Chapitre 5

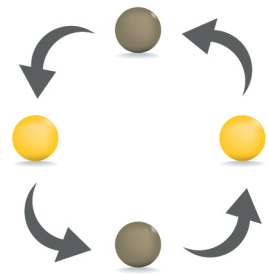
# Les boucles

*Difficulté :*

Les boucles sont un concept nouveau pour vous. Elles vont vous permettre de répéter une certaine opération autant de fois que nécessaire. Le concept risque de vous sembler un peu théorique car les applications pratiques présentées dans ce chapitre ne vous paraîtront probablement pas très intéressantes. Toutefois, il est impératif que cette notion soit comprise avant que vous ne passiez à la suite. Viendra vite le moment où vous aurez du mal à écrire une application sans boucle.

En outre, les boucles peuvent permettre de parcourir certaines séquences comme les chaînes de caractères pour, par exemple, en extraire chaque caractère.

Alors, on commence ?



## En quoi cela consiste-t-il ?

Comme je l'ai dit juste au-dessus, les boucles constituent un moyen de répéter un certain nombre de fois des instructions de votre programme. Prenons un exemple simple, même s'il est assez peu réjouissant en lui-même : écrire un programme affichant la table de multiplication par 7, de 1 * 7 à 10 * 7.

... bah quoi ?

Bon, ce n'est qu'un exemple, ne faites pas cette tête, et puis je suis sûr que ce sera utile pour certains. Dans un premier temps, vous devriez arriver au programme suivant :

```
1  print(" 1 * 7 =", 1 * 7)
2  print(" 2 * 7 =", 2 * 7)
3  print(" 3 * 7 =", 3 * 7)
4  print(" 4 * 7 =", 4 * 7)
5  print(" 5 * 7 =", 5 * 7)
6  print(" 6 * 7 =", 6 * 7)
7  print(" 7 * 7 =", 7 * 7)
8  print(" 8 * 7 =", 8 * 7)
9  print(" 9 * 7 =", 9 * 7)
10 print("10 * 7 =", 10 * 7)
```

... et le résultat :

```
1   1 * 7 = 7
2   2 * 7 = 14
3   3 * 7 = 21
4   4 * 7 = 28
5   5 * 7 = 35
6   6 * 7 = 42
7   7 * 7 = 49
8   8 * 7 = 56
9   9 * 7 = 63
10 10 * 7 = 70
```

Je vous rappelle que vous pouvez enregistrer vos codes dans des fichiers. Vous trouverez la marche à suivre à la page 349 de ce livre.

Bon, c'est sûrement la première idée qui vous est venue et cela fonctionne, très bien même. Seulement, vous reconnaîtrez qu'un programme comme cela n'est pas bien utile. Essayons donc le même programme mais, cette fois-ci, en utilisant une variable ; ainsi, si on décide d'afficher la table de multiplication de 6, on n'aura qu'à changer la valeur de la variable ! Pour cet exemple, on utilise une variable `nb` qui contiendra 7. Les instructions seront légèrement différentes mais vous devriez toujours pouvoir écrire ce programme :

```
1  nb = 7
2  print(" 1 *", nb, "=", 1 * nb)
```

```
print(" 2 *", nb, "=", 2 * nb)
print(" 3 *", nb, "=", 3 * nb)
print(" 4 *", nb, "=", 4 * nb)
print(" 5 *", nb, "=", 5 * nb)
print(" 6 *", nb, "=", 6 * nb)
print(" 7 *", nb, "=", 7 * nb)
print(" 8 *", nb, "=", 8 * nb)
print(" 9 *", nb, "=", 9 * nb)
print("10 *", nb, "=", 10 * nb)
```

Le résultat est le même, vous pouvez vérifier. Mais le code est quand-même un peu plus intéressant : on peut changer la table de multiplication à afficher en changeant la valeur de la variable `nb`.

Mais ce programme reste assez peu pratique et il accomplit une tâche bien répétitive. Les programmeurs étant très paresseux, ils préfèrent utiliser les boucles.

