Das Python-Tutorium

Version 2.1

Guido van Rossum
Fred L. Drake, Jr., Hrsg.

15. April 2001

PythonLabs
E-Mail: python-docs@python.org

BEOPEN.COM TERMS AND CONDITIONS FOR PYTHON 2.0
BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI OPEN SOURCE LICENSE AGREEMENT

Python 1.6 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1012. This Agreement may also be obtained from a proxy server on the Internet using the following URL: `http://hdl.handle.net/1895.22/1012`.

CWI PERMISSIONS STATEMENT AND DISCLAIMER

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

VORWORT DES ÜBERSETZERS

Der Text dieses Tutoriums wurde mit größter Sorgfalt aus dem englischen Original übersetzt (inklusive der Kommentare in den Code-Beispielen, nicht jedoch der Code selbst). Dennoch können Fehler nicht vollständig ausgeschlossen werden. Der Übersetzer (Dinu C. Gherman) übernimmt keinerlei juristische Verantwortung für evtl. verbliebene fehlerhafte Angaben in dieser Übersetzung und schließt jede Haftung für deren Folgen aus. Das Werk einschließlich aller seiner Teile ist in seiner englischen Originalfassung urheberrechtlich geschützt (siehe obiges Copyright).

Zusammenfassung

Python ist eine leicht zu lernende und mächtige Programmiersprache. Sie bietet effiziente, abstrakte Datenstrukturen und einen einfachen aber effektiven Ansatz zum objektorientierten Programmieren. Die elegante Syntax und dynamische Bindung von Python machen sie – zusammen mit der Tatsache, dass es interpretiert wird – zu einer idealen Sprache für Skripte und zur schnellen Applikationsentwicklung in vielen Bereichen und auf den meisten Plattformen.

Der Python-Interpreter und die umfangreiche Standard-Bibliothek sind im Quellcode und in Binärform für alle wichtigen Plattformen auf Pythons Web-Site, `http://www.python.org`, kostenlos verfügbar und können frei weiterverbreitet werden. Am gleichen Ort befinden sich auch Distributionen von und Hinweise auf viele Python-Module, -Programme und -Werkzeuge sowie zusätzliche Dokumentation von Dritten.

Der Python-Interpreter kann leicht um neue Funktionen und Datentypen erweitert werden, die in C oder C++ (oder anderen von C aufrufbaren Sprachen) implementiert sind. Python ist ebenso als Erweiterungssprache für in C geschriebene Anwendungen geeignet.

Dieses Tutorium führt informell in die grundlegenden Konzepte und Eigenschaften der Sprache Python und des Drumherums ein. Es ist von Vorteil, einen Python-Interpreter zur Hand zu haben, um parallel zum Tutorium eigene Erfahrungen damit zu machen, aber alle Beispiele sind in sich abgeschlossen, so dass das Tutorium auch ohne Rechner gelesen werden kann.

Eine Beschreibung der Standard-Objekte und -Module findet sich in der *Python-Bibliotheks-Referenz* (engl. *Python Library Reference*). Die *Python-Referenz* (engl. *Python Reference*) enthält eine formalere Darstellung der Sprache. Um Erweiterungen in C oder C++ zu schreiben, lese man *Erweiterung und Einbettung* (engl. *Extension and Embedding*) und *Python/C API-Referenz* (engl. *Python/C API Reference*).

Dieses Tutorium versucht nicht, umfassend zu sein und jede einzelne Eigenschaft oder auch nur jede häufig benutzte zu behandeln. Stattdessen führt es viele der bemerkenswertesten Eigenschaften von Python ein und wird Ihnen einen guten Einblick in die Konzepte und den Stil der Sprache geben. Nach der Lektüre werden Sie in der Lage sein, Python-Module und -Programme zu lesen und zu schreiben, und Sie werden besser auf die Benutzung der verschiedenen Python-Bibliotheken vorbereitet sein, wie sie in der *Python-Bibliotheks-Referenz* beschrieben sind.

INHALTSVERZEICHNIS

KAPITEL

EINS

Einige Leckerbissen vorweg

Wenn Sie jemals ein langes Shell-Skript geschrieben haben, kennen Sie vermutlich dieses Gefühl: Sie würden liebend gern noch eine weitere Eigenschaft hinzufügen, aber es ist bereits sehr langsam, sehr groß und sehr kompliziert. Oder die Eigenschaft verlangt einen Systemaufruf oder andere Funktion, die nur von C aus verfügbar ist... Normalerweise ist das aktuelle Problem nicht ernst genug, um ganz neu in C geschrieben zu werden. Vielleicht braucht man nur Strings variabler Länge oder Datentypen (wie sortierte Listen von Dateinamen), die bereits leicht in der Shell zu verwenden sind, aber eine Menge Arbeit in C machen, oder Sie sind einfach nicht vertraut genug mit C.

Eine andere Situation: Vielleicht müssen Sie mit mehreren C-Bibliotheken arbeiten und der übliche schreiben/übersetzen/testen/neu-übersetzen-Zyklus ist zu langsam. Sie müssen Software schneller entwickeln. Möglicherweise haben Sie ein Programm geschrieben, das eine Erweiterungssprache brauchen könnte, aber Sie wollen keine eigene Sprache entwerfen, einen Interpreter dafür schreiben und testen und dann an Ihre Anwendung koppeln.

In solchen Fällen könnte Python genau die richtige Sprache für Sie sein. Python ist einfach zu benutzen, aber es ist eine echte Programmiersprache, die sehr viel mehr Struktur und Unterstützung für große Programme bereitstellt als eine Shell. Auf der anderen Seite bietet sie eine viel bessere Fehlerkontrolle als C und hat (weil es eine echte *Hochsprache* ist) eingebaute abstrakte Datentypen wie flexible Listen und Dictionaries (engl. für Wörterbuch), für deren effiziente Implementierung in C Sie Tage bräuchten. Wegen seiner abstrakteren Datentypen kann Python bei einer wesentlich größeren Klasse von Problemen angewendet werden als *Awk* und sogar *Perl* bzw. viele Dinge sind mindestens so einfach in Python wie in diesen anderen Sprachen.

Python erlaubt es, Ihr Programm in Module aufzuteilen, die in anderen Python-Programmen wiederverwendet werden können. Zu Python gehört eine große Sammlung von Standard-Modulen, die man als Basis für eigene Programme benutzen kann – oder als Beispiele, um in Python programmieren zu lernen. Es gibt auch eingebaute Module, die Dateiein- und ausgabe, Systemaufrufe und sogar Schnittstellen zu GUI-Bibliotheken wie Tk zur Verfügung stellen.

Python ist eine interpretierte Sprache, die Ihnen beträchtliche Zeit während der Programmentwicklung sparen helfen kann, da kein Übersetzen und Binden notwendig ist. Der Interpreter kann interaktiv benutzt werden, wodurch man leicht mit den Eigenschaften der Sprache experimentieren, eigene Wegwerf-Programme schreiben und Funktionen während der Programmentwicklung von oben nach unten testen kann. Er gibt auch einen praktischen Taschenrechner ab.

Python erlaubt es, sehr kompakte und leserliche Programme zu schreiben. In Python geschriebene Programme sind aus folgenden Gründen normalerweise wesentlich kürzer als C- oder C++-Programme:

- die abstrakten Datentypen erlauben komplexe Operationen in einem einzigen Ausdruck,
- die Gruppierung von Ausdrücken erfolgt durch Einrückung anstatt durch Anfangs- und Ende-Klammern,
- Variablen und Argumente müssen nicht deklariert werden.

Python ist *erweiterbar*: Wenn Sie in C programmieren können, ist es leicht, eine neue eingebaute Funktion oder ein Modul dem Interpreter hinzuzufügen, entweder um kritische Operationen mit maximaler Geschwindigkeit auszuführen oder um Python-Programme an Bibliotheken zu binden, die nur in Binärform vorliegen (wie z.B. herstellerabhängi-

ge Graphikbibliotheken). Wenn Sie schon einmal dabei sind, können Sie den Python-Interpreter an eine C-Anwendung binden und ihn als Erweiterung oder als Kommandosprache für diese Anwendung verwenden.

Übrigens, die Sprache wurde benannt nach einer BBC-Fernsehserie namens „Monty Python's Flying Circus“ und hat nichts mit bösen Reptilien zu tun. Die Verwendung von Monty Python-Seitenhieben in der Dokumentation ist nicht nur erlaubt, sondern sogar empfohlen.

1.1 Weitere Schritte

Nun, da Sie ganz begeistert sind von Python, werden Sie es mehr im Detail untersuchen wollen. Da der beste Weg, eine Sprache zu lernen, der ist, sie zu benutzen, sind Sie hiermit dazu eingeladen.

Im nächsten Kapitel wird die Benutzung des Interpreters erklärt. Dies ist eher nüchterne Information, aber essentiell, um die späteren Beispiele auszuprobieren.

Der Rest dieses Tutoriums führt anhand von Beispielen in verschiedene Eigenschaften der Sprache Python und der Konzepte dahinter ein, beginnend mit einfachen Ausdrücken, Anweisungen und Datentypen, über Funktionen und Module bis hin zu fortgeschrittenen Konzepten wie Ausnahmen und benutzerdefinierten Klassen.

KAPITEL

ZWEI

Benutzung des Python-Interpreters

2.1 Aufruf des Interpreters

Der Python-Interpreter ist normalerweise als '/usr/local/bin/python' auf den Rechnern installiert, wo er verfügbar ist. Das Verzeichnis '/usr/local/bin' in den Suchpfad der eigenen UNIX-Shell aufzunehmen, erlaubt es, ihn mit dem Kommando

```
python
```

in der Shell aufzurufen. Da die Wahl des Verzeichnisses für den Interpreter eine Option bei der Installation ist, sind andere Plätze auch möglich. Fragen Sie bei Ihrem lokalen Python-Guru oder Systemadministrator nach. (Eine beliebte Alternative ist z.B. '/usr/local/python'.)

Die Eingabe eines Dateiende-Zeichens (engl. EOF) (also `Control-D` unter UNIX, `Control-Z` unter DOS oder Windows) unter dem primären Prompt veranlasst den Interpreter, sich mit einem Exit-Status von Null zu beenden. Falls das bei Ihnen nicht geht, können Sie den Interpreter beenden, indem Sie folgende Kommandos eingeben: '`import sys; sys.exit()`'.

Die Eigenschaften des Interpreters bzgl. der Zeilen-Editierung sind nicht sehr komfortabel. Unter UNIX mag wer immer den Interpreter installiert hat auch die Verwendung der GNU Readline-Bibliothek eingeschaltet haben, die weitergehende interaktive Editier- und Merk-Eigenschaften erlaubt. Der vielleicht schnellste Test, um zu sehen, ob der GNU-Kommandozeilen-Editor unterstützt wird, ist der, `Control-P` gleich nach dem ersten Python-Prompt zu tippen. Sollte es piepen, so haben sie die erweiterten Editier-Eigenschaften (siehe Angang A für eine Einführung in die Tastaturbelegung). Sollte nichts passieren, oder wenn ein `^P` erscheint, sind diese Eigenschaften nicht verfügbar und Sie können nur die Rückschrittaste (engl. backspace) drücken, um vorherige Zeichen in der Zeile zu löschen.

Der Interpreter arbeitet ein wenig wie eine UNIX-Shell: wenn er mit einer Standard-Eingabe aufgerufen wird, die mit einem konsolenähnlichen (tty) Gerät verbunden ist, liest er Kommandos und führt sie interaktiv aus. Wird er mit einem Dateinamen als Argument oder einer Datei als Standard-Eingabe aufgerufen, liest er ein *Skript* aus dieser Datei.

Ein dritter Weg, den Interpreter zu starten, ist '**python -c** *command* `[arg] ...`', wodurch die Anweisungen in *command* ausgeführt werden, analog zur Option **-c** einer Shell. Da Python-Anweisungen oft Leerzeichen und andere Zeichen beinhalten, die für eine Shell von besonderer Bedeutung sind, ist es am besten, man setzt *command* gänzlich in doppelte Anführungszeichen.

Man beachte, dass zwischen '`python file`' und '`python <file`' ein Unterschied besteht. In letzterem Fall werden Eingabeaufforderungen des Programms wie z.B. Aufrufe von `input()` und `raw_input()` von *file* aus bedient. Da diese Datei bereits vom Parser bis zum Ende gelesen wurde, bevor das Programm ausgeführt wird, wird das Programm sofort ein EOF erhalten. Im ersteren Fall (der der Normalfall ist) werden sie von der Datei aus bedient, die mit der Standard-Eingabe des Python-Interpreters verbunden ist.

Wenn eine Skript-Datei verwendet wird, ist es manchmal nützlich, das Skript auszuführen und anschließend in den

interaktiven Modus zu gehen. Dies kann erreicht werden, indem die Option **-i** vor das Skript gesetzt wird. (Das funktioniert nicht, wenn das Skript von der Standard-Eingabe gelesen wird – aus den gleichen Gründen wie im vorherigen Absatz.)

2.1.1 Argumentübergabe

Falls dem Interpreter bekannt, werden der Name des Skriptes und weitere Argumente dahinter dem Skript in der Variablen `sys.argv` übergeben, die eine Liste von Strings ist. Ihre Länge ist mindestens eins. Wenn kein Skript und keine Argumente übergeben werden, ist `sys.argv[0]` ein leerer String. Wenn der Skriptname als `'-'` übergeben wird, d.h. Standard-Eingabe, wird `sys.argv[0]` auf `'-'` gesetzt. Wenn **-c** *command* verwendet wird, wird `sys.argv[0]` auf `'-c'` gesetzt. Optionen, die nach **-c** *command* gefunden werden, werden vom Python-Interpreter selbst nicht behandelt, sondern in `sys.argv` belassen.

2.1.2 Interaktiver Modus

Wenn Kommandos von einem konsolenähnlichen Gerät gelesen werden, so sagt man, dass der Interpreter im *interaktiven Modus* ist. In diesem Modus erwartet er das nächste Kommando nach dem *primären Prompt*, normalerweise drei größer-als-Zeichen (‘»> ’). Das Warten auf Fortsetzungszeilen wird durch den *sekundären Prompt*, standardmäßig bestehend aus drei Punkten, (‘... ’) angezeigt.

Der Interpreter gibt eine Willkommensnachricht mit seiner Versionsnummer und einem Copyright aus, bevor die Ausgabe des primären Prompts erfolgt, z.B. so:

```
python
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06)  [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

Fortsetzungszeilen werden zu Beginn eines Konstruktes benötigt, das aus mehreren Zeilen besteht. Man betrachte etwa folgende `if`-Anweisung:

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

2.2 Der Interpreter und seine Umgebung

2.2.1 Fehlerbehandlung

Tritt ein Fehler auf, so gibt der Interpreter eine Fehlermeldung und einen „stack trace“ aus. Im interaktiven Modus kehrt er dann zum primären Prompt zurück. Wenn die Eingabe von einer Datei kam, terminiert er mit einem von Null verschiedenen Exit-Status, nachdem der „stack trace“ ausgegeben wurde. (Ausnahmen, die von einer `except`-Klausel in einer `try`-Anweisung behandelt werden, sind keine Fehler in diesem Zusammenhang.) Einige Fehler sind ausnahmslos fatal und verursachen eine Terminierung mit Exit-Status ungleich Null. Dies gilt für interne Inkonsistenzen und einige Fälle von nicht ausreichendem Hauptspeicher. Alle Fehlermeldungen werden auf die Standard-Fehlerausgabe geschrieben. Die normale Ausgabe von ausgeführten Kommandos wird auf die Standard-Ausgabe geschrieben.

Die Eingabe eines Unterbrechungszeichens (normalerweise Control-C oder DEL) beim primären oder sekundären Prompt annulliert die Eingabe und bewirkt die Rückkehr zum primären Prompt.[1] Ein Unterbrechungszeichen einzutippen, während ein Kommando ausgeführt wird, löst die `KeyboardInterrupt`-Ausnahme aus, die mit einer `try`-Anweisung behandelt werden kann.

2.2.2 Ausführbare Python-Skripte

Unter BSD-ähnlichen UNIX-Systemen können Python-Skripte direkt ausführbar gemacht werden, genau wie Shell-Skripte, indem die Zeile

```
#! /usr/bin/env python
```

(unter der Annahme, dass der Interpreter in der Variable $PATH ist) zu Beginn des Skriptes gesetzt wird und die Datei ausführbar gemacht wird. Das '`#!`' müssen die ersten zwei Zeichen der Datei sein. Man beachte, dass das Doppelkreuz-Zeichen, '#', in Python zur Markierung des Beginns eines Kommentars verwendet wird.

2.2.3 Die Datei für den interaktiven Interpreter-Start

Wenn Sie Python interaktiv verwenden, ist es oftmals nützlich, einige Standard-Kommandos jedes mal auszuführen, wenn der Interpreter gestartet wird. Man kann dies erreichen, indem eine Umgebungsvariable namens $PYTHONSTARTUP auf den Namen einer Datei mit Ihren Startkommandos gesetzt wird. Dies ist ähnlich zur '.profile'-Datei in UNIX-Shells.

Diese Datei wird nur in interaktiven Sitzungen gelesen, nicht wenn Python Kommandos aus einem Skript liest und auch nicht, wenn '/dev/tty' als explizite Quelle von Kommandos angegeben wird (was einer interaktiven Sitzung gleichkäme). Die Datei wird im gleichen Namensraum ausgeführt, in dem auch interaktive Kommandos ausgeführt werden, so dass Objekte, die es definiert oder importiert, ohne Qualifizierung in der interaktiven Sitzung verwendet werden können (Namensräume und Qualifizierung werden später noch erklärt).

Falls Sie eine weitere Start-Datei vom aktuellen Verzeichnis lesen wollen, können Sie dies in der globalen Start-Datei programmieren, z.B. '`if os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py')`'. Wenn Sie die Start-Datei in einem Skript verwenden wollen, müssen Sie dies im Skript explizit tun:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    execfile(filename)
```

[1] Ein Problem mit dem GNU Readline-Paket kann dies verhindern.

KAPITEL

DREI

Eine informelle Einführung in Python

In den folgenden Beispielen unterscheiden sich Ein- und Ausgabe durch die An- oder Abwesenheit von Prompts (‘»> ’ und ‘... ’). Um das jeweilige Beispiel zu wiederholen, müssen Sie alles hinter dem Prompt eingeben, wenn ein solcher da ist. Zeilen ohne Prompt stammen vom Interpreter. Man beachte, dass ein sekundärer Prompt (‘...’) in einer Zeile allein bedeutet, dass Sie eine Leerzeile eingeben müssen. Dies ist erforderlich, um ein Kommando über mehrere Zeilen zu beenden.

Viele der Beispiele in diesem Tutorium, selbst die, die interaktiv eingegeben werden, beinhalten Kommentare. Kommentare in Python beginnen mit einem Doppelkreuz-Zeichen, ‘#’, und gelten bis zum Zeilenende. Ein Kommentar kann am Zeilenanfang auftreten, oder nach Leerzeichen oder Code, aber nicht in einem String-Literal. Ein Doppelkreuz-Zeichen in einem String-Literal ist lediglich ein Doppelkreuz.

Einige Beispiele:

```
# Dies ist der erste Kommentar.
SPAM = 1                 # Dies ist der zweite.
                         # ... Und nun ein dritter!
STRING = "# Dies ist kein Kommentar."
```

3.1 Verwendung von Python als Taschenrechner

Probieren wir ein paar einfache Python-Kommandos aus. Starten Sie den Interpreter und warten Sie bis zum primären Prompt, ‘»>’! (Es sollte nicht lange dauern.)

