

I Boucles et tests

1. Le test if et la boucle for

La commande `if` permet de distinguer plusieurs situations conduisant à un comportement différent du programme. La syntaxe est la suivante :

```
if cond 1 :  
    bloc de commandes 1  
elif cond 2 :  
    bloc de commandes 2  
elif cond 3 :  
    bloc de commandes 3  
...  
elif cond n :  
    bloc de commandes n  
else :  
    bloc de commandes n+1
```

Si `cond 1` est réalisée, le bloc de commandes 1 est exécuté, sinon la `cond 2` est testée et le bloc de commandes 2 est exécuté si `cond 2` est réalisée, ... Le bloc de commandes `n+1` n'est exécuté que si aucune des conditions 1 à `n` n'est réalisée. Le bloc de commande `i < n+1` est donc exécuté si toutes les `cond j < i` ne sont pas réalisées et si `cond i` est réalisée. Les commandes `elif` et `else` sont optionnelles.

Les conditions de test s'écrivent avec les opérateurs `==`, `!=`, `<`, `>`, `<=`, `>=`, les opérateurs logiques `and`, `or` et `not`, la commande `in ...`

```
>>> L = [1,2,5,-4,0,-1,2,-1]  
  
>>> r = 0  
>>> for k in range(len(L)) :  
    if L[k] < 0 :  
        r -= 1  
    elif L[k] > 0 :  
        r += 1  
  
>>> print(r)  
1
```

Un autre exemple, avec un test « composé » :

```
>>> L = list(range(10)) + ['a', 'b', 'c']  
  
>>> print(L)  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'a', 'b', 'c']  
  
>>> L1 = list(range(1,11,3))  
  
>>> print(L1)  
[1, 4, 7, 10]  
  
>>> for elt in L :  
    if elt in L1 or elt == 'b' :  
        L.remove(elt)  
  
>>> print(L)  
[0, 2, 3, 5, 6, 8, 9, 'a', 'c']
```

Les *conditions* doivent pouvoir être évaluées pour renvoyer un *booléen*; si `test` est une variable qui contient déjà un booléen, la « bonne syntaxe » est `if test :` et non `if test == True :` (même si cela fonctionne); de même, au lieu de `if test == False :`, on préférera utiliser `if not test :`

2. Boucle conditionnelle **while**

La syntaxe générale est la suivante :

```
| while condition :  
|     bloc de commandes
```

Le bloc de commandes est exécuté tant que la condition est réalisée. Il faut faire attention à ce que la condition soit réalisée au départ pour que le bloc de commandes soit exécuté au moins une fois et penser à modifier des paramètres de la condition dans le bloc de commandes pour que l'on sorte de la boucle au bout d'un certain temps.

```
>>> def fact(n) :  
|     r = 1  
|     k = 1  
|     while k <= n :  
|         r = r * k  
|         k += 1  
|     return r  
  
>>> fact(4)  
24  
  
>>> fact(1)  
1
```

La condition peut à nouveau être plus complexe et utiliser les connecteurs logiques **and**, **or** et **not**.

*Attention : Contrairement au « et » logique, le connecteur **and** n'est pas commutatif :*

Le code suivant produit une erreur

```
>>> L = [1,2,3]  
  
>>> def f(L) :  
|     i = 0  
|     while L[i]>0 and i<len(L) :  
|         i += 1  
|     return i  
  
>>> f(L)  
IndexError: list index out of range
```

Alors que le code suivant fonctionne correctement :

```
>>> def f(L) :  
|     i = 0  
|     while i<len(L) and L[i]>0 :  
|         i += 1  
|     return i  
  
>>> f(L)  
3
```

II Fonctions

1. Structure générale

On crée une fonction avec le mot clé **def** suivi du nom de la fonction (celui-ci ne doit pas être déjà utilisé) suivi de la liste des arguments entre parenthèses et séparés d'une virgule. Les parenthèses sont obligatoires même s'il n'y a pas d'arguments.

```
>>> def fct(n) :  
|     """ calcule 2 puissance n """ # indenter la suite !!!!! # commentaire affiché par help(fct)  
|     return 2**n # résultat de la fonction  
  
>>> fct(4)  
16
```

```

>>> def autre() :                # fonction sans argument
    n = input()                  # demande de saisie
    n = int(n)                   # input renvoie une chaîne de caractères
    print(3*n)                   # affichage simple du résultat

>>> autre()
3                                # on prend n=3
27

```

Plusieurs arguments :

```

>>> def diff(x,y) :
    return y-x
>>> diff(4,2)
-2

```

Un argument plus « complexe » :

```

>>> def somme(L) :
    s = 0
    for elt in L : # L doit être itérable
        s += elt
    return s

>>> somme(range(4))
6

```

2. Utilisation de la commande **return**

En Python, la commande **return** interrompt la fonction, il est donc possible de programmer une fonction en utilisant cette possibilité pour interrompre une boucle **for** par exemple.

La fonction suivante recherche par exemple si 0 est un élément de la liste L

```

def zero(L) :
    for elt in L :
        if elt == 0 :
            return True
    return False

```

Il est possible de faire autrement : il est en général suffisant de remplacer la boucle **for** par une boucle **while** et l'introduction d'un booléen :

```

def ZERO(L) :
    trouve = False
    k = 0
    while not trouve and k < len(L) :
        if L[k] == 0 :
            trouve = True
        k += 1
    return trouve

```

Même si la commande **return** interrompt la fonction, on peut utiliser plusieurs fois la commande **return** dans une seule fonction :

```

def valAbs(x) :
    if x > 0 :
        return x
    else :
        return -x

```

Une fonction ne contient pas obligatoirement la commande **return** mais dans ce cas, son résultat sera **None** (même si elle peut afficher quelque chose par l'intermédiaire d'un **print**).

<pre> def f(x) : return x**2 </pre>	<pre> def g(x) : print(x**2) </pre>
---	---

Ces deux fonction semblent avoir le même effet mais alors que **f** calcule effectivement x^2 , **g** ne fait que l'afficher :

```

>>> a = f(2) # n'affiche rien mais a vaut 4

>>> a**2
16

>>> b = g(2) # là on affiche !
4

>>> b**2
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unsupported operand type(s) for ** or pow(): 'NoneType' and 'int'

>>> print(b)
None

```

Dans cet exemple, `b` contient `None` (ce qui est déjà quelque chose en soit : un test `if b == None` est possible par exemple) mais absolument pas le résultat attendu de la fonction `g`.

3. Variables locales et globales

On parle de variable *globale* lorsque cette variable est définie en dehors de la fonction ; une telle variable est utilisable, et modifiable, par la fonction. Une variable *locale* est définie à l'intérieur de la fonction, et sa valeur disparaît à la fin de l'exécution de la fonction.

```

>>> y = 2

>>> def f(x) :
        return x+y

>>> f(3)
5

```

Ici `y` est une variable globale.

```

>>> y = 2

>>> def f(x) :
        y = 1
        return x+y

>>> f(3)
4

>>> y
2

```

Ici la fonction crée une variable locale, nommée `y` aussi, la variable globale `y`, définie avant, n'est donc pas modifiée.

```

>>> y = 2

>>> def f(x) :
        y = y+1
        return x+y

>>> f(3)
UnboundLocalError: local variable 'y' referenced before assignment

```

Le problème vient ici du fait que la fonction crée une variable locale `y`, qui n'est donc pas la variable globale précédente, et l'affectation `y = y+1` pose problème puisque cette variable locale n'est pas initialisée avant.

Il existe différentes façon de contourner ce problème :

- Éviter de nommer une variable locale de la même façon qu'une variable globale.
- Rajouter un argument supplémentaire à la fonction et intégrer les variables globales aux arguments de la fonction (définir ici une fonction `f(x,y)`, mais cela ne permettra pas de modifier le contenu d'une éventuelle variable appelée `y` et définie en dehors de `f`).
- Déclarer les variables globales utilisées par la fonction avec la commande `global` au début de la fonction

```

>>> y = 2

>>> def f(x) :
    global y
    y = y+1
    return x+y

>>> f(3)
6
>>> y
3

```

Le fait de déclarer `y` comme variable globale permet à la fonction de modifier sa valeur.

4. Effets de Bord

En python, une fonction peut modifier la valeur de certaines variables passées en paramètres au cours de leur exécution ; on parle de fonction « à effet de bord ».

On peut par exemple comparer les deux fonctions suivantes :

```

def f(L) :
    L.append(0)
    return L

def g(L) :
    L = L+[0]
    return L

```

Cette possibilité peut permettre de définir des fonctions sans la commande `return` : l'effet de la fonction est « visible » directement sur un de ses paramètres mais le résultat de la fonction est alors `None` (on parle plutôt de « procédure »).

```

def perm(L) :
    n = len(L)
    for i in range(n//2) :
        L[i], L[n-i-1] = L[n-i-1], L[i]

```

Cette fonction `perm`, retourne la liste `L` en modifiant la liste `L` elle même (c'est une action « sur place ») ; l'utilisation d'un `return` n'est pas indispensable car le « résultat » de la fonction `perm` est directement stocké dans la liste `L`

```

>>> L = list(range(4))

>>> L
[0, 1, 2, 3]

>>> perm(L) # apparemment rien ne se passe

>>> L
[3, 2, 1, 0]

>>> a = perm(L) # toujours rien

>>> print(a)
None

>>> L
[0, 1, 2, 3]

```

5. Sous-fonction

On peut également définir une fonction à l'intérieur d'une autre :

```

>>> def g(x) :
    y = x+1
    def h(t) :
        return t+x+y
    return h(x)

```

```
>>> g(1)
4
```

Ici la fonction **h** est définie à l'intérieur de la fonction **g** : la fonction **h** n'est alors pas utilisable directement, elle n'existe que pendant le temps de l'exécution de **g** (comme une variable locale).

```
>>> h(0)
NameError: name 'h' is not defined
```

La variable **y** est une variable locale de la fonction **g** mais devient une variable globale de la fonction **h** qui peut donc l'utiliser, comme le paramètre **x** de **g**.

