TP0

October 23, 2015

1 Introduction au python

Le but de ce TP est de se familiariser avec le langage python que l'on utilisera tout au long du semestre.

1.1 L'environnement IPython / Jupyter

Nous effectuons ce TP dans un environnement de notebook **Jupyter**. Les différentes cellules peuvent contenir du code que l'on envoie à un noyeau python. Le noyeau les exécute et nous donne éventuellement un retour à afficher.

Pour exécuter une cellule, cliquez dessus puis tapez "Maj + Entrée"

```
In [ ]: 39 + 3
```

L'environnement est *persistant*. C'est à dire que je peux utiliser les mêmes variables et fonctions dans différentes cellules.

Par exemple, dans la cellule suivante, j'initialise une variable que j'affiche dans une autre cellule.

```
In [ ]: a = 5
```

```
In [ ]: print a
```

L'ordre dans lequel on exécute les cellules est important. Par exemple, si vous changez la valeur de a puis ré-exécuter les deux cellules, le print sera modifié.

1.2 La base de la base

1.2.1 Structure de tests

En python, un bloc est délimité par une instruction terminant par deux points ":" suivie d'un retour à la ligne avec **indentation**. Plus qu'une convention, l'indentation fait partie de la syntaxe du langage. Voici des exemples avec les blocs *if*, *else* et *elif*. **Exécutez les cellules pour plusieurs valeurs de** a.

```
In [ ]: a = 10
        if a < 0:
            print -a
        else:
            print a
```

```
In [ ]: a = 10
        if a < 0:
            print "a est négatif"
        elif a<=10:
            print "a est positif, plus petit ou égal à 10"
        else:
            print "a est plus grand que 10"
```

Voici un bloc mal indenté, si vous l'exécutez il vous retournera une erreur. Observez l'erreur car vous risquez de la rencontrer régulièrement !

```
In [ ]: a = 10
        if a < 0:
        print -a
        else:
        print a
```

Exercice : corrigez l'indentation des cellules suivantes

```
In [ ]: a = 10
         b = 5
            if a < b:
                print a
                else:
                    print b
```

```
In [ ]: a = 100
        b = 30
        if a > b:
            q = a/b
             r = a%b
            if r != 0:
                print a, " = ",q,"*",b," + ",r
            else:
                print a, " est divisible par ", b
```

```
In [ ]: m = 12342
        h = 0
        if m > 60:
            h = m/60
            m = m%60
         print h,":",m
```

```
In [ ]: a = 5
        b = 6
        if a > 5:
            if b > 5:
                print "a et b sont supérieurs à 5"
             else:
                print "seul a est supérieur à 5"
         else:
            if b > 5:
                print "seul b est supérieur à 5"
            else:
                print "a et b sont inférieurs à 5"
```

Les valeurs booléennes peuvent être combinées à l'aide des opérateurs **and** et **or**. Voici quelques rappels sur les booléens.

```
In [ ]: print True and True
        print True and False
        print True or True
        print True or False
```

Il existe aussi un opérateur unaire **not**.

```
In [ ]: print not True
        print not False
```

Exercice : voici 3 variables qui représentent une date. Testez si cette date est valide. Testez aussi avec d'autres valeurs.

Rappel : les années bissextiles sont celles qui sont divisbles par 4 mais pas par 100 ou alors divisbles par 400. Par exemple, 1996, et 2004 sont des années bissextiles (divisibles par 4 et pas par 100), 2000 aussi car divisble par 400, par contre pas 2100.

```
In [ ]: j = 31
        m = 3
        annee = 1995
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

1.2.2 Boucle while

La boucle while python est très similaire à celles des autres langages, sa syntaxe suit la logique python (avec ":" et indentation). Voici un premier exemple.

```
In [ ]: a = 0
        while a < 10:
            a+=1
            print a
```

Exercice : recopier et compléter le schéma ci-dessous pour que le nombre d'affichage soit : 1. **égal à** $n/2$ 2. **égal à** $\log(n)$ 3. **égal à** $\sqrt(n)$

```
In [ ]: a = 0
        n = 100
        while ?? < n:
            print a
            ??
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

1.2.3 types de variables

Même si on ne les déclare pas, les variables python ont un type. Simplement, ce type est **dynamique** c'est-à-dire que le type de la variable s'adapte à la valeur qu'elle prend.

```
In [ ]: a = 10
        print type(a)
        a = "test"
        print type(a)
```

Voici quelques types de base : int, long, float, string, bool.

