); s
word: 32322322323223223223223223223232232232232...
sage: prefixe = s[:10000]
sage: map(prefixe.number_of_factors, range(15))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: [_[i+1] - _[i] for i in range(len(_)-1)]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

TESTS:

```
sage: tm = WordMorphism('a->ab,b->ba')
sage: fib = WordMorphism('a->ab,b->a')
sage: w = words.s_adic([fib,tm,tm,fib,tm,fib]*3,'a')
sage: w
word: abaaabaababaabaaababaaabababaaabaababaabaa...
sage: w.length()
32400
sage: w.parent()
Finite words over {'a', 'b'}
sage: type(w)
<class 'sage.combinat.words.word.FiniteWord_iter_with_caching'>

sage: words.s_adic([fib,tm,tm,fib,tm,fib],'aaaaaaa')
word: abaaabaababaabaaababaaabababaaabaababa

sage: words.s_adic([0,1,0,1,0,1,0,1],'a',[tm,fib])
word: abbaababababbaabbaababbaababbaababababbaabba...
```

```
sage: words.s_adic([fib,fib],'bb')
Traceback (most recent call last):
...
ValueError: The hypothesis of the algorithm used is not satisfied: the image of the 2-th let
```

Test on different letters:

```
sage: tm = WordMorphism({0:[0,1], 1:[1,0]})
sage: fib = WordMorphism({0:[0,1], 1:[0]})
sage: f = lambda n: tm if n == 0 else fib
sage: words.s_adic(words.ThueMorseWord(), repeat(0), f)
word: 01100101100110011001011001011001100110010110...
```

Testing the message error for the third argument:

```
sage: words.s_adic(words.ThueMorseWord(), repeat(0), 5)
Traceback (most recent call last):
...
TypeError: morphisms (=5) must be None, callable or provide a __getitem__ method.
```

AUTHORS:

- Sebastien Labbe (2009-12-18): initial version

5.1.332 Datatypes for words defined by iterators and callables

class sage.combinat.words.word_infinite_datatypes.**WordDatatype_callable**(*parent*, *callable*, *length=None*)

Bases: sage.combinat.words.word_datatypes.WordDatatype

Datatype for a word defined by a callable.

class sage.combinat.words.word_infinite_datatypes.**WordDatatype_callable_with_caching**(*parent*, *callable*, *length=No*

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_callable

Datatype for a word defined by a callable.

flush()

Empty the associated cache of letters.

EXAMPLES:

The first 40 (by default) values are always cached:

```
sage: w = words.ThueMorseWord()
sage: w._letter_cache
{0: 0, 1: 1, 2: 1, 3: 0, 4: 1, 5: 0, 6: 0, 7: 1, 8: 1, 9: 0, 10: 0, 11: 1, 12: 0, 13: 1, 14:
sage: w[100]
1
sage: w._letter_cache
{0: 0, 1: 1, 2: 1, 3: 0, 4: 1, 5: 0, 6: 0, 7: 1, 8: 1, 9: 0, 10: 0, 11: 1, 12: 0, 13: 1, 14:
sage: w.flush()
sage: w._letter_cache
{}
```

class sage.combinat.words.word_infinite_datatypes.**WordDatatype_iter**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_datatypes.WordDatatype

INPUT:

- `parent` - a parent
- `iter` - an iterator
- `length` - (default: `None`) the length of the word

EXAMPLES:

```
sage: w = Word(iter("abbabaab"), length="unknown", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length() is None
False
sage: w.length()
8
sage: s = "abbabaabbaabab babaabab baabbabaabbaabab babbaabbabaabab"
sage: w = Word(iter(s), length="unknown", caching=False); w
word: abbabaabbaababbabaababbaabbabaabbaababba...
sage: w.length() is None
True

sage: w = Word(iter("abbabaab"), length="finite", caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8, caching=False); w
word: abbabaab
sage: isinstance(w, sage.combinat.words.word_infinite_datatypes.WordDatatype_iter)
True
sage: w.length()
8
```

class sage.combinat.words.word_infinite_datatypes.**WordDatatype_iter_with_caching**(*parent*, *iter*, *length=None*)

Bases: sage.combinat.words.word_infinite_datatypes.WordDatatype_iter

INPUT:

- `parent` - a parent
- `iter` - an iterator
- `length` - (default: `None`) the length of the word

EXAMPLES:

```
sage: import itertools
sage: Word(itertools.cycle("abbabaab"))
word: abbabaababbabaababbabaababbabaababbabaab...
sage: w = Word(iter("abbabaab"), length="finite"); w
word: abbabaab
sage: w.length()
8
```

```
sage: w = Word(iter("abbabaab"), length="unknown"); w
word: abbabaab
sage: w.length()
8
sage: list(w)
['a', 'b', 'b', 'a', 'b', 'a', 'a', 'b']
sage: w.length()
8
sage: w = Word(iter("abbabaab"), length=8)
sage: w._len
8
```

flush()
Delete the cached values.

EXAMPLES:

```
sage: from itertools import count
sage: w = Word(count())
sage: w._last_index, len(w._list)
(39, 40)
sage: w[43]
43
sage: w._last_index, len(w._list)
(43, 44)
sage: w.flush()
sage: w._last_index, w._list
(-1, [])
```

5.1.333 User-customizable options for words

sage.combinat.words.word_options.**WordOptions**(***kwargs*)
Sets the global options for elements of the word class. The defaults are for words to be displayed in list notation.

INPUT:

- `display` - 'string' (default), or 'list', words are displayed in string or list notation.
- `truncate` - boolean (default: True), whether to truncate the string output of long words (see truncate_length below).
- `truncate_length` - integer (default: 40), if the length of the word is greater than this integer, then the word is truncated.
- `letter_separator` - (string, default: ",") if the string representation of letters have length greater than 1, then the letters are separated by this string in the string representation of the word.

If no parameters are set, then the function returns a copy of the options dictionary.

EXAMPLES:

```
sage: w = Word([2,1,3,12])
sage: u = Word("abba")
sage: WordOptions(display='list')
sage: w
word: [2, 1, 3, 12]
sage: u
word: ['a', 'b', 'b', 'a']
sage: WordOptions(display='string')
sage: w
```

```
word: 2,1,3,12
sage: u
word: abba
```

5.1.334 Combinatorial classes of words.

To define a new class of words, please refer to the documentation file: sage/combinat/words/notes/word_inheritance_howto.txt

AUTHORS:

- Franco Saliola (2008-12-17): merged into sage
- Sebastien Labbe (2008-12-17): merged into sage
- Arnaud Bergeron (2008-12-17): merged into sage
- Sebastien Labbe (2009-07-21): Improved morphism iterator (trac ticket #6571).
- Vincent Delecroix (2015): classes simplifications (trac ticket #19619)

EXAMPLES:

```
sage: Words()
Finite and infinite words over Set of Python objects of type 'object'
sage: Words(4)
Finite and infinite words over {1, 2, 3, 4}
sage: Words(4,5)
Words of length 5 over {1, 2, 3, 4}

sage: FiniteWords('ab')
Finite words over {'a', 'b'}
sage: InfiniteWords('natural numbers')
Infinite words over Non negative integers
```

class sage.combinat.words.words.**AbstractLanguage**(*alphabet=None*, *category=None*)

Bases: sage.structure.parent.Parent

Abstract base class

This is *not* to be used by any means. This class gather previous features of set of words (prior to trac ticket #19619). In the future that class might simply disappear or become a common base class for all languages. In the latter case, its name would possibly change to Language.

alphabet()

EXAMPLES:

```
sage: Words(NN).alphabet()
Non negative integer semiring

sage: InfiniteWords([1,2,3]).alphabet()
{1, 2, 3}
sage: InfiniteWords('ab').alphabet()
{'a', 'b'}

sage: FiniteWords([1,2,3]).alphabet()
{1, 2, 3}
sage: FiniteWords().alphabet()
Set of Python objects of type 'object'
```

has_letter(*letter*)
Returns True if the alphabet of self contains the given letter.

INPUT:

- `letter` - a letter

EXAMPLES:

```
sage: W = Words()
sage: W.has_letter('a')
doctest:...: DeprecationWarning: has_letter is deprecated. Use 'letter
in W.alphabet()' instead
See http://trac.sagemath.org/19619 for details.
True
sage: W.has_letter(1)
True
sage: W.has_letter({})
True
sage: W.has_letter([])
True
sage: W.has_letter(range(5))
True
sage: W.has_letter(Permutation([]))
True

sage: W = Words(['a','b','c'])
sage: W.has_letter('a')
True
sage: W.has_letter('d')
False
sage: W.has_letter(8)
False
```

identity_morphism()
Returns the identity morphism from self to itself.

EXAMPLES:

```
sage: W = Words('ab')
sage: W.identity_morphism()
WordMorphism: a->a, b->b

sage: W = Words(range(3))
sage: W.identity_morphism()
WordMorphism: 0->0, 1->1, 2->2
```

There is no support yet for infinite alphabet:

```
sage: W = Words(alphabet=Alphabet(name='NN'))
sage: W
Finite and infinite words over Non negative integers
sage: W.identity_morphism()
Traceback (most recent call last):
...
NotImplementedError: size of alphabet must be finite
```

size_of_alphabet()
Returns the size of the alphabet.

EXAMPLES:

```
sage: Words().size_of_alphabet()
doctest:...: DeprecationWarning: size_of_alphabet is deprecated. Use
W.alphabet().cardinality() instead
See http://trac.sagemath.org/19619 for details.
+Infinity
sage: Word('abaccefa').parent().size_of_alphabet()
+Infinity
sage: Words('abcdef').size_of_alphabet()
6
sage: Words('').size_of_alphabet()
0
sage: Words('456').size_of_alphabet()
3
```

class sage.combinat.words.words.**FiniteOrInfiniteWords**(*alphabet*)

Bases: sage.combinat.words.words.AbstractLanguage

INPUT:

- alphabet – the underlying alphabet

TESTS:

```
sage: loads(dumps(Words())) == Words()
True
```

cardinality()

Return the cardinality of this set of words.

EXAMPLES:

```
sage: Words('abcd').cardinality()
+Infinity
sage: Words('a').cardinality()
+Infinity
sage: Words('').cardinality()
1
```

factors()

Return the set of finite words.

EXAMPLES:

```
sage: Words('ab').finite_words()
Finite words over {'a', 'b'}
```

finite_words()

Return the set of finite words.

EXAMPLES:

```
sage: Words('ab').finite_words()
Finite words over {'a', 'b'}
```

infinite_words()

Return the set of infinite words.

EXAMPLES:

```
sage: Words('ab').infinite_words()
Infinite words over {'a', 'b'}
```

iterate_by_length(*length*)
Return an iterator over the words of given length.

EXAMPLES:

```
sage: for w in Words('ab').iterate_by_length(3):
....:       print w,
aaa aab aba abb baa bab bba bbb
```

shift()
Return the set of infinite words.

EXAMPLES:

```
sage: Words('ab').infinite_words()
Infinite words over {'a', 'b'}
```

class sage.combinat.words.words.**FiniteWords**(*alphabet=None*, *category=None*)
Bases: sage.combinat.words.words.AbstractLanguage

The set of finite words over a fixed alphabet.

EXAMPLES:

```
sage: W = FiniteWords('ab')
sage: W
Finite words over {'a', 'b'}
```

cardinality()
Return the cardinality of this set.

EXAMPLES:

```
sage: FiniteWords('').cardinality()
1
sage: FiniteWords('a').cardinality()
+Infinity
```

factors()
Return itself.

EXAMPLES:

```
sage: FiniteWords('ab').factors()
Finite words over {'a', 'b'}
```

iter_morphisms(*arg=None*, *codomain=None*, *min_length=1*)
Iterate over all morphisms with domain `self` and the given codomain.

INPUT:

- `arg` - (optional, default: None) It can be one of the following :
  - `None` - then the method iterates through all morphisms.
  - tuple (a, b) of two integers - It specifies the range `range(a, b)` of values to consider for the sum of the length of the image of each letter in the alphabet.
  - list of nonnegative integers - The length of the list must be equal to the size of the alphabet, and the i-th integer of `arg` determines the length of the word mapped to by the i-th letter of the (ordered) alphabet.
- `codomain` - (default: None) a combinatorial class of words. By default, `codomain` is `self`.

- `min_length` - (default: 1) nonnegative integer. If `arg` is not specified, then iterate through all the morphisms where the length of the images of each letter in the alphabet is at least `min_length`. This is ignored if `arg` is a list.

OUTPUT:

iterator

EXAMPLES:

Iterator over all non-erasing morphisms:

```
sage: W = FiniteWords('ab')
sage: it = W.iter_morphisms()
sage: for _ in range(7): next(it)
WordMorphism: a->a, b->a
WordMorphism: a->a, b->b
WordMorphism: a->b, b->a
WordMorphism: a->b, b->b
WordMorphism: a->aa, b->a
WordMorphism: a->aa, b->b
WordMorphism: a->ab, b->a
```

Iterator over all morphisms including erasing morphisms:

```
sage: W = FiniteWords('ab')
sage: it = W.iter_morphisms(min_length=0)
sage: for _ in range(7): next(it)
WordMorphism: a->, b->
WordMorphism: a->a, b->
WordMorphism: a->b, b->
WordMorphism: a->, b->a
WordMorphism: a->, b->b
WordMorphism: a->aa, b->
WordMorphism: a->ab, b->
```

Iterator over morphisms where the sum of the lengths of the images of the letters is in a specific range:

```
sage: for m in W.iter_morphisms((0, 3), min_length=0): m
WordMorphism: a->aa, b->
WordMorphism: a->ab, b->
WordMorphism: a->ba, b->
WordMorphism: a->bb, b->
WordMorphism: a->a, b->a
WordMorphism: a->a, b->b
WordMorphism: a->b, b->a
WordMorphism: a->b, b->b
WordMorphism: a->a, b->
WordMorphism: a->b, b->
WordMorphism: a->, b->aa
WordMorphism: a->, b->ab
WordMorphism: a->, b->ba
WordMorphism: a->, b->bb
WordMorphism: a->, b->a
WordMorphism: a->, b->b
WordMorphism: a->, b->

sage: for m in W.iter_morphisms( (2, 4) ): m
WordMorphism: a->aa, b->a
WordMorphism: a->aa, b->b
WordMorphism: a->ab, b->a
```

```
WordMorphism: a->ab, b->b
WordMorphism: a->ba, b->a
WordMorphism: a->ba, b->b
WordMorphism: a->bb, b->a
WordMorphism: a->bb, b->b
WordMorphism: a->a, b->aa
WordMorphism: a->a, b->ab
WordMorphism: a->a, b->ba
WordMorphism: a->a, b->bb
WordMorphism: a->b, b->aa
WordMorphism: a->b, b->ab
WordMorphism: a->b, b->ba
WordMorphism: a->b, b->bb
WordMorphism: a->a, b->a
WordMorphism: a->a, b->b
WordMorphism: a->b, b->a
WordMorphism: a->b, b->b
```