## La boucle `while`

La boucle que je vais présenter se retrouve dans la plupart des autres langages de programmation et porte le même nom. Elle permet de répéter un **bloc d'instructions** tant qu'une condition est vraie (`while` signifie « tant que » en anglais). J'espère que le concept de **bloc d'instructions** est clair pour vous, sinon je vous renvoie au chapitre précédent.

La syntaxe de `while` est :

```
while condition:
    # instruction 1
    # instruction 2
    # ...
    # instruction N
```

Vous devriez reconnaître la forme d'un bloc d'instructions, du moins je l'espère.

Quelle condition va-t-on utiliser ?

Eh bien, c'est là le point important. Dans cet exemple, on va créer une variable qui sera incrémentée dans le bloc d'instructions. Tant que cette variable sera inférieure à 10, le bloc s'exécutera pour afficher la table.

Si ce n'est pas clair, regardez ce code, quelques commentaires suffiront pour le comprendre :

```
nb = 7 # On garde la variable contenant le nombre dont on veut
    la table de multiplication
i = 0 # C'est notre variable compteur que nous allons incré
    menter dans la boucle
```

```
3
4 while i < 10: # Tant que i est strictement inférieure à 10
5     print(i + 1, "*", nb, "=", (i + 1) * nb)
6     i += 1 # On incrémente i de 1 à chaque tour de boucle
```

Analysons ce code ligne par ligne :

1. On instancie la variable `nb` qui accueille le nombre sur lequel nous allons travailler (en l'occurence, 7). Vous pouvez bien entendu faire saisir ce nombre par l'utilisateur, vous savez le faire à présent.
2. On instancie la variable `i` qui sera notre compteur durant la boucle. `i` est un standard utilisé quand il est question de boucles et de variables s'incrémentant mais il va de soi que vous auriez pu lui donner un autre nom. On l'initialise à 0.
3. Un saut de ligne ne fait jamais de mal !
4. On trouve ici l'instruction `while` qui se décode, comme je l'ai indiqué en commentaire, en « **tant que i est strictement inférieure à 10** ». N'oubliez pas les deux points à la fin de la ligne.
5. La ligne du `print`, vous devez la reconnaître. Maintenant, la plus grande partie de la ligne affichée est constituée de variables, à part les signes mathématiques. Vous remarquez qu'à chaque fois qu'on utilise `i` dans cette ligne, pour l'affichage ou le calcul, on lui ajoute 1 : cela est dû au fait qu'en programmation, on a l'habitude (habitude que vous devrez prendre) de commencer à compter à partir de 0. Seulement ce n'est pas le cas de la table de multiplication, qui va de 1 à 10 et non de 0 à 9, comme c'est le cas pour les valeurs de `i`. Certes, j'aurais pu changer la condition et la valeur initiale de `i`, ou même placer l'incrémentation de `i` avant l'affichage, mais j'ai voulu prendre le cas le plus courant, le format de boucle que vous retrouverez le plus souvent. Rien ne vous empêche de faire les tests et je vous y encourage même.
6. Ici, on incrémente la variable `i` de 1. Si on est dans le premier tour de boucle, `i` passe donc de 0 à 1. Et alors, puisqu'il s'agit de la fin du bloc d'instructions, on revient à l'instruction `while`. `while` vérifie que la valeur de `i` est toujours inférieure à 10. Si c'est le cas (et ça l'est pour l'instant), on exécute à nouveau le bloc d'instructions. En tout, on exécute ce bloc 10 fois, jusqu'à ce que `i` passe de 9 à 10. Alors, l'instruction `while` vérifie la condition, se rend compte qu'elle est à présent fausse (la valeur de `i` n'est pas inférieure à 10 puisqu'elle est maintenant égale à 10) et s'arrête. S'il y avait du code après le bloc, il serait à présent exécuté.

N'oubliez pas d'incrémenter `i` ! Sinon, vous créez ce qu'on appelle une boucle infinie, puisque la valeur de `i` n'est jamais supérieure à 10 et la condition du `while`, par conséquent, toujours vraie... La boucle s'exécute donc à l'infini, du moins en théorie. Si votre ordinateur se lance dans une boucle infinie à cause de votre programme, pour interrompre la boucle, vous devrez taper CTRL + C dans la fenêtre de l'interpréteur (sous Windows ou Linux). Python ne le fera pas tout seul car, pour lui, il se passe bel et bien quelque chose. De toute façon, il est incapable de différencier une boucle infinie d'une boucle finie : c'est au programmeur de le faire.