Dans ce cadre là, la fonction **g** doit absolument appeler la sous-fonction **h** au cours de son exécution, sinon elle ne sert à rien.

III Utilisation des listes

1. Initialisations

On aura souvent besoin de travailler avec des listes, que l'on devra donc initialiser. Il y a en général deux façons de faire :

- Soit on initialise la liste avec une liste vide (**L = []**) puis on rajoute des éléments par concaténation ou avec la méthode **append()** ; ceci est particulièrement efficace lorsque les éléments à inclure dans cette liste sont toujours à placer à la fin.
- Soit on initialise la liste avec des éléments quelconques (**L = [0 for _ in range(10)]** par ex) et on modifie les valeurs de cette liste en utilisant le caractère mutable des listes : **L[4] = 1**

Il faut par contre faire attention à ne pas mélanger les deux méthodes : le code suivant produit une erreur

```
>>> L = []
>>> L[0] = 1
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: list assignment index out of range
```

Il faut aussi faire attention lors de l'utilisation de listes de listes : comparer les initialisations suivantes, et en particulier l'effet de **L[0][1] = 10** par ex :

```
L1 = [[0,0] * 4]
L2 = [[0,0]] * 4
L3 = [[0,0] for i in range(4)]
L4 = [[0 for i in range(2)] for j in range(4)]
L5 = [[0 for i in range(2)]] * 4
```

2. Opérations sur les listes

Les éléments d'une liste sont indicés à partir de 0. L'élément d'indice *i* d'une liste s'écrit **<liste>[i]**, les listes sont des objets **mutables**, on peut donc modifier les éléments individuellement :

```
>>> liste=[1, 2.5 ,[1, 2], 'toto']
>>> len(liste) # longueur de la liste
4
>>> liste[3] # accès à un élément par son indice
'toto'
>>> liste[1]=3 # modification d'un élément
>>> print(liste)
[1, 3, [1, 2], 'toto']
```

Pour extraire une sous-liste d'une liste on utilise le « slicing » (tranche) :

```
>>> liste=[1, 2 ,[1, 2], 'toto', 4.2]
>>> liste[:3] # les 3 premiers
[1, 2, [1, 2]]
>>> print(liste[1::2]) # de 2 en 2 à partir du deuxième
[2, 'toto']
>>> print(liste[1:]) # tout à partir du deuxième
[2, [1, 2], 'toto', 4.2]
>>> print(liste[-3:]) # les 3 derniers
[[1, 2], 'toto', 4.2]
```

Pour rajouter des éléments :

```
>>> liste1 = ['lundi', 'mardi', 'mercredi']
>>> liste2 = ['jeudi', 'vendredi']
>>> liste3 = liste1 + liste2          # concaténation
>>> print(liste3)
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
>>> liste4 = [0] * 7                  # répétition (équivalent à 6 concaté
nations)
>>> print(liste4)
[0, 0, 0, 0, 0, 0, 0]
```

Les méthodes **pop** et **append** sont particulièrement efficaces pour agir sur le dernier élément de la liste :

```
>>> liste = [0, 1, 2, 3]
>>> liste.append(4) # ajout de 4 en fin de liste
>>> print(liste)
[0, 1, 2, 3, 4]
>>> a = liste.pop() # suppression et stockage du dernier élément
>>> print(liste)
[0, 1, 2, 3]
>>> a
4
```

Enfin, on peut tester l'appartenance d'un élément à une liste avec la commande **in** : **elt in L** renvoie **True** ou **False** selon que **elt** fait ou non partie de la liste **L**.

3. Copie de listes

Le code **M = L** ne permet pas de faire une copie satisfaisante de la liste **L** : la liste **M** produite n'est pas indépendante de la liste **L**

```
>>> L = [0, 1]
>>> M = L
>>> M
[0, 1]
>>> M[0] = 2
>>> L
[2, 1]
```

Il vaut mieux copier la liste **L** élément par élément :

```
M = []
for elt in L :
    M.append(elt)
```

ou directement **M = L[:]**

Lors de l'utilisation d'une liste de listes, c'est encore plus compliqué : **M = L[:]** n'est plus satisfaisant (c'est équivalent à **M = [l for l in L]** qui ne marche pas non plus).