Iterator over morphisms with specific image lengths:

```
sage: for m in W.iter_morphisms([0, 0]): m
WordMorphism: a->, b->
sage: for m in W.iter_morphisms([0, 1]): m
WordMorphism: a->, b->a
WordMorphism: a->, b->b
sage: for m in W.iter_morphisms([2, 1]): m
WordMorphism: a->aa, b->a
WordMorphism: a->aa, b->b
WordMorphism: a->ab, b->a
WordMorphism: a->ab, b->b
WordMorphism: a->ba, b->a
WordMorphism: a->ba, b->b
WordMorphism: a->bb, b->a
WordMorphism: a->bb, b->b
sage: for m in W.iter_morphisms([2, 2]): m
WordMorphism: a->aa, b->aa
WordMorphism: a->aa, b->ab
WordMorphism: a->aa, b->ba
WordMorphism: a->aa, b->bb
WordMorphism: a->ab, b->aa
WordMorphism: a->ab, b->ab
WordMorphism: a->ab, b->ba
WordMorphism: a->ab, b->bb
WordMorphism: a->ba, b->aa
WordMorphism: a->ba, b->ab
WordMorphism: a->ba, b->ba
WordMorphism: a->ba, b->bb
WordMorphism: a->bb, b->aa
WordMorphism: a->bb, b->ab
WordMorphism: a->bb, b->ba
WordMorphism: a->bb, b->bb
```

The codomain may be specified as well:

```
sage: Y = FiniteWords('xyz')
sage: for m in W.iter_morphisms([0, 2], codomain=Y): m
WordMorphism: a->, b->xx
WordMorphism: a->, b->xy
WordMorphism: a->, b->xz
```

```
WordMorphism: a->, b->yx
WordMorphism: a->, b->yy
WordMorphism: a->, b->yz
WordMorphism: a->, b->zx
WordMorphism: a->, b->zy
WordMorphism: a->, b->zz
sage: for m in Y.iter_morphisms([0,2,1], codomain=W): m
WordMorphism: x->, y->aa, z->a
WordMorphism: x->, y->aa, z->b
WordMorphism: x->, y->ab, z->a
WordMorphism: x->, y->ab, z->b
WordMorphism: x->, y->ba, z->a
WordMorphism: x->, y->ba, z->b
WordMorphism: x->, y->bb, z->a
WordMorphism: x->, y->bb, z->b
sage: it = W.iter_morphisms(codomain=Y)
sage: for _ in range(10): next(it)
WordMorphism: a->x, b->x
WordMorphism: a->x, b->y
WordMorphism: a->x, b->z
WordMorphism: a->y, b->x
WordMorphism: a->y, b->y
WordMorphism: a->y, b->z
WordMorphism: a->z, b->x
WordMorphism: a->z, b->y
WordMorphism: a->z, b->z
WordMorphism: a->xx, b->x
```

TESTS:

```
sage: list(W.iter_morphisms([1,0]))
[WordMorphism: a->a, b->, WordMorphism: a->b, b->]
sage: list(W.iter_morphisms([0,0], codomain=Y))
[WordMorphism: a->, b->]
sage: list(W.iter_morphisms([0, 1, 2]))
Traceback (most recent call last):
...
TypeError: arg (=[0, 1, 2]) must be an iterable of 2 integers
sage: list(W.iter_morphisms([0, 'a']))
Traceback (most recent call last):
...
TypeError: arg (=[0, 'a']) must be an iterable of 2 integers
sage: list(W.iter_morphisms([0, 1], codomain='a'))
Traceback (most recent call last):
...
TypeError: codomain (=a) must be an instance of FiniteWords
```

iterate_by_length(*l=1*)

Returns an iterator over all the words of self of length l.

INPUT:

- `l` - integer (default: 1), the length of the desired words

EXAMPLES:

```
sage: W = FiniteWords('ab')
sage: list(W.iterate_by_length(1))
[word: a, word: b]
sage: list(W.iterate_by_length(2))
```

```
[word: aa, word: ab, word: ba, word: bb]
sage: list(W.iterate_by_length(3))
[word: aaa,
 word: aab,
 word: aba,
 word: abb,
 word: baa,
 word: bab,
 word: bba,
 word: bbb]
sage: list(W.iterate_by_length('a'))
Traceback (most recent call last):
...
TypeError: the parameter l ('a') must be an integer
```

random_element(*length=None*, **args*, ***kwds*)

Returns a random finite word on the given alphabet.

INPUT:

- `length` – (optional) the length of the word. If not set, will use a uniformly random number between 0 and 10.
- all other argument are transmitted to the random generator of the alphabet

EXAMPLE:

```
sage: W = FiniteWords(5)
sage: W.random_element() # random
word: 5114325445423521544531411434451152142155...

sage: W = FiniteWords(ZZ)
sage: W.random_element() # random
word: 5,-1,-1,-1,0,0,0,0,-3,-11
sage: W.random_element(length=4, x=0, y=4) # random
word: 1003
```

TESTS:

```
sage: _ = FiniteWords(GF(5)).random_element()
```

shift()

Return the set of infinite words on the same alphabet.

EXAMPLES:

```
sage: FiniteWords('ab').shift()
Infinite words over {'a', 'b'}
```

class sage.combinat.words.words.**InfiniteWords**(*alphabet=None*, *category=None*)

Bases: sage.combinat.words.words.AbstractLanguage

INPUT:

- `alphabet` – the unerlying alphabet

TESTS:

```
sage: loads(dumps(FiniteWords('ab'))) == FiniteWords('ab')
True
sage: loads(dumps(InfiniteWords('ab'))) == InfiniteWords('ab')
True
```

```
sage: Words('abc').cmp_letters
<built-in function cmp>
sage: Words('bac').cmp_letters
<bound method FiniteOrInfiniteWords._cmp_letters ...>
```

cardinality()

Return the cardinality of this set

EXAMPLES:

```
sage: InfiniteWords('ab').cardinality()
+Infinity
sage: InfiniteWords('a').cardinality()
1
sage: InfiniteWords('').cardinality()
0
```

factors()

Return the set of finite words on the same alphabet.

EXAMPLES:

```
sage: InfiniteWords('ab').factors()
Finite words over {'a', 'b'}
```

random_element(**args*, ***kwds*)

Return a random infinite word.

EXAMPLES:

```
sage: W = InfiniteWords('ab')
sage: W.random_element() # random
word: abbbabbaabbbabbabbaabaabbabbbbbbbbbaabbbb...

sage: W = InfiniteWords(ZZ)
sage: W.random_element(x=2,y=4) # random
word: 33332233222322333332233232232222233233233...
```

shift()

Return itself.

EXAMPLES:

```
sage: InfiniteWords('ab').shift()
Infinite words over {'a', 'b'}
```

sage.combinat.words.words.**Words**(*alphabet=None*, *length=None*, *finite=True*, *infinite=True*)

Returns the combinatorial class of words of length k over an alphabet.

EXAMPLES:

```
sage: Words()
Finite and infinite words over Set of Python objects of type 'object'
sage: Words(length=7)
Words of length 7 over Set of Python objects of type 'object'
sage: Words(5)
Finite and infinite words over {1, 2, 3, 4, 5}
sage: Words(5, 3)
Words of length 3 over {1, 2, 3, 4, 5}
sage: Words(5, infinite=False)
Finite words over {1, 2, 3, 4, 5}
```

```
sage: Words(5, finite=False)
Infinite words over {1, 2, 3, 4, 5}
sage: Words('ab')
Finite and infinite words over {'a', 'b'}
sage: Words('ab', 2)
Words of length 2 over {'a', 'b'}
sage: Words('ab', infinite=False)
Finite words over {'a', 'b'}
sage: Words('ab', finite=False)
Infinite words over {'a', 'b'}
sage: Words('positive integers', finite=False)
Infinite words over Positive integers
sage: Words('natural numbers')
Finite and infinite words over Non negative integers
```

class sage.combinat.words.words.**Words_all**
Bases: sage.combinat.words.words.FiniteOrInfiniteWords

Deprecated class used for unpickle support only!

class sage.combinat.words.words.**Words_n**(*words*, *n*)
Bases: sage.structure.parent.Parent

The set of words of fixed length on a given alphabet.

alphabet()
Return the underlying alphabet.

EXAMPLES:

```
sage: Words([0,1], 4).alphabet()
{0, 1}
```

cardinality()
Returns the number of words of length n from alphabet.

EXAMPLES:

```
sage: Words(['a','b','c'], 4).cardinality()
81
sage: Words(3, 4).cardinality()
81
sage: Words(0,0).cardinality()
1
sage: Words(5,0).cardinality()
1
sage: Words(['a','b','c'],0).cardinality()
1
sage: Words(0,1).cardinality()
0
sage: Words(5,1).cardinality()
5
sage: Words(['a','b','c'],1).cardinality()
3
sage: Words(7,13).cardinality()
96889010407
sage: Words(['a','b','c','d','e','f','g'],13).cardinality()
96889010407
```

iterate_by_length(*length*)

All words in this class are of the same length, so use iterator instead.

TESTS:

```
sage: W = Words(['a', 'b'], 2)
sage: list(W.iterate_by_length(2))
[word: aa, word: ab, word: ba, word: bb]
sage: list(W.iterate_by_length(1))
[]
```

list()

Returns a list of all the words contained in self.

EXAMPLES:

```
sage: Words(0,0).list()
[word: ]
sage: Words(5,0).list()
[word: ]
sage: Words(['a','b','c'],0).list()
[word: ]
sage: Words(5,1).list()
[word: 1, word: 2, word: 3, word: 4, word: 5]
sage: Words(['a','b','c'],2).list()
[word: aa, word: ab, word: ac, word: ba, word: bb, word: bc, word: ca, word: cb, word: cc]
```

random_element(**args*, ***kwds*)

Return a random word in this set.

EXAMPLES:

```
sage: W = Words('ab', 4)
sage: W.random_element()  # random
word: bbab
sage: W.random_element() in W
True

sage: W = Words(ZZ, 5)
sage: W.random_element()  # random
word: 1,2,2,-1,12
sage: W.random_element() in W
True
```

TESTS:

```
sage: _ = Words(GF(5),4).random_element()
```

Check that trac ticket #18283 is fixed:

```
sage: w = Words('abc', 5).random_element()
sage: w.length()
5
```

5.1.335 Yang-Baxter Graphs

class sage.combinat.yang_baxter_graph.**SwapIncreasingOperator**(*i*)

Bases: sage.combinat.yang_baxter_graph.SwapOperator

The operator that swaps the items in positions `i` and `i+1`.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapOperator
sage: s3 = SwapOperator(3)
sage: s3 == loads(dumps(s3))
True
```

class sage.combinat.yang_baxter_graph.**SwapOperator**(*i*)

Bases: sage.structure.sage_object.SageObject

The operator that swaps the items in positions i and i+1.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapOperator
sage: s3 = SwapOperator(3)
sage: s3 == loads(dumps(s3))
True
```

position()

self is the operator that swaps positions i and i+1. This method returns i.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapOperator
sage: s3 = SwapOperator(3)
sage: s3.position()
3
```

sage.combinat.yang_baxter_graph.**YangBaxterGraph**(*partition=None*, *root=None*, *operators=None*)

Construct the Yang-Baxter graph from root by repeated application of operators, or the Yang-Baxter graph associated to partition.

INPUT:

The user needs to provide either partition or both root and operators, where

- partition – a partition of a positive integer
- root – the root vertex
- operator - a function that maps vertices u to a list of tuples of the form (v, l) where v is a successor of u and l is the label of the edge from u to v.

OUTPUT:

- Either:
  - YangBaxterGraph_partition - if partition is defined
  - YangBaxterGraph_generic - if partition is None

EXAMPLES:

The Yang-Baxter graph defined by a partition $[p_1, \ldots, p_k]$ is the labelled directed graph with vertex set obtained by bubble-sorting $(p_k - 1, p_k - 2, \ldots, 0, \ldots, p_1 - 1, p_1 - 2, \ldots, 0)$; there is an arrow from u to v labelled by i if v is obtained by swapping the i-th and $(i + 1)$-th elements of u. For example, if the partition is $[3, 1]$, then we begin with $(0, 2, 1, 0)$ and generate all tuples obtained from it by swapping two adjacent entries if they are increasing:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: bubbleswaps = [SwapIncreasingOperator(i) for i in range(3)]
sage: Y = YangBaxterGraph(root=(0,2,1,0), operators=bubbleswaps); Y
Yang-Baxter graph with root vertex (0, 2, 1, 0)
```

```
sage: Y.vertices()
[(2, 0, 1, 0), (2, 1, 0, 0), (0, 2, 1, 0)]
```

The `partition` keyword is a shorthand for the above construction.

```
sage: Y = YangBaxterGraph(partition=[3,1]); Y
Yang-Baxter graph of [3, 1], with top vertex (0, 2, 1, 0)
sage: Y.vertices()
[(0, 2, 1, 0), (2, 0, 1, 0), (2, 1, 0, 0)]
```

The permutahedron can be realized as a Yang-Baxter graph.

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: swappers = [SwapIncreasingOperator(i) for i in range(3)]
sage: Y = YangBaxterGraph(root=(1,2,3,4), operators=swappers); Y
Yang-Baxter graph with root vertex (1, 2, 3, 4)
sage: Y.plot()
Graphics object consisting of 97 graphics primitives
```

The Cayley graph of a finite group can be realized as a Yang-Baxter graph.

```
sage: def left_multiplication_by(g):
...       return lambda h : h*g
sage: G = CyclicPermutationGroup(4)
sage: operators = [ left_multiplication_by(gen) for gen in G.gens() ]
sage: Y = YangBaxterGraph(root=G.identity(), operators=operators); Y
Yang-Baxter graph with root vertex ()
sage: Y.plot(edge_labels=False)
Graphics object consisting of 9 graphics primitives

sage: G = SymmetricGroup(4)
sage: operators = [left_multiplication_by(gen) for gen in G.gens()]
sage: Y = YangBaxterGraph(root=G.identity(), operators=operators); Y
Yang-Baxter graph with root vertex ()
sage: Y.plot(edge_labels=False)
Graphics object consisting of 96 graphics primitives
```

AUTHORS:

- Franco Saliola (2009-04-23)

class sage.combinat.yang_baxter_graph.**YangBaxterGraph_generic**(*root*, *operators*)

Bases: sage.structure.sage_object.SageObject

A class to model the Yang-Baxter graph defined by `root` and `operators`.

INPUT:

- `root` – the root vertex of the graph
- `operators` – a list of callables that map vertices to (new) vertices.

Note: This is a lazy implementation: the digraph is only computed when it is needed.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(4)]
sage: Y = YangBaxterGraph(root=(1,0,2,1,0), operators=ops); Y
Yang-Baxter graph with root vertex (1, 0, 2, 1, 0)
```

```
sage: loads(dumps(Y)) == Y
True
```

AUTHORS:

- Franco Saliola (2009-04-23)

edges()

Returns the (labelled) edges of `self`.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(3)]
sage: Y = YangBaxterGraph(root=(0,2,1,0), operators=ops)
sage: Y.edges()
[((0, 2, 1, 0), (2, 0, 1, 0), Swap-if-increasing at position 0), ((2, 0, 1, 0), (2, 1, 0, 0)
```

plot(**args*, ***kwds*)

Plots `self` as a digraph.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(4)]
sage: Y = YangBaxterGraph(root=(1,0,2,1,0), operators=ops)
sage: Y.plot()
Graphics object consisting of 16 graphics primitives
sage: Y.plot(edge_labels=False)
Graphics object consisting of 11 graphics primitives
```

relabel_edges(*edge_dict*, *inplace=True*)

Relabel the edges of `self`.