3.1.1 Zahlen

Der Interpreter verhält sich wie ein einfacher Taschenrechner: man kann einen Ausdruck eingeben und er wird dessen Wert ausgeben. Die Syntax von Ausdrücken ist simpel: die Operatoren `+`, `-`, `*` und `/` funktionieren wie in den meisten anderen Sprachen (z.B. Pascal oder C). Klammern können zur Gruppierung verwendet werden.

```
>>> 2+2
4
>>> # Dies ist ein Kommentar.
... 2+2
4
>>> 2+2  # Und ein Kommentar in einer Code-Zeile.
4
>>> (50-5*6)/4
5
>>> # Integer-Division ergibt die untere Ganzzahl:
... 7/3
2
>>> 7/-3
-3
```

Wie in C wird das Gleich-Zeichen (‘=’) benutzt, um einen Wert einer Variablen zuzuweisen. Der Wert einer Zuweisung wird nicht ausgegeben:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Ein Wert kann mehreren Variablen gleichzeitig zugewiesen werden:

```
>>> x = y = z = 0  # Setze x, y und z auf Null.
>>> x
0
>>> y
0
>>> z
0
```

Es gibt volle Unterstützung für Fließkommazahlen. Operatoren mit gemischten Operandentypen konvertieren den Ganzzahl-Operanden nach Fließkomma [1]:

```
>>> 4 * 2.5 / 3.3
3.0303030303
>>> 7.0 / 2
3.5
```

Komplexe Zahlen werden auch unterstützt. Imaginäre Zahlen werden mit dem Suffix ‘`j`’ oder ‘`J`’ ausgegeben. Komplexe Zahlen mit einem Realteil ungleich Null werden als ‘`(`*real*`+`*imag*`j)`’ ausgegeben und können mit der Funktion ‘`complex(`*real*`, `*imag*`)`’ erzeugt werden.

[1] Als Trennzeichen wird ein Punkt verwendet, kein Komma. Diese Besonderheit der englischen Sprache findet man bei allen Programmiersprachen und sogar bei fast allen Taschenrechnern. [Der Übersetzer]

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0, 1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Komplexe Zahlen werden immer als zwei Fließkommazahlen dargestellt, dem realen und dem imaginären Teil. Um diese Teile von einer komplexen Zahl *z* zu extrahieren, verwende man *z*`.real` und *z*`.imag`.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Die Konvertierungsfunktionen nach Fließkomma- und Ganzzahl (`float()`, `int()` und `long()`) funktionieren nicht für komplexe Zahlen – es gibt keine korrekte Möglichkeit, eine komplexe Zahl in eine reelle zu konvertieren. Man verwende `abs(`*z*`)`, um ihren Betrag (als Fließkommazahl) oder `z.real`, um ihren Realteil zu erhalten.

```
>>> a=1.5+0.5j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g. abs(z)
>>> a.real
1.5
>>> abs(a)
1.58113883008
```

Im interaktiven Modus wird der zuletzt ausgegebene Ausdruck der Variablen `_` zugewiesen. Das bedeutet, wenn man Python als Taschenrechner benutzt, ist es etwas einfacher, Berechnungen weiterzuführen, z.B.:

```
>>> tax = 17.5 / 100
>>> price = 3.50
>>> price * tax
0.6125
>>> price + _
4.1125
>>> round(_, 2)
4.11
```

Diese Variable sollte von der Benutzerin nur lesend verwendet werden. Weisen Sie ihr keinen Wert explizit zu – Sie würden eine unabhängige lokale Variable mit dem gleichen Namen erzeugen, die die eingebaute mit ihrem speziellen Verhalten ausblendet.

3.1.2 Strings

Abgesehen von Zahlen kann Python auch Strings (Zeichenketten) manipulieren, die auf verschiedene Weise ausgedrückt werden können. Sie können in einfachen oder doppelten Anführungszeichen stehen.

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

String-Literale können sich auf verschiedene Arten über mehrere Zeilen erstrecken. Zeilenvorschübe können mit Rückwärts-Schrägstrichen maskiert werden, z.B.:

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
    Note that whitespace at the beginning of the line is\
 significant.\n"
print hello
```

was folgendes ausgeben würde:

```
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant.
```

Strings können auch innerhalb eines Paares von dreifachen Anführungszeichen """ oder ''' auftreten. Das Zeilenende (engl. EOL) muss nicht maskiert werden, wenn man dreifache Anführungszeichen verwendet, es taucht jedoch im String selbst auf:

```
print """
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
```

erzeugt die folgende Ausgabe:

```
Usage: thingy [OPTIONS]
   -h                        Display this usage message
   -H hostname               Hostname to connect to
```

Der Interpreter gibt das Resultat von String-Operationen genauso aus, wie es für die Eingabe verwendet würde: in Anführungszeichen und unter Voranstellung von Rückwärts-Schrägstrichen für Anführungszeichen und andere lustige Zeichen, um den exakten Wert anzuzeigen. Ein String steht in doppelten Anführungszeichen, wenn er einzelne, aber keine doppelten Anführungszeichen enthält, sonst steht er in einzelnen Anführungszeichen. Die später beschriebene `print`-Anweisung kann benutzt werden, um Strings ohne Anführungszeichen oder Maskierungen auszugeben.

Strings können mit dem Operator `+` aneinandergefügt (zusammengeklebt) und mit `*` vervielfältigt werden.

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Zwei String-Literale nebeneinander werden automatisch aneinandergefügt. Die obere Zeile hätte man auch schreiben können als '`word = 'Help' 'A''`. Das funktioniert nur mit zwei String-Literalen, nicht mit beliebigen String-Ausdrücken:

```
>>> import string
>>> 'str' 'ing'                   #  <-  Das ist ok.
'string'
>>> string.strip('str') + 'ing'   #  <-  Das ist ok.
'string'
>>> string.strip('str') 'ing'     #  <-  Das ist ungueltig.
  File "<stdin>", line 1
    string.strip('str') 'ing'
                        ^
SyntaxError: invalid syntax
```

Strings können indiziert werden. Wie in C hat das erste Zeichen den Index 0. Es gibt keinen eigenen Datentyp für ein einzelnes Zeichen. Ein Zeichen ist einfach ein String der Länge eins. Wie in der Sprache Icon können Teilstrings mit der *Teilbereichs-Notation* spezifiziert werden: zwei Indizes, getrennt durch einen Doppelpunkt (das Zeichen *am* zweiten Index gehört nicht mehr dazu).

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Im Gegensatz zu Strings in C können Strings in Python nicht verändert werden. Die Zuweisung an eine durch einen Index angegebene Position im String führt zu einem Fehler:

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Es ist jedoch sehr einfach und effizient, einen neuen String mit dem kombinierten Inhalt zu erzeugen:

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4:]
'SplatA'
```

Teilbereiche haben sinnvolle Voreinstellungen. Ein ausgelassener erster Index wird als Null, ein ausgelassener zweiter als die Länge des Strings interpretiert.

```
>>> word[:2]    # Die ersten beiden Zeichen.
'He'
>>> word[2:]    # Alles ausser den ersten beiden Zeichen.
'lpA'
```

Dies ist eine nützliche Invariante von Teilbereichs-Operationen: `s[:i] + s[i:]` ergibt `s`.

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Ungültige Teilbereiche werden anständig behandelt: ein Index, der zu groß ist, wird mit der Stringlänge ersetzt, eine obere Grenze, die kleiner als die untere ist, ergibt einen leeren String.

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Indizes dürfen negativ sein. Es wird dann vom rechten Rand des Strings aus gezählt, z.B.:

```
>>> word[-1]     # Das letzte Zeichen.
'A'
>>> word[-2]     # Das vorletzte Zeichen.
'p'
>>> word[-2:]    # Die letzten beiden Zeichen.
'pA'
>>> word[:-2]    # Alles ausser den beiden letzten Zeichen.
'Hel'
```

Vorsicht aber bei -0: es ist identisch mit 0, zählt also nicht von rechts!

```
>>> word[-0]     # (Da -0 gleich 0 ist.)
'H'
```

Negative Bereichsindizes, die außerhalb des erlaubten Bereichs liegen, werden abgeschnitten, aber versuchen Sie das nicht bei ein-elementigen (Nicht-Bereichs-) Indizes:

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # Fehler.
Traceback (most recent call last):
  File "<stdin>", line 1
IndexError: string index out of range
```

Am besten kann man sich merken, wie Teilbereiche funktionieren, indem man sich denkt, daß die Randindizes *zwischen* die Zeichen zeigen, wobei die linke Kante des ersten Zeichens mit 0 bezeichnet wird. Dann hat die rechte Kante des letzten Zeichens von *n* Zeichen den Index *n*, z.B.:

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

Die obere Zahlenreihe bezeichnet die Positionen der Indizes 0...5 im String, die untere entspricht den jeweiligen negativen Indizes. Der Bereich von *i* bis *j* besteht aus allen Zeichen zwischen den mit *i* und *j* markierten Kanten.

Für nicht-negative Indizes ist die Länge des Teilbereichs gleich der Differenz der Indizes, falls beide innerhalb der erlaubten Grenzen liegen. So ist z.B. die Länge von `word[1:3]` gleich 2.

Die eingebaute Funktion `len()` ergibt die Länge eines Strings:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

3.1.3 Unicode-Strings

Ab Python 2.0 steht dem Programmierer ein neuer Datentyp zur Speicherung von Text zur Verfügung: das Unicode-Objekt. Man kann es benutzen, um Unicode-Daten zu speichern und zu manipulieren (siehe http://www.unicode.org). Außerdem harmoniert es gut mit existierenden String-Objekten, wobei, wenn notwendig, automa-

tisch eine Konvertierung vorgenommen wird.

Unicode hat den Vorteil, jedem Zeichen eines modernen oder alten Schriftsatzes eine eindeutige Nummer zuzuordnen. Bislang waren nur 256 Nummern für Schriftzeichen möglich, und Texte waren normalerweise an eine Code-Seite gebunden, die die Nummern auf Schriftzeichen abbildete. Dies führte zu sehr großer Verwirrung besonders bei der sogenannten „Internalization“ (häufig geschrieben als ‘`i18n`’ — ‘`i`’ + 18 Zeichen + ‘`n`’) von Software, d.h. der Anpassung an die jeweilige Landessprache. Unicode löst diese Probleme, indem es eine Code-Seite für alle Schriftsätze definiert.

Unicode-Strings in Python zu erzeugen ist ebenso einfach wie bei normalen Strings:

```
>>> u’Hello World!’
u’Hello World!’
```

Der kleine Buchstabe ‘`u`’ vor dem Anführungszeichen gibt an, dass ein Unicode-String erzeugt werden soll. Wenn der String Sonderzeichen enthalten soll, so kann man dies mit *Unicode-Maskierungscodes* in Python erreichen. Folgendes Beispiel zeigt, wie:

```
>>> u’Hello\u0020World!’
u’Hello World!’
```

Die Maskierungssequenz (engl. escape sequence) `\u0020` gibt an, dass das Unicode-Zeichen mit der Ordnungszahl 0x0020 (das Leerzeichen) an der angegeben Position stehen soll.

Andere Zeichen werden interpretiert, indem ihre jeweiligen Nummernwerte direkt als Unicode-Nummer verwendet werden. Wenn Sie mit String-Literalen in der Standard-Codierung Latin-1 umgehen, die in vielen westlichen Ländern benutzt wird, werden Sie es schätzen, dass die unteren 256 Unicode-Zeichen identisch sind mit den 256 Zeichen der Standard-Codierung Latin-1.

Den Experten sei gesagt, dass es auch einen rohen Modus wie jenen für normale Strings gibt. Man muss dem String ein kleines ’r’ voranstellen, damit Python die *Roh-Unicode-Maskierung* verwendet. Dann wird die obige `\uXXXX`-Konvertierung nur dann angewendet, wenn eine ungerade Anzahl von Rückwärtsschrägstrichen vor dem kleinen ’u’ steht.

```
>>> ur’Hello\u0020World !’
u’Hello World !’
>>> ur’Hello\\u0020World !’
u’Hello\\\\u0020World !’
```

Der Roh-Modus ist dann besonders nützlich, wenn man eine Menge von Rückwärtsschrägstrichen eingeben muss, z.B. in regulären Ausdrücken.

Abgesehen von diesen Standard-Codierungen bietet Python eine ganze Reihe anderer Möglichkeiten, um aus einer bekannten Codierung Unicode-Strings zu erzeugen.

Die eingebaute Funktion `unicode()` bietet einen Zugang zu allen registrierten Unicode-Codecs (COdier-er und DECodierer). Einige der bekannteren Codierungen, die von diesen Codecs konvertiert weren können, sind *Latin-1*, *ASCII*, *UTF-8* und *UTF-16*. Die beiden letzten sind Codierungen mit variabler Länge, die jedes Unicode-Zeichen in einem oder mehreren Bytes speichern. Die voreingestellte Codierung ist normalerweise auf ASCII gesetzt, die Zeichen von 0 bis 127 weitergibt und alle anderen Zeichen mit einem Fehler zurückweist. Wenn ein Unicode-String ausgegeben, in eine Dateien abspeichert oder mit `str()` konvertiert wird, findet die Konvertierung mit dieser voreingestellten Codierung statt.

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"äöü"
u'\xe4\xf6\xfc'
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
```

Um einen Unicode-String in einen 8-Bit String mit einer bestimmten Codierung zu konvertieren, stellen Unicode-Objekte die Methode `encode()` zur Verfügung, die ein Argument erwartet, den Namen der Codierung. Dabei werden Namen mit kleinen Buchstaben bevorzugt.

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

Wenn Ihre Daten in einer bestimmten Codierung vorliegen, und Sie wollen daraus den entsprechenden Unicode-String erzeugen, können Sie dazu die Funktion `unicode()` mit dem Namen der Codierung als zweitem Argument verwenden.

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

3.1.4 Listen

Python kennt eine Anzahl von *zusammengesetzten* Datentypen, die verwendet werden, um andere Werte zu gruppieren. Der vielseitigste ist die *Liste*, die als Liste von durch Kommata getrennten Werten (Elementen) innerhalb von eckigen Klammern steht. Listenelemente brauchen nicht alle vom gleichen Typ zu sein.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Genau wie String-Indizes beginnen Listen-Indizes bei 0, und man kann Teilbereiche von Listen bilden, Listen zusammenfügen, und so weiter:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boe!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boe!']
```

Im Gegensatz zu Strings, die *unveränderlich* sind, ist es möglich, einzelne Elemente von Listen zu ändern:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Die Zuweisung an Teilbereiche ist ebenso möglich, und das kann sogar die Länge der Liste verändern:

```
>>> # Ersetze einige Elemente:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Entferne einige:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Fuege einige ein:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> a[:0] = a     # Fuege (eine Kopie von) sich selbst am Anfang ein.
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
```

Die eingebaute Funktion `len()` gilt auch für Listen:

```
>>> len(a)
8
```

Es ist möglich, Listen zu verschachteln (Listen zu erzeugen, die andere Listen enthalten), z.B.:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # Siehe Abschnitt 5.1.
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Man beachte, dass `p[1]` und `q` wirklich dasselbe Objekt bezeichnen! Wir werden auf die *Objekt-Semantik* später zurückkommen.

3.2 Erste Schritte in Richtung Programmierung

Natürlich können wir Python für kompliziertere Dinge verwenden, als nur, um zwei und zwei zu addieren. Zum Beispiel können wir einen ersten Teil der *Fibonacci*-Folge wie folgt schreiben:

```
>>> # Fibonacci-Reihe:
... # Die Summe zweier Elemente ergibt das naechste.
... a, b = 0, 1
>>> while b < 10:
...       print b
...       a, b = b, a+b
...
1
1
2
3
5
8
```

Dieses Beispiel demonstriert mehrere neue Eigenschaften.

- Die erste Zeile enthält eine *Mehrfachzuweisung*: die Variablen `a` und `b` erhalten gleichzeitig die neuen Werte 0 und 1. In der letzten Zeile wird das nochmals gemacht, um zu zeigen, dass die Ausdrücke auf der rechten Seite alle ausgewertet werden, bevor eine Zuweisung stattfindet. Die Ausdrücke auf der rechten Seite werden von links nach rechts ausgewertet.
- Die `while`-Schleife wird durchlaufen, solange die Bedingung (hier: `b < 10`) wahr bleibt. In Python ist, genau wie in C, jede nicht-negative ganze Zahl wahr; Null ist falsch. Die Bedingung darf auch ein String oder eine Liste sein, ja sogar jede beliebige Sequenz; alles mit einer Länge verschieden von Null ist wahr, leere Sequenzen sind falsch. Der in diesem Beispiel verwendete Test ist ein einfacher Vergleich. Die üblichen Vergleichsoperatoren werden genau wie in C geschrieben: `<` (kleiner-als), `>` (größer-als), `==` (gleich), `<=` (kleiner-gleich), `>=` (größer-gleich) und `!=` (ungleich).
- Der *Rumpf* der Schleife ist *eingerückt*: Einrückung ist Pythons Art, Anweisungen zu gruppieren. Python stellt (noch!) keine intelligenten Mechanismen zum Editieren von Eingabezeilen zur Verfügung, so dass man die

Tabulatortaste oder Leerzeichen für jede eingerückte Zeile eingeben muss. In der Praxis werden Sie komplizierte Eingaben für Python mit einem Text-Editor eingeben, von denen die meisten eine automatische Einrückung bieten. Wenn eine zusammengesetzte Anweisung interaktiv eingegeben wird, muss sie durch eine Leerzeile abgeschlossen werden, um das Ende anzuzeigen (da der Parser nicht raten kann, wann Sie die letzte Zeile eingegeben haben). Man beachte, dass alle Zeilen eines Blocks um den gleichen Betrag eingerückt sein müssen.

- Die `print`-Anweisung gibt den Wert (oder die Werte) des Ausdrucks (oder der Ausdrücke) aus, den (oder die) es bekommt. Sie unterscheidet sich von der Ausgabe eines reinen Ausdrucks, den Sie haben wollen (wie wir es vorher im Taschenrechner-Beispiel gemacht haben), in der Art wie sie Mehrfach-Ausdrücke und Strings behandelt. Strings werden ohne Anführungszeichen ausgegeben, und es wird ein Leerzeichen zwischen die Elemente gesetzt, damit man die Dinge schön formatieren kann, etwa so:

```
>>> i = 256*256
>>> print 'The value of i is', i
The value of i is 65536
```

Ein Komma am Zeilenende verhindert nach der Ausgabe den Sprung in die neue Zeile:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Man beachte, dass der Interpreter eine neue Zeile einfügt, bevor er den nächsten Prompt ausgibt, falls die letzte Zeile nicht leer war.

KAPITEL

VIER

Weitere Kontrollstrukturen

Neben der eben eingeführten `while`-Anweisung kennt Python die bei anderen Sprachen üblichen Kontrollfluss-Anweisungen – mit einigen Tricks.

4.1 `if`-Anweisungen

Die vielleicht bekannteste Anweisung ist die `if`-Anweisung, z.B.:

```
>>> x = int(raw_input("Bitte geben Sie eine Zahl ein: "))
>>> if x < 0:
...      x = 0
...      print 'Negative changed to zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Single'
... else:
...      print 'More'
...
```

Es kann keine oder mehrere `elif`-Teile geben, und der `else`-Teil ist optional. Das Schlüsselwort '`elif`' ist eine Kurzschreibweise für „else if" und ist nützlich, um exzessive Einrückungen zu vermeiden. Eine Sequenz der Art `if` … `elif` … `elif` … ist ein Ersatz für `switch`- oder `case`-Anweisungen in anderen Sprachen.

4.2 `for`-Anweisungen

Die `for`-Anweisung in Python unterscheidet sich ein wenig von dem, was Sie aus C oder Pascal gewohnt sind. Anstatt stets über eine arithmetische Folge von Zahlen zu iterieren (wie in Pascal) oder der Benutzerin die Möglichkeit zu geben, die Schrittweite und Endbedingung der Iteration zu definieren (wie in C), iteriert die `for`-Anweisung in Python über die Elemente einer Sequenz (z.B. einer Liste oder eines Strings) in der Reihenfolge ihres Auftretens in dieser Sequenz. Beispiel:

