Premiers calculs en Python

In [ ]:

```
```

In [1]:

```
a=1.2
print a
b=a+3
print b
c=1
d=float(1)
e=1.0
print type(a)
print type(b)
print type(c)
print type(d)
print type(e)
```

```
1.2
4.2
<type 'float'>
<type 'float'>
<type 'int'>
<type 'float'>
<type 'float'>
```

Affectation : Ici, a est une variable, en interne elle a été automatiquement typée en flottant « float » parce qu'il y a un point décimal. a est l'identifiant de la variable (attention à ne pas utiliser le mots réservés comme identifiant), = est l'opérateur d'affectation.

Forcer le typage d'une variable : d = float(1) Même sans point décimal, d est considéré comme float.

Connaître le type d'un objet type(nom_de_variable) affiche le type interne d'une variable

Bases de Python

1.1 Les Types simples

- les entiers sont représentés par:
  - le type `int` entre -2^{31} et 2^{31},
  - le type `long` au-delà, en valeur exacte et uniquement limité par la mémoire disponible. Il est parfois caractérisé par le suffixe `L` à la fin de la valeur littérale;
- les réels sont représentés par le type `float`, en virgule flottante avec environ 15 chiffres (décimaux) significatifs. Les réels représentables sont ceux de valeur absolue comprise entre 10^{-300} et 10^{300} environ. Deux valeurs spéciales sont également possibles pour indiquer l'échec de la représentation: `nan` (not a number) et `inf` (infinity);
- les nombres complexes sont représentés par le type `complex`, implémenté en mémoire par deux `float`. Ils s'écrivent sous la forme `a + bj` où `a` et `b` sont les représentations **littérales** de deux `float`.

- le type booléen : `bool` peut prendre les valeurs `True` ou `False`. Attention aux majuscules, python y est sensible

In [2]:

```
# Entiers
print 1234567890
print type(1234567890)
print 1234567890 * 1234567890
print type(1234567890 * 1234567890)
1234567890 * 1234567890
```

```
1234567890
<type 'int'>
1524157875019052100
<type 'int'>
```

Out[2]:

```
1524157875019052100
```

In [3]:

```
# Réels
print 0.5
print type(0.5)
print 4.2 * 10000000000000000000
print .456e-3
print 1e500
print 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 - 1.0
```

```
0.5
<type 'float'>
4.2e+19
0.000456
inf
-1.11022302463e-16
```

In [4]:

```
# Complexes
print 3 + 4j
print type(3 + 4j)
print abs(3 + 4j)
print 1j    # Correct
print j     # Incorrect: il manque le "1"
```

```
(3+4j)
<type 'complex'>
5.0
1j

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
 last)
<ipython-input-4-92eca83f9e0f> in <module>()
      4 print abs(3 + 4j)
      5 print 1j    # Correct
----> 6 print j     # Incorrect: il manque le "1"

NameError: name 'j' is not defined
```

In [6]:

```
# Booléens
print True
print type(False)
```

```
True
<type 'bool'>
```

1.2 Types itérables

Il s'agit du regroupement de plusieurs valeurs dans les cases d'un conteneur. Par défaut, `python` en propose au moins quatre:

- le type `tuple`, séparé par des virgules et enclos par des parenthèses. Une fois un `tuple` créé, il est impossible de modifier ses éléments ou sa taille. Les éléments d'un tuple ne sont pas forcément de même type : collection d'objets hétérogènes. Accès indicé

In [7]:

```
#définition d'un tuple
t1 = (2,6,8,10,15,26)
print(t1)
#taille du tuple
print(len(t1))
#accès indicé
a = t1[0]
print(a)
#modification ?
t1[2] = 3
print t1
```

```
(2, 6, 8, 10, 15, 26)
6
2

---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
 last)
<ipython-input-7-5c009c220f9b> in <module>()
      8 print(a)
      9 #modification ?
---> 10 t1[2] = 3
     11 print t1

TypeError: 'tuple' object does not support item assignment
```

In [8]:

```
#plage d'indices
b = t1[2:5]
print(b)
#autre plage
c = t1[:4]
print(c)
```

```
(8, 10, 15)
(2, 6, 8, 10)
```

In [9]:

```
#Un peu plus loin en tuples... concaténation
t2 = (7,9,31)
t3 = t1 + t2
print(t3)

#replication
t4 = 2 * t2
print(t4)

#tuples d'objets hétérogènes
v1 = (3,6,"toto",True,34.1)
print(v1)

#tuple de tuples
x = ((2,3,5),(6,7,9),(1,8))
print(x)

#accès indicé
print(x[2][1])

#accès aux tailles
print(len(x))
```

```
(2, 6, 8, 10, 15, 26, 7, 9, 31)
(7, 9, 31, 7, 9, 31)
(3, 6, 'toto', True, 34.1)
((2, 3, 5), (6, 7, 9), (1, 8))
8
3
```