INPUT:

- `edge_dict` – a dictionary keyed by the (unlabelled) edges.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(3)]
sage: Y = YangBaxterGraph(root=(0,2,1,0), operators=ops)
sage: def relabel_op(op, u):
...       i = op.position()
...       return u[:i] + u[i:i+2][::-1] + u[i+2:]
sage: Y.edges()
[((0, 2, 1, 0), (2, 0, 1, 0), Swap-if-increasing at position 0), ((2, 0, 1, 0), (2, 1, 0, 0)
sage: d = {((0,2,1,0),(2,0,1,0)):17, ((2,0,1,0),(2,1,0,0)):27}
sage: Y.relabel_edges(d, inplace=False).edges()
[((0, 2, 1, 0), (2, 0, 1, 0), 17), ((2, 0, 1, 0), (2, 1, 0, 0), 27)]
sage: Y.edges()
[((0, 2, 1, 0), (2, 0, 1, 0), Swap-if-increasing at position 0), ((2, 0, 1, 0), (2, 1, 0, 0)
sage: Y.relabel_edges(d, inplace=True)
sage: Y.edges()
[((0, 2, 1, 0), (2, 0, 1, 0), 17), ((2, 0, 1, 0), (2, 1, 0, 0), 27)]
```

relabel_vertices(*v*, *relabel_operator*, *inplace=True*)

Relabel the vertices `u` of `self` by the object obtained from `u` by applying the `relabel_operator` to `v` along a path from `self.root()` to `u`.

Note that the `self.root()` is paired with `v`.

INPUT:

- `v` – tuple, Permutation, CombinatorialObject
- `inplace` – if `True`, modifies `self`; otherwise returns a modified copy of `self`.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(3)]
sage: Y = YangBaxterGraph(root=(0,2,1,0), operators=ops)
sage: def relabel_op(op, u):
...       i = op.position()
...       return u[:i] + u[i:i+2][::-1] + u[i+2:]
sage: d = Y.relabel_vertices((1,2,3,4), relabel_op, inplace=False); d
Yang-Baxter graph with root vertex (1, 2, 3, 4)
sage: Y.vertices()
[(2, 0, 1, 0), (2, 1, 0, 0), (0, 2, 1, 0)]
sage: e = Y.relabel_vertices((1,2,3,4), relabel_op); e
sage: Y.vertices()
[(2, 1, 3, 4), (1, 2, 3, 4), (2, 3, 1, 4)]
```

root()

Returns the root vertex of `self`.

If `self` is the Yang-Baxter graph of the partition $[p_1, p_2, \ldots, p_k]$, then this is the vertex $(p_k - 1, p_k - 2, \ldots, 0, \ldots, p_1 - 1, p_1 - 2, \ldots, 0)$.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(4)]
sage: Y = YangBaxterGraph(root=(1,0,2,1,0), operators=ops)
sage: Y.root()
(1, 0, 2, 1, 0)
sage: Y = YangBaxterGraph(root=(0,1,0,2,1,0), operators=ops)
sage: Y.root()
(0, 1, 0, 2, 1, 0)
sage: Y = YangBaxterGraph(root=(1,0,3,2,1,0), operators=ops)
sage: Y.root()
(1, 0, 3, 2, 1, 0)
sage: Y = YangBaxterGraph(partition=[3,2])
sage: Y.root()
(1, 0, 2, 1, 0)
```

successors(*v*)

Return the successors of the vertex `v`.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(4)]
sage: Y = YangBaxterGraph(root=(1,0,2,1,0), operators=ops)
sage: Y.successors(Y.root())
[(1, 2, 0, 1, 0)]
sage: Y.successors((1, 2, 0, 1, 0))
[(1, 2, 1, 0, 0), (2, 1, 0, 1, 0)]
```

vertex_relabelling_dict(*v*, *relabel_operator*)

Return a dictionary pairing vertices `u` of `self` with the object obtained from `v` by applying the

relabel_operator along a path from the root to u. Note that the root is paired with v.

INPUT:

- v – an object
- relabel_operator – function mapping a vertex and a label to the image of the vertex

OUTPUT:

- dictionary pairing vertices with the corresponding image of v

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(3)]
sage: Y = YangBaxterGraph(root=(0,2,1,0), operators=ops)
sage: def relabel_operator(op, u):
...       i = op.position()
...       return u[:i] + u[i:i+2][::-1] + u[i+2:]
sage: Y.vertex_relabelling_dict((1,2,3,4), relabel_operator)
{(0, 2, 1, 0): (1, 2, 3, 4),
 (2, 0, 1, 0): (2, 1, 3, 4),
 (2, 1, 0, 0): (2, 3, 1, 4)}
```

vertices()

Returns the vertices of self.

EXAMPLES:

```
sage: from sage.combinat.yang_baxter_graph import SwapIncreasingOperator
sage: ops = [SwapIncreasingOperator(i) for i in range(3)]
sage: Y = YangBaxterGraph(root=(0,2,1,0), operators=ops)
sage: Y.vertices()
[(2, 0, 1, 0), (2, 1, 0, 0), (0, 2, 1, 0)]
```

class sage.combinat.yang_baxter_graph.**YangBaxterGraph_partition**(*partition*)

Bases: sage.combinat.yang_baxter_graph.YangBaxterGraph_generic

A class to model the Yang-Baxter graph of a partition.

The Yang-Baxter graph defined by a partition $[p_1, \ldots, p_k]$ is the labelled directed graph with vertex set obtained by bubble-sorting $(p_k - 1, p_k - 2, \ldots, 0, \ldots, p_1 - 1, p_1 - 2, \ldots, 0)$; there is an arrow from u to v labelled by i if v is obtained by swapping the i-th and $(i + 1)$-th elements of u.

Note: This is a lazy implementation: the digraph is only computed when it is needed.

EXAMPLES:

```
sage: Y = YangBaxterGraph(partition=[3,2,1]); Y
Yang-Baxter graph of [3, 2, 1], with top vertex (0, 1, 0, 2, 1, 0)
sage: loads(dumps(Y)) == Y
True
```

AUTHORS:

- Franco Saliola (2009-04-23)

relabel_vertices(*v*, *inplace=True*)

Relabel the vertices of self with the object obtained from v by applying the transpositions corresponding to the edge labels along some path from the root to the vertex.

INPUT:

- v – tuple, Permutation, CombinatorialObject
- inplace – if True, modifies self; otherwise returns a modified copy of self.

EXAMPLES:

```
sage: Y = YangBaxterGraph(partition=[3,1]); Y
Yang-Baxter graph of [3, 1], with top vertex (0, 2, 1, 0)
sage: d = Y.relabel_vertices((1,2,3,4), inplace=False); d
Digraph on 3 vertices
sage: Y.vertices()
[(0, 2, 1, 0), (2, 0, 1, 0), (2, 1, 0, 0)]
sage: e = Y.relabel_vertices((1,2,3,4)); e
sage: Y.vertices()
[(1, 2, 3, 4), (2, 1, 3, 4), (2, 3, 1, 4)]
```

vertex_relabelling_dict(*v*)

Return a dictionary pairing vertices u of self with the object obtained from v by applying transpositions corresponding to the edges labels along a path from the root to u.

Note that the root is paired with v.

INPUT:

- v – an object

OUTPUT:

- dictionary pairing vertices with the corresponding image of v

EXAMPLES:

```
sage: Y = YangBaxterGraph(partition=[3,1])
sage: Y.vertex_relabelling_dict((1,2,3,4))
{(0, 2, 1, 0): (1, 2, 3, 4),
 (2, 0, 1, 0): (2, 1, 3, 4),
 (2, 1, 0, 0): (2, 3, 1, 4)}
sage: Y.vertex_relabelling_dict((4,3,2,1))
{(0, 2, 1, 0): (4, 3, 2, 1),
 (2, 0, 1, 0): (3, 4, 2, 1),
 (2, 1, 0, 0): (3, 2, 4, 1)}
```

5.1.336 C-Finite Sequences

C-finite infinite sequences satisfy homogenous linear recurrences with constant coefficients:

$$a_{n+d} = c_0 a_n + c_1 a_{n+1} + \cdots + c_{d-1} a_{n+d-1}, \quad d > 0.$$

CFiniteSequences are completely defined by their ordinary generating function (o.g.f., which is always a fraction of polynomials over $\mathbb{Z}$ or $\mathbb{Q}$).

EXAMPLES:

```
sage: fibo = CFiniteSequence(x/(1-x-x^2))        # the Fibonacci sequence
sage: fibo
C-finite sequence, generated by x/(-x^2 - x + 1)
sage: fibo.parent()
The ring of C-Finite sequences in x over Rational Field
sage: fibo.parent().category()
Category of commutative rings
sage: C.<x> = CFiniteSequences(QQ);
```

```
sage: fibo.parent() == C
True
sage: C
The ring of C-Finite sequences in x over Rational Field
sage: C(x/(1-x-x^2))
C-finite sequence, generated by x/(-x^2 - x + 1)
sage: C(x/(1-x-x^2)) == fibo
True
sage: var('y')
y
sage: CFiniteSequence(y/(1-y-y^2))
C-finite sequence, generated by y/(-y^2 - y + 1)
sage: CFiniteSequence(y/(1-y-y^2)) == fibo
False
```

Finite subsets of the sequence are accessible via python slices:

```
sage: fibo[137]                 #the 137th term of the Fibonacci sequence
19134702400093278081449423917
sage: fibo[137] == fibonacci(137)
True
sage: fibo[0:12]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
sage: fibo[14:4:-2]
[377, 144, 55, 21, 8]
```

They can be created also from the coefficients and start values of a recurrence:

```
sage: r = C.from_recurrence([1,1],[0,1])
sage: r == fibo
True
```

Given enough values, the o.g.f. of a C-finite sequence can be guessed:

```
sage: r = C.guess([0,1,1,2,3,5,8])
sage: r == fibo
True
```

See also:

`fibonacci()`, `BinaryRecurrenceSequence`

AUTHORS:

- Ralf Stephan (2014): initial version

REFERENCES:

class `sage.rings.cfinite_sequence.`**`CFiniteSequence`**(*parent*, *ogf*)

Bases: `sage.structure.element.FieldElement`

Create a C-finite sequence given its ordinary generating function.

INPUT:

- `ogf` – a rational function, the ordinary generating function (can be a an element from the symbolic ring, fraction field or polynomial ring)

OUTPUT:

- A CFiniteSequence object

EXAMPLES:

```
sage: CFiniteSequence((2-x)/(1-x-x^2))      # the Lucas sequence
C-finite sequence, generated by (-x + 2)/(-x^2 - x + 1)
sage: CFiniteSequence(x/(1-x)^3)            # triangular numbers
C-finite sequence, generated by x/(-x^3 + 3*x^2 - 3*x + 1)
```

Polynomials are interpreted as finite sequences, or recurrences of degree 0:

```
sage: CFiniteSequence(x^2-4*x^5)
Finite sequence [1, 0, 0, -4], offset = 2
sage: CFiniteSequence(1)
Finite sequence [1], offset = 0
```

This implementation allows any polynomial fraction as o.g.f. by interpreting any power of x dividing the o.g.f. numerator or denominator as a right or left shift of the sequence offset:

```
sage: CFiniteSequence(x^2+3/x)
Finite sequence [3, 0, 0, 1], offset = -1
sage: CFiniteSequence(1/x+4/x^3)
Finite sequence [4, 0, 1], offset = -3
sage: P = LaurentPolynomialRing(QQ.fraction_field(), 'X')
sage: X=P.gen()
sage: CFiniteSequence(1/(1-X))
C-finite sequence, generated by 1/(-X + 1)
```

The o.g.f. is always normalized to get a denominator constant coefficient of $+1$:

```
sage: CFiniteSequence(1/(x-2))
C-finite sequence, generated by -1/2/(-1/2*x + 1)
```

The given `ogf` is used to create an appropriate parent: it can be a symbolic expression, a polynomial , or a fraction field element as long as it can be coerced into a proper fraction field over the rationals:

```
sage: var('x')
x
sage: f1 = CFiniteSequence((2-x)/(1-x-x^2))
sage: P.<x> = QQ[]
sage: f2 = CFiniteSequence((2-x)/(1-x-x^2))
sage: f1 == f2
True
sage: f1.parent()
The ring of C-Finite sequences in x over Rational Field
sage: f1.ogf().parent()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: CFiniteSequence(log(x))
Traceback (most recent call last):
...
TypeError: unable to convert log(x) to a rational
```

TESTS:

```
sage: P.<x> = QQ[]
sage: CFiniteSequence(0.1/(1-x))
C-finite sequence, generated by 1/10/(-x + 1)
sage: CFiniteSequence(pi/(1-x))
Traceback (most recent call last):
...
TypeError: unable to convert -pi to a rational
sage: P.<x,y> = QQ[]
sage: CFiniteSequence(x*y)
```

```
Traceback (most recent call last):
...
NotImplementedError: Multidimensional o.g.f. not implemented.
```

coefficients()
Return the coefficients of the recurrence representation of the C-finite sequence.

OUTPUT:

- A list of values

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: lucas = C((2-x)/(1-x-x^2))    # the Lucas sequence
sage: lucas.coefficients()
[1, 1]
```

denominator()
Return the numerator of the o.g.f of `self`.

EXAMPLES:

```
sage: f = CFiniteSequence((2-x)/(1-x-x^2)); f
C-finite sequence, generated by (-x + 2)/(-x^2 - x + 1)
sage: f.denominator()
-x^2 - x + 1
```

numerator()
Return the numerator of the o.g.f of `self`.

EXAMPLES:

```
sage: f = CFiniteSequence((2-x)/(1-x-x^2)); f
C-finite sequence, generated by (-x + 2)/(-x^2 - x + 1)
sage: f.numerator()
-x + 2
```

ogf()
Return the ordinary generating function associated with the CFiniteSequence.

This is always a fraction of polynomials in the base ring.

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: r = C.from_recurrence([2],[1])
sage: r.ogf()
1/(-2*x + 1)
sage: C(0).ogf()
0
```

recurrence_repr()
Return a string with the recurrence representation of the C-finite sequence.

OUTPUT:

- A string

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: C((2-x)/(1-x-x^2)).recurrence_repr()
```

```
'Homogenous linear recurrence with constant coefficients of degree 2: a(n+2) = a(n+1) + a(n)
sage: C(x/(1-x)^3).recurrence_repr()
'Homogenous linear recurrence with constant coefficients of degree 3: a(n+3) = 3*a(n+2) - 3*
sage: C(1).recurrence_repr()
'Finite sequence [1], offset 0'
sage: r = C((-2*x^3 + x^2 - x + 1)/(2*x^2 - 3*x + 1))
sage: r.recurrence_repr()
'Homogenous linear recurrence with constant coefficients of degree 2: a(n+2) = 3*a(n+1) - 2*
sage: r = CFiniteSequence(x^3/(1-x-x^2))
sage: r.recurrence_repr()
'Homogenous linear recurrence with constant coefficients of degree 2: a(n+2) = a(n+1) + a(n)
```

series(*n*)

Return the Laurent power series associated with the CFiniteSequence, with precision n.