```
>>> # Messe einige Strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

Es ist nicht sicher, die Sequenz zu modifizieren, über die gerade iteriert wird (das kann nur mit veränderlichen Sequenztypen passieren, z.B. Listen). Wenn Sie die Liste modifizieren müssen, über die Sie iterieren, z.B. um die einzelnen Elemente zu kopieren, müssen Sie über eine Kopie iterieren. Mit der Teilbereichs-Notation ist dies sehr bequem:

```
>>> for x in a[:]: # Erstelle eine Teilbereichs-Kopie der ganzen Liste.
...    if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3 Die `range()`-Funktion

Falls Sie über eine Sequenz von Zahlen iterieren müssen, kommt die eingebaute Funktion `range()` sehr gelegen. Sie erzeugt Listen, die arithmetischen Aufzählungen entsprechen, z.B.:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Der angegebene Endpunkt ist nie Teil der erzeugten Liste. `range(10)` erzeugt eine Liste von 10 Werten, exakt die gültigen Indizes für Elemente einer Sequenz der Länge 10. Es ist möglich, den Bereich bei einer anderen Zahl anfangen zu lassen oder eine andere ganzzahlige Schrittweite anzugeben (sogar eine negative):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Um die Indizes einer Sequenz zu durchlaufen, kombiniere man `range()` und `len()` wie folgt:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

4.4 `break`- und `continue`-Anweisungen und `else`-Klauseln in Schleifen

Die `break`-Anweisung bricht, wie in C, aus der kleinsten umgebenden `for`- oder `while`-Schleife aus.

Die `continue`-Anweisung, auch aus C geliehen, setzt die Schleife mit der nächsten Iteration fort.

Schleifen-Anweisungen dürfen eine `else`-Klausel haben. Sie wird ausgeführt, wenn die Schleife vollständig durch die Liste gelaufen ist (mit `for`) oder wenn die Bedingung falsch wird (mit `while`), aber nicht, wenn die Schleife durch eine `break`-Anweisung terminiert wird. Dies wird bei der folgenden Schleife deutlich, die nach Primzahlen sucht:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...            print n, 'equals', x, '*', n/x
...            break
...     else:
...          print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

4.5 `pass`-Anweisungen

Die `pass`-Anweisung tut nichts. Sie kann benutzt werden, wenn eine Anweisung syntaktisch notwendig ist, ohne dass das Programm wirklich etwas tun muss. Beispiel:

```
>>> while 1:
...       pass # Aktives Warten auf eine Tastatur-Unterbrechung.
...
```

4.6 Definition von Funktionen

Wir können eine Funktion schreiben, die die Fibonacci-Folge bis zu einer beliebigen Grenze ausgibt:

```
>>> def fib(n):    # Schreibe Fibonacci-Reihe bis n.
...     "Gib Fibonacci-Reihe bis n aus."
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Rufe nun die gerade definierte Funktion auf:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Das Schlüsselwort `def` leitet die *Definition* einer Funktion ein. Ihm muss der Funktionsname folgen sowie eine Liste von formalen Parametern in runden Klammern sowie ein Doppelpunkt. Die Anweisungen, die den Rumpf der Funktion ausmachen, beginnen in der folgenden Zeile und müssen eingerückt sein. Die erste Anweisung des Rumpfes kann optional ein String-Literal sein. Dieses String-Literal ist der Dokumentations-String der Funktion, auch *Docstring* genannt.

Es gibt Werkzeuge, die Docstrings verwenden, um automatisch online oder gedruckte Dokumentation zu erzeugen oder um die Benutzerin interaktiv im Code stöbern zu lassen. Es ist von Vorteil, Docstrings in Ihrem Code zu verwenden. Versuchen Sie, es sich zur Gewohnheit zu machen!

Die *Ausführung* einer Funktion führt zu einer neuen Symboltabelle mit den lokalen Variablen dieser Funktion. Genauer: alle Variablenzuweisungen in einer Funktion speichern den Wert in der lokalen Symboltabelle, wohingegen Referenzen auf Variablen erst in der lokalen Symboltabelle gesucht werden, dann in der globalen Symboltabelle und dann in der Tabelle der eingebauten Namen. Daher kann in einer Funktion globalen Variablen nicht direkt ein Wert zugewiesen werden (solange sie nicht in in einer `global`-Anweisung vorkommen), obwohl sie referenziert werden können.

Die eigentlichen Parameter (Argumente) eines Funktionsaufrufs werden dann in die lokale Symboltabelle der aufrufenden Funktion eingefügt, wenn die Funktion aufgerufen wird. Daher werden die Argumente mit *Werte-Semantik* (engl. *call by value*) übergeben (wobei der *Wert* immer eine Objekt-*Referenz* und nicht der Wert des Objektes ist).[1] Wenn eine Funktion eine andere Funktion aufruft, wird eine neue lokale Symboltabelle für diesen Aufruf erzeugt. [2]

Eine Funktionsdefinition trägt den Namen der Funktion in die aktuelle Symboltabelle ein. Der Wert des Funktionsnamens hat einen Typ, der vom Interpreter als eine benutzerdefinierte Funktion erkannt wird. Dieser Wert kann einem anderen Namen zugewiesen werden, der dann auch als Funktion verwendet werden kann. Dies kann als allgemeiner Mechanismus zur Umbenennung von Funktionen dienen:

```
>>> fib
<function object at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

Sie könnten einwenden, dass `fib` keine Funktion, sondern eine Prozedur ist. In Python wie in C sind Prozeduren lediglich Funktionen, die keinen Wert zurück geben. Tatsächlich ist es so, technisch gesehen, dass Prozeduren sehr

[1] Tatsächlich wäre *Objekt-Referenz-Semantik* (engl. *call by object reference*) eine bessere Beschreibung, da wenn ein veränderliches Objekt übergeben wird, die aufrufende Funktion alle von der aufgerufenen Funktion daran vorgenommen Änderungen sehen wird, z.B. Elemente, die in einer Liste eingefügt wurden.

[2] Die aufgerufene Funktion kann *nicht* auf die lokalen Symbole der aufrufenden Funktion zugreifen, wenn diese nicht als Parameter übergeben werden. [Der Übersetzer]

wohl einen Wert zurück geben, wenn auch einen langweiligen. Dieser Wert heißt `None` (ein eingebauter Name). Die Ausgabe des Wertes `None` wird normalerweise vom Interpreter unterdrückt, falls es der einzige Ausgabewert ist. Wenn man wirklich will, kann man ihn dennoch sehen:

```
>>> print fib(0)
None
```

Es ist einfach, eine Funktion zu schreiben, die eine Liste von Zahlen der Fibonacci-Folge zurück gibt, anstatt sie direkt auszudrucken:

```
>>> def fib2(n): # Gib Fibonacci-Reihe bis n aus.
...     "Gib eine Liste mit der Fibonacci-Reihe bis n zurueck."
...     result = []
...     a, b = 0, 1
...     while b < n:
...         result.append(b)    # Siehe unten.
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # Rufe sie auf.
>>> f100                # Gib das Ergebnis aus.
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Dieses Beispiel demonstriert, wie üblich, einige neue Eigenschaften von Python:

- Die `return`-Anweisung verlässt eine Prozedur mit der Rückgabe eines Wertes. `return` ohne einen Ausdruck als Argument gibt sie `None` zurück. Beim Erreichen des Endes einer Prozedur wird ebenfalls `None` zurück gegeben.
- Die Anweisung `result.append(b)` ruft eine *Methode* des Listen-Objektes `result` auf. Eine Methode ist eine Funktion, die zu einem Objekt ‘gehört’ und `obj.methodname` heißt, wobei `obj` irgendein Objekt ist (was auch ein Ausdruck sein kann), und `methodname` der Name einer Methode, die vom Objekt-Typ definiert wird. Verschiedene Typen definieren verschiedene Methoden. Methoden verschiedener Typen dürfen den gleichen Namen haben ohne dass eine Doppeldeutigkeit entstünde. (Es ist möglich, mit Hilfe von *Klassen* eigene Objekt-Typen und -Methoden zu definieren, wie später in diesem Tutorium noch diskutiert wird.) Die Methode `append()` im obigen Beispiel ist für ListenObjekte definiert. Sie fügt ein neues Element ans Ende einer Liste hinzu. In diesem Beispiel ist das äquivalent zu ‘`result = result + [b]`’, aber effizienter.

4.7 Mehr über die Definition von Funktionen

Es ist auch möglich, Funktionen mit einer variablen Anzahl von Argumenten zu definieren. Es gibt drei Formen, die kombiniert werden können:

4.7.1 Argumente mit Vorgabewerten

Die nützlichste Form besteht darin, einen Vorgabewert für ein oder mehrere Argumente zu spezifizieren.

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while 1:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'): return 1
        if ok in ('n', 'no', 'nop', 'nope'): return 0
        retries = retries - 1
        if retries < 0: raise IOError, 'refusenik user'
        print complaint
```

Diese Funktion kann entweder so aufgerufen werden: `ask_ok('Wollen Sie wirklich beenden?')` oder so: `ask_ok('Ist es OK, die Datei zu überschreiben?', 2)`.

Die Vorgabewerte werden im *definierenden* Geltungsbereich zum Zeitpunkt der Funktionsdefinition ausgewertet. So wird z.B.

```
i = 5
def f(arg = i): print arg
i = 6
f()
```

den Wert `5` ausgeben.

Wichtiger Hinweis: Ein Vorgabewert wird nur einmal ausgewertet. Das macht dann einen Unterschied, wenn es sich um ein veränderliches Objekt handelt, etwa eine Liste oder ein Dictionary. Die folgende Funktion zum Beispiel sammelt die ihr übergebenen Argumente in aufeinander folgenden Aufrufen:

```
def f(a, l = []):
    l.append(a)
    return l
print f(1)
print f(2)
print f(3)
```

Das wird folgendes ausgeben:

```
[1]
[1, 2]
[1, 2, 3]
```

Wenn Sie nicht wollen, dass aufeinander folgende Aufrufe sich das veränderliche Objekt teilen, können Sie stattdessen die folgende Funktion verwenden:

```
def f(a, l = None):
    if l is None:
        l = []
    l.append(a)
    return l
```

4.7.2 Schlüsselwort-Argumente

Funktionen können auch mit Schlüsselwort-Argumenten in der Form '*keyword = value*' aufgerufen werden. Diese Funktion zum Beispiel,

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "Volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

könnte in allen folgenden Varianten aufgerufen werden:

```
parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

aber diese Aufrufe wären alle ungültig:

```
parrot()                     # Notwendiges Argument fehlt.
parrot(voltage=5.0, 'dead')  # Schluesselwort-Argument folgt Schluesselwort.
parrot(110, voltage=220)     # Doppelter Wert fuer ein Argument.
parrot(actor='John Cleese')  # Unbekanntes Schluesselwort.
```

Allgemein gilt, dass Positions-Argumente vor Schlüssel-Argumenten in einer Argumentliste stehen müssen, wobei die Schlüssel aus den formalen Parameternamen gewählt werden müssen. Es ist unwichtig, ob ein formaler Parameter einen Vorgabewert hat oder nicht. Keinem Argument darf ein Wert mehr als einmal zugewiesen werden – Namen von formalen Parametern, die mit Positions-Argumenten korrespondieren, können nicht als Schlüssel im gleichen Aufruf verwendet werden. Im folgenden Beispiel wird wegen dieser Einschränkung ein Fehler ausgelöst:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: keyword parameter redefined
```

Wenn ein letzter formaler Parameter der Form ***name* existiert, so wird ihm ein Dictionary zugewiesen, der alle Schlüssel-Argumente enthält, deren Schlüssel nicht mit einem formalen Parameter korrespondiert. Das darf kombiniert werden mit einem formalen Parameter der Form **name* (beschrieben im nächsten Unterabschnitt), der ein Tupel mit den Positions-Argumenten zugewiesen bekommt, die zusätzlich zu denen der formalen Parameterliste existieren. (**name* muss vor ***name* stehen.) Wenn wir z.B. die folgende Funktion definieren:

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, '?'
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments: print arg
    print '-'*40
    for kw in keywords.keys(): print kw, ':', keywords[kw]
```

könnte sie wie folgt aufgerufen werden:

```
cheeseshop('Limburger', "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           client='John Cleese',
           shopkeeper='Michael Palin',
           sketch='Cheese Shop Sketch')
```

und natürlich bestünde die Ausgabe in:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

4.7.3 Variable Argumentlisten

Schließlich besteht die am wenigsten benutzte Möglichkeit darin, dass eine Funktion mit einer beliebigen Anzahl von Argumenten aufgerufen werden kann. Diese Argumente werden in einem Tupel zusammengefasst. Vor der variablen Anzahl von Argumenten können beliebig viele normale Argumente stehen (auch keine).

```
def fprintf(file, format, *args):
    file.write(format % args)
```

4.7.4 Lambda-Formen

Wegen der großen Nachfrage wurden einige der üblichen Eigenschaften von funktionalen Programmiersprachen in Python aufgenommen. Mit dem `lambda`-Schlüsselwort können kleine, anonyme Funktionen erzeugt werden. Dies ist eine Funktion, die die Summe zweier Argumente zurück gibt: '`lambda a, b: a+b`'. Lambda-Formen können verwendet werden, wann immer Funktions-Objekte benötigt werden. Syntaktisch sind sie beschränkt auf einen einzigen Ausdruck. Semantisch sind sie nichts als ein syntaktisches Zuckerl für eine normale Funktionsdefinition. Wie bei verschachtelten Funktionsdefinitionen können lambda-Formen keine Variablen des äußeren Geltungsbereichs referenzieren, aber das kann durch vernünftige Verwendung von Argumenten mit Vorgabewerten umgangen werden, z.B.:

```
>>> def make_incrementor(n):
...     return lambda x, incr=n: x+incr
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
>>>
```

4.7.5 Dokumentations-Strings

Allmählich formieren sich Konventionen über den Inhalt und das Format von Dokumentations-Strings.

Die erste Zeile sollte immer eine sehr knappe Zusammenfassung der Aufgabe des Objektes sein. Um es kurz zu machen, sollte sie nicht explizit den Namen oder Typ des Objektes enthalten, da diese mit anderen Mitteln zur Verfügung stehen (außer, wenn der Name ein Verb ist, das die Operation einer Funktion beschreibt). Diese Zeile sollte mit einem Großbuchstaben beginnen und mit einem Punkt enden.

Falls der Docstring noch mehr Zeilen enthält, sollte die zweite Zeile leer sein, um eine sichtbare Trennung von Zusammenfassung und Rest der Beschreibung vorzunehmen. Die folgenden Zeilen sollten ein oder mehrere Absätze sein, die Aufrufkonventionen, Seiteneffekte, etc. des Objektes beschreiben.

Der Python-Parser belässt die Einrückung von mehrzeiligen String-Literalen, d.h. dass Dokumentations-Werkzeuge diese Einrückung selbst entfernen müssen. Dies kann mit den folgenden Konventionen geschehen: Die erste nicht-leere Zeile *nach* der ersten Zeile des Strings bestimmt das Maß der Einrückung für den gesamten Docstring. (Wir können die erste Zeile nicht auswerten, da sie normalerweise direkt nach den – eingerückten – öffnenden Anführungszeichen kommt, so dass ihre Einrückung im String-Literal nicht ersichtlich ist.) Leerzeichen, die „äquivalent“ zu dieser Einrückung sind, werden dann vom linken Rand aller Zeilen entfernt. Zeilen, die weniger eingerückt sind, sollten nicht vorkommen, aber wenn doch, sollte der gesamte linke Leerraum entfernt werden. Der Test auf Leerraum sollte nach Expansion von Tabulatoren (meist auf acht Zeichen) erfolgen.

Hier ein Beispiel für einen mehrzeiligen Docstring:

```
>>> def my_function():
...     """Tut nichts, aber dokumentiert dies auch.
...
...     Nein, es tut wirklich nichts.
...     """
...     pass
...
>>> print my_function.__doc__
Tut nichts, aber dokumentiert dies auch.

    Nein, es tut wirklich nichts.
```

KAPITEL

FÜNF

Datenstrukturen

Dieses Kapitel beschreibt einige Dinge, die Sie zuvor bereits gelernt haben, mehr im Detail und behandelt auch einige neue.

5.1 Mehr über Listen

Der Datentyp Liste hat noch weitere Methoden. Dies sind alle Methoden von Listen-Objekten:

`append(x)` Füge ein Element an das End der Liste an. Äquivalent zu `a[len(a):] = [x]`.

`extend(L)` Erweitere die Liste durch Anfügen aller Elemente einer gegebenen Liste. Äquivalent zu `a[len(a):] = L`.

`insert(i, x)` Füge ein Element an einer bestimmten Stelle ein. Das erste Argument ist der Index, vor dem eingefügt werden soll, so dass `a.insert(0, x)` zu Beginn einfügt und `a.insert(len(a), x)` äquivalent zu `a.append(x)` ist.

`remove(x)` Entferne das erste Element der Liste, dessen Wert `x` ist. Falls kein solches existiert, so ist dies ein Fehler.

`pop([i])` Entferne das Element an der angegebenen Position in der Liste und gib es zurück. Falls kein Index angegeben wird, gibt `a.pop()` das letzte Element der Liste zurück. Dieses Element wird ebenfalls aus der Liste entfernt.

`index(x)` Gib den Index des ersten Elements in der Liste zurück, dessen Wert gleich `x` ist. Falls kein solcher existiert, so ist dies ein Fehler.

`count(x)` Return the number of times `x` appears in the list.

`sort()` Sortiere die Elemente der Liste, in der Liste selbst.

`reverse()` Invertiere die Reihenfolge der Listenelemente, in dieser selbst.

Ein Beispiel, das den Großteil aller Listen-Methoden verwendet:

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

5.1.1 Verwendung von Listen als Stapel

Mit den Listenmethoden kann man Listen sehr einfach als Stapel verwenden, stack, wobei das zuletzt hinzugefügte Elemente gleichzeitig jenes darstellt, das als erstes vom Stapel genommen wird (engl. „last-in, first-out“). Um ein Element oben auf dem Stapel abzulegen, verwendet man `append()`. Um ein Element vom Stapel zu entfernen, verwendet man `pop()` ohne expliziten Index. Beispiel:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2 Verwendung von Listen als Schlangen

Man kann eine Liste auch sehr bequem als Schlange (engl. queue) verwenden, wobei das zuerst hinzugefügte Element als letztes daraus wieder entfernt wird (engl. „first-in, first-out“). Um ein Element an das Ende einer Schlange anzufügen, verwendet man `append()`. Um eines vom Anfang der Schlange zu erhalten, verwendet man `pop()` mit `0` als Index. Beispiel:

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")           # Ergibt Terry
>>> queue.append("Graham")          # Ergibt Graham
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

5.1.3 Werkzeuge zur funktionalen Programmierung

Es gibt drei eingebaute Funktionen, die alle sehr nützlich in Kombination mit Listen sind: `filter()`, `map()` und `reduce()`.

'`filter(`*function*`, `*sequence*`)`' gibt eine Sequenz (wenn möglich, des selben Typs) zurück, die aus den Elementen der Sequenz besteht, für die *function*`(`*item*`)` wahr ist, z.B. um Primzahlen zu berechnen:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

'`map(`*function*`, `*sequence*`)`' ruft *function*`(`*item*`)` für jedes Listenelement auf und gibt eine Liste der zurück gegebenen Werte zurück. Zum Beispiel könnte man Kubikzahlen wie folgt berechnen:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Man darf mehr als eine Sequenz übergeben; die Funktion muss dann genausoviele Argumente haben wie es Sequenzen gibt, und sie wird mit dem entsprechenden Element der jeweiligen Sequenz (oder `None`, falls eine Sequenz kürzer als die andere ist) aufgerufen. Falls `None` als Funktion übergeben wird, wird es mit einer Funktion ersetzt, die ihr(e) Argument(e) zurück gibt.