- le type `list`, séparé par des virgules et enclos par des crochets. Le plus polyvalent En gros, une liste, c'est comme un tuple, mais DE TAILLE DYNAMIQUE et MODIFIABLE

In [10]:

```
#définition d'une liste
L1 = [2,6,8,10,15,26]
print(L1)
#taille de la tuple = 6
print(len(L1))
#accès indicé = 2
a = L1[0]
print(a)
#modification ! Possible !
L1[2] = 3
print(L1)
```

```
[2, 6, 8, 10, 15, 26]
6
2
[2, 6, 3, 10, 15, 26]
```

Mauvaise nouvelle... une manière un peu étrange de définir les liste d'entiers de 1 à n... ATTENTION !

In [11]:

```
#liste de base : les entiers de 1 à n
n=20
l1=range(1,n)
print l1
l2=range(n)
print l2
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Les autres mécanismes associés aux tuples sont transposables (plages d'indices, objets hétérogènes, concaténation, réplication, liste de listes...)

In [12]:

```
#autre liste
L2 = [32,69,28,69]
print(L2)
#ajout
L2.append(21)
print(L2)
#insertion à l'indice 1
L2.insert(1,53)
print(L2)
#suppression elt n°3
del L2[3]
print(L2)
#accès + suppression elt n°1
a = L2.pop(1)
print(a)
#inversion
L2.reverse()
print(L2)
#étendre
L2.extend([34,55])
print(L2)
```

```
[32, 69, 28, 69]
[32, 69, 28, 69, 21]
[32, 53, 69, 28, 69, 21]
[32, 53, 69, 69, 21]
53
[21, 69, 69, 32]
[21, 69, 69, 32, 34, 55]
```

Une bonne nouvelle : les "List Compréhension" C'est un objet simple (et concis) pour générer une liste à partir d'une fonction ou d'une autre liste.

In [13]:

```
#Exemple : Monter tous les chiffres de "source" au carré
#Façon naïve (et longue) avec une boucle :
source = [1,5,8,12,7]
resultat1 = []
for v in source:
    resultat1=resultat1+[v**2]
print(resultat1)

#Façon "List Compréhension"
resultat2 = [v**2 for v in source]
print(resultat2)
```

```
[1, 25, 64, 144, 49]
[1, 25, 64, 144, 49]
```

Une mauvaise nouvelle : Une variable de type liste est en fait une référence => Problème de copies...

In [14]:

```
#L3
L3 = [61,92,17]
print(L3)
#affectation ?
L4 = L3
print(L4)
#modification d'une valeur
L4[1] = 55
#répercussions sur L4 bien sur... mais aussi sur L3 :-/
print(L4)
print(L3)
```

```
[61, 92, 17]
[61, 92, 17]
[61, 55, 17]
[61, 55, 17]
```

In [15]:

```
from copy import copy
#L3
L3 = [61,92,17]
print L3
#copie des valeurs (en python 3 L3.copy()) fonctionne aussi
L4 = copy(L3)
print(L4)
L4[1] = 55
print(L4)
print(L3)
```

```
[61, 92, 17]
[61, 92, 17]
[61, 55, 17]
[61, 92, 17]
```

- le type `str`, sans séparateur et enclos par des guillements simples ' ou doubles ". Il ne peut contenir que des caractères;

- le type `set`, séparé par des virgules et enclos pas des accolades. Chaque élément ne peut apparaitre qu'une fois. Il n'est pas possible d'accéder aux éléments directement par indexage. Il n'est pas possible de créer l'ensemble vide par la valeur littérale `{}`, à la place utiliser `set()` ou `set([])`;

In [16]:

```
#définir une chaîne
s1 = "bonjour le monde"
print(s1)
#longueur
long = len(s1)
print(long)
#accès indicé
s2 = s1[:7]
print(s2)
#non modifiable
s1[0] = "B"
```

```
bonjour le monde
16
bonjour

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
 last)
<ipython-input-16-8dffb060792f> in <module>()
      9 print(s2)
     10 #non modifiable
---> 11 s1[0] = "B"

TypeError: 'str' object does not support item assignment
```

In [17]:

```
#méthodes associées
S = s1.upper()
print(S)
#recherche d'une sous-chaîne
id = S.find("JO")
print(id)
#nb d'occurences
nb = S.count("ON")
print(nb)
#remplacement de « O » par « A »
SA = S.replace("O","A")
print(SA)
```