INPUT:

- n – a nonnegative integer

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: r = C.from_recurrence([-1,2],[0,1])
sage: s = r.series(4); s
x + 2*x^2 + 3*x^3 + 4*x^4 + O(x^5)
sage: type(s)
<type 'sage.rings.laurent_series_ring_element.LaurentSeries'>
```

sage.rings.cfinite_sequence.**CFiniteSequences**(*base_ring*, *names=None*, *category=None*)

Return the ring of C-Finite sequences.

The ring is defined over a base ring ($\mathbb{Z}$ or $\mathbb{Q}$) and each element is represented by its ordinary generating function (ogf) which is a rational function over the base ring.

INPUT:

- `base_ring` – the base ring to construct the fraction field representing the C-Finite sequences
- `names` – (optional) the list of variables.

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: C
The ring of C-Finite sequences in x over Rational Field
sage: C.an_element()
C-finite sequence, generated by (-x + 2)/(-x^2 - x + 1)
sage: C.category()
Category of commutative rings
sage: C.one()
Finite sequence [1], offset = 0
sage: C.zero()
Constant infinite sequence 0.
sage: C(x)
Finite sequence [1], offset = 1
sage: C(1/x)
Finite sequence [1], offset = -1
sage: C((-x + 2)/(-x^2 - x + 1))
C-finite sequence, generated by (-x + 2)/(-x^2 - x + 1)
```

TESTS:

```
sage: TestSuite(C).run()
```

class sage.rings.cfinite_sequence.**CFiniteSequences_generic**(*polynomial_ring*, *category*)

Bases: sage.rings.ring.CommutativeRing, sage.structure.unique_representation.UniqueRepresentation

The class representing the ring of C-Finite Sequences

TESTS:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: from sage.rings.cfinite_sequence import CFiniteSequences_generic
sage: isinstance(C,CFiniteSequences_generic)
True
sage: type(C)
<class 'sage.rings.cfinite_sequence.CFiniteSequences_generic_with_category'>
sage: C
The ring of C-Finite sequences in x over Rational Field
```

Element

alias of CFiniteSequence

an_element()

Return an element of C-Finite Sequences.

OUTPUT:

The Lucas sequence.

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ);
sage: C.an_element()
C-finite sequence, generated by (-x + 2)/(-x^2 - x + 1)
```

fraction_field()

Return the faction field used to represent the elements of `self`.

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ);
sage: C.fraction_field()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

from_recurrence(*coefficients*, *values*)

Create a C-finite sequence given the coefficients c and starting values a of a homogenous linear recurrence.

$$a_{n+d} = c_0 a_n + c_1 a_{n+1} + \cdots + c_{d-1} a_{n+d-1}, \quad d \geq 0.$$

INPUT:

- `coefficients` – a list of rationals
- `values` – start values, a list of rationals

OUTPUT:

- A CFiniteSequence object

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: C.from_recurrence([1,1],[0,1])    # Fibonacci numbers
C-finite sequence, generated by x/(-x^2 - x + 1)
```

```
sage: C.from_recurrence([-1,2],[0,1])     # natural numbers
C-finite sequence, generated by x/(x^2 - 2*x + 1)
sage: r = C.from_recurrence([-1],[1])
sage: s = C.from_recurrence([-1],[1,-1])
sage: r == s
True
sage: r = C(x^3/(1-x-x^2))
sage: s = C.from_recurrence([1,1],[0,0,0,1,1])
sage: r == s
True
sage: C.from_recurrence(1,1)
Traceback (most recent call last):
...
ValueError: Wrong type for recurrence coefficient list.
```

gen (*i=0*)

Return the i-th generator of `self`.

INPUT:

- `i` – an integer (default:0)

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ);
sage: C.gen()
x
sage: x == C.gen()
True
```

TESTS:

```
sage: C.gen(2)
Traceback (most recent call last):
...
ValueError: The ring of C-Finite sequences in x over Rational Field has only one generator (
```

guess (*sequence*, *algorithm='sage'*)

Return the minimal CFiniteSequence that generates the sequence.

Assume the first value has index 0.

INPUT:

- `sequence` – list of integers
- **`algorithm` – string**
  - 'sage' - the default is to use Sage's matrix kernel function
  - 'pari' - use Pari's implementation of LLL
  - 'bm' - use Sage's Berlekamp-Massey algorithm

OUTPUT:

- a CFiniteSequence, or 0 if none could be found

With the default kernel method, trailing zeroes are chopped off before a guessing attempt. This may reduce the data below the accepted length of six values.

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ)
sage: C.guess([1,2,4,8,16,32])
C-finite sequence, generated by 1/(-2*x + 1)
sage: r = C.guess([1,2,3,4,5])
Traceback (most recent call last):
...
ValueError: Sequence too short for guessing.
```

With Berlekamp-Massey, if an odd number of values is given, the last one is dropped. So with an odd number of values the result may not generate the last value:

```
sage: r = C.guess([1,2,4,8,9], algorithm='bm'); r
C-finite sequence, generated by 1/(-2*x + 1)
sage: r[0:5]
[1, 2, 4, 8, 16]
```

ngens()

Return the number of generators of `self`

EXAMPLES:

```
sage: from sage.rings.cfinite_sequence import CFiniteSequences
sage: C.<x> = CFiniteSequences(QQ);
sage: C.ngens()
1
```

polynomial_ring()

Return the polynomial ring used to represent the elements of `self`.

EXAMPLES:

```
sage: C.<x> = CFiniteSequences(QQ);
sage: C.polynomial_ring()
Univariate Polynomial Ring in x over Rational Field
```

CHAPTER

SIX

INDICES AND TABLES

- Index
- Module Index
- Search Page

[BjBr] Bjorner and Brenti. Combinatorics of Coxeter Groups. Springer, 2005.

[Erik] H. Erikson. Computational and Combinatorial Aspects of Coxeter Groups. Thesis, 1995.

[D2012] tom denton. Canonical Decompositions of Affine Permutations, Affine Codes, and Split k-Schur Functions. Electronic Journal of Combinatorics, 2012.

[MiRoRu] W. H. Mills, David P Robbins, Howard Rumsey Jr., *Alternating sign matrices and descending plane partitions*, Journal of Combinatorial Theory, Series A, Volume 34, Issue 3, May 1983, Pages 340–359. http://www.sciencedirect.com/science/article/pii/0097316583900687

[EKLP92] N. Elkies, G. Kuperberg, M. Larsen, J. Propp, *Alternating-Sign Matrices and Domino Tilings*, Journal of Algebraic Combinatorics, volume 1 (1992), p. 111-132.

[Aval07] J.-C. Aval. *Keys and alternating sign matrices*. Sem. Lothar. Combin. 59 (2007/10), Art. B59f, 13 pp.

[LascouxPreprint] A. Lascoux. *Chern and Yang through ice*. Preprint.

[Gir12] Samuele Giraudo, *Algebraic and combinatorial structures on pairs of twin binary trees*, Arxiv 1204.4776v1.

[SV13] Silliman and Vogt. "Powers in Lucas Sequences via Galois Representations." Proceedings of the American Mathematical Society, 2013. Arxiv 1307.5078v2

[BMS06] Bugeaud, Mignotte, and Siksek. "Classical and modular approaches to exponential Diophantine equations: I. Fibonacci and Lucas perfect powers." Annals of Math, 2006.

[SS] Shorey and Stewart. "On the Diophantine equation a x^{2t} + b x^t y + c y^2 = d and pure powers in recurrence sequences." Mathematica Scandinavica, 1983.

[LodayRonco] Jean-Louis Loday and Maria O. Ronco. *Hopf algebra of the planar binary trees*, Advances in Mathematics, volume 139, issue 2, 10 November 1998, pp. 293-309. http://www.sciencedirect.com/science/article/pii/S0001870898917595

[HNT05] Florent Hivert, Jean-Christophe Novelli, and Jean-Yves Thibon. *The algebra of binary search trees*, Arxiv math/0401089v2.

[CP12] Gregory Chatel, Viviane Pons. *Counting smaller trees in the Tamari order*, Arxiv 1212.0751v1.

[DG94] S. Dulucq and O. Guibert. Mots de piles, tableaux standards et permutations de Baxter, proceedings of Formal Power Series and Algebraic Combinatorics, 1994.

[BW88] Anders Bjoerner, Michelle L. Wachs, *Generalized quotients in Coxeter groups*. Transactions of the American Mathematical Society, vol. 308, no. 1, July 1988. http://www.ams.org/journals/tran/1988-308-01/S0002-9947-1988-0946427-X/S0002-9947-1988-0946427-X.pdf

[Read04] Nathan Reading. *Cambrian Lattices*. Arxiv math/0402086v2.

[BDP2013] Thomas Brüstle, Grégoire Dupont, Matthieu Pérotin *On Maximal Green Sequences* Arxiv 1205.2050

[FZ2007] Sergey Fomin, Andrei Zelevinsky "Cluster Algebras IV: coefficients" Arxiv 0602259

[LeeLiZe] Lee-Li-Zelevinsky, Greedy elements in rank 2 cluster algebras, Arxiv 1208.2391

[CLS] C. Ceballos, J.-P. Labbe, C. Stump, *Subword complexes, cluster complexes, and generalized multi-associahedra*, J. Algebr. Comb. **39** (2014) pp. 17-51. doi:10.1007/s10801-013-0437-x, Arxiv 1108.1776.

[MS14] Jennifer Morse and Anne Schilling. *Crystal approach to affine Schubert calculus*. Int. Math. Res. Not. (2015). doi:10.1093/imrn/rnv194, Arxiv 1408.0320.

[LP2008] C. Lenart and A. Postnikov. *A combinatorial model for crystals of Kac-Moody algebras*. Trans. Amer. Math. Soc. 360 (2008), 4349-4381.

[St2003] J. Stembridge, *A local characterization of simply-laced crystals*, Trans. Amer. Math. Soc. 355 (2003), no. 12, 4807-4823.

[Kashiwara93] M. Kashiwara. The Crystal Base and Littelmann's Refined Demazure Character Formula. Duke Math. J. **71** (3), pp. 839–858, 1993.

[NZ97] T. Nakashima and A. Zelevinsky. Polyhedral Realizations of Crystal Bases for Quantized Kac-Moody Algebras. Adv. Math. **131**, pp. 253–278, 1997.

[KS10] J.-A. Kim and D.-U. Shin. Generalized Young walls and crystal bases for quantum affine algebra of type A. Proc. Amer. Math. Soc. 138(11), pp. 3877–3889, 2010.

[KLRS] S.-J. Kang, K.-H. Lee, H. Ryu, and B. Salisbury. A combinatorial description of the affine Gindikin-Karpelevich formula of type $A_n^{(1)}$. Arxiv 1203.1640.

[BN10] D. Bump and M. Nakasuji. Integration on p-adic groups and crystal bases. Proc. Amer. Math. Soc. 138(5), pp. 1595–1605.

[LS12] K.-H. Lee and B. Salisbury. Young tableaux, canonical bases, and the Gindikin-Karpelevich formula. Arxiv 1205.6006.

[HL08] J. Hong and H. Lee. Young tableaux and crystal $B(\infty)$ for finite simple Lie algebras. J. Algebra 320, pp. 3680–3693, 2008.

[KM94] S.-J. Kang and K. C. Misra. Crystal bases and tensor product decompositions of $U_q(G_2)$-modules. J. Algebra 163, pp. 675–691, 1994.

[K93] M. Kashiwara. *The crystal base and Littelmann's refined Demazure character formula*. Duke Math. J. **71**. 1993.

[KMOY07] M. Kashiwara, K. C. Misra, M. Okado, D. Yamada. *Perfect crystals for* $U_q(D_4^{(3)})$, J. Algebra. **317** (2007).

[Shimozono02] M. Shimozono *Affine type A crystal structure on tensor products of rectangles, Demazure characters, and nilpotent varieties*, J. Algebraic Combin. **15** (2002). no. 2. 151-187. Arxiv math.QA/9804039.

[Schilling08] A. Schilling. "Combinatorial structure of Kirillov-Reshetikhin crystals of type $D_n(1)$, $B_n(1)$, $A_{2n-1}(2)$". J. Algebra. **319** (2008). 2938-2962. Arxiv 0704.2046.

[JS2010] B. Jones, A. Schilling. "Affine structures and a tableau model for E_6 crystals", J. Algebra. **324** (2010). 2512-2542. doi:10.1016/j.bbr.2011.03.031, Arxiv 0909.2442.

[FOS09] G. Fourier, M. Okado, A. Schilling. *Kirillov-Reshetikhin crystals for nonexceptional types*. Advances in Mathematics. **222** (2009). Issue 3. 1080-1116. Arxiv 0810.5067.

[LOS12] C. Lecouvey, M. Okado, M. Shimozono. "Affine crystals, one-dimensional sums and parabolic Lusztig q-analogues". Mathematische Zeitschrift. **271** (2012). Issue 3-4. 819-865. doi:10.1007/s00209-011-0892-9, Arxiv 1002.3715.

[FOS2010] G. Fourier, M. Okado, A. Schilling. Perfectness of Kirillov-Reshetikhin crystals for nonexceptional types Contemp. Math. 506 (2010) 127-143 (arXiv:0811.1604 [math.RT])

[HK02] *Introduction to Quantum Groups and Crystal Bases.* Jin Hong and Seok-Jin Kang. 2002. Volume 42. Graduate Studies in Mathematics. American Mathematical Society.

[KN94] M. Kashiwara and T. Nakashima. Crystal graphs for representations of the q-analogue of classical Lie algebras. J. Algebra **165**, no. 2, pp. 295–345, 1994.

[Littelmann95] P. Littelmann, Paths and root operators in representation theory. Ann. of Math. (2) 142 (1995), no. 3, 499-525.

[LNSSS2013] C. Lenart, S. Naito, D. Sagaki, A. Schilling, M. Shimozono, A uniform model for Kirillov-Reshetikhin crystals. Extended abstract. DMTCS proc, to appear ({{{Arxiv 1211.6019}}})

[LZ11] Bin Li and Hechun Zhang. *Path realization of crystal* $B(\infty)$. Front. Math. China, **6** (4), (2011) pp. 689–706. doi:10.1007/s11464-010-0073-x

[KKS07] S.-J. Kang, J.-A. Kim, and D.-U. Shin. Modified Nakajima Monomials and the Crystal $B(\infty)$. J. Algebra **308**, pp. 524–535, 2007.

[Kash03] M. Kashiwara. Realizations of Crystals. Combinatorial and geometric representation theory (Seoul, 2001), Contemp. Math. **325**, Amer. Math. Soc., pp. 133–139, 2003.

[Kash95] M. Kashiwara. The crystal base and Littelmann's refined Demazure character formula. Duke Math. J. **71** (1993), no. 3, 839-858.

[SchillingTingley2011] A. Schilling, P. Tingley. Demazure crystals, Kirillov-Reshetikhin crystals, and the energy function. Electronic Journal of Combinatorics. **19(2)**. 2012. Arxiv 1104.2359

[RCES] Alley CATs in search of good homes Ruskey, R. Cohen, P. Eades, A. Scott Congressus numerantium, 1994 Pages 97–110

[DerUB] http://www.u-bourgogne.fr/LE2I/jl.baril/derange.pdf

[Martinez08] http://www.siam.org/proceedings/analco/2008/anl08_022martinezc.pdf

[GR1989] C. Reutenauer, A. M. Garsia. *A decomposition of Solomon's descent algebra.* Adv. Math. **77** (1989). http://www.lacim.uqam.ca/~christo/Publi%C3%A9s/1989/Decomposition%20Solomon.pdf

[Atkinson] M. D. Atkinson. *Solomon's descent algebra revisited.* Bull. London Math. Soc. 24 (1992) 545-551. http://www.cs.otago.ac.nz/staffpriv/mike/Papers/Descent/DescAlgRevisited.pdf

[MR-Desc] C. Malvenuto, C. Reutenauer, *Duality between quasi-symmetric functions and the Solomon descent algebra*, Journal of Algebra 177 (1995), no. 3, 967-982. http://www.lacim.uqam.ca/~christo/Publi%C3%A9s/1995/Duality.pdf

[Schocker2004] Manfred Schocker, *The descent algebra of the symmetric group*. Fields Inst. Comm. 40 (2004), pp. 145-161. http://www.mathematik.uni-bielefeld.de/~ringel/schocker-neu.ps

[ClaytonSmith] On the existence of $(v, 5, 1)$-BIBD. http://www.argilo.net/files/bibd.pdf Clayton Smith

[Denniston69] R. H. F. Denniston, Some maximal arcs in finite projective planes. Journal of Combinatorial Theory 6, no. 3 (1969): 317-319. http://dx.doi.org/10.1016/S0021-9800(69)80095-5

[AndHonk97] A short course in Combinatorial Designs, Ian Anderson, Iiro Honkala, Internet Editions, Spring 1997, http://www.utu.fi/~honkala/designs.ps

[Stinson91] D.R. Stinson, A survey of Kirkman triple systems and related designs, Volume 92, Issues 1-3, 17 November 1991, Pages 371-393, Discrete Mathematics, http://dx.doi.org/10.1016/0012-365X(91)90294-C.

[RCW71] D. K. Ray-Chaudhuri, R. M. Wilson, Solution of Kirkman's schoolgirl problem, Volume 19, Pages 187-203, Proceedings of Symposia in Pure Mathematics

[BJL99] T. Beth, D. Jungnickel, H. Lenz, Design Theory 2ed. Cambridge University Press 1999

[Hu57] Daniel R. Hughes, "A class of non-Desarguesian projective planes", The Canadian Journal of Mathematics (1957), http://cms.math.ca/cjm/v9/p378

[We07] Charles Weibel, "Survey of Non-Desarguesian planes" (2007), notices of the AMS, vol. 54 num. 10, pages 1294–1303

[CvL] P. Cameron, J. H. van Lint, Designs, graphs, codes and their links, London Math. Soc., 1991.

[DesignHandbook] Handbook of Combinatorial Designs (2ed) Charles Colbourn, Jeffrey Dinitz Chapman & Hall/CRC 2012

[HT95] W. Huffman and V. Tonchev, The existence of extremal self-dual $[50, 25, 10]$ codes and quasi-symmetric $2 - (49, 9, 6)$ designs, Designs, Codes and Cryptography September 1995, Volume 6, Issue 2, pp 97-106

[Hanani75] Haim Hanani, Balanced incomplete block designs and related designs, http://dx.doi.org/10.1016/0012-365X(75)90040-0, Discrete Mathematics, Volume 11, Issue 3, 1975, Pages 255-369.

[JulianAbel13] Existence of Five MOLS of Orders 18 and 60 R. Julian R. Abel Journal of Combinatorial Designs 2013

[KY04] S. Klee and L. Yates, Tight Subdesigns of the Higman-Sims Design, Rose-Hulman Undergraduate Math. J 5.2 (2004). https://www.rose-hulman.edu/mathjournal/archives/2004/vol5-n2/paper9/v5n2-9pd.pdf

[Todorov12] D.T. Todorov, Four mutually orthogonal Latin squares of order 14, Journal of Combinatorial Designs 2012, vol.20 n.8 pp.363-367

[BJL99-1] T. Beth, D. Jungnickel, H. Lenz "Design theory Vol. I." Second edition. Encyclopedia of Mathematics and its Applications, 69. Cambridge University Press, (1999).

[BLJ99-2] T. Beth, D. Jungnickel, H. Lenz "Design theory Vol. II." Second edition. Encyclopedia of Mathematics and its Applications, 78. Cambridge University Press, (1999).

[Bo39] R. C. Bose, "On the construction of balanced incomplete block designs", Ann. Eugenics, vol. 9, (1939), 353–399.

[Bu95] M. Buratti "On simple radical difference families", J. of Combinatorial Designs, vol. 3, no. 2 (1995)

[Wi72] R. M. Wilson "Cyclotomy and difference families in elementary Abelian groups", J. of Num. Th., 4 (1972), pp. 17-47.

[McF1973] Robert L. McFarland "A family of difference sets in non-cyclic groups" Journal of Combinatorial Theory (A) vol 15 (1973). http://dx.doi.org/10.1016/0097-3165(73)90031-9

[D2009] P. Dobcsanyi et al. DesignTheory.org http://designtheory.org/database/

[Stinson2004] Douglas R. Stinson, Combinatorial designs: construction and analysis, Springer, 2004.

[ColDin01] Charles Colbourn, Jeffrey Dinitz, Mutually orthogonal latin squares: a brief survey of constructions, Volume 95, Issues 1-2, Pages 9-48, Journal of Statistical Planning and Inference, Springer, 1 May 2001.

[CD96] Making the MOLS table Charles Colbourn and Jeffrey Dinitz Computational and constructive design theory vol 368,pages 67-134 1996

[AbelThesis] On the Existence of Balanced Incomplete Block Designs and Transversal Designs, Julian R. Abel, PhD Thesis, University of New South Wales, 1995

[AbelCheng1994] R.J.R. Abel and Y.W. Cheng, Some new MOLS of order 2np for p a prime power, The Australasian Journal of Combinatorics, vol 10 (1994)

[BvR82] More mutually orthogonal Latin squares, Andries Brouwer and John van Rees Discrete Mathematics vol.39, num.3, pages 263-281 1982 http://oai.cwi.nl/oai/asset/304/0304A.pdf

[HananiBIBD] Balanced incomplete block designs and related designs, Haim Hanani, Discrete Mathematics 11.3 (1975) pages 255-369.

[Brouwer80] A Series of Separable Designs with Application to Pairwise Orthogonal Latin Squares, Andries E. Brouwer, Vol. 1, n. 1, pp. 39-41, European Journal of Combinatorics, 1980 http://www.sciencedirect.com/science/article/pii/S0195669880800199

[Greig99] Designs from projective planes and PBD bases Malcolm Greig Journal of Combinatorial Designs vol. 7, num. 5, pp. 341–374 1999

[DukesLing14] A three-factor product construction for mutually orthogonal latin squares, Peter J. Dukes, Alan C.H. Ling, http://arxiv.org/abs/1401.1466

[Rees00] Truncated Transversal Designs: A New Lower Bound on the Number of Idempotent MOLS of Side, Rolf S. Rees, Journal of Combinatorial Theory, Series A 90.2 (2000): 257-266.

[Rees93] Two new direct product-type constructions for resolvable group-divisible designs, Rolf S. Rees, Journal of Combinatorial Designs 1.1 (1993): 15-26.

[Thwarts] Thwarts in transversal designs Charles J.Colbourn, Jeffrey H. Dinitz, Mieczyslaw Wojtas. Designs, Codes and Cryptography 5, no. 3 (1995): 189-197.

[OS64] Finite projective planes with affine subplanes, T. G. Ostrom and F. A. Sherk. Canad. Math. Bull vol7 num.4 (1964)

[AC07] Concerning eight mutually orthogonal latin squares Julian R. Abel, Nicholas Cavenagh Journal of Combinatorial Designs Vol. 15, n.3, pp. 255-261 2007

[Hanani60] Haim Hanani, On quadruple systems, pages 145–157, vol. 12, Canadadian Journal of Mathematics, 1960 http://cms.math.ca/cjm/v12/cjm1960v12.0145-0157.pdf

[HH12] Victoria Horan and Glenn Hurlbert, Overlap Cycles for Steiner Quadruple Systems, 2012, http://arxiv.org/abs/1204.3215

[Naz96] Maxim Nazarov, Young's Orthogonal Form for Brauer's Centralizer Algebra. Journal of Algebra 182 (1996), 664–693.