Kombiniert man diese zwei Spezialfälle, so sehen wir, dass '`map(None, `*list1*`, `*list2*`)`' eine bequeme Art ist, ein Paar von Listen in eine Liste von Paaren umzuwandeln, z.B.:

```
>>> seq = range(8)
>>> def square(x): return x*x
...
>>> map(None, seq, map(square, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49)]
```

'`reduce(`*func*`, `*sequence*`)`' gibt einen einzigen Wert zurück, der sich ergibt, wenn man die zwei-stellige Funktion *func* auf die ersten zwei Elemente der Sequenz, dann auf das Resultat und das nächste Element, etc. anwendet. Um etwa die Summe der Zahlen von 1 bis 10 zu berechnen:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

Falls die Sequenz nur ein Element hat, wird dieses eine zurück gegeben. Falls sie leer ist, wird eine Ausnahme ausgelöst.

Ein drittes Argument kann übergeben werden, um einen Startwert anzugeben. In diesem Fall wird der Startwert bei einer leeren Sequenz zurück gegeben. Die Funktion wird dann auf den Startwert und das erste Element erstmalig angewendet, dann auf das Resultat und das nächste Element, u.s.w. Beispiel:

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

5.1.4 „List Comprehensions“

Mit „List comprehensions“ kann man auf sehr kompakte Art und Weise Listen zu erzeugen, ohne `map()`, `filter()` und/oder `lambda` zu benutzen. Die resultierende Listendefinition neigen oft dazu klarer zu sein als solche Listen, die mit diesen Konstrukten erzeugt werden. Jede „list comprehension“ besteht aus einem Ausdruck, gefolgt von einer `for`-Klausel und Null oder mehr `for`- oder `if`-Klauseln. Das Ergebnis ist eine Liste, die sich aus der Auswertung des Ausdrucks im Kontext der ihm folgenden `for`- und `if`-Klauseln. Falls der Ausdruck zu einem Tupel ausgewertet werden würde, muss dieses in Klammern stehen.

```
>>> freshfruit = [’  banana’, ’  loganberry ’, ’passion fruit  ’]
>>> [weapon.strip() for weapon in freshfruit]
[’banana’, ’loganberry’, ’passion fruit’]
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [{x: x**2} for x in vec]
[{2: 4}, {4: 16}, {6: 36}]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec]  # Fehler - bei Tupeln sind Klammern notwendig.
  File "<stdin>", line 1
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
```

5.2 Die `del`-Anweisung

Es gibt eine Möglichkeit, ein Element aus einer Liste zu entfernen, indem man seinen Index anstatt seinen Wert angibt: die `del`-Anweisung. Sie kann auch verwendet werden, um Teilbereiche aus einer Liste zu entfernen (was wir bereits mit der Zuweisung einer leeren Liste an den Bereich getan haben). Beispiel:

```
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
```

`del` kann auch verwendet werden, um Variablen gänzlich zu löschen:

```
>>> del a
```

Die anschließende Referenzierung (Verwendung) des Namens `a` ist ein Fehler (jedenfalls solange ihm kein anderer Wert zugewiesen wird). Wir werden später noch andere Möglichkeiten sehen, `del` zu verwenden.

5.3 Tupel und Sequenzen

Wir haben gesehen, dass Listen und Strings viele gemeinsame Eigenschaften haben, etwa Indizierungs- und Teilbereichs-Operatoren. Es sind zwei Beispiele für *Sequenz*-Datentypen. Da Python eine sich entwickelnde Sprache ist, könnten andere Sequenz-Typen noch hinzukommen. Es gibt noch einen anderen Standard-Sequenz-Typ: das *Tupel*.

Ein Tupel besteht aus einer Anzahl von Werten, die durch Kommata getrennt sind, z.B.:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tupel duerfen verschachtelt sein:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Wie man sieht, werden ausgegebene Tupel immer in runde Klammern gesetzt, so dass verschachtelte Tupel korrekt interpretiert werden. Bei der Eingabe können sie mit oder ohne umgebende Klammern verwendet werden, obwohl die Klammern oft sowieso notwendig sind (wenn ein Tupel Teil eines größeren Ausdrucks ist).

Tupel haben viele Anwendungen, etwa als (x, y)-Koordinatenpaare, Mitarbeiter-Datensätze aus der Datenbank, etc. Genau wie Strings sind Tupel unveränderlich: man kann einzelnen Elementen von Tupeln nichts zuweisen (man kann jedoch einen Großteil dieses Effekts durch Teilbereichs-Bildung und Aneinanderfügung simulieren).

Ein besonderes Problem ist die Erzeugung von Tupeln mit 0 oder nur 1 Element: die Syntax bietet einige Tricks, um das zu ermöglichen. Leere Tupel werden durch ein leeres Klammernpaar erzeugt. Ein Tupel mit genau einem Element wird erzeugt, indem man diesem Element ein Komma nachstellt (ein einzelnes Element in Klammern genügt nicht). Hässlich – aber wirkungsvoll. Beispiel:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- Beachte abschliessendes Komma!
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

Die Anweisung `t = 12345, 54321, 'hello!'` ist ein Beispiel für das sogenannte *Tupel-Einpacken* (engl. *tuple packing*): die Werte `12345`, `54321` und `'hello!'` werden in ein Tupel eingepackt. Die umgekehrte Operation ist auch möglich, z.B.:

```
>>> x, y, z = t
```

Dies nennt man, sinnigerweise, *Tupel-Auspacken* (engl. *tuple unpacking*). Das Auspacken von Tupeln verlangt, dass die Liste der Variablen auf der linken Seite die gleiche Anzahl von Elementen hat wie das Tupel selbst. Man beachte, dass Mehrfachzuweisungen lediglich eine Kombination von Ein- und Auspacken von Tupeln ist!

Gelegentlich ist die entsprechende Operation auf Listen nützlich: *Listen-Auspacken*. Sie wird unterstützt, indem die Variablenliste in eckige Klammern gesetzt wird:

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> [a1, a2, a3, a4] = a
```

5.4 Dictionaries

Ein anderer nützlicher, in Python eingebauter Datentyp ist das sogenannte *Dictionary* (engl. für Wörterbuch, Verzeichnis). Dictionaries kann man manchmal in anderen Sprachen unter den Begriffen „assoziativer Speicher“ oder „assoziative Vektoren“ wiederfinden. Im Gegensatz zu Sequenzen, die mit einem Zahlen-Intervall indiziert werden, werden Dictionaries über *Schlüssel* indiziert, die irgendeinen unveränderlichen Typ haben können. Strings und Zahlen können immer solche Schlüssel sein. Tupel können als Schlüssel verwendet werden, wenn sie nur Strings, Zahlen oder Tupel enthalten. Listen können nicht als Schlüssel verwendet werden, da Listen mit der `append()`-Methode direkt verändert werden können.

Am besten stellt man sich Dictionaries als eine ungeordnete Menge von *Schlüssel:Wert*-Paaren vor, unter der Randbedingung, dass die Schlüssel eindeutig sein müssen (innerhalb eines Dictionaries). Ein Paar geschwungener Klammern erzeugt ein leeres Dictionary: `{}`. Eine durch Kommata getrennte Liste von Schlüssel:Wert-Paaren innerhalb der Klammern fügt die initialen Schlüssel:Wert-Paare in das Dictionary ein. So werden Dictionaries auch ausgegeben.

Die Hauptoperationen auf einem Dictionary sind das Speichern eines Wertes unter einem Schlüssel und das Abrufen dieses Wertes bei Angabe des Schlüssels. Es ist auch möglich, ein Schlüssel:Wert-Paar mit `del` zu löschen. Wird beim Speichern ein Schlüssel verwendet, der bereits existiert, so wird der alte damit assoziierte Schlüssel vergessen. Es ist ein Fehler, einen Wert mit einem nicht existierenden Schlüssel abzurufen.

Die `keys()`-Methode eines Dictionary-Objektes gibt eine Liste aller in dem Dictionary verwendeten Schlüssel zurück – in zufälliger Reihenfolge (brauchen Sie sie sortiert, dann wenden Sie einfach die `sort()`-Methode auf die Liste der Schlüssel an). Um zu testen, ob ein einzelner Schlüssel in einem Dictionary vorkommt, verwende man die `has_key()`-Methode eines Dictionaries.

Hier ist ein kleines Beispiel mit einem Dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
1
```

5.5 Mehr über Bedingungen

Die in `while`- und `if`-Anweisungen oben verwendeten Bedingungen können neben Vergleichen auch andere Operatoren beinhalten.

Die Vergleichsoperatoren `in` und `not in` prüfen, ob ein Wert in einer Sequenz vorkommt (oder nicht). Die Opera-

toren `is` und `is not` vergleichen, ob zwei Objekte wirklich das gleiche Objekt sind. Das ist nur für veränderliche Objekte wie Listen von Bedeutung. Alle Vergleichsoperatoren haben die gleiche Priorität, die kleiner ist als die aller numerischen Operatoren.

Vergleiche können verkettet werden. `a < b == c` z.B. prüft, ob `a` kleiner ist als `b` und außerdem, ob `b` gleich `c` ist.

Vergleiche können mit den Booleschen Operatoren `and` und `or` kombiniert werden, und das Resultat eines Vergleichs (oder irgendeines anderen Booleschen Ausdrucks) kann mit `not` negiert werden. All diese Operatoren haben wiederum niedrigere Priorität als Vergleichsoperatoren. Unter ihnen hat `not` die höchste Priorität und `or` die geringste, so dass `A and not B or C` äquivalent ist zu `(A and (not B)) or C`. Natürlich können Klammern gesetzt werden, um die gewünschte Zusammensetzung auszudrücken.

Die Booleschen Operatoren `and` und `or` sind sogenannte *Abkürzungs*-Operatoren (engl. *shortcut*): ihre Argumente werden von links nach rechts ausgewertet, und die Auswertung wird beendet, sobald das Ergebnis feststeht. Wenn etwa `A` und `C` wahr sind, aber `B` ist falsch, so wertet `A and B and C` den Ausdruck C nicht aus. Allgemein gilt, dass der Rückgabewert eines Abkürzungs-Operators (wenn als allgemeiner Wert und nicht als Boolescher Ausdruck verwendet) das zuletzt ausgewertete Argument ist.

Es ist möglich, das Resultat eines Vergleichs oder eines anderen Booleschen Ausdrucks einer Variablen zuzuweisen, z.B.

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Man beachte dass in Python (anders als in C) Zuweisungen in Ausdrücken nicht erlaubt sind. C-Programmierer mögen darüber murren, aber dies verhindert eine sehr häufig auftretende Sorte von Fehlern in C-Programmen, nämlich die Verwendung von `=` in einem Ausdruck, wo eigentlich `==` gemeint war.

5.6 Vergleich von Sequenzen und anderen Typen

Sequenz-Objekte dürfen mit anderen Objekten vom gleichen Sequenz-Typ verglichen werden. Der Vergleich basiert auf *lexikographischer* Ordnung: zuerst werden die ersten beiden Elemente verglichen, und wenn sie sich unterscheiden, bestimmt dies bereits das Resultat. Wenn sie gleich sind, werden die nächsten beiden Elemente verglichen, und so weiter, bis eine der beiden Sequenzen erschöpft ist. Wenn zwei zu vergleichende Elemente selbst Sequenzen vom gleichen Typ sind, wird der lexikographische Vergleich rekursiv fortgesetzt. Falls alle Elemente einer Sequenz gleich sind, werden die Sequenzen als gleich betrachtet. Falls eine Sequenz eine Anfangssequenz der anderen ist, ist die gekürzte Sequenz die kleinere. Die lexikographische Ordnung für Strings verwendet die ASCII-Ordnung für einzelne Zeichen. Einige Beispiele für Vergleiche von Sequenzen des selben Typs:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Man beachte, dass es erlaubt ist, Objekte verschiedenen Typs zu vergleichen. Das Resultat ist deterministisch, aber beliebig: die Typen sind nach ihrem Namen geordnet. Daher ist eine Liste immer kleiner als ein String, ein String immer kleiner als ein Tupel, etc. Gemischte numerische Typen werden nach ihrem numerischen Wert verglichen, d.h.

0 ist gleich 0.0, etc.[1]

[1] Man sollte sich nicht auf diese Regeln verlassen, da sie sich in einer zukünftigen Version der Sprache ändern könnten.

KAPITEL

SECHS

Module

Wenn Sie den Python-Interpreter verlassen und wieder starten, sind alle Definitionen, die Sie vorher eingegeben haben, verloren. Wenn Sie daher längere Programme schreiben wollen, ist es besser, einen Text-Editor zu benutzen, um die Eingabe für den Interpreter in eine Datei zu schreiben, die Sie mit dem Interpreter aufrufen. Das versteht man unter der Erzeugung eines *Skriptes*. Mit zunehmender Länge Ihres Programmes möchten Sie es vielleicht in mehrere Dateien aufteilen, um den Code besser warten zu können. Sie werden sicher auch praktische Funktionen benutzen wollen, die Sie in verschiedenen Programmen geschrieben haben, ohne die Definition in jedes einzelne Programm zu kopieren.

Um dies zu unterstützen, bietet Python die Möglichkeit, Definitionen in einer Datei abzulegen, um diese in einem Skript oder einer interaktiven Sitzung mit dem Interpreter zu benutzen. Eine solche Datei wird ein *Modul* genannt. Die Definitionen eines Moduls können in anderen Modulen oder im *Hauptmodul*, d.h. der Sammlung von Variablen und Definitionen, zu denen man von der obersten Aufrufebene Zugang hat, *importiert* werden.

Ein Modul ist eine Datei mit Python-Definitionen und -Anweisungen. Der Dateiname ist der Modulname mit dem Zusatz '.py'. Innerhalb eines Moduls ist der Modulname (als String) als Wert der globalen Variablen `__name__` verfügbar. Benutzen Sie z.B. Ihren Lieblings-Text-Editor, um eine Datei 'fibo.py' im aktuellen Verzeichnis mit folgendem Inhalt zu erzeugen:

```
# Fibonacci numbers module

def fib(n):    # Gib Fibonacci-Reihe bis n aus.
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # Gib Fibonacci-Reihe bis n aus.
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Starten sie den Python-Interpreter und importieren Sie nun diesen Modul mit folgendem Kommando:

```
>>> import fibo
```

Das trägt nicht die Namen der in `fibo` definierten Funktionen in die Symboltabelle ein, sondern nur den Modulnamen `fibo`. Mit diesem Modulnamen hat man Zugriff auf die Funktionen:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Wenn Sie vorhaben, eine Funktion oft zu benutzen, können Sie sie einem lokalen Namen zuweisen:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 Mehr über Module

Ein Modul kann ausführbare Anweisungen wie auch Funktionsdefinitionen enthalten. Diese Anweisungen dienen der Initialisierung des Moduls. Sie werden nur dann ausgeführt, wenn das Modul das *erste* mal importiert wird.[1]

Jedes Modul hat seine eigene Symboltabelle, die von allen im Modul definierten Funktionen als globale Symboltabelle verwendet wird. Daher kann die Autorin eines Moduls globale Variablen innerhalb eines Moduls verwenden, ohne Angst vor versehentlichen, späteren Namenskonflikten mit den globalen Variablen der Benutzerin haben zu müssen. Auf der anderen Seite – wenn man weiß, was man macht – kann man die globalen Variablen eines Moduls mit der gleichen Notation wie für Funktionen manipulieren, `modname.itemname`.

Module können andere Module importieren. Es ist üblich, wenn auch nicht notwendig, alle `import`-Anweisungen zu Beginn eines Moduls (d.h. Skriptes) zu setzen. Die importierten Modulnamen werden in die globale Symboltabelle des importierenden Moduls eingetragen.

Es gibt eine Variante der `import`-Anweisung, die Namen aus einem Modulverzeichnis in die Symboltabelle des importierenden Moduls einträgt. Beispiel:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Das trägt nicht den Namen des Moduls, von dem die Symbole übernommen werden, in die lokale Symboltabelle ein, d.h. `fibo` in obigem Beispiel ist nicht definiert.

Es gibt sogar eine Variante, alle Namen zu importieren, die ein Modul definiert:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Damit werden alle Namen mit Ausnahme derer, die mit einem Unterstrich beginnen (_), importiert.

[1] Tatsächlich sind Funktionsdefinitionen auch „Anweisungen“ die „ausgeführt“ werden; die Ausführung bewirkt, dass der Funktionsname in die Symboltabelle eingetragen wird.

6.1.1 Der Modul-Suchpfad

Wenn ein Modul namens `spam` importiert wird, sucht der Interpreter nach einer Datei mit Namen ‘spam.py’ im aktuellen Verzeichnis und anschließend in der Liste der Verzeichnisse, die durch die Umgebungsvariable $PYTHONPATH spezifiziert wird. Diese hat die gleiche Syntax wie die Shell-Variable $PATH, d.h. eine Liste von Verzeichnisnamen. Wenn $PYTHONPATH nicht gesetzt ist oder wenn die Datei dort nicht gefunden wird, wird die Suche in einem von der Installation abhängigen Standardpfad fortgesetzt, unter UNIX normalerweise ‘/usr/local/lib/python’.

Tatsächlich werden Module in einer Liste von Verzeichnissen gesucht, die durch die Variable `sys.path` bestimmt wird, die vom Verzeichnis mit dem Eingabe-Skript initialisiert wird (oder dem aktuellen Verzeichnis), $PYTHONPATH und der von der Installation abhängigen Standard-Einstellung. Das erlaubt es Python-Programmen, den Modul-Suchpfad zu modifizieren oder zu ersetzen. Siehe dazu auch den späteren Absatz über Standard-Module.

6.1.2 „Übersetzte“ Python-Dateien

Zur Beschleunigung der Startzeit für kurze Programme, die eine Menge Standard-Module verwenden, wird angenommen, dass eine Datei mit Namen ‘spam.pyc’ in einem Verzeichnis, in dem ‘spam.py’ gefunden wird, eine zuvor „Byte-übersetzte“ Version des Moduls `spam` darstellt. Der Zeitpunkt der letzten Änderung von ‘spam.py’, das zur Erzeugung von ‘spam.pyc’ verwendet wurde, wird in ‘spam.pyc’ festgehalten und die Datei ‘.pyc’ wird ignoriert, falls diese zwei Punkte nicht übereinstimmen.

Normalerweise muss man nichts tun, um die Datei ‘spam.pyc’ zu erzeugen. Immer wenn ‘spam.py’ erfolgreich übersetzt wurde, wird der Versuch unternommen, die übersetzte Version nach ‘spam.pyc’ zu schreiben. Falls dieser Versuch schiefgeht, so ist das kein Fehler. Wird aus irgendeinem Grund diese Datei nicht vollständig geschrieben, so wird die resultierende Datei ‘spam.pyc’ als unvollständig erkannt und später ignoriert. Der Inhalt der Datei ‘spam.pyc’ ist Plattform-unabhängig, so dass ein Python-Modulverzeichnis über Maschinen mit verschiedenen Architekturen hinweg verwendet werden kann.

Einige Tips für die Experten:

- Wird der Python-Interpreter mit **-O**-Option aufgerufen, so wird optimierter Code erzeugt und in ‘.pyo’-Dateien gespeichert. Der Optimierer hilft im Moment nicht viel; er entfernt lediglich `assert`-Anweisungen und `SET_LINENO`-Befehle. Mit **-O** wird der *ganze* Bytecode optimiert; `.pyc`-Dateien werden ignoriert und `.py`-Dateien werden in optimierten Bytecode übersetzt.
- Übergibt man dem Python-Interpreter zwei **-O** Optionen (**-OO**), so veranlasst dies den Bytecode-Übersetzer, Optimierungen vorzunehmen, die in seltenen Fällen zu fehlerhaften Programmen führen können. Im Moment werden nur `__doc__`-Strings vom Bytecode entfernt, was zu kompakteren ‘.pyo’-Dateien führt. Da einige Programme sich darauf verlassen könnten, dass diese präsent sind, sollte man diese Option nur dann nutzen, wenn man sich über die Folgen klar ist.
- Ein Programm arbeitet nicht schneller oder langsamer, wenn es von einer ‘.pyc’- oder ‘.pyo’-Datei gelesen wird, als wenn es von einer ‘.py’-Datei gelesen wird. Das Einzige, was bei ‘.pyc’- oder ‘.pyo’-Dateien schneller geschieht, ist die Geschwindigkeit, mit der sie geladen werden.
- Wird ein Skript ausgeführt, indem sein Name in der Kommandozeile angegeben wird, so wird der Bytecode für dieses Skript nie in eine ‘.pyc’- oder ‘.pyo’-Datei geschrieben. Daher kann die Startzeit eines langen Skriptes reduziert werden, indem der meiste Code in ein Modul transferiert wird und der Rest in ein kleines Lade-Skript, das dieses Modul importiert.
- Es ist möglich, eine Datei ‘spam.pyc’ (oder ‘spam.pyo’ falls **-O** verwendet wird) ohne ein Modul ‘spam.py’ im gleichen Verzeichnis zu verwenden. Man kann dies dazu benutzen, um eine Bibliothek von Python-Code in einem Format zu verbreiten, das einigermaßen schwer zurück zu übersetzen ist (engl. *reverse engineering*).
- Das Modul `compileall` kann ‘.pyc’-Dateien (oder ‘.pyo’-Dateien bei Verwendung von **-O**) für alle Module in einem Verzeichnis erzeugen.

6.2 Standard-Module

Python wird mit einer Bibliothek von Standard-Modulen verbreitet, die in einem separaten Dokument, der *Python Bibliotheks-Referenz* (fortan nur noch „Bibliotheks-Referenz“ genannt) beschrieben wird. Einige Module sind in den Interpreter eingebaut und ermöglichen den Zugang zu Operationen, die nicht zum Kern der Sprache gehören, aber dennoch eingebaut sind, entweder aus Effizienzgründen oder um Zugang zu Betriebssystem-Primitiven, wie Systemaufrufen zu ermöglichen. Die Auswahl dieser Module ist abhängig von der jeweiligen Konfiguration. So wird das `amoeba`-Modul nur auf Systemen bereitgestellt, die irgendwie Amoeba-Primitive unterstützen. Ein Modul verdient besondere Beachtung: `sys`, das in jedem Python-Interpreter eingebaut ist. Die Variablen `sys.ps1` und `sys.ps2` definieren die Strings, die als primäre und sekundäre Prompts verwendet werden:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

Diese zwei Variablen sind nur definiert, wenn der Interpreter im interaktiven Modus ist.

Die Variable `sys.path` ist eine Liste von Strings, die den Modul-Suchpfad des Interpreters bestimmt. Sie wird mit dem Standardpfad initialisiert, der der Umgebungsvariablen $PYTHONPATH entnommen wird, oder von einem eingebauten Standard-Wert, falls $PYTHONPATH nicht gesetzt ist. Man kann sie mit den normalen Listen-Operationen modifizieren, z.B.:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 Die `dir()`-Funktion

Die eingebaute Funktion `dir()` wird verwendet, um herauszufinden, welche Namen in einem Modul definiert sind. Sie gibt eine sortierte Liste von Strings zurück:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__name__', 'argv', 'builtin_module_names', 'copyright', 'exit',
'maxint', 'modules', 'path', 'ps1', 'ps2', 'setprofile', 'settrace',
'stderr', 'stdin', 'stdout', 'version']
```

Ohne Argumente listet `dir()` all die Namen auf, die bisher definiert worden sind:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__name__', 'a', 'fib', 'fibo', 'sys']
```

Man beachte, dass alle Arten von Namen aufgezählt werden: Variablen, Module, Funktionen, etc.

`dir()` führt die Namen von eingebauten Funktionen und Variablen nicht auf. Wenn Sie dennoch eine Liste davon benötigen: sie werden im Standard-Modul `__builtin__` definiert:

```
>>> import __builtin__
>>> dir(__builtin__)
['AccessError', 'AttributeError', 'ConflictError', 'EOFError', 'IOError',
'ImportError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'MemoryError', 'NameError', 'None', 'OverflowError', 'RuntimeError',
'SyntaxError', 'SystemError', 'SystemExit', 'TypeError', 'ValueError',
'ZeroDivisionError', '__name__', 'abs', 'apply', 'chr', 'cmp', 'coerce',
'compile', 'dir', 'divmod', 'eval', 'execfile', 'filter', 'float',
'getattr', 'hasattr', 'hash', 'hex', 'id', 'input', 'int', 'len', 'long',
'map', 'max', 'min', 'oct', 'open', 'ord', 'pow', 'range', 'raw_input',
'reduce', 'reload', 'repr', 'round', 'setattr', 'str', 'type', 'xrange']
```

6.4 Pakete

Pakete bieten die Möglichkeit, den Namensraum von Python zu strukturieren, indem „qualifizierte Modulnamen“ verwendet werden. Zum Beispiel beschreibt der Modulname `A.B` ein Untermodul namens ‘`B`’ im Paket ‘`A`’. Genauso wie die Verwendung von Modulen die Autoren verschiedener Module davor bewahrt, sich Sorgen um ihre globalen Variablen zu machen, bewahrt die Verwendung von qualifizierten Modulnamen die Autoren von Paketen mit mehreren Modulen wie NumPy oder Python Imaging Library davor, sich Sorgen um die Verwendung der eigenen Modulnamen durch andere Autoren zu machen.

Angenommen, Sie wollen eine Sammlung von Modulen (ein „Paket“) für die einheitliche Handhabung von Audio-Dateien und -Daten entwerfen. Es gibt viele verschiedene Audio-Dateiformate (normalerweise an ihrer Erweiterung zu erkennen, z.B. ‘.wav’, ‘.aiff’, ‘.au’), so dass Sie vermutlich eine wachsende Anzahl von Modulen für die Umwandlung zwischen diesen verschiedenen Formaten erstellen und warten müssen. Ebenso gibt es viele verschiedene Operationen, die Sie vielleicht auf Audio-Daten vornehmen möchten (z.B. Mixen, Echo hinzufügen, eine Equalizer-Funktion anwenden, einen künstlichen Stereo-Effekt erzeugen), so dass Sie zusätzlich eine nicht enden wollende Kette von Modulen schreiben müssen, um auch diese Operationen zu implementieren. Dies wäre eine mögliche Struktur Ihres Pakets (ausgedrückt als hierarchisches Dateisystem):

```
Sound/                          Top-level package
      __init__.py               Initialize the sound package
      Formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      Effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      Filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Die ‘__init__.py’-Dateien werden benötigt, um Python die Verzeichnisse als Pakete behandeln zu lassen, was verhindert, dass Verzeichnisse mit gängigen Namen wie z.B. ‘`string`’ unabsichtlich spätere Module im Suchpfad verdecken. Im einfachsten Fall kann ‘__init__.py’ einfach eine leere Datei sein, aber sie kann auch Initialisierungscode für das Paket enthalten oder die Variable `__all__` setzen, die später beschrieben wird.

Benutzerinnen des Paketes können einzelne Module daraus importieren, z.B.:

```
import Sound.Effects.echo
```

Das lädt das Untermodul `Sound.Effects.echo`. Es muss mit seinem vollständigen Namen referenziert werden, z.B.:

```
Sound.Effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Eine alternative Art, ein Untermodul zu importieren, ist:

```
from Sound.Effects import echo
```

Das lädt auch das Untermodul `echo` und macht es ohne sein Paket-Präfix verfügbar:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Wieder eine andere Variante ist, die gewünschte Funktion oder Variable direkt zu importieren:

```
from Sound.Effects.echo import echofilter
```

Wieder lädt dies das Untermodul `echo`, aber dessen Funktion `echofilter()` wird direkt verfügbar gemacht:

```
echofilter(input, output, delay=0.7, atten=4)
```

Man beachte, dass bei Verwendung von `from` *package* `import` *item*, das zu ladende Element entweder ein Untermodul (oder Unterpaket) des Paketes sein kann oder ein anderer darin definierter Name, etwa eine Funktion, Klasse oder Variable. Die `import`-Anweisung prüft zuerst, ob das Element im Paket definiert ist. Wenn nicht, nimmt sie an, es handelt sich um ein Modul und versucht, es zu laden. Wenn dieser Versuch schiefgeht, wird ein `ImportError` ausgelöst.

Umgekehrt muss bei Verwendung der `import` *item.subitem.subsubitem*-Syntax jedes Element, bis auf das letzte ein Paket sein. Das letzte kann ein Modul oder ein Paket sein, aber keine Klasse oder Funktion, die im vorherigen Element definiert wäre.

6.4.1 Importieren von * aus einem Paket

Was passiert nun, wenn die Benutzerin `from Sound.Effects import *` schreibt? Idealerweise würde man hoffen, dass dies nun im Dateisystem sucht und erkennt, welche Untermodule im Paket da sind und sie alle importiert. Dummerweise funktioniert diese Operation auf der Mac- und Windows-Plattform nicht so gut, da das Dateisystem oft nicht über exakte Angaben der Schreibweise eines Dateinamens verfügt! Auf diesen Plattformen gibt es keine Garantie, zu wissen, ob ‘ECHO.PY’ als Modul `echo`, `Echo` oder `ECHO` importiert werden soll. (Zum Beispiel hat Windows 95 die ärgerliche Eigenschaft, alle Dateinamen mit einem großen Anfangsbuchstaben anzuzeigen.) Die Einschränkung von Dateinamen auf 8+3 Buchstaben unter DOS bringt weitere interessante Probleme für lange Modulnamen mit sich.

Die einzige Lösung für die Autorin eines Paketes ist es, einen expliziten Index des Paketes anzugeben. Die import-Anweisung verwendet dabei folgende Konvention: falls die Datei ‘__init__.py’ eines Paketes eine Liste namens `__all__` definiert, wird sie als Liste von Modulnamen aufgefasst, die importiert werden sollen, wenn Code der Art `from` *package* `import *` gefunden wird. Es liegt an der Autorin des Paketes, diese Liste auf dem neuesten Stand zu halten, wenn eine neue Version des Paketes veröffentlicht wird. Autorinnen können diese Eigenschaft auch bewusst *nicht* unterstützen, wenn sie keinen Nutzen darin sehen, * aus ihrem Paket zu importieren. Die Datei `Sounds/Effects/__init__.py` könnte z.B. den folgenden Code enthalten:

```
__all__ = ["echo", "surround", "reverse"]
```

Das würde bedeuten, dass `from Sound.Effects import *` die angegebenen drei Untermodule vom `Sound`-Paket importieren würde.

Falls `__all__` nicht definiert ist, importiert die Anweisung `from Sound.Effects import *` *nicht* alle Untermodule des `Sound.Effects`-Paketes in den aktuellen Namensraum, sondern stellt nur sicher, dass das Paket `Sound.Effects` importiert wird (möglicherweise, indem sein Initialisierungscode, ‘__init__.py’, ausgeführt wird) und importiert dann, welche Namen auch immer im Paket definiert sind. Das beinhaltet alle in ‘__init__.py’ definierten Namen (und explizit geladenen Untermodule) und ebenso alle Untermodule des Pakets, die von vorherigen import-Anweisungen explizit geladen wurden, z.B.

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

In diesem Beispiel werden die Module `echo` und `surround` in den aktuellen Namensraum importiert, da sie im Paket `Sound.Effects` definiert sind, wenn die `from ... import`-Anweisung ausgeführt wird. (Das funktioniert auch, wenn `__all__` definiert ist.)

Man beachte, dass im Allgemeinen davon abgeraten wird, * aus einem Modul oder Paket zu importieren, da es häufig zu schlecht lesbarem Code führt. Es ist jedoch in Ordnung, wenn man es in einer interaktiven Sitzung benutzt, um Tipp-Arbeit zu sparen. Außerdem sind gewisse Module so entworfen, dass sie Namen nach einem bestimmten Muster exportieren.

Es ist aber nichts falsch daran, `from Package import specific_submodule` zu verwenden! Tatsächlich ist dies die empfohlene Notation, solange das importierende Modul keine Untermodule mit dem gleichen Namen aus verschiedenen Paketen verwenden muss.

6.4.2 Referenzen innerhalb von Paketen

Untermodule müssen häufig aufeinander Bezug nehmen (sich referenzieren). Zum Beispiel könnte das Modul `surround` das `echo`-Modul benutzen. Tatsächlich kommen solche Referenzen so oft vor, dass die `import`-Anweisung zuerst ins umgebende Paket nachschaut, bevor der Modul-Suchpfad inspiziert wird. Daher kann das umgebende Modul einfach `import echo` oder `from echo import echofilter` benutzen. Falls das importierte Modul im aktuellen Paket nicht gefunden wird (das Paket, dessen aktuelles Modul ein Untermodul ist), schaut die `import`-Anweisung auf oberster Ebene nach einem Modul mit dem angegebenen Namen.

Wenn Pakete in Unterpakete organisiert werden (wie mit dem `Sound`-Paket im Beispiel), gibt es keine Abkürzung, um Untermodule von Schwester-Paketen zu referenzieren, sondern es muss der vollständige Name des Paketes verwendet werden. Falls z.B. das Modul `Sound.Filters.vocoder` das `echo`-Modul im `Sound.Effects`-Paket benutzen muss, kann es `from Sound.Effects import echo` verwenden.

KAPITEL

SIEBEN

Ein- und Ausgabe

Es gibt verschiedene Wege, um die Ausgabe eines Programmes darzustellen. Daten können in einer von Menschen lesbaren Form ausgegeben werden oder zur späteren Verwendung in eine Datei geschrieben werden. Dieses Kapitel diskutiert einige der Möglichkeiten.

7.1 Ansprechendere Formatierung von Ausgaben

Bisher haben wir zwei Arten kennengelernt, Werte auszugeben: *Ausdrücke* und die `print`-Anweisung. (Eine weitere ist die Methode `write()` von Datei-Objekten zu verwenden. Die Standard-Ausgabe kann durch `sys.stdout` als Datei-Objekt referenziert werden. Siehe die Bibliotheks-Referenz für weitere Information zu diesem Thema.)

Oftmals möchte man die Formatierung seiner Ausgabe besser gestalten, als nur durch Leerzeichen zwischen den Werten. Es gibt zwei Arten, seine Ausgabe zu formatieren. Die erste ist, die gesamte Handhabung von Strings selbst zu übernehmen. Mit den Operationen zur Teilbereichs-Bildung und Aneinanderfügung von Strings kann man jedes vorstellbare Format erzeugen. Das Standard-Modul `string` enthält einige nützliche Operationen, um Strings bis zu einer vorgegebenen Spaltenbreite aufzufüllen. Diese Operationen werden in Kürze besprochen. Die zweite Art ist, den Operator `%` mit einem String als erstem Argument zu verwenden. `%` interpretiert das linke Argument als `sprintf()`-ähnliches Format wie in C, das auf das rechte Argument angewendet wird und gibt den String nach der Anwendung dieses Operators zurück.

Eine Frage jedoch bleibt: wie wandelt man Werte in Strings um? Zum Glück kennt Python eine Möglichkeit, jeden Wert in einen String umzuwandeln, nämlich, ihn der Funktion `repr()` zu übergeben oder einfach, den Wert zwischen umgekehrten einfachen Anführungszeichen zu schreiben (`‘`). Einige Beispiele:

```
>>> x = 10 * 3.14
>>> y = 200*200
>>> s = 'The value of x is ' + 'x' + ', and y is ' + 'y' + '...'
>>> print s
The value of x is 31.400000000000002, and y is 40000...
>>> # Umgekehrte Anfuehrungszeichen funktionieren
>>> # nicht nur bei Zahlen:
... p = [x, y]
>>> ps = repr(p)
>>> ps
'[31.400000000000002, 40000]'
>>> # Die Konvertierung eines Strings fuegt Anfuehrungszeichen
>>> # und Rueckwaerts-Schraegstriche hinzu:
... hello = 'hello, world\n'
>>> hellos = 'hello'
>>> print hellos
'hello, world\n'
>>> # Das Argument von umgekehrten Anfuehrungszeichen
>>> # darf ein Tuple sein:
... 'x, y, ('spam', 'eggs')'
"(31.400000000000002, 40000, ('spam', 'eggs'))"
```

Dies sind zwei Arten, eine Tabelle mit Quadrat- und Kubikzahlen auszugeben:

```
>>> import string
>>> for x in range(1, 11):
...     print string.rjust('x', 2), string.rjust('x*x', 3),
...     # Beachte abschliessendes Komma in der Zeile zuvor!
...     print string.rjust('x*x*x', 4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
>>> for x in range(1, 11):
...     print '%2d %3d %4d' % (x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Man beachte, dass ein Leerzeichen zwischen den Spalten durch die Art hinzugefügt wurde, wie `print` funktioniert:

es fügt immer ein Leerzeichen zwischen seinen Argumenten ein.)

Dieses Beispiel demonstriert, wie man mit der Funktion `string.rjust()` einen String in einer Tabelle mit vorgegebener Breite rechtsbündig ausgeben kann, indem links Leerzeichen hinzugefügt werden. Es gibt ähnliche Funktionen `string.ljust()` und `string.center()`. Diese Funktionen geben nichts aus, sondern geben einfach einen neuen String zurück. Falls der Eingabestring zu lang ist, schneiden sie ihn nicht ab, sondern geben ihn unverändert zurück. Das wird die Anordnung der Spalten etwas durcheinanderbringen, aber das ist normalerweise immer noch besser, als die Alternative, nämlich einen verfälschten Wert auszugeben. Wenn man wirklich das Abschneiden braucht, kann man immer eine Teilbereichs-Operation verwenden, wie z.B. in '`string.ljust(x, n)[0:n]`'.)

Es gibt eine weitere Funktion, `string.zfill()`, die einen numerischen String links mit Nullen auffüllt. Sie kann auch mit Plus- und Minus-Vorzeichen umgehen:

```
>>> import string
>>> string.zfill('12', 5)
'00012'
>>> string.zfill('-3.14', 7)
'-003.14'
>>> string.zfill('3.14159265359', 5)
'3.14159265359'
```

Die Verwendung des %-Operators sieht so aus:

```
>>> import math
>>> print 'The value of PI is approximately %5.3f.' % math.pi
The value of PI is approximately 3.142.
```

Gibt es mehr als eine Format-Anweisung in dem String, übergibt man ein Tupel als rechten Operanden, z.B.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print '%-10s ==> %10d' % (name, phone)
...
Jack       ==>       4098
Dcab       ==>    7678
Sjoerd     ==>       4127
```

Die meisten Format-Anweisungen funktionieren genau wie in C und verlangen, dass man den passenden Typ übergibt. Tut man dies jedoch nicht, so wird eine Ausnahme ausgelöst und es wird kein „core dump" erzeugt. Das `%s`-Format ist nicht ganz so streng: wenn das entsprechende Argument kein String ist, wird es mit der eingebauten Funktion `str()` in einen solchen umgewandelt. Die Verwendung von * zur Übergabe der Breite oder Genauigkeit als separates (Ganzzahl-) Argument wird auch unterstützt. Die C-Format-Anweisungen `%n` und `%p` werden nicht unterstützt.

Hat man einen wirklich langen Format-String, den man nicht aufteilen möchte, wäre es nett, die Variablen mit Namen zu referenzieren, anstatt ihre Position anzugeben. Das kann geschehen, indem eine Erweiterung von C-Format-Anweisungen in der Form `%(name)format` verwendet wird, z.B.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab: %(Dcab)d' % table
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Das ist besonders nützlich in Kombination mit der eingebauten Funktion `vars()`, die ein Dictionary mit allen lokalen Variablen zurück gibt.

7.2 Dateien lesen und schreiben

`open()` gibt ein Datei-Objekt zurück und wird meist mit zwei Argumenten verwendet: '`open(`*filename*`, `*mode*`)`'.

```
>>> f=open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

Das erste Argument ist ein String mit dem Dateinamen. Das zweite Argument ist ein weiterer String, der ein paar Zeichen enthält, die beschreiben, wie die Datei verwendet werden soll. *mode* kann `'r'` sein, wenn die Datei nur gelesen wird, `'w'` um nur zu schreiben (eine bereits existierende Datei gleichen Namens wird überschrieben), und `'a'` öffnet eine Datei, um Daten anzufügen; alle Daten, die in die Datei geschrieben werden, werden automatisch an das Ende angefügt. `'r+'` öffnet eine Datei sowohl zum Lesen wie auch zum Schreiben. Das *mode*-Argument ist optional. Wenn es ausgelassen wird, wird `'r'` angenommen.

Unter Windows und auf dem Macintosh, öffnet ein dem Modus angefügtes `'b'` die Datei im Binär-Modus, d.h. es gibt auch Modi wie diese: `'rb'`, `'wb'`, und `'r+b'`. Windows macht einen Unterschied zwischen Text- und Binär-Dateien; die Endzeilenmarkierung (EOL) in Text-Dateien werden automatisch leicht verändert, wenn Daten gelesen oder geschrieben werden. Diese Modifikation hinter den Kulissen ist in Ordnung für Text-Dateien in ASCII, aber sie wird Binär-Dateien wie JPEGs oder '.EXE'-Dateien korrumpieren. Seien Sie sehr vorsichtig, um auf jeden Fall den Binär-Modus für solche Dateien zu verwenden. (Man beachte, dass die exakte Semantik für den Text-Modus auf dem Macintosh von der verwendeten C-Bibliothek abhängig ist.)

7.2.1 Methoden von Datei-Objekten

Die restlichen Beispiele in diesem Abschnitt setzen voraus, dass eine Datei namens `f` bereits erzeugt wurde.

Um den Inhalt einer Datei zu lesen, rufen Sie `f.read(`*size*`)` auf, was eine gewisse Menge Daten liest und in einem String zurück gibt. *size* ist ein optionales numerisches Argument. Wird *size* weggelassen oder ist es negativ, so wird der gesamte Inhalt der Datei gelesen und zurück gegeben. Es ist Ihr Problem, wenn die Datei doppelt so groß ist wie Ihr Hauptspeicher. Anderenfalls werden höchstens *size* Bytes gelesen und zurück gegeben. Wird das Ende der Datei erreicht, so gibt `f.read()` einen leeren String zurück ().

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` liest eine einzelne Zeile aus der Datei. Ein Zeilenvorschub-Zeichen (`\n`) wird am Ende des Strings stehengelassen es sei denn, es handelt sich um die letzte Zeile der Datei, wenn sie nicht mit einem Zeilenvorschub-Zeichen endet. Das macht den Rückgabewert eindeutig: wenn `f.readline()` einen leeren String zurück gibt, wurde das Ende der Datei erreicht, während ein Leerzeichen durch ein `'\n'` dargestellt wird, einem String, der nur ein einziges Zeilenvorschub-Zeichen enthält.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

`f.readlines()` verwendet wiederholt `f.readline()` und gibt eine Liste mit allen Zeilen der Datei zurück.

```
>>> f.readlines()
['This is the first line of the file.\n', 'Second line of the file\n']
```

`f.write(`*string*`)` schreibt den Inhalt von *string* in die Datei und gibt `None` zurück.

```
>>> f.write('This is a test\n')
```

`f.tell()` gibt eine ganze Zahl zurück, die für die aktuelle Position in der Datei steht, gemessen in Bytes vom Anfang der Datei aus. Um die Position im Datei-Objekt zu verändern, verwende man '`f.seek(`*offset*`, `*from_what*`)`'. Die Position wird berechnet, indem *offset* zu einem Bezugspunkt hinzu gezählt wird. Der Bezugspunkt wird durch das *from_what*-Argument ausgewählt. Ein Wert von 0 für *from_what* zählt vom Anfang der Datei, 1 von der aktuellen Position und 2 vom Ende der Datei aus. *from_what* kann weggelassen werden und wird zum Standardwert 0 ergänzt, was gleichbedeutend mit dem Anfang der Datei ist.

```
>>> f=open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Springe zum 5-ten Byte in der Datei.
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Springe zum 3-ten Byte vor dem Ende.
>>> f.read(1)
'd'
```

Wenn Sie die Datei nicht mehr brauchen, rufen Sie `f.close()` auf, um die Datei zu schließen und um etwaige Systemressourcen freizugeben, die die geöffnete Datei noch belegt. Nach `f.close()` werden alle weiteren Versuche, die Datei zu verwenden, automatisch scheitern:

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Datei-Objekte haben ein paar weitere Methoden, wie z.B. `isatty()` und `truncate()`, die seltener verwendet werden. Für eine vollständige Abhandlung von Datei-Objekten ziehe man die Bibliotheks-Referenz zu Rate.

7.2.2 Das `pickle`-Modul

Strings können sehr einfach in eine Datei geschrieben und von dort wieder gelesen werden. Für Zahlen muss man etwas mehr Aufwand treiben, da die `read()`-Methode nur Strings zurück gibt, die man einer Funktion wie `string.atoi()` übergeben muss, die dann einen String wie z.B. `'123'` nehmen und seinen numerischen Wert, 123, zurück geben. Wenn man jedoch komplexere Datentypen wie z.B. Listen, Klassen oder Instanzen abspeichern möchte, werden die Dinge sehr viel komplizierter.

Damit man nicht ständig Code zum Speichern komplexer Datenstrukturen schreiben und korrigieren muss, stellt Python ein Standard-Modul namens `pickle` zur Verfügung (engl. pickle = einlegen, haltbar machen). Dies ist ein erstaunliches Modul, das fast jedes Python-Objekt (selbst einige Formen von Python-Code!) in eine String-Repräsentation umwandeln kann. Dieser Prozess wird (in Anlehnung an das Englische) *Pickeln* genannt. Die Re-

konstruktion des Objektes aus der String-Repräsentation wird dual dazu *Entpickeln* genannt.[1] Zwischen dem Pickeln und Entpickeln könnte der String, der das Objekt repräsentiert, in einer Datei oder als Daten gespeichert oder über eine Netzwerkverbindung an einen entfernten Rechner geschickt worden sein.

Wenn Sie ein Objekt `x`, und ein zum Schreiben geöffnetes Datei-Objekt `f` haben, braucht es zur einfachsten Art des Pickelns nur eine Zeile Code:

```
pickle.dump(x, f)
```

Und um das Objekt wieder zu entpickeln (mit `f` als einem zum Lesen geöffneten Datei-Objekt):

```
x = pickle.load(f)
```

(Es gibt viele Varianten hiervon, je nachdem, ob man z.B. viele Objekte pickeln möchte oder die gepickelten Daten nicht in eine Datei schreiben möchte. Die vollständige Dokumentation für `pickle` findet man in der Bibliotheks-Referenz.

`pickle` ist der Standard-Weg, um Python-Objekte zu erzeugen, die gespeichert und von anderen Programmen oder vom gleichen Programm zu einem späteren Zeitpunkt wiederverwendet werden können. Der technische Ausdruck dafür ist ein *persistentes* Objekt. Weil die Verwendung von `pickle` so weitverbreitet ist, versuchen viele Autoren von Python-Erweiterungen sicherzustellen, dass neue Datentypen wie z.B. Matrizen sauber ge- und entpickelt werden können.

[1] Das hat natürlich nichts mit Pickeln im deutschen Sprachgebrauch zu tun. [Der Übersetzer]

KAPITEL

ACHT

Fehler und Ausnahmen

Bisher wurden Fehlermeldungen lediglich erwähnt, aber wenn Sie die Beispiele ausprobiert haben, werden Sie einige gesehen haben. Es gibt (mindestens) zwei unterscheidbare Arten von Fehlern: *Syntaxfehler* und *Ausnahmen*.

8.1 Syntaxfehler

Syntaxfehler, auch als Parsing-Fehler bekannt, sind vielleicht die gängigste Art von Beschwerden, die Sie bekommen, während Sie noch Python lernen:

```
>>> while 1 print 'Hello world'
  File "<stdin>", line 1
    while 1 print 'Hello world'
                ^
SyntaxError: invalid syntax
```

Der Parser wiederholt die anstößige Zeile und gibt einen kleinen 'Pfeil' an der Stelle aus, wo der Fehler erkannt wurde. Der Fehler wurde von dem Zeichen (engl. token) *vor* dem Pfeil verursacht (oder zumindest dort entdeckt). Im Beispiel wird der Fehler beim Schlüsselwort `print` erkannt, da ein Doppelpunkt (`':'`) davor fehlt.

Der Dateiname und die Zeilennummer werden mit ausgegeben, so dass man weiß, wo man nachschauen muss, falls die Eingabe aus einem Skript kam.

8.2 Ausnahmen

Selbst wenn eine Anweisung syntaktisch korrekt ist, kann sie einen Fehler verursachen, wenn man versucht, sie auszuführen. Während des Programmablaufs festgestellte Fehler werden *Ausnahmen* genannt und führen nicht bedingungslos zum Abbruch des Programms. Sie werden bald lernen, sie in Python-Programmen zu handhaben. Die meisten Ausnahmen jedoch werden nicht von Programmen behandelt und resultieren in Fehlermeldungen und einem Programmabbruch wie z.B.:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1
ZeroDivisionError: integer division or modulo
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1
NameError: spam
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1
TypeError: illegal argument type for built-in operation
```

Die letzte Zeile der Fehlermeldung erklärt, was passiert ist. Ausnahmen sind in verschiedene Typen unterteilt und der jeweilige Typ wird als Teil der Nachricht ausgegeben. Die Typen im Beispiel sind `ZeroDivisionError`, `NameError` und `TypeError`. Der String, der als Ausnahme-Typ ausgegeben wird, ist der Name des eingebauten Namens der aufgetretenen Ausnahme. Das gilt für alle eingebauten Ausnahmen, muss aber nicht für benutzerdefinierte Ausnahmen zutreffen (obwohl es eine nützliche Konvention ist). Standard-Ausnahme-Namen sind eingebaute Bezeichner (nicht reservierte Schlüsselworte).

Der Rest der Zeile ist ein Detail, dessen Interpretation und Bedeutung vom Ausnahme-Typen abhängen.

Der vorhergehende Teil der Fehlermeldung zeigt den Kontext, in dem der Fehler aufgetreten ist, in Form eines „stack backtrace“ an. Im Allgemeinen enthält er die Zeile des Quellcodes aus dem „stack backtrace“ es werden allerdings keine Zeilen von der Standard-Eingabe angezeigt.

Die Bibliotheks-Referenz zählt die eingebauten Ausnahmen auf und erklärt, was sie bedeuten.

8.3 Ausnahme-Behandlung

Es ist möglich, Programme zu schreiben, die ausgewählte Ausnahmen abfangen und behandeln. Betrachten Sie das folgende Beispiel, in dem vom Benutzer so lange eine Eingabe verlangt wird, bis eine gültige Ganzzahl eingegeben wurde, erlaubt es ihm aber, das Programm abzubrechen (mit `Control-C` oder was immer das Betriebssystem unterstützt). Man beachte, dass eine vom Benutzer verursachte Unterbrechung durch das Auslösen der Ausnahme `KeyboardInterrupt` signalisiert wird.

```
>>> while 1:
...     try:
...         x = int(raw_input("Bitte eine Zahl eingeben: "))
...         break
...     except ValueError:
...         print "Hoppla! Das war keine gueltige Zahl. Neuer Versuch..."
...
```

Die `try`-Anweisung funktioniert wie folgt:

- Zuerst wird die *`try`-Klausel* (die Anweisungen zwischen den Schlüsselworten `try` und `except`) ausgeführt.
- Wenn keine Ausnahme auftritt, wird die *`except`-Klausel* übersprungen, und die Ausführung der `try`-Anweisung ist beendet.
- Falls eine Ausnahme während der Ausführung der `try`-Klausel auftritt, wird der Rest der Klausel übersprungen. Wenn dann deren Typ mit der Ausnahme nach dem Schlüsselwort `except` übereinstimmt, wird der Rest der

try-Klausel übersprungen, die except-Klausel wird ausgeführt, und die Ausführung wird nach der try-Anweisung fortgesetzt.

- Tritt eine Ausnahme auf, die nicht mit der Ausnahme in der except-Klausel übereinstimmt, wird sie nach außen an weitere try-Anweisungen weitergereicht. Wenn keine Behandlung erfolgt, so ist es eine *unbehandelte Ausnahme* und die Ausführung terminiert mit einer Meldung wie weiter oben beschrieben.

Eine try-Anweisung darf mehr als eine except-Klausel haben, um Behandlungsroutinen für verschiedene Ausnahmen zu spezifizieren. Höchstens eine Routine wird ausgeführt. Behandlungsroutinen behandeln nur Ausnahmen, die in der entsprechenden try-Klausel auftreten, nicht in anderen Behandlungsroutinen der gleichen try-Anweisung. Eine except-Klausel darf mehrere Ausnahmen in einem Tupel benennen, z.B.:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Die letzte except-Klausel darf den (oder die) Ausnahme-Namen weglassen, um dann als Joker zu dienen. Verwenden Sie dies mit größter Vorsicht, denn es ist einfach, einen echten Programmierfehler damit auszublenden! Sie kann auch dazu verwendet werden, um eine Fehlermeldung auszugeben und dann die Ausnahme erneut auszulösen (was dem Aufrufer ebenfalls erlaubt, die Ausnahme zu behandeln):

```
import string, sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(string.strip(s))
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

Die try … except-Anweisung hat eine optionale *else-Klausel*, die, wenn vorhanden, nach allen except-Klauseln stehen muss. Sie ist nützlich als Platz für Code, der ausgeführt werden muss, wenn die try-Klausel keine Ausnahme auslöst. Beispiel:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

Es ist besser, die else-Klausel zu benutzen, als zusätzlichen Code in der try-Klausel hinzu zu fügen, weil dadurch verhindert wird, dass versehentlich eine Ausnahme abgefangen wird, die nicht von dem Code ausgelöst worden ist, der durch die Anweisung try … except geschützt wird.

Tritt eine Ausnahme auf, kann sie einen zugehörigen Wert haben, auch als *Argument* der Ausnahme bekannt. Das Vorhandensein und der Typ des Argumentes hängen vom Ausnahme-Typ ab. Für Ausnahme-Typen mit einem Argu-

ment darf die `except`-Klausel eine Variable nach dem Ausnahme-Namen (oder -Liste) spezifizieren, um den Wert des Argumentes zu erhalten. Beispiel:

```
>>> try:
...     spam()
... except NameError, x:
...     print 'name', x, 'undefined'
...
name spam undefined
```

Hat eine Ausnahme ein Argument, wird es als letzter Teil ('Detail') der Nachricht für unbehandelte Ausnahmen ausgegeben.

Ausnahme-Behandlungsroutinen behandeln Ausnahmen nicht nur, wenn sie direkt in der `try`-Klausel vorkommen, sondern auch, wenn sie innerhalb von Funktionen auftreten, die (sogar indirekt) in der `try`-Klausel aufgerufen werden, z.B.:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError, detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo
```

8.4 Auslösen von Ausnahmen

Die `raise`-Anweisung erlaubt, die Auslösung einer spezifizierten Ausnahme zu erzwingen, z.B.:

```
>>> raise NameError, 'HiThere'
Traceback (most recent call last):
  File "<stdin>", line 1
NameError: HiThere
```

Das erste Argument für `raise` bezeichnet die Ausnahme, die ausgelöst werden soll. Das optionale zweite Argument spezifiziert ein Argument der Ausnahme.

8.5 Benutzerdefinierte Ausnahmen

Programme dürfen ihre eigenen Ausnahme-Typen erzeugen, indem ein String einer Variablen zugewiesen wird, z.B.:

```
>>> class MyError:
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return ‘self.value‘
...
>>> try:
...     raise MyError(2*2)
... except MyError, e:
...     print ’My exception occurred, value:’, e.value
...
My exception occurred, value: 4
>>> raise MyError, 1
Traceback (most recent call last):
  File "<stdin>", line 1
__main__.MyError: 1
```

Viele Standard-Module verwenden dies, um Fehler zu melden, die in Funktionen auftreten können, die sie definieren.

Weitere Information über Klassen wird im Kapitel 9, „Klassen“ vorgestellt.

8.6 Definition von Aufräum-Aktionen

Die `try`-Anweisung hat eine andere optionale Klausel, die dazu gedacht ist, Aufräum-Aktionen zu definieren, die unter allen Umständen ausgeführt werden müssen, z.B.:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print ’Goodbye, world!’
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2
KeyboardInterrupt
```

Eine *`finally`-Klausel* wird immer ausgeführt, ob eine Ausnahme in der `try`-Klausel aufgetreten ist, oder nicht. Falls eine Ausnahme aufgetreten ist, wird sie erneut ausgelöst, nachdem die `finally`-Klausel ausgeführt wurde. Die `finally`-Klausel wird auch „auf dem Weg hinaus“ ausgeführt, wenn die `try`-Anweisung mit einer `break`- oder `return`-Anweisung verlassen wird.

Eine `try`-Anweisung muss entweder eine oder mehrere `except`-Klauseln oder eine `finally`-Klausel haben, aber nicht beides.

KAPITEL

NEUN

Klassen

Der Klassen-Mechanismus von Python fügt Klassen mit einem Minimum an neuer Syntax und Semantik der Sprache hinzu. Es ist eine Mischung von Klassen-Mechanismen, wie sie in C++ und Modula-3 anzutreffen sind. Genau wie bei Modulen, setzen Klassen in Python keine absolute Grenze zwischen Definition und Benutzung, sondern verlassen sich eher auf die Vernunft des Benutzers, nicht „in die Definition einzubrechen.“ Die wichtigsten Eigenschaften von Klassen werden jedoch mit voller Kraft erhalten: der Klassen-Vererbungs-Mechanismus erlaubt Mehrfach-Vererbung von Klassen, eine abgeleitete Klasse kann Methoden ihrer Oberklasse (oder -Klassen) überschreiben, eine Methode kann die Methode gleichen Namens in der Oberklasse aufrufen. Objekte können beliebige Mengen privater Daten enthalten.

In C++-Terminologie ausgedrückt sind alle Klassenmitglieder (engl. class members), inklusive der Datenmitglieder, *öffentlich*, und alle Mitglieder-Funktionen sind *virtuell*. Es gibt keine speziellen Konstruktoren oder Destruktoren. Wie in Modula-3 gibt es keine Kurzformen, um die Mitglieder von Objekten von ihren Methoden zu referenzieren: die Methodenfunktion wird mit einem expliziten ersten Argument deklariert, das das Objekt repräsentiert, welches implizit beim Aufruf übergeben wird. Wie in Smalltalk sind Klassen ihrerseits Objekte, allerdings im weiteren Sinne des Wortes: in Python sind alle Datentypen Objekte. Das stellt die Semantik beim Import und der Umbenennung dar. Aber, genau wie in C++ oder Modula-3, können eingebaute Datentypen nicht als Oberklassen für die Erweiterung durch die Benutzerin verwendet werden. Außerdem können wie in C++, aber nicht in Modula-3, die meisten eingebauten Operatoren mit spezieller Syntax (arithmetische Operatoren, Indizierung, etc.) in Instanzen von Klassen umdefiniert werden.

9.1 Ein Wort zur Terminologie

Da es keine universell anerkannte Terminologie gibt, in der man über Klassen sprechen könnte, werde ich gelegentlich Begriffe aus der Smalltalk- und C++-Welt verwenden. (Ich würde gern Begriffe aus Modula-3 verwenden, da seine objektorientierte Semantik Python näher steht, als die von C++, aber ich vermute, dass weniger Leser davon gehört haben.)

Ich muss Sie auch dahingehend warnen, dass es eine begriffliche Falle für Leserinnen mit Erfahrung in objektorientierter Programmierung gibt: das Wort „Objekt“ bedeutet in Python nicht notwendigerweise eine Klassen-Instanz. Wie in C++ und Modula-3, und im Gegensatz zu Smalltalk sind nicht alle Typen in Python mit Klassen gleichzusetzen: die fundamentalen, eingebauten Typen wie Ganzzahlen und Listen sind keine, und auch etwas exotischere Typen wie Dateien sind keine. *Alle* Python-Typen jedoch haben eine gewisse Semantik gemein, die am besten mit dem Wort Objekt beschrieben werden kann.