On peut par exemple faire de la façon suivante :

```
N = []
for l in L :
    N.append(l[:])
```

IV Structure immuables

1. Chaînes de caractères

Une chaîne de caractères en python peut être délimitée par des apostrophes (simple quotes) ou bien par des guillemets (double quotes). Exemple :

```
>>> chaine1 = "C'est un premier message."
>>> chaine2 = 'Message "deux".'
>>> chaine3 = 'C\'est un autre message.'
>>> print(chaine1, chaine2, chaine3)
C'est un premier message. Message "deux". C'est un autre message.
```

```
>>> type(chainel)
str
>>> len(chainel)    # nombre de caractères
25
```

L'instruction **print** insère un espace entre chaque chaîne. On notera dans la troisième variable, comment insérer l'apostrophe dans une chaîne délimitée par deux apostrophes, grâce au caractère d'échappement `\` (**backslash**). On peut le rencontrer dans plusieurs circonstances à l'intérieur d'une chaîne, notamment :

- `\'` : pour insérer l'apostrophe dans une chaîne délimitée par deux apostrophes,
- `\"` : pour insérer un guillemet dans une chaîne délimitée par deux guillemets,
- `\n` : pour insérer un saut de ligne,
- `\t` : pour insérer une tabulation.

Pour insérer le caractère backslash il suffit de le « doubler » :

```
>>> c1 = 'C\'est une chaîne avec \\'
>>> print(c1)
C'est une chaîne avec \
```

Notons enfin qu'il est possible de délimiter une chaîne par des triples apostrophes (ou guillemets), ce qui permet de l'écrire sur plusieurs lignes. Ceci est souvent utilisé pour documenter les fonctions lors de leur définition.

Deux chaînes peuvent être comparées avec les opérateurs `==` (égalité), `<`, `<=`, `>`, `>=` (comparaisons selon l'ordre lexicographique).

Les caractères d'une chaîne sont indicés à partir de 0. Si **nom** désigne une chaîne, le caractère d'indice *i* est **nom[i]**.

Il n'est pas possible de modifier isolément un caractère d'une chaîne : les chaînes de caractères sont des objets **non mutables** (contrairement aux listes).

```
>>> nom = 'Le corbeau'
>>> print(nom[0], nom[3], nom[5])
L c r
>>> type(nom[0])
str    # c'est une chaîne contenant un seul caractère
>>> nom[1] = 'E'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Plus généralement il est possible d'extraire une partie d'une chaîne (« slicing »), le résultat sera une chaîne. La syntaxe générale est la suivante :

`<chaîne>[indice début:indice fin:pas]`

```
>>> nom = 'Hélène'
>>> print(nom[:3]) # les 3 premiers caractères
Hél
>>> print(nom[1::2]) # de 2 en 2 à partir du deuxième
éeè
>>> print(nom[1:]) # à partir du deuxième
élène
>>> print(nom[-3:]) # les 3 derniers
ène
```

Une chaîne est un objet itérable, elle peut être parcourue à l'aide d'une boucle **for** :

```
>>> for z in 'Hélène': print(z)
H
é
l
è
n
e
```

La concaténation est l'opération qui consiste à juxtaposer deux chaînes pour en faire une seule. En python c'est l'opérateur `+` et les deux opérandes doivent être de même type :

```
>>> nom1 = "Bonjour "
>>> nom2 = "vous!"
>>> nom3 = nom1 + nom2 # concaténation
```

```

>>> nom4 = nom1 * 4      # répétition
>>> print(nom3)
Bonjour vous!
>>> print(nom4)
Bonjour Bonjour Bonjour Bonjour
>>> nom5 = nom1 + 45
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects

```

Pour créer une chaîne, on peut utiliser la commande **str** (particulièrement utile dans les boucles) :

```

>>> ['k' for k in range(4)]
['k', 'k', 'k', 'k']
>>> [str(k) for k in range(4)]
['0', '1', '2', '3']

```

Contrairement aux listes, les méthodes **pop** et **append** ne sont pas utilisables pour les chaînes de caractères.

2. Les tuples

Un tuple est similaire à une liste à deux différences près :

- Un tuple est délimité par une paire de parenthèses et non de crochets.
- Un tuple est **non mutable** (contrairement aux listes), ce qui peut être une sécurité dans certains cas. Ce type est également moins gourmand en ressources que les listes.

```

>>> t = (1,2,3,4,5,6)
>>> type(t)
<type 'tuple'>
>>> t[2]
3
>>> t[1:5]      # slicing
(2, 3, 4, 5)
>>> t[1] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> len(t)
6
>>> t+(7,8)      # concaténation
(1,2,3,4,5,6,7,8)
>>> (1,2)*3      # répétition
(1,2,1,2,1,2)
>>> for i in t : print(i)  # un tuple est aussi itérable
1
2
3
4
5
6

```