[GL1996] J.J. Graham and G.I. Lehrer, Cellular algebras. Inventiones mathematicae 123 (1996), 1–34.

[HR2005] Tom Halverson and Arun Ram, *Partition algebras*. European Journal of Combinatorics **26** (2005), 869–921.

[Sta-EC2] Richard P. Stanley. *Enumerative Combinatorics*, Volume 2. Cambridge University Press, 2001.

[StaCat98] Richard Stanley. *Exercises on Catalan and Related Numbers excerpted from Enumerative Combinatorics, vol. 2 (CUP 1999)*, version of 23 June 1998. http://www-math.mit.edu/~rstan/ec/catalan.pdf

[Hag2008] James Haglund. *The q,t – Catalan Numbers and the Space of Diagonal Harmonics: With an Appendix on the Combinatorics of Macdonald Polynomials*. University of Pennsylvania, Philadelphia – AMS, 2008, 167 pp.

[DS1992] A. Denise, R. Simion, Two combinatorial statistics on Dyck paths, Discrete Math 137 (1992), 155–176.

[Kra2001] C. Krattenthaler – Permutations with restricted patterns and Dyck paths, Adv. Appl. Math. 27 (2001), 510–530.

[BK2001] J. Bandlow, K. Killpatrick – An area-to_inv bijection between Dyck paths and 312-avoiding permutations, Electronic Journal of Combinatorics, Volume 8, Issue 1 (2001).

[EP2004] S. Elizalde, I. Pak. *Bijections for refined restricted permutations**. JCTA 105(2) 2004.

[CK2008] A. Claesson, S. Kitaev. *Classification of bijections between '321'- and '132'- avoiding permutations*. Seminaire Lotharingien de Combinatoire **60** 2008. Arxiv 0805.1325.

[Knu1973] D. Knuth. *The Art of Computer Programming, Vol. III*. Addison-Wesley. Reading, MA. 1973.

[Stu2008] C. Stump – More bijective Catalan combinatorics on permutations and on colored permutations, Preprint. Arxiv 0808.2822.

[Cha2005] F. Chapoton, Une Base Symétrique de l'algèbre des Coinvariants Quasi-Symétriques, Electronic Journal of Combinatorics Vol 12(1) (2005) N16.

[AI] P. Arnoux, S. Ito, Pisot substitutions and Rauzy fractals, Bull. Belg. Math. Soc. 8 (2), 2001, pp. 181–207

[SAI] Y. Sano, P. Arnoux, S. Ito, Higher dimensional extensions of substitutions and their dual maps, J. Anal. Math. 83, 2001, pp. 183–206

[HKP2015] Clemens Heuberger, Sara Kropf, and Helmut Prodinger, *Output sum of transducers: Limiting distribution and periodic fluctuation*, Electron. J. Combin. 22 (2015), #P2.19.

[HKW2015] Clemens Heuberger, Sara Kropf and Stephan Wagner, *Variances and Covariances in the Central Limit Theorem for the Output of a Transducer*, European J. Combin. 49 (2015), 167-187, doi:10.1016/j.ejc.2015.03.004.

[HKP2015a] Clemens Heuberger, Sara Kropf, and Helmut Prodinger, *Analysis of Carries in Signed Digit Expansions*, Arxiv 1503.08816.

[P1964] William Parry, *Intrinsic Markov chains*, Transactions of the American Mathematical Society 112, 1964, pp. 55-66. doi:10.1090/S0002-9947-1964-0161372-1.

[S1948] Claude E. Shannon, *A mathematical theory of communication*, The Bell System Technical Journal 27, 1948, 379-423, doi:10.1002/j.1538-7305.1948.tb01338.x.

[HP2007] Clemens Heuberger and Helmut Prodinger, *The Hamming Weight of the Non-Adjacent-Form under Various Input Statistics*, Periodica Mathematica Hungarica Vol. 55 (1), 2007, pp. 81–96, doi:10.1007/s10998-007-3081-z.

[FGT1992] Philippe Flajolet, Danièle Gardy, Loÿs Thimonier, *Birthday paradox, coupon collectors, caching algorithms and self-organizing search*, Discrete Appl. Math. 39 (1992), 207–229, doi:10.1016/0166-218X(92)90177-C.

[FHP2015] Uta Freiberg, Clemens Heuberger, Helmut Prodinger, *Application of Smirnov Words to Waiting Time Distributions of Runs*, Arxiv 1503.08096.

[S1986] Gábor J. Székely, *Paradoxes in Probability Theory and Mathematical Statistics*, D. Reidel Publishing Company.

[BaWo2012] Javier Baliosian and Dina Wonsever, *Finite State Transducers*, chapter in *Handbook of Finite State Based Models and Applications*, edited by Jiacun Wang, Chapman and Hall/CRC, 2012.

[ChLi] F. Chapoton and M. Livernet, *Pre-Lie algebras and the rooted trees operad*, International Math. Research Notices (2001) no 8, pages 395-408.

[Liv] M. Livernet, *A rigidity theorem for pre-Lie algebras*, J. Pure Appl. Algebra 207 (2006), no 1, pages 1-18.

[Propp2001] James Propp. *The Many Faces of Alternating Sign Matrices*, Discrete Mathematics and Theoretical Computer Science 43 (2001): 58 Arxiv math/0208125

[Striker2015] Jessica Striker. *The toggle group, homomesy, and the Razumov-Stroganov correspondence*, Electron. J. Combin. 22 (2015) no. 2 Arxiv 1503.08898

[Wieland00] B. Wieland. *A large dihedral symmetry of the set of alternating sign matrices*. Electron. J. Combin. 7 (2000).

[BBF] B. Brubaker, D. Bump, and S. Friedberg. Weyl Group Multiple Dirichlet Series: Type A Combinatorial Theory. Ann. of Math. Stud., vol. 175, Princeton Univ. Press, New Jersey, 2011.

[GC50] I. M. Gelfand and M. L. Cetlin. Finite-Dimensional Representations of the Group of Unimodular Matrices. Dokl. Akad. Nauk SSSR **71**, pp. 825–828, 1950.

[Tok88] T. Tokuyama. A Generating Function of Strict Gelfand Patterns and Some Formulas on Characters of General Linear Groups. J. Math. Soc. Japan **40** (4), pp. 671–685, 1988.

[Knuth-TAOCP2A] D. Knuth "The art of computer programming", fascicules 2A, "generating all n-tuples"

[Knuth-TAOCP3A] D. Knuth "The art of computer programming", fascicule 3A "generating all combinations"

[Ryser63] H. J. Ryser, Combinatorial Mathematics, Carus Monographs, MAA, 1963.

[Gale57] D. Gale, A theorem on flows in networks, Pacific J. Math. 7(1957)1073-1082.

[PCh2013] Gregory Chatel and Viviane Pons. *Counting smaller trees in the Tamari order*. FPSAC. (2013). Arxiv 1212.0751v1.

[Pons2013] Viviane Pons, *Combinatoire algebrique liee aux ordres sur les permutations*. PhD Thesis. (2013). Arxiv 1310.1805v1.

[TamBrack1962] Dov Tamari. *The algebra of bracketings and their enumeration*. Nieuw Arch. Wisk. (1962).

[HuangTamari1972] Samuel Huang and Dov Tamari. *Problems of associativity: A simple proof for the lattice property of systems ordered by a semi-associative law*. J. Combinatorial Theory Ser. A. (1972). http://www.sciencedirect.com/science/article/pii/0097316572900039 .

[ChapTamari08] Frederic Chapoton. *Sur le nombre d'intervalles dans les treillis de Tamari*. Sem. Lothar. Combin. (2008). Arxiv math/0602368v1.

[LLMS2006] T. Lam, L. Lapointe, J. Morse, M. Shimozono, Affine insertion and Pieri rules for the affine Grassmannian, Memoirs of the AMS, 208 (2010), no. 977, Arxiv math.CO/0609110

[LLMSSZ2013] T. Lam, L. Lapointe, J. Morse, A. Schilling, M. Shimozono, M. Zabrocki, k-Schur functions and affine Schubert calculus, preprint Arxiv 1301.3569

[KL79] D. Kazhdan and G. Lusztig. *Representations of Coxeter groups and Hecke algebras*. Invent. Math. **53** (1979). no. 2, 165–184. doi:10.1007/BF01390031 MathSciNet MR0560412

[Dy93] M. J. Dyer. *Hecke algebras and shellings of Bruhat intervals*. Compositio Mathematica, 1993, 89(1): 91-115.

[BB05] A. Bjorner, F. Brenti. *Combinatorics of Coxeter groups*. New York: Springer, 2005.

[KnutsonPurbhoo10] A. Knutson, K. Purbhoo, Product and puzzle formulae for GL_n Belkale-Kumar coefficients, Arxiv 1008.4979

[KT2003] Allen Knutson, Terence Tao, Puzzles and (equivariant) cohomology of Grassmannians, Duke Math. J. 119 (2003) 221

[CoskunVakil06] I. Coskun, R. Vakil, Geometric positivity in the cohomology of homogeneous spaces and generalized Schubert calculus, Arxiv math/0610538

[KTW] Allen Knutson, Terence Tao, Christopher Woodward, The honeycomb model of GL(n) tensor products II: Puzzles determine facets of the Littlewood-Richardson cone, Arxiv math/0107011

[BuchKreschTamvakis03] 1. Buch, A. Kresch, H. Tamvakis, Gromov-Witten invariants on Grassmannian, Arxiv math/0306388

[Buch00] 1. Buch, A Littlewood-Richardson rule for the K-theory of Grassmannians, Arxiv math.AG/0004137

[HadaSloa] N.J.A. Sloane's Library of Hadamard Matrices, at http://neilsloane.com/hadamard/

[HadaWiki] Hadamard matrices on Wikipedia, Wikipedia article Hadamard_matrix

[Hora] K. J. Horadam, Hadamard Matrices and Their Applications, Princeton University Press, 2006.

[CP16] N. Cohen, D. Pasechnik, Implementing Brouwer's database of strongly regular graphs, http://arxiv.org/abs/1601.00181

[BH12] A. Brouwer and W. Haemers, Spectra of graphs, Springer, 2012, http://homepages.cwi.nl/~aeb/math/ipm/ipm.pdf

[HX10] W. Haemers and Q. Xiang, Strongly regular graphs with parameters $(4m^4, 2m^4+m^2, m^4+m^2, m^4+m^2)$ exist for all $m > 1$, European Journal of Combinatorics, Volume 31, Issue 6, August 2010, Pages 1553-1559, http://dx.doi.org/10.1016/j.ejc.2009.07.009.

[SWW72] A Street, W. Wallis, J. Wallis, Combinatorics: Room squares, sum-free sets, Hadamard matrices. Lecture notes in Mathematics 292 (1972).

[Ha83] M. Hall, Combinatorial Theory, 2nd edition, Wiley, 1983

[GS70s] J.M. Goethals and J. J. Seidel, A skew Hadamard matrix of order 36, J. Aust. Math. Soc. 11(1970), 343-344

[Wall71] J. Wallis, A skew-Hadamard matrix of order 92, Bull. Aust. Math. Soc. 5(1971), 203-204

[KoSt08] C. Koukouvinos, S. Stylianou On skew-Hadamard matrices, Discrete Math. 308(2008) 2723-2731

[JacMat96] Mark T. Jacobson and Peter Matthews, "Generating uniformly distributed random Latin squares", Journal of Combinatorial Designs, 4 (1996)

[NCSF] Gelfand, Krob, Lascoux, Leclerc, Retakh, Thibon, *Noncommutative Symmetric Functions*, Adv. Math. 112 (1995), no. 2, 218-348.

[QSCHUR] Haglund, Luoto, Mason, van Willigenburg, *Quasisymmetric Schur functions*, J. Comb. Theory Ser. A 118 (2011), 463-490. http://www.sciencedirect.com/science/article/pii/S0097316509001745 , Arxiv 0810.2489v2.

[Tev2007] Lenny Tevlin, *Noncommutative Analogs of Monomial Symmetric Functions, Cauchy Identity, and Hall Scalar Product*, Arxiv 0712.2201v1.

[Ges] I. Gessel, *Multipartite P-partitions and inner products of skew Schur functions*, Contemp. Math. **34** (1984), 289-301. http://people.brandeis.edu/~gessel/homepage/papers/multipartite.pdf

[MR] C. Malvenuto and C. Reutenauer, *Duality between quasi-symmetric functions and the Solomon descent algebra*, J. Algebra **177** (1995), no. 3, 967-982. http://www.mat.uniroma1.it/people/malvenuto/Duality.pdf

[GriRei2014] Darij Grinberg, Victor Reiner, *Hopf algebras in combinatorics*, 30 September 2014. Arxiv 1409.8356v1.

[Mal1993] Claudia Malvenuto, *Produits et coproduits des fonctions quasi-symetriques et de l'algebre des descentes*, thesis, November 1993. http://www1.mat.uniroma1.it/people/malvenuto/Thesis.pdf

[Haz2004] Michiel Hazewinkel, *Explicit polynomial generators for the ring of quasisymmetric functions over the integers*. Arxiv math/0410366v1

[Rad1979] David E. Radford, *A natural ring basis for the shuffle algebra and an application to group schemes*, J. Algebra **58** (1979), 432-454.

[NCSF1] Israel Gelfand, D. Krob, Alain Lascoux, B. Leclerc, V. S. Retakh, J.-Y. Thibon, *Noncommutative symmetric functions*. Arxiv hep-th/9407124v1

[NCSF2] D. Krob, B. Leclerc, J.-Y. Thibon, *Noncommutative symmetric functions II: Transformations of alphabets*. http://www-igm.univ-mlv.fr/~jyt/ARTICLES/NCSF2.ps

[LMvW13] Kurt Luoto, Stefan Mykytiuk and Stephanie van Willigenburg, *An introduction to quasisymmetric Schur functions – Hopf algebras, quasisymmetric functions, and Young composition tableaux*, May 23, 2013, Springer. http://www.math.ubc.ca/%7Esteph/papers/QuasiSchurBook.pdf

[BBSSZ2012] Chris Berg, Nantel Bergeron, Franco Saliola, Luis Serrano, Mike Zabrocki, *A lift of the Schur and Hall-Littlewood bases to non-commutative symmetric functions*, Arxiv 1208.5191v3.

[SW2010] John Shareshian and Michelle Wachs. *Eulerian quasisymmetric functions*. (2010). Arxiv 0812.0764v2

[HHL05] *A combinatorial formula for Macdonald polynomials*. Haiman, Haglund, and Loehr. J. Amer. Math. Soc. 18 (2005), no. 3, 735-761.

[LW12] *Quasisymmetric expansions of Schur-function plethysms*. Loehr and Warrington. Proc. Amer. Math. Soc. 140 (2012), no. 4, 1159-1171.

[KT97] *Noncommutative symmetric functions IV: Quantum linear groups and Hecke algebras at* $q = 0$. Krob and Thibon. Journal of Algebraic Combinatorics 6 (1997), 339-376.

[HNT06] F. Hivert, J.-C. Novelli, J.-Y. Thibon. *Commutative combinatorial Hopf algebras*. (2006). Arxiv 0605262v1.

[BZ05] N. Bergeron, M. Zabrocki. *The Hopf algebra of symmetric functions and quasisymmetric functions in non-commutative variables are free and cofree*. (2005). Arxiv math/0509265v3.

[BHRZ06] N. Bergeron, C. Hohlweg, M. Rosas, M. Zabrocki. *Grothendieck bialgebras, partition lattices, and symmetric functions in noncommutative variables*. Electronic Journal of Combinatorics. **13** (2006).

[RS06] M. Rosas, B. Sagan. *Symmetric functions in noncommuting variables*. Trans. Amer. Math. Soc. **358** (2006). no. 1, 215-232. Arxiv math/0208168.

[BRRZ08] N. Bergeron, C. Reutenauer, M. Rosas, M. Zabrocki. *Invariants and coinvariants of the symmetric group in noncommuting variables*. Canad. J. Math. **60** (2008). 266-296. Arxiv math/0502082

[BT13] N. Bergeron, N. Thiem. *A supercharacter table decomposition via power-sum symmetric functions*. Int. J. Algebra Comput. **23**, 763 (2013). doi:10.1142/S0218196713400171. Arxiv 1112.4901.

[Beck] M. Beck, Stanford Math Circle - Parking Functions, October 2010, http://math.stanford.edu/circle/parkingBeck.pdf

[Hag08] The q,t – Catalan Numbers and the Space of Diagonal Harmonics: With an Appendix on the Combinatorics of Macdonald Polynomials, James Haglund, University of Pennsylvania, Philadelphia – AMS, 2008, 167 pp.

[Shin] H. Shin, Forests and Parking Functions, slides from talk September 24, 2008, http://www.emis.de/journals/SLC/wpapers/s61vortrag/shin.pdf

[GXZ] A. M. Garsia, G. Xin, M. Zabrocki, A three shuffle case of the compositional parking function conjecture, Arxiv 1208.5796v1

[Ker] Kerber, A. 'Algebraic Combinatorics via Finite Group Actions' 1.3 p24

[AG1988] George E. Andrews, F. G. Garvan, *Dyson's crank of a partition*. Bull. Amer. Math. Soc. (N.S.) Volume 18, Number 2 (1988), 167-171. http://projecteuclid.org/euclid.bams/1183554533

[ORV] Grigori Olshanski, Amitai Regev, Anatoly Vershik, *Frobenius-Schur functions*, Arxiv math/0110077v1. Possibly newer version at http://www.wisdom.weizmann.ac.il/~regev/papers/FrobeniusSchurFunctions.ps

[AssafDEG] Sami Assaf. *Dual equivalence graphs and a combinatorial proof of LLT and Macdonald positivity*. (2008). Arxiv 1005.3759v5.

[LM2006] MR2167475 (2006j:05214) L. Lapointe, J. Morse. Tableaux on $k+1$-cores, reduced words for affine permutations, and k-Schur expansions. J. Combin. Theory Ser. A 112 (2005), no. 1, 44–81.

[LM2004] Lapointe, L. and Morse, J. 'Order Ideals in Weak Subposets of Young's Lattice and Associated Unimodality Conjectures'. Ann. Combin. (2004)

[JamesKerber] Gordon James, Adalbert Kerber, *The Representation Theory of the Symmetric Group*, Encyclopedia of Mathematics and its Applications, vol. 16, Addison-Wesley 1981.