Objekte haben ein eigenständiges Dasein, und mehrere Namen (in mehreren Geltungsbereichen) können mit dem gleichen Objekt verbunden sein. Das ist in anderen Sprachen als *aliasing* (in etwa: Pseudonyme) bekannt. In einer ersten flüchtigen Betrachtung von Python wird das normalerweise nicht besonders gewürdigt und kann gefahrlos ignoriert werden, wenn man es mit unveränderlichen Grund-Datentypen zu tun hat (Zahlen, Strings, Tupel). Aliasing hat jedoch einen (beabsichtigten!) Effekt auf die Semantik von Python-Code, auch wenn veränderliche Objekte wie Listen, Dictionaries und die meisten Typen, die Einheiten außerhalb des Programmes repräsentieren (Dateien, Fenster, etc.).

Das wird normalerweise von einem Programm vorteilhaft eingesetzt, da solche Aliase sich in vielerlei Hinsicht wie Zeiger verhalten. Die Übergabe eines Objektes z.B. ist eine einfache Operation, da die Implementation von Python nur einen Zeiger übergibt; und wenn eine Funktion ein als Argument übergebenes Objekt modifiziert, wird der Aufrufer die Modifikation sehen – das macht zwei verschiedene Übergabe-Mechanismen für Argumente, wie etwa in Pascal überflüssig.

9.2 Geltungsbereiche und Namensräume in Python

Bevor Klassen eingeführt werden, muss ich Ihnen erst etwas über die Regeln für Geltungsbereiche in Python erzählen. Klassen-Definitionen spielen mit ein paar niedlichen Tricks in Namensräumen, und sie müssen wissen, wie Geltungsbereiche und Namensräume funktionieren, um zu verstehen, was vor sich geht. Und schließlich ist dieses Wissen auch hilfreich für jeden fortgeschrittenen Python-Programmierer.

Fangen wir mit ein paar Definitionen an.

Ein *Namensraum* ist eine Abbildung von Namen nach Objekten. Die meisten Namensräume sind im Moment als Python-Dictionaries implementiert, aber davon bemerkt man man normalerweise nichts (außer bei der Laufzeit-Geschwindigkeit), und es könnte sich in Zukunft ändern. Beispiele für Namensräume sind: die Menge eingebauter Namen (Funktionen wie z.B. `abs()` und Namen eingebauter Ausnahmen), die globalen Namen in einem Modul und die lokalen Namen bei einem Funktionsaufruf. In gewissem Sinne bildet die Menge der Attribute eines Objektes auch einen Namensraum. Die wichtigste Eigenschaft von Namensräumen, die man kennen muss, ist die, dass es absolut keinen Zusammenhang zwischen Namen in verschiedenen Namensräumen gibt. Zum Beispiel dürfen zwei verschiedene Module eine Funktion namens „maximize“ definieren, ohne dass es irgendwelche Probleme damit gäbe – Benutzerinnen der Module müssen sie mit dem Modulnamen als Präfix versehen.

Übrigens verwende ich das Wort *Attribut* für alle Namen, die nach einem Punkt stehen. In dem Ausdruck `z.real` ist `real` ein Attribut des Objektes `z`. Streng genommen sind Referenzen auf Namen in Modulen Attribut-Referenzen: in dem Ausdruck `modname.funcname`, ist `modname` ein Modul-Objekt und `funcname` ist ein Attribut davon. In diesem Fall gibt es eine simple Abbildung zwischen den Attributen des Moduls und den im Modul definierten globalen Namen: sie teilen sich den gleichen Namensraum![1]

Attribute dürfen nur-lesbar oder schreibbar sein. In letzterem Fall ist die Zuweisung an Attribute erlaubt. Modul-Attribute sind schreibbar: Sie können schreiben: '`modname.the_answer = 42`'. Schreibbare Attribute dürfen auch mit der `del`-Anweisung gelöscht werden, z.B. '`del modname.the_answer`'.

Namensräume werden zu verschiedenen Zeitpunkten erzeugt und haben verschiedene Lebenszeiten. Der Namensraum, der die eingebauten Namen enthält, wird erzeugt, wenn der Python-Interpreter gestartet wird, und wird niemals gelöscht. Der globale Namensraum eines Moduls wird erzeugt, wenn die Definition des Moduls eingelesen wird. Normalerweise existieren Modul-Namensräume auch, bis der Interpreter terminiert. Die Anweisungen, die in der obersten Ebene des Interpreters ausgeführt werden (entweder aus einem Skript oder aus einer Sitzung interaktiv gelesen), werden als Teil eines Moduls namens `__main__` betrachtet, so dass sie ihren eigenen globalen Namensraum haben. (Die eingebauten Namen existieren ebenfalls in einem Modul. Es wird `__builtin__` genannt.)

Der lokale Namensraum für eine Funktion wird erzeugt, wenn die Funktion aufgerufen wird, und gelöscht, wenn sie einen Wert zurück gibt, oder eine Ausnahme auslöst, die nicht in der Funktion behandelt wird. (Tatsächlich wäre „vergessen“ das bessere Wort, um zu beschreiben, was eigentlich passiert.) Natürlich haben rekursive Aufrufe alle ihre eigenen Namensräume.

Ein *Geltungsbereich* ist ein Bereich des Quelltextes eines Python-Programmes (bisher nur Code genannt), wo ein Namensraum direkt zugänglich ist. „Direkt zugänglich“ bedeutet hier, dass versucht wird, eine nicht-qualifizierte Referenz auf einen Namen innerhalb dieses Namensraumes zu finden.

Obwohl Geltungsbereiche statisch bestimmt werden, ist ihre Verwendung dynamisch. Zu jedem Zeitpunkt während der Ausführung werden genau drei verschachtelte Namensräume verwendet (d.h. es sind genau drei Namensräume

[1] Außer in einem: Modul-Objekte haben ein geheimes nur-lesbares Attribut namens `__dict__`, das das Dictionary zurück gibt, mit dem der Namensraum des Moduls implementiert ist. Der Name `__dict__` ist ein Attribut, aber kein globaler Name. Offensichtlich verletzt seine Verwendung die Abstraktion der Implementierung von Namensräumen und sollte auf Programme wie post-mortem-Debugger beschränkt werden.

direkt zugänglich): der innerste Geltungsbereich, in dem als erstes gesucht wird, enthält die lokalen Namen, der mittlere Geltungsbereich, in dem anschließend gesucht wird, enthält die globalen Namen des aktuellen Moduls, und der äußerste Geltungsbereich (wo zuletzt gesucht wird) ist der Namensraum, der eingebaute Namen enthält.

Normalerweise referenziert der lokale Geltungsbereich die lokalen Namen der (textuell) aktuellen Funktion. Außerhalb von Funktionen referenziert der lokale Geltungsbereich den gleichen Namensraum wie der globale Geltungsbereich: den Modul-Namensraum. Klassen-Definitionen fügen dem lokalen Geltungsbereich einen weiteren Namensraum hinzu.

Es ist wichtig, zu begreifen, dass Geltungsbereiche textuell bestimmt werden: der globale Geltungsbereich einer Funktion, die in einem Modul definiert wurde, ist der Namensraum dieses Moduls, unabhängig davon, von wo und mit welchem Alias die Funktion aufgerufen wird. Auf der anderen Seite vollzieht sich die tatsächliche Suche nach Namen dynamisch, d.h. zur Laufzeit. Allerdings entwickelt sich die Sprachdefinition in Richtung von statischer Namens-Auflösung, d.h. zur Zeit der „Übersetzung“ – verlassen Sie sich also nicht auf dynamische Namens-Auflösung! (Tatsächlich werden lokale Variablen bereits statisch bestimmt.)

Ein besonderer Umstand von Python ist der, dass sich Zuweisungen immer im innersten Geltungsbereich abspielen. Zuweisungen kopieren keine Daten, sie binden lediglich Namen an Objekte. Das gleiche gilt für Lösch-Operationen: die Anweisung ‘`del x`’ entfernt die Bindung von `x` vom Namensraum, der im lokalen Geltungsbereich referenziert wird. Tatsächlich, verwenden alle Operationen, die neue Namen einführen, den lokalen Geltungsbereich: insbesondere binden import-Anweisungen und Funktionsdefinitionen den Modul- oder Funktionsnamen an den lokalen Geltungsbereich. (Die `global`-Anweisung kann verwendet werden, um anzuzeigen, dass gewisse Variablen im globalen Geltungsbereich zuhause sind.)

9.3 Ein erster Blick auf Klassen

Klassen führen ein wenig neue Syntax, drei neue Objekt-Typen und ein wenig neue Semantik ein.

9.3.1 Syntax der Klassen-Definition

Die einfachste Form der Klassen-Definition sieht so aus:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Klassen-Definitionen müssen wie Funktionsdefinitionen (`def`-Anweisungen) auch ausgeführt werden, bevor sie einen Effekt haben können. (Man könnte sich etwa vorstellen, eine Klassen-Definition in einen Zweig einer `if`-Anweisung oder innerhalb einer Funktion zu plazieren.)

In der Praxis werden die Anweisungen innerhalb einer Klassen-Definition normalerweise Funktionsdefinitionen sein, aber andere Anweisungen sind auch erlaubt und manchmal auch nützlich – wir werden später noch einmal darauf zurück kommen. Die Funktionsdefinitionen innerhalb einer Klasse haben normalerweise eine eigentümliche Form einer Argumentliste, die von den Aufrufkonventionen für Methoden diktiert wird – auch das wird später erklärt.

Wenn eine Klassen-Definition betreten wird, wird ein neuer Namensraum erzeugt und als lokaler Geltungsbereich verwendet – daher gehen alle Zuweisungen an lokale Variablen in diesen neuen Namensraum. Insbesondere binden Funktionsdefinitionen den Namen einer neuen Funktion hier an.

Wird eine Klassen-Definition normal verlassen (also am Ende), so wird ein *Klassen-Objekt* erzeugt. Dies ist im wesentlichen eine Hülle um den Inhalt des Namensraumes, der von der Klassen-Definition erzeugt wird. Wir werden im

nächsten Abschnitt noch mehr über Klassen-Objekte erfahren. Der ursprüngliche lokale Geltungsbereich (derjenige, der aktuell war, gerade bevor die Klassen-Definition betreten wurde), wird wiedereingesetzt, und das Klassen-Objekt wird hier an den Klassennamen gebunden, der im Kopf der Klassen-Definition angegeben wurde (`ClassName` im Beispiel).

9.3.2 Klassen-Objekte

Klassen-Objekte unterstützen zwei Arten von Operationen: Attribut-Referenzen und Instanziierung .

Attribut-Referenzen verwenden die Standard-Syntax, die für alle Attribut-Referenzen in Python verwendet wird: `obj.name`. Gültige Attributnamen sind all die Namen, die im Namensraum der Klasse waren, als das Klassen-Objekt erzeugt wurde. Wenn also die Klassen-Definition so aussah:

```
class MyClass:
    "A simple example class"
    i = 12345
    def f(x):
        return 'hello world'
```

dann sind `MyClass.i` und `MyClass.f` gültige Attribut-Referenzen, die jeweils eine Ganzzahl und ein Methoden-Objekt zurück geben. An Klassenattribute kann auch zugewiesen werden, so dass man den Wert von `MyClass.i` mittels Zuweisung ändern kann. `__doc__` ist auch ein gültiges Attribut, das nur-lesbar ist und den zur Klasse gehörenden Dokumentations-String zurück gibt: `"A simple example class"`).

Die *Instanziierung* von Klassen verwendet die Funktions-Notation. Tun Sie einfach so, als ob das Klassen-Objekt eine parameterlose Funktion wäre, die eine neue Instanz der Klasse zurück gibt. Zum Beispiel erzeugt (unter Verwendung obiger Klasse):

```
x = MyClass()
```

eine neue *Instanz* der Klasse und weist dieses Objekt der lokalen Variablen `x` zu.

Die Instanziierungs-Operation (ein Klassen-Objekt „aufzurufen“) erzeugt ein leeres Objekt. Viele Klassen wollen Objekte mit einem bekannten Anfangszustand erzeugen. Daher darf eine Klasse eine spezielle Methode namens `__init__()` wie folgt definieren:

```
    def __init__(self):
        self.empty()
```

Wenn eine Klasse eine `__init__()`-Methode definiert, wird bei der Klassen-Instanziierung automatisch die Methode `__init__()` für die neu erzeugte Klassen-Instanz aufgerufen. Im `Bag`-Beispiel kann man eine neue, initialisierte Instanz wie folgt erhalten:

```
x = Bag()
```

Natürlich darf die Methode `__init__()` Argumente haben, und so eine größere Flexibilität haben. In diesem Fall werden Argumente, die an den Klassen-Instanziierungsoperator übergeben werden, an `__init__()` weitergeben. Beispiel:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Instanz-Objekte

Was können wir nun mit Instanz-Objekten tun? Die einzigen Operationen, die von Instanz-Objekten verstanden werden, sind Attribut-Referenzen. Es gibt zwei Arten gültiger Attributnamen.

Die ersten werde ich *Datenattribute* nennen. Diese entsprechen den „Instanzvariablen“ in Smalltalk, und den „Datenmitgliedern“ in C++. Datenattribute müssen nicht deklariert werden, sondern sie werden wie lokale Variablen erzeugt, wenn ihnen zum ersten mal etwas zugewiesen wird. Wenn z.B. `x` eine Instanz der oben definierten Klasse `MyClass` ist, wird der folgende Code klaglos den Wert `16` ausgeben:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

Die zweite Art von Attribut-Referenzen sind *Methoden*. Eine Methode ist eine Funktion, die zu einem Objekt „gehört.“ (In Python gilt der Begriff Methode nicht ausschließlich für Klassen-Instanzen: andere Objekt-Typen können ebenfalls Methoden haben. Listen-Objekte z.B. haben Methoden namens append, insert, remove, sort, u.s.w. Im weiteren werden wir jedoch den Begriff Methode nur ausschließlich verwenden, um darunter Methoden von Instanzen von Klassen-Objekten zu verstehen, außer wir drücken explizit etwas anderes aus.)

Gültige Methodennamen eines Instanz-Objektes hängen von seiner Klasse ab. Per definitionem definieren alle Attribute einer Klasse, die (benutzerdefinierte) Funktions-Objekte darstellen, entsprechende Methoden ihrer Instanzen. In unserem Beispiel ist `x.f` eine gültige Methoden-Referenz, da `MyClass.f` eine Funktion ist, aber `x.i` ist keine, da `MyClass.i` keine Funktion ist. Aber `x.f` ist nicht das gleiche wie `MyClass.f` – es ist ein *Methoden-Objekt*, kein Funktions-Objekt.

9.3.4 Methoden-Objekte

Normalerweise wird eine Methode sofort aufgerufen, z.B.:

```
x.f()
```

In unserem Beispiel würde das den String `'hello world'` zurück geben. Es ist jedoch nicht notwendig, eine Methode direkt aufzurufen: `x.f` ist ein Methoden-Objekt und kann anderweitig gespeichert und später aufgerufen werden. Zum Beispiel wird:

```
xf = x.f
while 1:
    print xf()
```

‘`hello world`’ bis zum Ende aller Tage ausgeben.

Was passiert genau, wenn eine Methode aufgerufen wird? Sie werden vielleicht bemerkt haben, dass `x.f()` oben ohne ein Argument aufgerufen wurde, selbst, wenn die Funktionsdefinition für `f` ein Argument spezifizierte. Was geschah mit dem Argument? Natürlich löst Python eine Ausnahme aus, wenn eine Funktion, die eine Argument verlangt, ohne ein solches aufgerufen wird – selbst, wenn das Argument tatsächlich gar nicht verwendet wird. . .

Vielleicht haben Sie es bereits erraten: das Besondere an Methoden ist, dass das Objekt als erstes Argument der Funktion übergeben wird. In unserem Beispiel ist der Aufruf von `x.f()` exakt äquivalent zu `MyClass.f(x)`. Allgemein gilt, dass der Aufruf einer Methode mit einer Liste von *n* Argumenten dazu äquivalent ist, die entsprechende Funktion mit einer Argumentliste aufzurufen, die gebildet wird, indem das Objekt der Methode vor dem ersten Argument eingefügt wird.

Wenn Sie noch nicht verstehen, wie Methoden funktionieren, kann ein Blick in die Implementierung die Dinge vielleicht klären. Wenn ein Instanzattribut referenziert wird, das kein Datenattribut ist, wird seine Klasse abgesucht. Wenn der Name ein gültiges Klassenattribut bezeichnet, das ein Funktions-Objekt ist, wird ein Methoden-Objekt erzeugt, indem (Zeiger auf) das Instanz-Objekt und das Funktions-Objekt in ein abstraktes Objekt verpackt werden: das Methoden-Objekt. Wenn das Methoden-Objekt mit einer Argumentliste aufgerufen wird, wird es wieder ausgepackt, eine neue Argumentliste wird vom Instanz-Objekt und der ursprünglichen Argumentliste gebildet, und das Funktions-Objekt wird mit dieser neuen Argumentliste aufgerufen.

9.4 Diverse Bemerkungen

[Diese sollten vielleicht sorgfältiger plaziert werden. . .]

Datenattribute überschreiben Methodenattribute mit dem gleichen Namen. Um versehentliche Namenskonflikte zu vermeiden, die in großen Programmen schwer zu findende Fehler verursachen können, ist es ratsam, irgendeine Konvention zu verwenden, die die Chancen für solche Konflikte minimiert, z.B. Methodennamen mit großen Anfangsbuchstaben beginnen zu lassen, Datenattribute mit einem eindeutigen String-Präfix zu versehen (vielleicht nur ein Unterstrich), oder Verben für Methoden und Substantive für Datenattribute zu verwenden.

Datenattribute dürfen von Methoden genauso wie von gewöhnlichen Benutzern („Klienten“) eines Objektes referenziert werden. Mit anderen Worten, Klassen sind unbrauchbar, um pure abstrakte Datentypen zu implementieren. Tatsächlich gibt es in Python keine Möglichkeit, Datenkapselung zu erzwingen – es basiert alles auf Konventionen. (Auf der anderen Seite kann die in C geschriebene Python-Implementation, Implementierungsdetails vollständig verbergen und den Zugang zu einem Objekt kontrollieren, wenn notwendig. Das kann von Python-Erweiterungen verwendet werden, die in C geschrieben wurden.)

Klienten sollten Datenattribute vorsichtig verwenden – Klienten könnten Invarianten durcheinanderbringen, die von Methoden verwaltet werden, indem sie auf deren Datenattributen herumtrampeln . Man beachte, dass Klienten eigene Datenattribute zu einem Instanz-Objekt hinzufügen können, ohne die Gültigkeit der Methoden zu beeinflussen, solange wie Namenskonflikte vermieden werden. Wieder kann hier einem eine Namenskonvention eine Menge Kopfschmerzen ersparen.

Es gibt keine Kurzform, um Datenattribute (oder andere Methoden!) aus Methoden heraus zu referenzieren. Ich finde, dass das die Lesbarkeit von Methoden eher erhöht: es gibt keine Möglichkeit, beim schnellen Durchschauen einer Methode, lokale Variablen mit Instanzvariablen zu verwechseln.

Gewöhnlich wird das erste Argument einer Methode `self` genannt. Dies ist nichts weiter als eine Konvention: der Name `self` hat absolut keine spezielle Bedeutung für Python. (Man beachte jedoch, dass bei Nichtbeachtung der Konvention Ihr Code für andere Python-Programmierer weniger lesbar sein könnte, und es ist auch vorstellbar, dass

ein *Klassen-Browser* geschrieben wird, der sich auf diese Konvention verlässt.)

