

Chapitre 0 : la boîte à outils pour commencer à "pythoner"

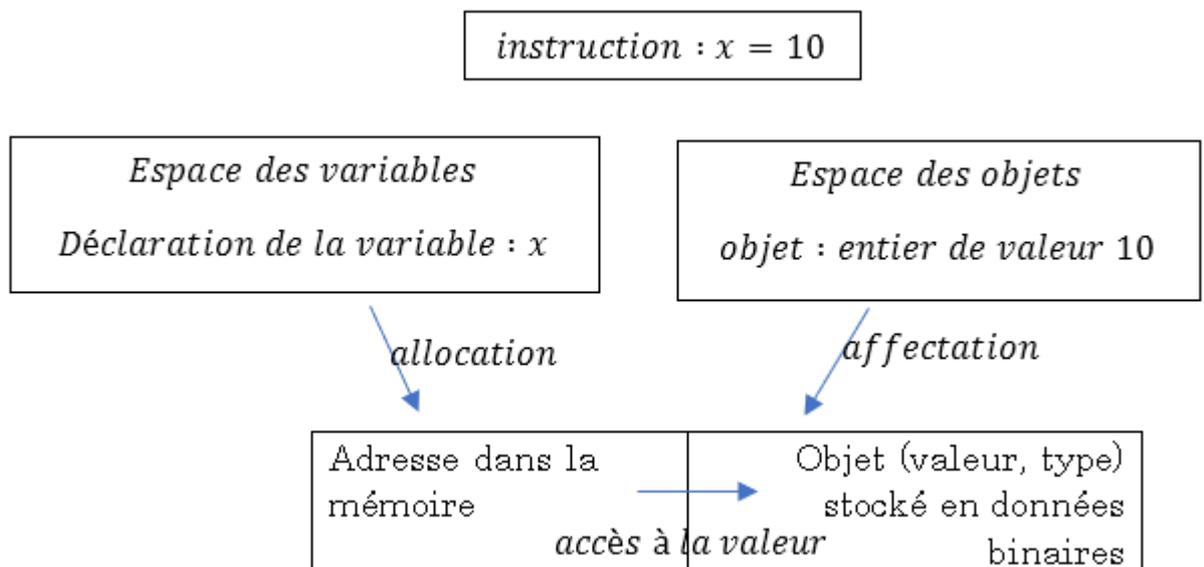
Quelques généralités :

- Pour installer python sous windows : <http://winpython.github.io/> (<http://winpython.github.io/>)
- pour installer python sous mac ou linux : <https://pyzo.org/start.html> (<https://pyzo.org/start.html>)
- plusieurs IDE sont possibles : spyder, pyzo, pycharm, jupyter, tonny,...
- Vous pouvez aussi travailler en ligne avec python tutor
- Vous pouvez travailler sur l'ENT avec Capytal

I- Opérateur d'affectation "=" (code quizinière : DMK4L7)

Analysons les effets de l'instruction très simple $x = 10$:

- 1) Parmi tous les **objets** possibles (entiers, nombre décimaux, données textuelles, ...), c'est l'objet "entier 10" qui a été choisi (avec toutes ses propriétés mathématiques)
- 2) Une variable x est **déclarée**
- 3) Le signe "=" réalise **une allocation** (un espace réservé dans la mémoire) et **une affectation** (x est liée à la valeur 10 par l'adresse mémoire) : on dit que la variable x référence l'objet 10 dans la mémoire



```
In [10]: 1 # Exemple
        2 x=10
        3 print(id(x)) #permet d'avoir l'adresse dans la mémoire de x
```

1997239839312

```
In [11]: 1 y=x
2 print(y, id(y)) #même adresse, même valeur => permet une économie de la
```

10 1997239839312

```
In [15]: 1 x=11.0#il est possible de faire un typage dynamique, ici x est associ
2 print(x)
3 print(id(x)) #Nouvelle allocaction et affectation de la variable x
4 #(la nouvelle affectation délée x de son ancienne valeur)
```

11.0
1997343293072

```
In [16]: 1 z=x+10
2 print(z) #on a accès à la valeur de x pour d'autres calculs
```

21.0

```
In [18]: 1 x=x+10#le typage dynamique permet aussi un lecture de l'ancienne vale
2 print(x)
3 print(id(x))
```

22.0
1997343290928

```
In [5]: 1 #Rq : le x=x+10 et le x+=10
2 x=10
3 x+=10
4 print(x) #20
5 #cependant cette notation présente quelques subtilités
6 L=[0,1,2,3]
7 L=L+"a" #fonctionne
8 print(L)
9 L+="a" #fonctionne !
10 print(L)
11 L=L+"a" #ne fonctionne pas
```

20
[0, 1, 2, 3, 'a']
[0, 1, 2, 3, 'a', 'a']

```
-----
-----
TypeError                                Traceback (most recent call
last)
<ipython-input-5-7845a97b4050> in <module>()
     9 L+="a"#fonctionne !
    10 print(L)
----> 11 L=L+"a"#ne fonctionne pas

TypeError: can only concatenate list (not "str") to list
```

```
In [1]: 1 #Rq : la copy par référence a l'avantage de limité l'encombrement de
2 #mais s'accompagne d'effet de bord
3 L=[0,1,2,3,4]
4 L_bis=L
5 L_bis.append(5)
6 print(L) #effet de bord
```

[0, 1, 2, 3, 4, 5]

II- Les opérations numériques sur les entiers et les nombres décimaux (code quizinière : 99BZ9Y)

```
In [2]: 1 x,y=10,3
2 #la somme
3 print(x+y)
```

13

```
In [3]: 1 #la soustraction
2 print(x-y)
```

7

```
In [4]: 1 #produit de x par y
2 print(x*y)
```

30

```
In [5]: 1 # x puissance y
2 print(x**y)
```

1000

Rappels sur la division euclidienne :

- le dividende a : quantité à diviser
- le diviseur b : quantité par laquelle on divise
- le quotient Q, le reste R : $a//b$, $a\%b$ tels que $a=b*Q+R$

```
In [6]: 1 # Division de x par y
2 print(x/y) # on pourra remarquer que le nombre limité de chiffres sign.
3 #et l'erreur d'arrondi liée au principe
4 # de numérisation des nombres décimaux abouti à une erreur
5 # si on numérise sur 4 bits alors :  $1*2^1+1*2^0+0*2^{-1}+1*2^{-2}=2+1+0.2$ 
6 #on comprend qu'en limitant le nombre de bits sur ce type de nombre n
```

3.3333333333333335

```
In [25]: 1 # Quotient Q de la division entière de x par y
2 print(x//y) # x = y*Q + R (où R est le reste, x est le dividende), y
```

3

```
In [26]: 1 # Reste de la division de x par y
2 print(x%y)
```

1

```
In [27]: 1 # doublet (Quotient, Reste)
2 divmod(x,y) #un tuple est renvoyé
```

Out[27]: (3, 1)

```
In [13]: 1 #valeur absolue
2 abs(-x)
```

Out[13]: 10

```
In [7]: 1 #Conversion d'un nombre décimal en entier le plus proche de 0
2 print(int(x/y))
3 #rq : int(3.9) donne 3 !
```

3

```
In [8]: 1 #conversion d'un entier en nombre décimal
2 print(float(x))
```

10.0

III- Les booléens (code quizinière : E7Y8ZQ)

- x or y : Vrai si x et/ou y sont vrais

```
In [33]: 1 print(True or True) #True
2 print(False or False) # False
3 print(False or True) #True
4 print(True or False) #True
5 print(True or rien) #True car opérateur paresseux (pas d'évaluation du
```

```
True
False
True
True
True
```

- x and y : Vrai si x et y sont vrais

```
In [9]: 1 print(True and False) #False
2 print(False and False) # False
3 print(False and True) #False
4 print(False and rien) #False car opérateur paresseux (pas d'évaluation
5 print(True and True) # True
```

```
False
False
False
False
True
```

- not x : Vrai si x faux, faux si x vrai

```
In [21]: 1 print(not False) #True
2 print(not True) #False
3 #les booléens peuvent être vis comme un sous type des entiers
4 print(1==True) #True
5 print(0==False) #True
6 print(not 0) #True
7 print(not 1) #False
8 print(not 10.4) #False (tout nombre différent de 0 peut être interprét
9 i=10
10 while i:
11     i=i-1
12     print(i)
```

```
True
False
True
True
True
False
False
9
8
7
6
5
4
3
2
1
0
```

A noter que ces opérateurs suivent un ordre de priorité (not>and>or>xor)

```
In [1]: 1 #XOR est python est obtenu avec ^
2 print(True^True)
3 print(True^False)
4 print(False^True)
5 print(False^False)
```

```
False
True
True
False
```

Certains opérateurs sur les types numériques renvoient un résultat booléen :

```
In [1]: 1 x=10
2 y=3
3 print(x<y) # x est-il plus strictement plus petit que y ?
4 print(x>y) # x est-il plus strictement plus grand que y ?
5 print(x<=y) # x est-il inférieur ou égale à y ?
6 print(x>=y) # x est-il inférieur ou égale à y ?
7 print(x==y) # x est-il identique à y ?
8 print (x!=y) # x est-il différent de y ?
```

```
False
True
False
True
False
True
```

```
In [1]: 1 #avec les erreurs d'arrondi, on met en évidence que chercher des cond.
2 #avec des float est dangereux
3 print(0.1+0.2==0.3)
```

```
False
```

IV- Les fonctions (code quizinière : OXD95W)

La définition d'une fonction se fait à l'aide du mot clé *def* :

```
In [18]: 1 def nom_fonction(variable1,variable2):
2     #corps de la fonction indenté
3     return #pour renvoyer un résultat
```

```
In [10]: 1 #exemple f1(x)=2x+4
2 def f1(x):
3     y=2*x+4
4     return y
5 valeur = f1(2) # On appelle la fonction f1 et on récupère f1(2)
6 print(valeur)
7 #Pour généraliser à toute fonction affine
8 def f2(a,b,x):
9     return a*x+b
10 print(f2(2,4,2))
```

8
8

```
In [11]: 1 #Rq : renvoyer et afficher un résultat :
2 # sous jupyter : si la cellule ne contient qu'une fonction alors le r
3 # =>le print n'est pas n'est pas nécessaire en présence du return
4 f1(2)
```

Out[11]: 8

```
In [12]: 1 #le return permet de récupérer la valeur (mais on perd l'affichage) :
2 a=f1(2) #on peut alors utiliser la variable a pour d'autres calculs
```

On distingue les variables locales et globales :

- les variables globales ne sont pas définies dans le corps de la fonction. S'il s'agit d'un nombre, cette variable ne peut être modifiée,
- les variables locales sont définies dans le corps de la fonction et leur lecture n'est possible que pendant l'appel de cette fonction

Si une variable est appelée dans le corps de la fonction alors elle est d'abord recherchée dans l'espace des variables locales puis globales

```
In [13]: 1 #exemple 1 : si a n'est que variable globale
2 a=2
3 def f3():
4     print(a)
5 print(a)
6 f3()
7 print(a)
```

2
2
2

```
In [23]: 1 #exemple 2 : si a est variable locale et globale
2 a=2
3 def f4():
4     a=3
5     print(a)
6 print(a)
7 f4()
8 print(a)
```

```
2
3
2
```

```
In [2]: 1 #exemple 3 : si a est passé en argument alors il appartient à l'espace
2 a=2
3 def f5(a):
4     a=a+2
5     b=a+3
6     return a,b
7 print(a)
8 print(f5(4)) #renvoie un tuple
9 print(a)
```

```
2
(6, 9)
2
```

In [24]:

```

1  #exemple supplémentaire :
2  a=2
3  def f6():
4      a=a+2#pose pb car sur on cherche à modifier une variable globale
5      b=a+3
6      return a,b
7  print(a)
8  print(f6())
9  print(a)

```

2

UnboundLocalError

Traceback (most recent call

last)

<ipython-input-24-efe74b3944c2> in <module>

```

6      return a,b
7  print(a)
----> 8  print(f5())
9  print(a)

```

<ipython-input-24-efe74b3944c2> in f5()

```

2  a=2
3  def f5():
----> 4      a=a+2
5      b=a+3
6      return a,b

```

UnboundLocalError: local variable 'a' referenced before assignment

In [14]:

```

1  #exemple supplémentaire
2  a=2
3  def f6():
4      global a#permet de modifier une variable gloable non mutable....à
5      a=a+2#a est modifiable
6      b=a+3
7      return a,b
8  print(a)
9  print(f6())
10 print(a)

```

2

(4, 7)

4

V- Les chaînes de caractères

Les données textuelles ou chaînes de caractères ont les propriétés suivantes :

- création :

```
In [16]: 1 mot1="Bonjour"#création manuelle
2 mot2='à l\'ensemble de \
3 la classe'#nécessité de mettre un antislash => double côte mieux !
4 #au passage l'anti-slash permet d'écrire une code sur 2 lignes une in.
5 # si on veut vraiment afficher sur deux lignes =>\n
6 phrase=mot1+"\n"+mot2
7 print(phrase)#concaténation
```

Bonjour
à l'ensemble de la classe

- Conversion:

```
In [17]: 1 #Effectué par le mot clé $str$
2 phrase=phrase+" rentrée "+str(2024)#concaténation possible si de même
3 print(phrase)
```

Bonjour
à l'ensemble de la classe rentrée 2024

- Sélection ou tranche

```
In [18]: 1 #sélection d'une lettre (indice de départ nul)
2 print(phrase[0])#le premier indice est à 0
3 #selection d'une sous chaîne [indice de départ : indice fin exclu]
4 print(phrase[0:7])
5 #remarque, on peut choisir le pas :
6 print(phrase[0:7:2])#ici pas de 2
```

B
Bonjour
Bnor

- longueur d'une chaîne

```
In [16]: 1 print(len(phrase))
```

44

VI- Structures conditionnelles (code quizinière : 54AXL3)

On a typiquement le squelette suivant :

```
In [13]: 1 x=float(input("rentre un nombre"))
2 if x >0 :#on notera la présence d'une indentation
3     print ("x est strictement positif")#instruction réalisée si la 1e
4 elif x<0 :
5     print ("x est strictement négatif")#instruction réalisée si la 1e
6 else :
7     print("x est nul")#instruction réalisée si les deux autres condit.
```

rentre un nombre 10

x est strictement positif

Rq : if et elif sont bien différents : un if est toujours testé, un elif et un else ne sont pas testés si le if précédent est True

```
In [17]: 1 x=11
2 if x >0 :#présence d'une indentation
3     print ("x est strictement positif")#instruction réalisée si la 1e
4 if x<10:
5     print ("x est plus petit que 10")#instruction réalisée si la 2e c
6 elif x<5 :
7     print ("x est plus petit que 5")#instruction réalisée si la 2e co
8 else :
9     print("x est nul")#instruction réalisée si les deux autres condit.
```

x est strictement positif

x est nul

```
In [1]: 1 x=11
2 if x>10:
3     print("if")
4 elif x>10:
5     print("elif")
6 elif x>10:
7     print("elif2")
8 #si if OK, les autres ne sont pas testé
```

if

On peut cumuler des conditions, il ne faut pas oublier que c'est *if* condition:

```
In [15]: 1 x=5
2 if (x>0 and x<10):
3     print("0<x<10")
```

0<x<10

```
In [16]: 1 x=5
2 if 0<x<10 :
3     print("0<x<10")
4
```

0<x<10