```
BONJOUR LE MONDE
3
2
BANJAUR LE MANDE
```

1.3 Opérateurs

Selon le type des opérandes, les opérateurs se comportent différemment. On sera par exemple attentif au fait que les opérateurs `//` (division euclidienne) et `/` (division) coïncident sur $\mathbb{Z} \times \mathbb{Z}$ en Python2. Si l'on souhaite un résultat flottant, il faut transtyper explicitement l'un des opérandes en `float`.

Opérateur	Opération
`=`	Affectation de variable. On peut affecter plusieurs variables en une fois, en regroupant variables et valeurs à affecter dans des conteneurs. Par exemple: `a,b=1,2` ou encore `[x,y]=(0,0)`
`+`	Addition, concaténation entre itérables de même type
`*`	Multiplication, répétition d'un itérable
`-`	Négation, opposé ou soustraction pour les valeurs numériques. Différence pour les ensembles
`/`	Division
`//`	Division entière
`**`	Exposant
`%`	Modulo
`not` `or` `and`	Opérations booléennes
`<` `<=` `==` `!=` `<>` `>=` `>`	Comparaison $< \leq = \neq \neq \geq >$
`in`	Présence d'une valeur dans un conteneur
`not in`	Absence d'une valeur dans un conteneur
`is`	Tester si une valeur est d'un type donné
`|` `&` `^`	Opérations bits à bits sur les valeurs numériques. Union, intersection ou différence symétrique pour les ensembles

Python propose aussi des opérateurs combinés avec une affectation, dits *in situ*. En voici quelques-uns.

Opérateur	Opération
`x += y`	`x = x + y`
`x *= y`	`x = x * y`
`x -= y`	`x = x - y`
`x /= y`	`x = x / y`
`x //= y`	`x = x // y`
`x **= y`	`x = x ** y`
`x %= y`	`x = x % y`

In [18]:

```
# Divisions
print 2 / 3
print 2 / 3.
print 2 / float(3)
print 1.5 // 0.6
```

```
0
0.666666666667
0.666666666667
2.0
```

In [19]:

```
# Conteneurs
print [1, 2] + [3, 4, 5]
print "bla" * 5
print {1, 2, 3} - {2, 3, 4}
print {1, 2, 3} | {2, 3, 4}
```

```
[1, 2, 3, 4, 5]
blablablablabla
set([1])
set([1, 2, 3, 4])
```

In [20]:

```
# Comparaison
print ["abc", "def"] > ["abc", "dee"]
print [] > [1]
print 2 in (1, 2, 3)
print 1.0 is float
```

```
True
False
True
False
```

2 Structures de contrôle

La structure syntaxique en Python est déterminée par les niveaux d'indentation du code.

Les blocs de code relatifs à une structure de contrôle (boucle ou branchement conditionnel) sont introduits par le symbole `:` à la fin de la ligne qui précède et doivent être indentés d'un niveau supplémentaire par rapport au niveau parent. Le retour au niveau d'indentation précédent signale au compilateur la fin du bloc.

En principe, on indente de quatre espaces supplémentaires à chaque niveau. Ceux-ci peuvent être obtenus facilement grâce à la touche de tabulation.

2.1 Branchements conditionnels

La syntaxe générale est

```
    if condition1:
        instructions
    elif condition2:
        instructions
    elif condition3:
        instructions
    else:
        instructions
```

In [21]:

```
a = -5

if a < 0:
    print 'a est negatif'
elif a>0:
    print 'a est positif'
else:
    print 'a est nul'
```

```
a est negatif
```

2.2 Boucles for (itération)

La syntaxe générale d'une boucle for est

```
    for variable in iterable:
        instructions
```

In [22]:

```
l = (1, 2, 3)

# Première méthode pour parcourir un itérable: par parcours direct
for x in l:
    print x

# Deuxième méthode pour parcourir un itérable: par parcours indexé
for k in range(len(l)):
    print l[k]
```

```
1
2
3
1
2
3
```

In [23]:

```
for c in "Hello":
    print c
```

```
H
e
l
l
o
```

In [24]:

```
# Somme des éléments d'une liste de réels
source=[1,2,3,6,10]
somme=0
for x in source:
    somme=somme+x
print somme

#Somme des entiers de 1 à n
n=20
somme=0
for k in range(1,n+1):
    somme=somme+k
print somme
```

```
22
210
```

In [25]:

```
# Somme des entiers pairs de 1 à n
n=25
s=0
for k in range(1,n+1):
    if k%2==0:
        s=s+k
print s

# verification
pairs=[k for k in range(1,n+1) if k%2==0]
so=sum(pairs)
print s==so
```

```
156
True
```

2.3 Boucles while (itération)

La syntaxe générale d'une boucle While est

```
while condition:
    instructions
```

3 Fonctions

3.1 Fonctions de base

Comme on l'a vu, une fonction s'évalue avec des parenthèses et peut également s'affecter à une variable (dans ce cas sans parenthèses).

Les fonctions peuvent prendre plusieurs paramètres, séparés par des virgules. Certains paramètres peuvent avoir des valeurs par défaut, il est alors possible de les omettre lors de l'évaluation.

Pour consulter la rubrique d'aide relative à une fonction, on peut taper son nom suivi ou précédé d'un point d'interrogation.

Voici une liste de quelques fonctions prédéfinies.

Fonction	**Usage**
`all(c)` ou `any(c)`	Indique si respectivement tous ou l'un des éléments du conteneur `c` convertis en booléens valent `True`
`input(prompt = '')`	Demande à l'utilisateur d'entrer une valeur avec une invite optionnelle
`len(c)`	Renvoie la longueur (`int`) du conteneur `c`
`map(f, c)`	Renvoie le conteneur de même type que `c` contenant les valeurs de la fonction `f` appliquée aux éléments de `c`
`min(c)` ou `max(c)`	Minimum et maximum d'un conteneur dont les éléments sont comparables entre eux
`range(start = 0, stop, step = 1)`	Renvoie une liste contenant les entiers de `min` à `max` exclus par pas de longueur `step`
`round(x, n = 0)`	Renvoie l'arrondi de la valeur numérique `x` sous flottante à (au plus) `n` chiffres décimaux après la virgule
`sum(c)`	Renvoie la somme des éléments du conteneur `c`
`type(x)`	Renvoie le type de `x`

In [26]:

```
l = range(100)
print l
print len(l)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 3
7, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54,
55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 7
2, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
100
```

In [27]:

```
l = range(-10, 0)
print l
print sum(l), min(l), max(l)
print map(abs, l)
```

```
[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1]
-55 -10 -1
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

In [29]:

```
x = input("Entrer une valeur : ")
print "La valeur est de type :", type(x)
```

```
Entrer une valeur : 2
La valeur est de type : <type 'int'>
```

3.2 Fonctions de classes

En réalité, la plupart des types (y compris numériques) sont représentés par Python en interne comme des classes. Une classe est un agrégat de données et de fonctions. Pour accéder à ceux-ci, on postfixe l'instance par un point suivi du nom du *champ* (donnée) ou de la *méthode* (fonction) correspondant.

Par exemple, une valeur de type `complex` possède:

- des champs `real` et `imag`;
- une méthode `conjugate()`.

In [30]:

```
print (5-3j).imag
z = 2 + 3j
print z.real
print z.conjugate()
```

```
-3.0
2.0
(2-3j)
```

Une méthode simple pour connaitre l'ensemble des champs et méthodes d'une classe, est de commencer à taper son nom suivi d'un point, puis d'appuyer sur la touche de tabulation.

Voici une courte liste de quelques champs et méthodes utiles.

Classe	Champ ou méthode	Utilisation
`complex`	`imag` `real`	Parties réelles et imaginaires
	`conjugate()`	Renvoie le conjugué
`list`	`append(x)`	Ajoute un élément en fin de liste
	`count(x)`	Renvoie le nombre d'occurences de `x`
	`index(x)`	Renvoie l'index de la première occurence

Classe	Champ ou méthode	Utilisation
	`insert(index, x)`	Insère un élément à l'index spécifié
	`pop(index = -1)`	Enlève un élément et renvoie sa valeur
	`remove(x)`	Enlève la première occurence de `x`
	`reverse()`	Inverse l'ordre des éléments
	`sort()`	Classe la liste
`str`	`find(substr)`	Renvoie la première occurence d'une sous-chaîne
	`upper()`	Renvoie la chaîne obtenue en remplaçant les minuscules par des majuscules
`set`	`add(x)`	Ajoute un élément (sauf s'il y est déjà)
	`clear()`	Enlève tous les éléments
	`discard(x)`	Enlève un élément (s'il y est)
`tuple`	`count(x)`	Renvoie le nombre d'occurences de `x`
	`index(x)`	Renvoie l'index de la première occurence

In [31]:

```
print "bonjour".find("jour")
print [1, 3, 2, 5, 1, 4, 7, 1, 8, 0, 6, 1, 2].count(1)
print (2+3j).conjugate()
```

```
3
4
(2-3j)
```

3.3 Définition de nouvelles fonctions

Voici un exemple de syntaxe.

```
def f(x, y=4, z=5):
    # instructions
    return valeur
```

Cette instruction stocke dans la variable `f` une fonction à trois variables `x`, `y` et `z`. Les variables `y` et `z` ont des valeurs par défaut. Le tableau suivant regroupe différentes manières d'appeler `f` ainsi que les valeurs prises par les paramètres `x`, `y` et `z`.

Appel	x	y	z
`f(1, 2, 3)`	1	2	3
`f(x=1, y=2, z=3)`	1	2	3
`f(1, 2)`	1	2	5
`f(1)`	1	4	5
`f(1, z=3)`	1	4	3