Exercice : créez des variables et jouez sur les valeurs pour obtenir différents types. Pour chacun des types, essayez ce que donne l'addition avec d'autres types ou avec eux mêmes quand le résultat n'est pas évident.

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

Comme vous l'avez peut-être remarqué, python (contrairement à C ou C++) passe de lui même des int au long si l'entier devient trop grand.

Exercice : écrivez une boucle pour déterminer le premier entier qui est stocké sous forme de long

```
In [ ]:
```

Votre boucle prend trop de temps ? Vous pouvez la stopper avec le bouton stop "interupt kernel" marqué d'un carré en haut du notebook. Et surtout, vous pouvez trouver plus efficace...

```
In [ ]:
```

1.3 Listes et for

Les boucles for en python sont très différentes de la version que l'on trouve par exemple, en C, java ou C++. En fait, ce sont par défaut des boucles *foreach*. Pour pouvoir les utilsier correctement, on va d'abord apprendre à se servir d'une structure fondamentale en python : les listes.

1.3.1 Listes

Les listes python sont en fait l'équivalent des tableaux. Comme on l'a vu en cours, elles sont codées par des systèmes de tableaux dynamiques. En particulier, ce ne sont pas des listes chaînées.

Voici commone l'on crée une liste, la liste vide :

```
In [ ]: l = []
        print l
```

liste contenant des valeurs :

```
In [ ]: l = [1,2,3]
        print l
```

On peut accéder aux différentes valeurs grâce à leur indice :

```
In [ ]: print l[0]
        print l[1]
        print l[2]
```

Contrairement au C ou au C++, le python vérifie que l'indice ne dépasse pas la taille de la liste.

```
In [ ]: print l[3]
```

Pour connaître la taille de la liste, on utilise la fonction *len*.

```
In [ ]: len(l)
```

** Exercice : Créez une liste et écrivez une boucle while qui affiche les différentes valeurs de la liste **

```
In [ ]:
```

```
In [ ]:
```

De nombreuses opérations sont possibles sur les listes.
Ajout d'un élément :

```
In [ ]: l.append(5)
        print l
```

Extension par une autre liste :

```
In [ ]: l.extend([4,4,3])
        print l
```

Somme de 2 listes :

```
In [ ]: [1,2,3] + [4,5,6]
```

Attention cependant, cette seconde méthode est moins efficace que la première, elle nécessite la création d'une troisième liste dans laquelle on recopie les valeurs des deux listes additionnées.
** Exercice : créez une liste vide et ajoutez lui tous les nombres pairs jusqu'à 100 à l'aide d'une boucle while **

```
In [ ]:
```

1.3.2 La fonction range

La fonction *range* permet de créer des listes d'entiers

```
In [ ]: range(10)
```

On peut l'utiliser de différentes façons : * *range*(n) – les entiers de 0 à n-1 * *range*(i,j) – les entiers de i à j-1 * *range*(i,j,k) – les entiers de i à j-1 avec un pas de k
Quelques exemples :

```
In [ ]: range(2,15)
```

```
In [ ]: range(2,15,3)
```

** Exercice : créez la liste **1.** des entiers de 0 à 19, **2.** des entiers de 1 à 10, **3.** des entiers pairs de 0 à 20 **4.** des entiers de 10 à 1 **5.** des entiers négatifs de -20 à -1 **6.** des entiers négatifs de -1 à -20**

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

1.3.3 La boucle for

Comme je l'ai écrit plus haut, la boucle *for* est en fait un *foreach*, elle permet de parcourir les valeurs d'une liste.

```
In [ ]: l = [2,2,4,2,3]
        for v in l:
            print v
```

En la combinant avec la fonction *range*, on obtient le comportement classique de la boucle for.
Ainsi on traduit le
Pour i allant de 0 à n-1
par

```
In [ ]: n=10
        for i in range(n):
            print i
```

Remarque : comme on l'a vu plus haut, en python 2, range retourne une liste. Donc quand on veut parcourir les éléments i entre 1 et 1 000 000, on est obligé de créer en mémoire une liste d'un million... Pour éviter ce problème, on pourra utiliser à la place *xrange* qui permet de parcourir les valeurs sans créer la liste (c'est un itérateur). Note : en python 3, il n'y aura plus de problèmes.

```
In [ ]: for i in xrange(1000000):
            # du code (je fais pas de print, ça prendrait trop de place)
            pass
        print i
```

```
In [ ]:
```