[Sut2002] Ruedi Suter. *Young's Lattice and Dihedral Symmetries*. Europ. J. Combinatorics (2002) 23, 233–238. http://www.sciencedirect.com/science/article/pii/S0195669801905414

[BOR09] Emmanuel Briand, Rosa Orellana, Mercedes Rosas. *The stability of the Kronecker products of Schur functions*. Arxiv 0907.4652v2.

[CO10] Jonathan Comes, Viktor Ostrik. *On blocks of Deligne's category* $\underline{\mathrm{Rep}}(S_t)$. Arxiv 0910.5695v2, http://pages.uoregon.edu/jcomes/blocks.pdf

[BdVO12] Christopher Bowman, Maud De Visscher, Rosa Orellana. *The partition algebra and the Kronecker coefficients*. Arxiv 1210.5579v6.

[LLMMSZ13] Thomas Lam, Luc Laponte, Jennifer Morse, Anne Schilling, Mark Shimozono, and Mike Zabrocki. k-Schur Functions and Affine Schubert Calculus. 2013. Arxiv 1301.3569.

[nw] Nijenhuis, Wilf, Combinatorial Algorithms, Academic Press (1978).

[DJM99] R. Dipper, G. James and A. Mathas "The cyclotomic q-Schur algebra", Math. Z, 229 (1999), 385-416.

[BK09] J. Brundan and A. Kleshchev "Graded decomposition numbers for cyclotomic Hecke algebras", Adv. Math., 222 (2009), 1883-1942"

[KMR] A. Kleshchev, A. Mathas, and A. Ram, *Universal Specht modules for cyclotomic Hecke algebras*, Proc. London Math. Soc. (2012) 105 (6): 1245-1289. Arxiv 1102.3519v1

[ZS98] Antoine Zoghbi, Ivan Stojmenovic, *Fast Algorithms for Generating Integer Partitions*, Intern. J. Computer Math., Vol. 70., pp. 319–332. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.1287

[MV] combinatorics of orthogonal polynomials (A. de Medicis et X.Viennot, Moments des q-polynomes de Laguerre et la bijection de Foata-Zeilberger, Adv. Appl. Math., 15 (1994), 262-304)

[McD] combinatorics of hyperoctahedral group, double coset algebra and zonal polynomials (I. G. Macdonald, Symmetric functions and Hall polynomials, Oxford University Press, second edition, 1995, chapter VII).

[CM] Benoit Collins, Sho Matsumoto, On some properties of orthogonal Weingarten functions, Arxiv 0903.5143.

[GarStan1984] A. M. Garsia, Dennis Stanton. *Group actions on Stanley-Reisner rings and invariants of permutation groups*. Adv. in Math. **51** (1984), 107-201. http://www.sciencedirect.com/science/article/pii/0001870884900057

[FoSc78] Dominique Foata, Marcel-Paul Schuetzenberger. *Major Index and Inversion Number of Permutations*. Mathematische Nachrichten, volume 83, Issue 1, pages 143-159, 1978. http://igm.univ-mlv.fr/~berstel/Mps/Travaux/A/1978-3MajorIndexMathNachr.pdf

[CC13] Mahir Bilen Can and Yonah Cherniavsky. *Omitting parentheses from the cyclic notation*. (2013). Arxiv 1308.0936v2.

[Mcd] I. G. Macdonald. Symmetric functions and Hall polynomials. Oxford University Press, second edition, 1995.

[RSW2011] Victor Reiner, Franco Saliola, Volkmar Welker. *Spectra of Symmetrized Shuffling Operators*. Arxiv 1102.2460v2.

[Mark94] George Markowsky. *Permutation lattices revisited*. Mathematical Social Sciences, 27 (1994), 59–72.

[OkounkovVershik2] A. M. Vershik, A. Yu. Okounkov. *A New Approach to the Representation Theory of the Symmetric Groups. 2*. http://uk.arxiv.org/abs/math/0503040v3.

[CST10] Tullio Ceccherini-Silberstein, Fabio Scarabotti, Filippo Tolli. *Representation Theory of the Symmetric Groups: The Okounkov-Vershik Approach, Character Formulas, and Partition Algebras*. CUP 2010.

[LodRon0102066] Jean-Louis Loday and Maria O. Ronco. Order structure on the algebra of permutations and of planar binary trees. Arxiv math/0102066v1.

[Gec81] Fundamentals of Computation Theory Gecseg, F. Proceedings of the 1981 International Fct-Conference Szeged, Hungaria, August 24-28, vol 117 Springer-Verlag, 1981

[Sta97] Stanley, Richard. Enumerative Combinatorics, Vol. 1. Cambridge University Press, 1997

[GW14] G. Gratzer and F. Wehrung, Lattice Theory: Special Topics and Applications Vol. 1, Springer, 2014.

[Platt76] C. R. Platt, Planar lattices and planar graphs, Journal of Combinatorial Theory Series B, Vol 21, no. 1 (1976): 30-39.

[Greene73] Curtis Greene. *On the Möbius algebra of a partially ordered set*. Advances in Mathematics, **10**, 1973. doi:10.1016/0001-8708(73)90106-0.

[Etienne98] Gwihen Etienne. *On the Möbius algebra of geometric lattices*. European Journal of Combinatorics, **19**, 1998. doi:10.1006/eujc.1998.0227.

[Rosen] K. Rosen *Handbook of Discrete and Combinatorial Mathematics* (1999), Chapman and Hall.

[Garg] V. Garg *Introduction to Lattice Theory with Computer Science Applications* (2015), Wiley.

[Striker2011] J. Striker. *A unifying poset perpective on alternating sign matrices, plane partitions, Catalan objects, tournaments, and tableaux*, Advances in Applied Mathematics 46 (2011), no. 4, 583-609. Arxiv 1408.5391

[JRJ94] Jourdan, Guy-Vincent; Rampon, Jean-Xavier; Jard, Claude (1994), "Computing on-line the lattice of maximal antichains of posets", Order 11 (3) p. 197-210, doi:10.1007/BF02115811

[FT00] Stefan Felsner, William T. Trotter, Dimension, Graph and Hypergraph Coloring, Order, 2000, Volume 17, Issue 2, pp 167-177, http://link.springer.com/article/10.1023%2FA%3A1006429830221

[Stan2009] Richard Stanley, *Promotion and evacuation*, Electron. J. Combin. 16 (2009), no. 2, Special volume in honor of Anders Björner, Research Paper 9, 24 pp.

[BF1999] Thomas Britz, Sergey Fomin, *Finite posets and Ferrers shapes*, Advances in Mathematics 158, pp. 86-127 (2001), Arxiv math/9912126 (the arXiv version has less errors).

[Bj1980] Anders Björner, *Shellable and Cohen-Macaulay partially ordered sets*, Trans. Amer. Math. Soc. 260 (1980), 159-183, doi:10.1090/S0002-9947-1980-0570784-2

[EnumComb1] Richard P. Stanley, *Enumerative Combinatorics, volume 1*, Second Edition, Cambridge University Press (2011). http://math.mit.edu/~rstan/ec/ec1/

[EPW14] Ben Elias, Nicholas Proudfoot, and Max Wakefield. *The Kazhdan-Lusztig polynomial of a matroid*. 2014. Arxiv 1412.7408.

[St1986] Richard Stanley. *Two poset polytopes*, Discrete Comput. Geom. (1986), doi:10.1007/BF02187680

[Propp1997] James Propp, *Generating Random Elements of Finite Distributive Lattices*, Electron. J. Combin. 4 (1997), no. 2, The Wilf Festschrift volume, Research Paper 15. http://www.combinatorics.org/ojs/index.php/eljc/article/view/v4i2r15

[CH2006] William Y.C. Chen and Qing-Hu Hou, *Factors of the Gaussian coefficients*, Discrete Mathematics 306 (2006), 1446-1449. doi:10.1016/j.disc.2006.03.031

[Bu87] Butler, Lynne M. *A unimodality result in the enumeration of subgroups of a finite abelian group*. Proceedings of the American Mathematical Society 101, no. 4 (1987): 771-775. doi:10.1090/S0002-9939-1987-0911049-8

[Delsarte48] S. Delsarte, *Fonctions de Möbius Sur Les Groupes Abeliens Finis*, Annals of Mathematics, second series, Vol. 45, No. 3, (Jul 1948), pp. 600-609. http://www.jstor.org/stable/1969047

[Ca1948] Leonard Carlitz, "q-Bernoulli numbers and polynomials". Duke Math J. 15, 987-1000 (1948), doi:10.1215/S0012-7094-48-01588-9

[Ca1954] Leonard Carlitz, "q-Bernoulli and Eulerian numbers". Trans Am Soc. 76, 332-350 (1954), doi:10.1090/S0002-9947-1954-0060538-2

[vanLeeuwen91] Marc. A. A. van Leeuwen, *Edge sequences, ribbon tableaux, and an action of affine permutations*. Europe J. Combinatorics. **20** (1999). http://wwwmathlabo.univ-poitiers.fr/~maavl/pdf/edgeseqs.pdf

[RC-MLT] Ben Salisbury and Travis Scrimshaw. *Connecting marginally large tableaux and rigged configurations via crystals*. Preprint. Arxiv 1505.07040.

[Kleber1] Michael Kleber. *Combinatorial structure of finite dimensional representations of Yangians: the simply-laced case*. Internat. Math. Res. Notices. (1997) no. 4. 187-201.

[Kleber2] Michael Kleber. *Finite dimensional representations of quantum affine algebras*. Ph.D. dissertation at University of California Berkeley. (1998). Arxiv math.QA/9809087.

[OSS03] Masato Okado, Anne Schilling, and Mark Shimozono. *Virtual crystals and Klebers algorithm*. Commun. Math. Phys. **238** (2003). 187-209. Arxiv math.QA/0209082.

[OSS13] Masato Okado, Reiho Sakamoto, and Anne Schilling. *Affine crystal structure on rigged configurations of type* $D_n^{(1)}$. J. Algebraic Combinatorics, **37** (2013). 571-599. Arxiv 1109.3523.

[SchScr] Anne Schilling and Travis Scrimshaw. *Crystal structure on rigged configurations and the filling map*. Arxiv 1409.2920.

[SalScr] Ben Salisbury and Travis Scrimshaw. *A rigged configuration model for* $B(\infty)$. Arxiv 1404.6539.

[HKOTT2002] G. Hatayama, A. Kuniba, M. Okado, T. Takagi, Z. Tsuboi. Paths, Crystals and Fermionic Formulae. Prog. Math. Phys. **23** (2002) Pages 205-272.

[CrysStructSchilling06] Anne Schilling. Crystal structure on rigged configurations. International Mathematics Research Notices. Volume 2006. (2006) Article ID 97376. Pages 1-27.

[RigConBijection] Masato Okado, Anne Schilling, Mark Shimozono. A crystal to rigged configuration bijection for non-exceptional affine algebras. Algebraic Combinatorics and Quantum Groups. Edited by N. Jing. World Scientific. (2003) Pages 85-124.

[BijectionDn] Anne Schilling. A bijection between type $D_n^{(1)}$ crystals and rigged configurations. J. Algebra. **285** (2005) 292-334

[BijectionLRT] Anatol N. Kirillov, Anne Schilling, Mark Shimozono. A bijection between Littlewood-Richardson tableaux and rigged configurations. Selecta Mathematica (N.S.). **8** (2002) Pages 67-135. (MathSciNet MR1890195).

[OSS2003] Masato Okado, Anne Schilling, and Mark Shimozono. Virtual crystals and fermionic formulas of type $D_{n+1}^{(2)}$, $A_{2n}^{(2)}$, and $C_n^{(1)}$. Representation Theory. **7** (2003) Arxiv math.QA/0105017.

[Sakamoto13] Reiho Sakamoto. Rigged configurations and Kashiwara operators. (2013) Arxiv 1302.4562v1.

[OSS2011] Masato Okado, Reiho Sakamoto, Anne Schilling, Affine crystal structure on rigged configurations of type $D_n^{(1)}$, J. Algebraic Combinatorics 37(3) (2013) 571-599 (Arxiv 1109.3523 [math.QA])

[FSS07] G. Fourier, A. Schilling, and M. Shimozono, Demazure structure inside Kirillov-Reshetikhin crystals, J. Algebra, Vol. 309, (2007), p. 386-404 http://arxiv.org/abs/math/0605451

[HST09] F. Hivert, A. Schilling, and N. M. Thiery, Hecke group algebras as quotients of affine Hecke algebras at level 0, JCT A, Vol. 116, (2009) p. 844-863 http://arxiv.org/abs/0804.3781

[HST2008] F. Hivert, A. Schilling, N. Thiery, Hecke group algebras as quotients of affine Hecke algebras at level 0, Journal of Combinatorial Theory, Series A 116 (2009) 844-863 (arXiv:0804.3781 [math.RT])

[Kac] Kac, *Infinite-dimensional Lie algebras*, Third Edition. Cambridge, 1990.

[KMPS] Kass, Moody, Patera and Slansky, *Affine Lie algebras, weight multiplicities, and branching rules*. Vols. 1, 2. University of California Press, Berkeley, CA, 1990.

[KacPeterson] Kac and Peterson. *Infinite-dimensional Lie algebras, theta functions and modular forms*. Adv. in Math. 53 (1984), no. 2, 125-264.

[Carter] Carter, *Lie algebras of finite and affine type*. Cambridge University Press, 2005

[HHL06] J. Haglund, M. Haiman and N. Loehr, A combinatorial formula for nonsymmetric Macdonald polynomials, Amer. J. Math. 130, No. 2 (2008), 359-383.

[LNSSS12] C. Lenart, S. Naito, D. Sagaki, A. Schilling, M. Shimozono, A uniform model for Kirillov-Reshetikhin crystals I: Lifting the parabolic quantum Bruhat graph, preprint arXiv.1211.2042 [math.QA]

[FoSta1994] 19. Fomin, R. Stanley. Schubert polynomials and the nilCoxeter algebra. Advances in Math., 1994.

[BH1994] 19. Billey, M. Haiman. Schubert polynomials for the classical groups. J. Amer. Math. Soc., 1994.

[TKLam1996] T.K. Lam. B and D analogues of stable Schubert polynomials and related insertion algorithms. PhD Thesis, MIT, 1996.

[Lam2008] 20. Lam. Schubert polynomials for the affine Grassmannian. J. Amer. Math. Soc., 2008.

[LSS2009] 20. Lam, A. Schilling, M. Shimozono. Schubert polynomials for the affine Grassmannian of the symplectic group. Mathematische Zeitschrift 264(4) (2010) 765-811 (arXiv:0710.2720 [math.CO])

[Pon2010] 19. Pon. Types B and D affine Stanley symmetric functions, unpublished PhD Thesis, UC Davis, 2010.

[Haiman06] 13. Haiman, Cherednik algebras, Macdonald polynomials and combinatorics, ICM 2006.

[Lusztig1985] G. Lusztig, *Equivariant K-theory and representations of Hecke algebras*, Proc. Amer. Math. Soc. 94 (1985), no. 2, 337-342.

[Cherednik1995] I. Cherednik, *Nonsymmetric Macdonald polynomials*. IMRN 10, 483-515 (1995).

[Kumar1987] S. Kumar, Demazure character formula in arbitrary Kac-Moody setting, Invent. Math. 89 (1987), no. 2, 395-423.

[Lascoux2003] Alain Lascoux, Symmetric functions and combinatorial operators on polynomials, CBMS Regional Conference Series in Mathematics, 99, 2003.

[Reiner97] Victor Reiner. *Non-crossing partitions for classical reflection groups*. Discrete Mathematics 177 (1997)

[Arm06] Drew Armstrong. *Generalized Noncrossing Partitions and Combinatorics of Coxeter Groups*. Arxiv math/0611106

[CFZ] Chapoton, Fomin, Zelevinsky - Polytopal realizations of generalized associahedra, Arxiv math/0202004.

[Iwahori] Iwahori, *Generalized Tits system (Bruhat decomposition) on p-adic semisimple groups*. 1966 Algebraic Groups and Discontinuous Subgroups (AMS Proc. Symp. Pure Math.., 1965) pp. 71-83 Amer. Math. Soc., Providence, R.I.

[Bour] Bourbaki, *Lie Groups and Lie Algebras* IV.2

[OSShimo03] M. Okado, A. Schilling, M. Shimozono. "Virtual crystals and fermionic formulas for type $D_{n+1}^{(2)}$, $A_{2n}^{(2)}$, and $C_n^{(1)}$". Representation Theory. **7** (2003). 101-163. doi:10.1.1.192.2095, Arxiv 0810.5067.

[SL000081] Sloane's OEIS sequence A000081

[Knu1970] Donald E. Knuth. *Permutations, matrices, and generalized Young tableaux*. Pacific J. Math. Volume 34, Number 3 (1970), pp. 709-727. http://projecteuclid.org/euclid.pjm/1102971948

[EG1987] Paul Edelman, Curtis Greene. *Balanced Tableaux*. Advances in Mathematics 63 (1987), pp. 42-99. http://www.sciencedirect.com/science/article/pii/0001870887900636

[BKSTY06] A. Buch, A. Kresch, M. Shimozono, H. Tamvakis, and A. Yong. *Stable Grothendieck polynomials and K-theoretic factor sequences*. Math. Ann. **340** Issue 2, (2008), pp. 359–382. Arxiv math/0601514v1.

[LM2011] A. Lauve, M. Mastnak. *The primitives and antipode in the Hopf algebra of symmetric functions in noncommuting variables*. Advances in Applied Mathematics. **47** (2011). 536-544. Arxiv 1006.0367v3 doi:10.1016/j.aam.2011.01.002.

[OZ2015] R. Orellana, M. Zabrocki, *Symmetric group characters as symmetric functions*, Arxiv 1510.00438.

[Mac1995] I. G. Macdonald, Symmetric functions and Hall polynomials, second ed., The Clarendon Press, Oxford University Press, New York, 1995, With contributions by A. Zelevinsky, Oxford Science Publications.