Jedes Funktions-Objekt, das ein Klassenattribut ist, definiert eine Methode für Instanzen dieser Klasse. Es ist nicht notwendig, dass die Funktionsdefinition textuell in der Klassen-Definition eingeschlossen ist. Ein Funktions-Objekt an eine lokale Variable in der Klasse zuzuweisen ist auch ok. Zum Beispiel:

```
# Funktion, die ausserhalb der Klasse definiert ist.
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

Nun sind `f`, `g` und `h` alles Attribute der Klasse `C`, die sich auf Funktions-Objekte beziehen, und dadurch sind sie alle Methoden von Instanzen von `C` – wobei `h` exakt äquivalent zu `g` ist. Man beachte, dass dieses Vorgehen normalerweise nur dazu dient, die Leserin zu verwirren.

Methoden dürfen andere Methoden aufrufen, indem sie Methodenattribute des `self`-Arguments verwenden, z.B.:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Methoden dürfen globale Namen genau wie gewöhnliche Funktionen referenzieren. Der globale Geltungsbereich, der mit einer Methode assoziiert ist, ist das Modul, das die Klassen-Definition enthält. (Die Klasse selbst wird nie als globaler Geltungsbereich verwendet!) Während man selten einen guten Grund für die Verwendung von globalen Daten in einer Methode findet, gibt es viele berechtigte Anwendungen für den globalen Geltungsbereich: einerseits können Funktionen und Module, die in den globalen Geltungsbereich importiert werden, von anderen Methoden und Funktionen, die darin definiert werden, benutzt werden. Normalerweise ist die Klasse, die die Methode definiert, selbst im globalen Geltungsbereich definiert, und im nächsten Abschnitt werden wir einige gute Gründe dafür finden, dass eine Methode ihre eigene Klasse referenzieren will.

9.5 Vererbung

Natürlich würde eine Spracheigenschaft den Namen „Klasse“ nicht verdienen, ohne Vererbung zu unterstützen. Die Syntax für eine abgeleitete Klassen-Definition sieht wie folgt aus:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Der Name `BaseClassName` muss in einem Geltungsbereich definiert werden, der die abgeleitete Klassen-Definition umfasst. Anstelle eines Namens für eine Oberklasse ist auch ein Ausdruck erlaubt. Das ist hilfreich, wenn die Oberklasse in einem anderen Modul definiert wird, z.B.:

```
class DerivedClassName(modname.BaseClassName):
```

Die Ausführug einer abgeleiteten Klassen-Definition verläuft genau wie die einer Oberklasse. Wenn das Klassen-Objekt konstruiert wird, wird die Oberklasse vermerkt. Diese wird verwendet, um Attribut-Referenzen aufzulösen: wenn ein gesuchtes Attribut in der Klasse nicht gefunden wird, wird es in der Oberklasse gesucht. Diese Regel wird rekursiv angewandt, wenn die Oberklasse ihrerseits von einer anderen Klasse abgeleitet ist.

Es gibt nichts besonderes bei der Instanziierung von abgeleiteten Klassen: `DerivedClassName()` erzeugt eine neue Instanz der Klasse. Methoden-Referenzen werden wie folgt aufgelöst: das entsprechende Klassenattribut wird gesucht, wobei die Kette von Oberklassen hinaufgestiegen wird, wenn notwendig. Wenn dies zu einem Funktions-Objekt führt, ist die Methoden-Referenz gültig.

Abgeleitete Klassen dürfen Methoden ihrer Oberklassen überschreiben. Da Methoden keine speziellen Privilegien beim Aufruf anderer Methoden desselben Objektes haben, kann eine Methode einer Oberklasse, die eine andere in derselben Oberklasse definierte Methode aufruft, schließlich tatsächlich eine Methode einer abgeleiteten Klasse aufrufen, die sie überschreibt. (Für C++-Programmierer: alle Methoden in Python sind tatsächlich `virtual`.)

Eine überschreibende Methode in einer abgeleiteten Klasse könnte aber eine Methode gleichen Namens in der Oberklasse auch erweitern, anstatt sie ersetzen zu wollen. Es gibt eine einfache Art, die Methode der Oberklasse direkt aufzurufen: rufen Sie einfach '`BaseClassName.methodname(self, arguments)`' auf. Dies ist gelegentlich auch für Klienten nützlich. (Man beachte, dass dies nur funktioniert, wenn die Oberklasse direkt im globalen Geltungsbereich definiert oder importiert ist.)

9.5.1 Mehrfach-Vererbung

Python unterstützt auch eine beschränkte Form von Mehrfach-Vererbung. Eine Klassen-Definition mit mehreren Oberklassen sieht wie folgt aus:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Die einzige Regel, die man braucht, um die Semantik zu erklären, ist die Auflösungs-Regel für Referenzen von Klassenattributen. Diese vollzieht sich in einer Tiefensuche und von links nach rechts. Wenn also ein Attribut in `DerivedClassName` nicht gefunden wird, wird es in `Base1`, dann (rekursiv) in den Oberklassen von `Base1`, und nur wenn es dort nicht gefunden wird, wird es in `Base2` gesucht, u.s.w.

(Einigen Leuten erscheint Breitensuche – die Suche in `Base2` und `Base3` vor den Oberklassen von `Base1` – natürlicher. Dazu müsste man jedoch wissen, ob ein spezielles Attribut von `Base1` tatsächlich in `Base1` definiert ist oder in einer seiner Oberklassen, bevor man die Konsequenzen eines Namenskonfliktes mit einem Attribut von `Base2` absehen kann. Tiefensuche macht keinen Unterschied zwischen direkten und von `Base1` ererbten Attributen.)

Es ist klar, dass undifferenzierte Verwendung von Mehrfach-Vererbung ein Alptraum in Sachen Wartung bedeutet, da Python sich auf Konventionen verlässt, um versehentliche Namenskonflikte zu vermeiden. Ein bekanntes Problem mit Mehrfach-Vererbung ist eine Klasse, die von zwei Oberklassen abgeleitet ist, die eine gemeinsame Oberklasse haben. Während es recht einfach ist, sich auszumalen, was in diesem Fall passiert (die Instanz wird eine einzige Kopie von

„Instanzvariablen“ oder Datenattributen haben, die von der gemeinsamen Oberklasse stammen), ist es nicht klar, ob diese Semantik in irgendeiner Art sinnvoll ist.

9.6 Private Variablen

Es gibt nur eine eingeschränkte Unterstützung von privaten Klassenbezeichnern. Jeder Bezeichner der Form `__spam` (mindestens zwei führende Unterstriche, höchstens ein abschließendes) wird textuell mit `_classname__spam` ersetzt, wobei `classname` der aktuelle Klassenname ohne führende Unterstriche ist. Diese Namensverstümmelung (engl. name mangling) wird ohne Berücksichtigung der syntaktisches Position des Bezeichners durchgeführt, so dass es verwendet werden kann, um private Klassen- und Instanzvariablen zu definieren, wie auch Methoden und globale Variablen und sogar, um private Instanzvariablen dieser Klasse in Instanzen anderer Klassen zu speichern. Der so gebildete Name kann abgeschnitten werden, wenn er mehr als 255 Zeichen lang ist. Außerhalb von Klassen oder wenn der Klassenname nur aus Unterstrichen besteht, findet keine Verstümmelung statt.

Diese Verstümmelung beabsichtigt, Klassen eine einfache Möglichkeit zu bieten, „private“ Instanzvariablen und -methoden zu definieren, ohne sich um von anderen Klassen definierte Instanzvariablen sorgen zu müssen oder sich mit Instanzvariablen in Code außerhalb der Klasse beschäftigen zu müssen. Man beachte, dass die Verstümmelungs-Regeln entworfen wurden, um Unfälle zu vermeiden. Es ist jedoch für einen entschlossenen Menschen immer noch möglich, sich Zugang zu einer Variablen zu verschaffen, die als privat gilt, und diese zu ändern. Das kann sogar nützlich sein, z.B. für den Debugger, und das ist ein Grund, warum dieses Loch nicht geschlossen wird. (Eine kleine Unzulänglichkeit am Rande: die Ableitung einer Klasse mit dem gleichen Namen wie dem der Oberklasse gestattet die Verwendung der privaten Variablen der Oberklasse.)

Man beachte ferner, dass Code, der an `exec()`, `eval()` oder `evalfile()` übergeben wird, den Klassennamen der aufrufenden Klasse nicht als aktuelle Klasse betrachtet; das ist ähnlich wie bei der `global`-Anweisung, deren Effekt genauso auf Code beschränkt ist, der zusammen Byte-übersetzt wurde. Dieselbe Einschränkung gilt für `getattr()`, `setattr()` und `delattr()` wie auch bei der direkten Referenzierung von `__dict__`.

Hier ist ein Beispiel einer Klasse, die ihre eigenen `__getattr__`- und `__setattr__`-Methoden implementiert und alle Attribute in einer privaten Variable abspeichert. In gewisser Weise funktioniert dies in Python 1.4 genauso wie in früheren Versionen:

```
class VirtualAttributes:
    __vdict = None
    __vdict_name = locals().keys()[0]

    def __init__(self):
        self.__dict__[self.__vdict_name] = {}

    def __getattr__(self, name):
        return self.__vdict[name]

    def __setattr__(self, name, value):
        self.__vdict[name] = value
```

9.7 Diverses

Manchmal ist es hilfreich, einen Datentyp zu benutzen, der ähnlich zu „record“ in Pascal oder „struct“ in C ist und eine Anzahl mit Namen versehener Daten gruppiert. Eine leere Klassen-Definition erledigt das auf elegante Weise, z.B.:

```
class Employee:
    pass

john = Employee() # Erzeuge einen leeren Mitarbeiter-Satz.

# Fuelle die Felder des Satzes.
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Einem Stück Python-Code, der einen speziellen abstrakten Datentyp erwartet, kann man oftmals eine Klasse übergeben, die die Methoden dieses Datentyps emuliert. Wenn Sie z.B. eine Funktion haben, die irgendwelche Daten eines Datei-Objektes formatiert, können Sie eine Klasse mit den Methoden `read()` und `readline()` definieren, die Daten von einem String-Puffer erhält, und diese als Argument übergeben.

Instanzmethoden-Objekte haben auch Attribute: `m.im_self` ist das Objekt, dessen Instanz diese Methode ist, und `m.im_func` ist das Funktions-Objekt, die der Methode entspricht.

9.7.1 Ausnahmen können Klassen sein

Benutzerdefinierte Ausnahmen sind nicht mehr auf String-Objekte eingeschränkt – es können auch Klassen sein. Mit diesem Mechanismus ist es möglich, erweiterbare Ausnahme-Hierarchien zu erzeugen.

Es gibt zwei neue gültige (semantische) Formen für die raise-Anweisung:

```
raise Class, instance

raise instance
```

In der ersten Form muss `instance` eine Instanz von `Class` oder einer davon abgeleiteten Klasse sein. Die zweite Form ist eine Abkürzung für:

```
raise instance.__class__, instance
```

Eine `except`-Klausel darf Klassen wie auch String-Objekte auflisten. Eine Klasse in einer `except`-Klausel ist kompatibel mit einer Ausnahme, wenn es dieselbe Klasse oder eine Oberklasse davon ist (aber nicht umgekehrt – eine `except`-Klausel, die eine abgeleitete Klasse auflistet, ist nicht kompatibel mit einer Oberklasse). Der folgende Code z.B. wird B, C, D in dieser Reihenfolge ausgeben:

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Man beachte, dass, falls die `except`-Klauseln umgekehrt werden (zuerst ‘`except B`’), B, B, B ausgegeben worden wäre – die erste passende `except`-Klausel wird ausgelöst.

Wenn eine Fehlermeldung für eine unbehandelte Ausnahme ausgegeben wird, die eine Klasse ist, wird der Klassenname ausgegeben, dann ein Doppelpunkt und ein Leerzeichen und schließlich die Instanz, nachdem sie mit der eingebauten Funktion `str()` konvertiert wurde.

KAPITEL

ZEHN

Was nun?

Die Lektüre dieses Tutoriums hat vermutlich Ihr Interesse an Python verstärkt, und nun drängen Sie darauf, Python bei Ihren eigenen täglichen Aufgaben zu verwenden. Was sollten Sie also tun?

Sie sollten die *Bibliotheks-Referenz* lesen oder zumindest durchblättern. Sie stellt umfangreiches (wenn auch dicht gedrängtes) Referenz-Material über Typen, Funktionen und Module bereit, die Ihnen eine Menge Zeit beim Schreiben von Programmen in Python sparen können. Die Standard-Python-Distribution beinhaltet *eine Menge* Quellcode in C wie auch in Python. Es gibt Module, um UNIX-Mailboxen zu lesen, Dokumente via HTTP zu empfangen, Zufallszahlen zu generieren, Kommandozeilen-Optionen zu parsen, CGI-Programme zu schreiben, Daten zu komprimieren und eine Menge mehr. Die Bibliotheks-Referenz durchzuforsten wird Ihnen einen Eindruck dessen vermitteln, was alles verfügbar ist.

Pythons Haupt-Web-Site ist `http://www.python.org/`. Sie enthält Code, Dokumentation und Verweise auf Python-bezogene Seiten im Web. Diese Web-Site wird an zahlreichen Orten weltweit gespiegelt, so z.B. in Europa, Japan und Australien. Ein Spiegel dürfte schneller als das Original sein, je nach Ihrem geographischen Aufenthaltsort. Eine informellere Site ist `http://starship.python.net`, die einen Haufen von Python-bezogenen, persönlichen Home-Pages enthält. Viele Leute stellen dort Software bereit.

Python-bezogene Fragen und Fehlerberichte können Sie an das Diskussionsforum (engl. newsgroup) comp.lang.python richten, oder sie an die Mailing-Liste python-list@python.org senden. Das Forum und die Mailing-Liste sind gekoppelt, so dass Nachrichten an eines automatisch auch an das andere geschickt werden. Es gibt ca. 120 Nachrichten pro Tag, die Fragen stellen (und beantworten), neue Eigenschaften vorschlagen und neue Module ankündigen. Bevor Sie dazu beitragen, prüfen Sie auch die Liste der häufig gestellten Fragen (engl. Frequently Asked Questions), auch FAQ genannt, unter `http://www.python.org/doc/FAQ.html` oder schauen Sie im ‘Misc/’-Verzeichnis der Python-Quellcode-Distribution hinein. Die Archive zu den Mailing-Listen sind unter `http://www.python.org/pipermail/` verfügbar. Die FAQ beantwortet viele der Fragen, die wieder und wieder kommen, und könnte auch die Lösung Ihres Problems bereits enthalten.

ANHANG

A

Interaktive Eingabe-Editierung und -Ergänzung

Einige Versionen des Python-Interpreters unterstützen das Editieren der aktuellen Eingabezeile und einen Ergänzungs-Mechanismus ähnlich dem der Korn- und GNU Bash-Shell. Dies ist unter Verwendung der *GNU Readline*-Bibliothek implementiert, die Emacs- und vi-ähnliches Editieren unterstützt. Diese Bibliothek hat ihre eigene Dokumentation, die ich hier nicht duplizieren werde. Die Grundlagen jedoch sind einfach erklärt. Das hier beschriebene interaktive Editieren und der Ergänzungs-Mechanismus sind optional unter UNIX und CygWin-Versionen des Interpreters verfügbar.

Dieses Kapitel dokumentiert *nicht* die Edititiermöglickeiten von Mark Hammonds PythonWin-Paket oder der Tk-basierten Umgebung IDLE, die mit Python verteilt wird. Die Kommandozeilenwiederholung, die in einem DOS-Terminal unter NT und einigen anderen DOS- und Windows-Varianten läuft, ist noch ein weiteres Biest.

A.1 Zeilen-Editierung

Falls unterstützt, ist die Eingabezeilen-Editierung immer dann aktiv, wenn der Interpreter einen primären oder sekundären Prompt ausgibt. Die aktuelle Zeile kann unter Verwendung der üblichen Steuerzeichen des Emacs-Editors editiert werden. Die wichtigsten davon sind: `C-A` (Control-A) bewegt den Zeiger an den Zeilenanfang, `C-E` ans Ende, `C-B` bewegt ihn eine Stelle nach links, `C-F` nach rechts. Die Rückschrittaste löscht Zeichen links vom Zeiger, `C-D` das Zeichen rechts davon. `C-K` löscht den Rest der Zeile rechts vom Zeiger, `C-Y` fügt den zuletzt gelöschten Teil wieder ein. `C-Unterstrich` macht die letzte Änderung rückgängig und kann mehrfach angewendet werden.

A.2 Zeilenwiederholung

Zeilenwiederholung funktioniert wie folgt. Alle nicht-leeren eingegebenen Zeilen werden in einem Puffer gespeichert und wenn ein neuer Prompt ausgegeben wird, befindet man sich auf einer neuen Zeile am unteren Ende dieses Puffers. `C-P` bewegt sich eine Zeile im Puffer nach oben (zurück), `C-N` eine nach unten. Jede Zeile im Puffer kann editiert werden. Ein Sternchen taucht vor dem Prompt auf, um zu markieren, dass eine Zeile modifiziert wurde. Beim Drücken der `Eingabetaste` wird die aktuelle Zeile an den Interpreter übergeben. `C-R` startet eine inkrementelle Rückwärtssuche, `C-S` startet eine Vorwärtssuche.

A.3 Tastenbelegungen

Die Tastenbelegungen und einige andere Parameter der Readline-Bibliothek können angepasst werden, indem Kommandos in einer Initialisierungs-Datei namens ‘~/.inputrc’ plaziert werden. Tastenbelegungen haben die Form:

```
key-name: function-name
```

oder

```
"string": function-name
```

und Optionen können gesetzt werden mit

```
set option-name value
```

Zum Beispiel:

```
# Ich bevorzuge vi-aehnliches Editieren:
set editing-mode vi

# Editiere Zeilen immer in einzelnen Zeilen:
set horizontal-scroll-mode On

# Belege einige Tasten neu:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Man beachte, dass die voreingestellte Belegung für `Tab` in Python die ist, einen Tabulator einzufügen, anstatt von Readlines standardmäßiger Dateinamenergänzungsfunktion. Wenn Sie darauf bestehen, können Sie das überschreiben, indem Sie

```
Tab: complete
```

in Ihrer '~/.inputrc'-Datei setzen. (Natürlich macht es das einem schwer, eingerückte Fortsetzungszeilen einzugeben. . .)

Die automatische Ergänzung von Variablen- und Modulnamen ist optional verfügbar. Um sie im interaktiven Modus des Interpreters zu aktivieren, fügen Sie folgendes in Ihre Startup-Datei ein: [1]

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

Das belegt die TAB-Taste mit der Ergänzungsfunktion so, dass bei doppeltem Drücken der TAB-Taste Ergänzungen vorgeschlagen werden. Die Funktion schaut dabei in den Namen von Python-Anweisungen, den aktuellen lokalen Variablen und den verfügbaren Modulnamen nach. Für qualifizierte Ausdrücke wie z.B. `string.a` wird sie den Ausdruck bis zum letzten '`.`' auswerten und dann Ergänzungen aus den Attributen des resultierenden Objektes vorschlagen. Man beachte, dass dabei Code in der Anwendung ausgeführt werden kann, falls ein Objekt mit einer `__getattr__()`-Methode Teil des Ausdrucks ist.

[1] Python führt den Inhalt der Datei aus, die durch die Umgebungsvariable $PYTHONSTARTUP bezeichnet wird, wenn Sie eine interaktive Interpreter-Sitzung starten.

A.4 Bemerkungen

Diese Eigenschaften stellen einen enormen Schritt vorwärts dar, verglichen mit bisherigen Versionen des Interpreters, wenngleich einige Wünsche offen bleiben. Es wäre nett, wenn die richtige Einrückung auf Fortsetzungszeilen vorgeschlagen werden könnte (der Parser weiß, ob ein Einrückungs-Token als nächstes verlangt wird). Der Ergänzungsmechanismus könnte die Symboltabelle des Interpreters verwenden. Ein Kommando zur Überprüfung (oder sogar zum Vorschlagen) von passenden Klammern, Anführungszeichen, etc. wäre ebenfalls sehr hilfreich.