** Exercice : utilisez une boucle for pour afficher le carré des entiers de 1 à 10**

```
In [ ]:
```

```
In [ ]:
```

** Ecrivez une boucle pour compter le nombre d'occurences de a dans la liste ci-dessous**

```
In [ ]: l = [2,3,3,2,4,4,2,3,2,4,5,2,2,3]
        a = 3
```

```
In [ ]:
```

```
In [ ]:
```

1.3.4 La compréhension de liste

La compréhension de liste (*list comprehension* en anglais) est un outil particulièrement pratique du python. Elle permet de combiner la création de liste avec une boucle *for* pour créer directement des listes spécifiques.
Exemple :
Les entiers pairs de 0 à 99 :

```
In [ ]: l = [2*i for i in xrange(50)]
        print l
```

** Exercice : en utilisant une syntaxe similare, créez la liste des i^2 pour $0 \leq i < 50$ ainsi que la liste des nombres impairs plus petits que 100**

In [ ]:

In [ ]:

Par ailleurs, on peut même rajouter des *if* dans la compréhension de liste. Voici une autre façon de créer les entiers pairs inférieurs à 100 :

```
In [ ]: l = [i for i in xrange(100) if i%2==0]
        print l
```

La condition à respecter se place après le *for* un peu comme dans un ensemble mathématique :

$$\{i; i \in [[0, 99]], i \text{ est pair}\}$$

Exercice : en utilisant une syntaxe similaire, créez la liste des entiers divisibles par 3, puis la liste des entiers divisibles à la fois par 2 et par 3

In [ ]:

In [ ]:

1.4 Fonctions

La dernière notion que nous verrons aujourd'hui est celle de **fonction**. Elle n'est pas différente en python de ce qu'on voit dans d'autres langages. La syntaxe suit la logique des autres blocs python.

```
In [ ]: def maFonction(a, b):
            if a > b:
                return a
            else:
                return b
```

L'exécution du bloc ci-dessus n'a rien affiché, mais elle a permis de définir *maFonction* que l'on pourra utiliser par la suite.

```
In [ ]: print maFonction(4,5)
        print maFonction(40,-2)
        print maFonction(-5,-6)
        print maFonction(0.222,4)
```

Si on essaie d'affecter à une variable un appel de fonction qui ne retourne pas de valeur, on récupère un objet null représenté par le mot clé python *none*.

```
In [ ]: def rienDuTout(a):
            return

In [ ]: b = rienDuTout(10)
        print b
        print b is None
```

Exercice : Voici un prototype de fonction très basique. Lisez la documentation, complétez le corps de la fonction, exécutez là puis exécutez les tests en dessous.

(Notez que les triples guillemets sont utilisés en python pour écrire des blocs de commentaires, pour une simple ligne de commentaire, on utilisera le caractère #)

```
In [ ]: """
        Retourne la valeur absolue d'en entier
        Input :
            - a, un entier
        """
        def absolue(a):
            # écrire le code
```

```
In [ ]: assert(absolue(4)==4)
        assert(absolue(-3)==3)
        assert(absolue(0)==0)
        print "Les tests passent !"
```

L'instruction *assert* vérifie simplement qu'une expression est *True*. Si c'est le cas, rien ne se passe, sinon, cela lève une exception.

```
In [ ]: assert(True)
```

```
In [ ]: assert(False)
```

** Exercice : même chose pour les fonctions suivantes **

```
In [ ]: """
        Retourne a à la puissance b
        Input :
            - a, un nombre (entier ou float)
            - b, un entier positif
        """
        def puissance(a,b):
            # pour l'instant, on se contente de la version en O(n) avec une simple boucle
```

```
In [ ]: assert(puissance(2,3)==2**3)
        assert(puissance(3,4)==3**4)
        assert(puissance(-1,5)==-1)
        print "Les tests passent !"
```

```
In [ ]: """
        Retourne la somme des entiers de 1 à n
        Input :
            - n, un entier
        """
        def sommen(n):
            # écrire le code
```

```
In [ ]: assert(sommen(10)==55)
        assert(sommen(13)==91)
        print "Les tests passent !"
```

```
In [ ]: """
        Retourne l'élément maximum d'une liste
        Input :
            - l, une liste
        """
        def maximumListe(l):
            # écrire le code
```

In [ ]:
```
assert(maximumListe([2,4,2,5,7,7,2])==7)
assert(maximumListe([14,4,2,5,7,7,2])==14)
assert(maximumListe([2,4,2,5,7,7,23])==23)
```

In [ ]: