chapitreprof

August 25, 2026

1 Chapitre 0 : la boîte à outils pour commencer à "pythoner" Code capytal : 0c13-1780985

Quelques généralités :

- Pour installer python sous Windows : http://winpython.github.io/
- pour installer python sous macOs ou Linux : https://pyzo.org/start.html
- plusieurs IDE sont possibles : spyder, pyzo, pycharm, jupyter, Thonny,...
- Vous pouvez aussi travailler en ligne avec Python Tutor
- Vous pouvez travailler sur l'ENT avec Capytale

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

En langage Python, l'opérateur d'affectation "=" permet de lier une variable à un objet (entier, nombre décimal, données textuelles, liste,...)

[50]:
```
# on choisit un nom de variable : ici x
x=10
```

[51]:
```
x
```

[51]:
```
10
```

[52]:
```
print(x)
```

```
10
```

[53]:
```
# Une fois définie, on peut utiliser cette variable pour d'autres calculs:
z=x+20
print(z)
```

```
30
```

[54]:
```
# On a un typage dynamique qui permet de faire évoluer le type de x au cours du␣
 ↪programme
#(un même nom peut être successivement associé à des objets de types différents)␣
 ↪:
x=x+10.0
print(x)
```

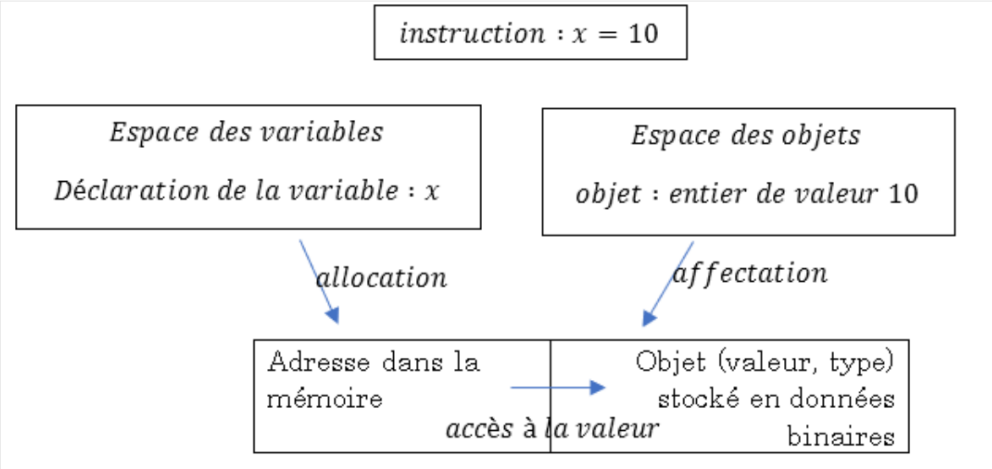
```
20.0
```

Rq :Un nom de variable :

- peut contenir des lettres, des chiffres et _ ;
- ne peut pas commencer par un chiffre ;
- ne peut contenir ni espace, ni tiret, ni caractère spécial ;
- est sensible aux majuscules et minuscules

Partie facultative :

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



[31]:
```
# Exemple
x=10
print(id(x))#permet d'avoir l'adresse dans la mémoire de x
```

```
140726426916040
```

[32]:
```
y=x
print(y,id(y))#même adresse, même valeur => permet une économie de la mémoire
 ↪(attention aux effets de bord)
```

```
10 140726426916040
```

[33]:
```
x=11.0#il est possible de faire un typage dynamique, ici x est associé à un float
print(x)
print(id(x))#Nouvelle allocaction et affectation de la variable x
#(la nouvelle affectation délie x de son ancienne valeur)
print(y,id(y))
```

```
11.0
1686994633584
10 140726426916040
```

[4]:
```
x=x+10#le typage dynamique permet aussi un lecture de l'ancienne valeur puis une␣
↪nouvelle affectation
print(x)
print(id(x))
```

```
21.0
1686964452176
```

[5]:
```
z=x+10
print(z)#on a accès à la valeur de x pour d'autres calculs
```

```
31.0
```

[5]:
```
#Rq : le x=x+10 et le x+=10
x=10
x+=10
print(x)#20
#cependant cette notation présente quelques subtilités
L=[0,1,2,3]
L=L+["a"]#fonctionne (on crée une nouvelle liste=
print(L)
L+="a"#fonctionne (pas de nouvelle liste, équivalent à extend)!
print(L)
L=L+"a"#ne fonctionne pas car la création n'aboutit 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
```

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

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

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

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

```
13
```

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

```
7
```

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

```
30
```

[5]:
```
# x puissance y
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 : tels que a=b*Q+R

[6]:
```
# Division de x par y
print(x/y)# on pourra remarquer que le nombre limité de chiffres significatifs
#et l'erreur d'arrondi liée au principe
# de numérisation des nombres décimaux abouti à une erreur
# si on numérise sur 4 bits alors : 1*2^1+1*2^0+0*2^-1+1*2^-2=2+1+0.25 =3,25
#on comprend qu'en limitant le nombre de bits sur ce type de nombre non dyadique␣
 ↪alors on a une erreur !
```

```
3.3333333333333335
```

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

```
3
```

[26]:
```
# Reste de la division de x par y
print(x%y)
# On pourra noter que on a un test d'identification des nombres pairs et impairs␣
 ↪:
print(5%2)#on a un reste de la division par 2 égale à 1 si impair
print(4%2)#on a un reste de la division par 2 égale à 0 si pair
```

```
1
```

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

[27]:
```
(3, 1)
```

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

[13]:
```
10
```

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

```
3
```

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

```
10.0
```

1.3 III- Les booléens (code quizinière : E7Y8ZQ)

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

[7]:
```
print(True or True)#True
print(False or False)# False
print(False or True)#True
print(True or False)#True
print(True or variable_inconnue)#True car opérateur paresseux (pas d'évaluation␣
 ↪du 2nd opérande si le 1e est vrai)
```

```
True
False
True
True
True
```

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

[9]:
```
print(True and False)#False
print(False and False)# False
print(False and True)#False
print(False and rien)#False car opérateur paresseux (pas d'évaluation du 2nd␣
 ↪opérande si le 1e est faux)
print(True and True)# True
```

```
False
False
False
```

```
False
True
```

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

[10]:
```
print(not False)#True
print(not True)#False
#les booléens peuvent être vis comme un sous type des entiers (int(True)=1 et␣
 ↪int(False)=0)
print(1==True)#True
print(0==False)#True
print(not 0)#True
print(not 1)#False
print(not 10.4)#False (tout nombre différent de 0 peut être interprété comme␣
 ↪True une fois converti en booléen)
print(10.4==True)#revient à 10.4==1 donc False
print(bool(10.4)==True)# revient à 1==1 donc True
i=10
while i:
    i=i-1
    print(i)
```

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

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

[2]:
```
print(not False or False and False)# ((not False) or (False and False))
```

```
True
```

Certains opérateurs sur les types numériques renvoient un résultat booléen :

[1]:
```
x=10
y=3
print(x<y)# x est-il strictement plus petit que y ?
print(x>y)# x est-il strictement plus grand que y ?
print(x<=y)# x est-il inférieur ou égal à y ?
print(x>=y)# x est-il supérieur ou égal à y ?
print(x==y)#  x et y ont-ils la même valeur ?
print (x!=y)# x est-il différent de y ?
```

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

[17]:
```
# PS : on n'écrit pas =< ou =>
print(2>=1)
print(2=>1)
```

```
  Cell In[17], line 3
    print(2=>1)
          ^
SyntaxError: expression cannot contain assignment, perhaps you meant "=="?
```

[1]:
```
#En raison des erreurs d'arrondi, il faut généralement éviter de tester␣
↪l'égalité exacte de deux nombres flottants calculés:
print(0.1+0.2==0.3)
```

```
False
```

1.4 IV- Les fonctions (code quizinière : OXD95W)

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

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

[18]:
```
#exemple f1(x)=2x+4
def f1(x):
    y=2*x+4
    return y
```

Pour exécuter une fonction, on l'appelle en donnant une valeur à chacun de ses arguments.:

[19]:
```
f1(2)# On appelle la fonction f1 et on récupère f1(2)
```

[19]: 8

Avec le mot clé *return*, on peut récupérer le résultat et l'affecter dans une autre variable pour un calcul ultérieur

[21]:
```
valeur=f1(2)
print(valeur*2)
```

16

[ ]:
```
#Pour généraliser à toute fonction affine, on peut rajouter d'autres arguments
def f2(a,b,x):
    return a*x+b
f2(2,4,2)
```

Rq : Un notebook affiche automatiquement la valeur renvoyée par la dernière expression de la cellule. Pour afficher les résultats d'expressions précédentes, il faut utiliser print() ou les regrouper dans la dernière expression.

[22]:
```
f1(2)
f1(4)#seul le dernier résultat est affiché
```

[22]: 12

[23]:
```
f1(2),f1(4)#les deux sont affichés
```

[23]: (8, 12)

On distingue les variables locales et globales :

- les variables globales ne sont pas définies dans le corps de la fonction.
- les variables locales sont définies dans le corps de la fonction et leur lecture n'est possible que pendant l'appel de cette fonction

[26]:
```
# En général, on commence un programme par les variables globales (par exemple␣
 ↪des constantes physiques)
g=9.81
def chute1(t):
    return 0.5*g*t**2
chute1(5)
```

[26]: 122.625

[30]:
```
def chute2(t):
    G=9.81
    return 0.5*G*t**2
print(G)# la varaible locale n'est pas accessible en dehors de l'appel de la␣
 ↪fonction
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[30], line 4
      2     G=9.81
      3     return 0.5*G*t**2
----> 4 chute2(5),print(G)

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

Pour aller plus loin :

Si une variable est appelées dans le corps de la fonction alors elle est d'abord recherchées dans l'espace des varaibles locales puis globales

[13]:
```
#exemple 1 : si a n'est que variable globale
a=2
def f3():
    print(a)
print(a)
f3()
print(a)
```

```
2
2
2
```

[23]:
```
#exemple 2 : si a est variable locale et globale
a=2
def f4():
    a=3
    print(a)
print(a)
f4()
print(a)
```

```
2
3
2
```

[2]:
```
#exemple 3 : si a est passé en argument alors il appartient à l'espace des␣
 ↪variables locales
a=2
def f5(a):
    a=a+2
    b=a+3
    return a,b
print(a)
print(f5(4))#renvoie un tuple
```

```
print(a)
```

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

[24]:
```
#exemple suplémentaire :
a=2
def f6():
    a=a+2#pose pb car sur on cherche à modifier une variable globale non mutable
 ↪(ici un entier)
    b=a+3
    return a,b
print(a)
print(f6())
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
```

[14]:
```
#exemple suplémentaire
a=2
def f6():
    global a#permet de modifier une variable gloable non mutable....à éviter
    a=a+2#a est modifiable
    b=a+3
    return a,b
print(a)
print(f6())
print(a)
```

```
2
```

```
(4, 7)
4
```

1.5 V- Les chaînes de caractères

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

- création :

[47]:
```
mot1="Bonjour"#création manuelle avec les quillemets doubles
mot2='à l\'ensemble de \
la classe'#nécessité de mettre un antislash => guillemets double à privilégier !
#au passage l'anti-slash permet d'écrire une code sur 2 lignes une instruction␣
 ↪un peu longue
# si on veut vraiment afficher sur deux lignes avec print =>\n
phrase=mot1+"\n"+mot2
print(phrase)#concaténation
```

```
Bonjour
à l'ensemble de la classe
```

- Conversion:

[48]:
```
#Effectué par le mot clé $str$
phrase=phrase+" pour la rentrée "+str(2026)#concaténation possible si de même␣
 ↪type
print(phrase)
```

```
Bonjour
à l'ensemble de la classe pour la rentrée 2026
```

[56]:
```
# On pourra signaler les f-string:
annee = 2026
phrase = f"Bonjour à l'ensemble de la classe pour la rentrée {annee}"
print(phrase)
```

```
Bonjour à l'ensemble de la classe pour la rentrée 2026
```

- Sélection ou tranche

[49]:
```
#sélection d'une lettre (indice de départ nul)
print(phrase[0])#le premier indice est à 0
#selection d'une sous chaîne [indice de départ : indice fin exclu]
print(phrase[0:7])
#remarque, on peut choisir le pas :
print(phrase[0:7:2])#ici pas de 2
```

```
B
Bonjour
Bnor
```

- longueur d'une chaîne

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

```
44
```

1.6 VI- Structures conditionnelles (code quizinière : 54AXL3)

On a typiquement le squelette suivant :

[13]:
```
x=float(input("rentre un nombre"))
if x >0 :#on notera la présence d'une indentation
    print ("x est strictement positif")#instruction réalisée si la 1e condition␣
 ↪est vraie
elif x<0 :
    print ("x est strictement négatif")#instruction réalisée si la 1e condition␣
 ↪est fausse et que la 2e est vraie
else :
    print("x est nul")#instruction réalisée si les deux autres conditions sont␣
 ↪fausses
```

```
rentre un nombre 10
```

```
x est strictement positif
```

Rq : if et elif sont bien différents : Plusieurs instructions if successives sont toutes testées. En revanche, dans une structure if–elif–else, Python s'arrête dès qu'une condition est vraie. Le bloc else est exécuté uniquement si aucune des conditions précédentes n'est vraie.

[1]:
```
# exemple pour lequel le if est nécessaire : on recherche le plus grand nombre␣
 ↪parmi 3
def f(a,b,c):
    maxi=a
    if b>maxi:
        maxi=b
    if c>maxi:
        maxi=c
    return maxi
f(10,20,30)# avec un elif => maxi=20 !
```

[1]:
```
30
```

[3]:
```
#autre exemple pour lequel le elif est nécessaire pour effectuer l'opération␣
 ↪demandée:
def f(a,b,operation):
    if operation=="+":
        return a+b
    elif operation=="-":
        return a-b
    elif operation=="*":
        return a*b
```

```
    elif operation=="/":
        if b!=0:
            return a/b
        else :
            return False
    return False
f(10,20,"/")
```

[3]: 0.5

[17]:
```
x=11
if x >0 :#présence d'une indentation
    print ("x est strictement positif")#instruction réalisée si la 1e condition␣
 ↪est vraie
if x<10:
    print ("x est plus petit que 10")#instruction réalisée si la 2e condition␣
 ↪est vraie
elif x<5 :
    print ("x est plus petit que 5")#instruction réalisée si la 2e condition est␣
 ↪fausse et que la 3e est vraie
else :
    print("x est nul")#instruction réalisée si les deux autres conditions sont␣
 ↪fausses
```

```
x est strictement positif
x est nul
```

[1]:
```
x=11
if x>10:
    print("if")
elif x>10:
    print("elif")
elif x>10:
    print("elif2")
#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:

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

```
0<x<10
```

[16]:
```
x=5
if 0<x<10 :
    print("0<x<10")
```

```
```

```
0<x<10
```