[Jack1970] H. Jack, *A class of symmetric functions with a parameter*, Proc. R. Soc. Edinburgh (A), 69, 1-18.

[Ma1995] I. G. Macdonald, *Symmetric functions and Hall polynomials*, second ed., The Clarendon Press, Oxford University Press, New York, 1995, With contributions by A. Zelevinsky, Oxford Science Publications.

[Mc1995] I. G. Macdonald, Symmetric functions and Hall polynomials, second ed., The Clarendon Press, Oxford University Press, New York, 1995, With contributions by A. Zelevinsky, Oxford Science Publications.

[Lam2006] T. Lam, Schubert polynomials for the affine Grassmannian, J. Amer. Math. Soc., 21 (2008), 259-281.

[LLMSSZ] T. Lam, L. Lapointe, J. Morse, A. Schilling, M. Shimozono, M. Zabrocki, k-Schur functions and affine Schubert calculus.

[LLT1997] Alain Lascoux, Bernard Leclerc, Jean-Yves Thibon, Ribbon tableaux, Hall-Littlewood functions, quantum affine algebras, and unipotent varieties, J. Math. Phys. 38 (1997), no. 2, 1041-1068, arXiv:q-alg/9512-31v1 [math.q.alg]

[LT2000] Bernard Leclerc and Jean-Yves Thibon, Littlewood-Richardson coefficients and Kazhdan-Lusztig polynomials, in: Combinatorial methods in representation theory (Kyoto) Adv. Stud. Pure Math., vol. 28, Kinokuniya, Tokyo, 2000, pp 155-220 arXiv:math/9809122v3 [math.q-alg]

[Macdonald1995] I. G. Macdonald, Symmetric functions and Hall polynomials, second ed., The Clarendon Press, Oxford University Press, New York, 1995, With contributions by A. Zelevinsky, Oxford Science Publications.

[GH1993] A. Garsia, M. Haiman, A graded representation module for Macdonald's polynomials, Proc. Nat. Acad. U.S.A. no. 90, 3607–3610.

[BGHT1999] F. Bergeron, A. M. Garsia, M. Haiman, and G. Tesler, Identities and positivity conjectures for some remarkable operators in the theory of symmetric functions, Methods Appl. Anal. 6 (1999), no. 3, 363–420.

[LLM1998] L. Lapointe, A. Lascoux, J. Morse, Determinantal Expressions for Macdonald Polynomials, IRMN no. 18 (1998). Arxiv math/9808050.

[BH2013] F. Bergeron, M. Haiman, Tableaux Formulas for Macdonald Polynomials, Special edition in honor of Christophe Reutenauer 60 birthday, International Journal of Algebra and Computation, Volume 23, Issue 4, (2013), pp. 833-852.

[Morse11] J. Morse, Combinatorics of the K-theory of affine Grassmannians, Adv. in Math., Volume 229, Issue 5, pp. 2950–2984.

[LamSchillingShimozono10] T. Lam, A. Schilling, M.Shimozono, K-theory Schubert calculus of the affine Grassmannian, Compositio Math. 146 (2010), 811-852.

[SZ.2001] M. Shimozono, M. Zabrocki, Hall-Littlewood vertex operators and generalized Kostka polynomials. Adv. Math. 158 (2001), no. 1, 66-85.

[Morse2011] J. Morse, Combinatorics of the K-theory of affine Grassmannians, Adv. in Math., Volume 229, Issue 5, pp. 2950–2984.

[LamSchillingShimozono2010] T. Lam, A. Schilling, M.Shimozono, K-theory Schubert calculus of the affine Grassmannian, Compositio Math. 146 (2010), 811-852.

[ClSt03] Peter Clifford, Richard P. Stanley, *Bottom Schur functions*. Arxiv math/0311382v2.

[ChariKleber2000] Vyjayanthi Chari and Michael Kleber. *Symmetric functions and representations of quantum affine algebras*. Arxiv math/0011161v1

[KoikeTerada1987] K. Koike, I. Terada, *Young-diagrammatic methods for the representation theory of the classical groups of type Bn, Cn, Dn*. J. Algebra 107 (1987), no. 2, 466-511.

[ShimozonoZabrocki2006] Mark Shimozono and Mike Zabrocki. *Deformed universal characters for classical and affine algebras*. Journal of Algebra, **299** (2006). Arxiv math/0404288.

[FD06] Francois Descouens, Making research on symmetric functions using MuPAD-Combinat. In Andres Iglesias and Nobuki Takayama, editors, 2nd International Congress on Mathematical Software (ICMS'06), volume 4151 of LNCS, pages 407-418, Castro Urdiales, Spain, September 2006. Springer-Verlag. Arxiv 0806.1873

[HT04] Florent Hivert and Nicolas M. Thiery, MuPAD-Combinat, an open-source package for research in algebraic combinatorics. Sem. Lothar. Combin., 51 :Art. B51z, 70 pp. (electronic), 2004. http://mupad-combinat.sf.net/.

[MAC] Ian Macdonald, Symmetric Functions and Orthogonal Polynomials, Second edition. With contributions by A. Zelevinsky. Oxford Mathematical Monographs. Oxford Science Publications. The Clarendon Press, Oxford University Press, New York, 1995. x+475 pp. ISBN: 0-19-853489-2

[STA] Richard Stanley, Enumerative combinatorics. Vol. 2. With a foreword by Gian-Carlo Rota and appendix 1 by Sergey Fomin. Cambridge Studies in Advanced Mathematics, 62. Cambridge University Press, Cambridge, 1999. xii+581 pp. ISBN: 0-521-56069-1; 0-521-78987-7

[ST94] Scharf, Thomas, Thibon, Jean-Yves, A Hopf-algebra approach to inner plethysm. Adv. Math. 104 (1994), no. 1, 30-58. doi:10.1006/aima.1994.1019

[SZ2001] M. Shimozono, M. Zabrocki, Hall-Littlewood vertex operators and generalized Kostka polynomials. Adv. Math. 158 (2001), no. 1, 66-85.

[King] King, R. Branching rules for $GL_m \supset \Sigma_n$ and the evaluation of inner plethysms. J. Math. Phys. 15, 258 (1974) doi:10.1063/1.1666632

[SchaThi1994] Thomas Scharf, Jean-Yves Thibon. *A Hopf-algebra approach to inner plethysm*. Advances in Mathematics 104 (1994), pp. 30-58. ftp://ftp.mathe2.uni-bayreuth.de/axel/papers/scharf:a_hopf_algebra_approach_to_inner_plethysm.ps.gz

[ShaWach2014] John Shareshian, Michelle L. Wachs. *Chromatic quasisymmetric functions*. Arxiv 1405.4629v1.

[GR1993] Ira M. Gessel, Christophe Reutenauer. *Counting Permutations with Given Cycle Structure and Descent Set*. Journal of Combinatorial Theory, Series A, 64 (1993), pp. 189–215.

[HazWitt1] Michiel Hazewinkel. *Witt vectors. Part 1*. Arxiv 0804.3888v1

[DoranIV1996] William F. Doran IV. *A Proof of Reutenauer's '-q_{(n)}' Conjecture*. Journal of combinatorial theory, Series A 74, pp. 342-344 (1996), article no. 0056. doi:10.1006/jcta.1996.0056

[BorWi2004] James Borger, Ben Wieland. *Plethystic algebra*. Arxiv math/0407227v1

[Banc2011] E. E. Bancroft, *Shard Intersections and Cambrian Congruence Classes in Type A.*, Ph.D. Thesis, North Carolina State University. 2011.

[Pete2013] T. Kyle Petersen, *On the shard intersection order of a Coxeter group*, SIAM J. Discrete Math. 27 (2013), no. 4, 1880-1912.

[Read2011] N. Reading, *Noncrossing partitions and the shard intersection order*, J. Algebraic Combin., 33 (2011), 483-530.

[EilLan53] On the groups $H(\pi, n)$, I, Samuel Eilenberg and Saunders Mac Lane, 1953.

[Green55] Green, J. A. *The characters of the finite general linear groups*. Trans. Amer. Math. Soc. 80 (1955), 402–447. doi:10.1090/S0002-9947-1955-0072878-2

[Morrison06] Morrison, Kent E. *Integer sequences and matrices over finite fields*. J. Integer Seq. 9 (2006), no. 2, Article 06.2.1, 28 pp. https://cs.uwaterloo.ca/journals/JIS/VOL9/Morrison/morrison37.html

[PSS13] Prasad, A., Singla, P., and Spallone, S., *Similarity of matrices over local rings of length two*. Arxiv 1212.6157

[NS] T. Nakanishi, S. Stella, Wonder of sine-Gordon Y-systems, to appear in Trans. Amer. Math. Soc., Arxiv 1212.6853

[Mac95] Macdonald I.-G., (1995), "Symmetric Functions and Hall Polynomials", Oxford Science Publication

[Stan2002] Richard P. Stanley, *The rank and minimal border strip decompositions of a skew partition*, J. Combin. Theory Ser. A 100 (2002), pp. 349-375. Arxiv math/0109092v1.

[Fulton97] William Fulton, *Young Tableaux*, Cambridge University Press 1997.

[BLL] F. Bergeron, G. Labelle, and P. Leroux. "Combinatorial species and tree-like structures". Encyclopedia of Mathematics and its Applications, vol. 67, Cambridge Univ. Press. 1998.

[BLL-Intro] Francois Bergeron, Gilbert Labelle, and Pierre Leroux. "Introduction to the Theory of Species of Structures", March 14, 2008.

[MM] M. Maia and M. Mendez. "On the arithmetic product of combinatorial species". Discrete Mathematics, vol. 308, issue 23, 2008, pp. 5407-5427. Arxiv math/0503436v2.

[Labelle] 7. Labelle. "New combinatorial computational methods arising from pseudo-singletons." DMTCS Proceedings 1, 2008.

[KnuMil] Knutson and Miller. *Subword complexes in Coxeter groups*. Adv. Math., 184(1):161-176, 2004.

[PilStu] Pilaud and Stump. *Brick polytopes of spherical subword complexes and generalized associahedra*. Adv. Math. 276:1-61, 2015.

[GS93] David M. Goldschmidt. *Group characters, symmetric functions, and the Hecke algebras*. AMS 1993.

[Ram1997] Arun Ram. *Seminormal representations of Weyl groups and Iwahori-Hecke algebras*. Proc. London Math. Soc. (3) **75** (1997). 99-133. Arxiv math/9511223v1. http://www.ms.unimelb.edu.au/~ram/Publications/1997PLMSv75p99.pdf

[EtRT] Pavel Etingof, Oleg Golberg, Sebastian Hensel, Tiankai Liu, Alex Schwendner, Dmitry Vaintrob, Elena Yudovina, "Introduction to representation theory", Arxiv 0901.0827v5.

[Las] Alain Lascoux, 'Young representations of the symmetric group.' http://phalanstere.univ-mlv.fr/~al/ARTICLES/ProcCrac.ps.gz

[Krat99] C. Krattenthaler, *Another Involution Principle-Free Bijective Proof of Stanley's Hook Content Formula*, Journal of Combinatorial Theory, Series A vol 88 Issue 1 (1999), 66-92, http://www.sciencedirect.com/science/article/pii/0012365X9290368P

[Ful1997] William Fulton, *Young Tableaux*. Cambridge University Press, 1997.

[BerKilGGI] A. N. Kirillov, A. D. Berenstein, *Groups generated by involutions, Gelfand–Tsetlin patterns, and combinatorics of Young tableaux*, Algebra i Analiz, 1995, Volume 7, Issue 1, pp. 92–152. http://math.uoregon.edu/~arkadiy/bk1.pdf

[LS90] A. Lascoux, M.-P. Schutzenberger. Keys and standard bases, invariant theory and tableaux. IMA Volumes in Math and its Applications (D. Stanton, ED.). Southend on Sea, UK, 19 (1990). 125-144.

[Willis10] M. Willis. A direct way to find the right key of a semistandard Young tableau. Arxiv 1110.6184v1.

[Ive2012] S. Iveson, *Tableaux on 'k + 1'-cores, reduced words for affine permutations, and 'k'-Schur expansions*, Operators on k-tableaux and the k-Littlewood-Richardson rule for a special case, UC Berkeley: Mathematics, Ph.D. Thesis, https://escholarship.org/uc/item/7pd1v1b5

[Hai1992] Mark D. Haiman, *Dual equivalence with applications, including a conjecture of Proctor*, Discrete Mathematics 99 (1992), 79-113, http://www.sciencedirect.com/science/article/pii/0012365X9290368P

[Sg2011] Bruce E. Sagan, *The cyclic sieving phenomenon: a survey*, Arxiv 1008.0790v3

[LLM01] L. Lapointe, A. Lascoux, J. Morse. *Tableau atoms and a new Macdonald positivity conjecture*. Arxiv math/0008073v2.

[S14] B. Salisbury. The flush statistic on semistandard Young tableaux. Arxiv 1401.1185

[Aval2008] Jean-Christope Aval. *Keys and Alternating Sign Matrices*, Seminaire Lotharingien de Combinatoire 59 (2008) B59f Arxiv 0711.2150

[DJM] R. Dipper, G. James and A. Mathas "The cyclotomic q-Schur algebra", Math. Z, 229 (1999), 385-416.

[BK] J. Brundan and A. Kleshchev "Graded decomposition numbers for cyclotomic Hecke algebras", Adv. Math., 222 (2009), 1883-1942"

[BMFPR] M. Bousquet-Melou, E. Fusy, L.-F. Preville Ratelle. *The number of intervals in the m-Tamari lattices*. Arxiv 1106.1498

[Knuth1] Knuth, Donald (2000). "Dancing links". Arxiv cs/0011047.

[CMS2012] Alexandre Casamayou, Nathann Cohen, Guillaume Connan, Thierry Dumont, Laurent Fousse, François Maltey, Matthias Meulien, Marc Mezzarobba, Clément Pernet, Nicolas M. Thiéry, Paul Zimmermann *Calcul Mathématique avec Sage* http://sagebook.gforge.inria.fr/

[BBGL08] A. Blondin Massé, S. Brlek, A. Garon, and S. Labbé, Combinatorial properties of f -palindromes in the Thue-Morse sequence. Pure Math. Appl., 19(2-3):39–52, 2008.

[BHNR04] S. Brlek, S. Hamel, M. Nivat, C. Reutenauer, On the Palindromic Complexity of Infinite Words, in J. Berstel, J. Karhumaki, D. Perrin, Eds, Combinatorics on Words with Applications, International Journal of Foundation of Computer Science, Vol. 15, No. 2 (2004) 293–306.

[DJP01] X. Droubay, J. Justin, G. Pirillo, Episturmian words and some constructions of de Luca and Rauzy, Theoret. Comput. Sci. 255, (2001), no. 1–2, 539–553.

[Sta11] Š. Starosta, On Theta-palindromic Richness, Theoret. Comp. Sci. 412 (2011) 1111–1121

[Arn2002] P. Arnoux, Sturmian sequences, in Substitutions in Dynamics, N. Pytheas Fogg (Ed.), Arithmetics, and Combinatorics (Lecture Notes in Mathematics, Vol. 1794), 2002.

[Ser1985] C. Series. The geometry of Markoff numbers. The Mathematical Intelligencer, 7(3):20–29, 1985.

[SU2009] J. Smillie and C. Ulcigrai. Symbolic coding for linear trajectories in the regular octagon, Arxiv 0905.0871, 2009.

[Mon2010] T. Monteil, The asymptotic language of smooth curves, talk at LaCIM2010.

[CassNic10] Cassaigne J., Nicolas F. Factor complexity. Combinatorics, automata and number theory, 163–247, Encyclopedia Math. Appl., 135, Cambridge Univ. Press, Cambridge, 2010.

[AC03] B. Adamczewski, J. Cassaigne, On the transcendence of real numbers with a regular expansion, J. Number Theory 103 (2003) 27–37.

[BmBGL07] A. Blondin-Masse, S. Brlek, A. Glen, and S. Labbe. On the critical exponent of generalized Thue-Morse words. *Discrete Math. Theor. Comput. Sci.* 9 (1):293–304, 2007.

[BmBGL09] A. Blondin-Masse, S. Brlek, A. Garon, and S. Labbe. Christoffel and Fibonacci Tiles, DGCI 2009, Montreal, to appear in LNCS.

[Loth02] M. Lothaire, Algebraic Combinatorics On Words, vol. 90 of Encyclopedia of Mathematics and its Applications, Cambridge University Press, U.K., 2002.

[Fogg] Pytheas Fogg, https://www.lirmm.fr/arith/wiki/PytheasFogg/S-adiques.

[Kolakoski66] William Kolakoski, proposal 5304, American Mathematical Monthly 72 (1965), 674; for a partial solution, see "Self Generating Runs," by Necdet Üçoluk, Amer. Math. Mon. 73 (1966), 681-2.

[BMP07] S. Brlek, G. Melançon, G. Paquin, Properties of the extremal infinite smooth words, Discrete Math. Theor. Comput. Sci. 9 (2007) 33–49.

[JP02] J. Justin, G. Pirillo, Episturmian words and episturmian morphisms, Theoret. Comput. Sci. 276 (2002) 281–313.

[GJ07] A. Glen, J. Justin, Episturmian words: a survey, Preprint, 2007, arXiv:0801.1655.

[Brlek89] Brlek, S. 1989. «Enumeration of the factors in the Thue-Morse word», *Discrete Appl. Math.*, vol. 24, p. 83–96.

[MH38] Morse, M., et G. A. Hedlund. 1938. «Symbolic dynamics», *American Journal of Mathematics*, vol. 60, p. 815–866.

[GK82] Greene, Daniel H.; Knuth, Donald E. (1982), "2.1.1 Constant coefficients - A) Homogeneous equations", Mathematics for the Analysis of Algorithms (2nd ed.), Birkhauser, p. 17.

[SZ94] Bruno Salvy and Paul Zimmermann. - Gfun: a Maple package for the manipulation of generating and holonomic functions in one variable. - Acm transactions on mathematical software, 20.2:163-177, 1994.

[Z11] Zeilberger, Doron. "The C-finite ansatz." The Ramanujan Journal (2011): 1-10.

C

r

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z