## La boucle `for`

Comme je l'ai dit précédemment, on retrouve l'instruction `while` dans la plupart des autres langages. Dans le C++ ou le Java, on retrouve également des instructions `for` mais qui n'ont pas le même sens. C'est assez particulier et c'est le point sur lequel je risque de manquer d'exemples dans l'immédiat, toute son utilité se révélant au chapitre sur les listes. Notez que, si vous avez fait du Perl ou du PHP, vous pouvez retrouver les boucles `for` sous un mot-clé assez proche : `foreach`.

L'instruction `for` travaille sur des séquences. Elle est en fait spécialisée dans le parcours d'une séquence de plusieurs données. Nous n'avons pas vu (et nous ne verrons pas tout de suite) ces séquences assez particulières mais très répandues, même si elles peuvent se révéler complexes. Toutefois, il en existe un type que nous avons rencontré depuis quelque temps déjà : les chaînes de caractères.

Les chaînes de caractères sont des séquences... de caractères ! Vous pouvez parcourir une chaîne de caractères (ce qui est également possible avec `while` mais nous verrons plus tard comment). Pour l'instant, intéressons-nous à `for`.

L'instruction `for` se construit ainsi :

```
1 for element in sequence:
```

`element` est une variable créée par le `for`, ce n'est pas à vous de l'instancier. Elle prend successivement chacune des valeurs figurant dans la séquence parcourue.

Ce n'est pas très clair ? Alors, comme d'habitude, tout s'éclaire avec le code !

```
1 chaine = "Bonjour les ZER0S"
2 for lettre in chaine:
3     print(lettre)
```

Ce qui nous donne le résultat suivant :

```
1  B
2  o
3  n
4  j
5  o
6  u
7  r
8
9  l
10 e
11 s
12
13 Z
14 E
15 R
16 0
17 S
```

Est-ce plus clair ? En fait, la variable `lettre` prend successivement la valeur de chaque lettre contenue dans la chaîne de caractères (d'abord B, puis o, puis n...). On affiche ces valeurs avec `print` et cette fonction revient à la ligne après chaque message, ce qui fait que toutes les lettres sont sur une seule colonne. Littéralement, la ligne 2 signifie « **pour lettre dans chaine** ». Arrivé à cette ligne, l'interpréteur va créer une variable `lettre` qui contiendra le premier élément de la chaîne (autrement dit, la première lettre). Après l'exécution du bloc, la variable `lettre` contient la seconde lettre, et ainsi de suite tant qu'il y a une lettre dans la chaîne.

Notez bien que, du coup, il est inutile d'incrémenter la variable `lettre` (ce qui serait d'ailleurs assez ridicule vu que ce n'est pas un nombre). Python se charge de l'incrémentation, c'est l'un des grands avantages de l'instruction `for`.

À l'instar des conditions que nous avons vues jusqu'ici, `in` peut être utilisée ailleurs que dans une boucle `for`.

```
1 chaine = "Bonjour les ZEROS"
2 for lettre in chaine:
3     if lettre in "AEIOUYaeiouy": # lettre est une voyelle
4         print(lettre)
5     else: # lettre est une consonne... ou plus exactement,
          lettre n'est pas une voyelle
6         print("*")
```

... ce qui donne :

```
1  *
2  o
3  *
4  *
5  o
6  u
7  *
8  *
9  *
10 e
11 *
12 *
13 *
14 E
15 *
16 *
17 *
```

Voilà ! L'interpréteur affiche les lettres si ce sont des voyelles et, sinon, il affiche des « * ». Notez bien que le 0 n'est pas affiché à la fin, Python ne se doute nullement qu'il s'agit d'un « o » stylisé.

Retenez bien cette utilisation de `in` dans une condition. On cherche à savoir si un élément quelconque est contenu dans un ensemble donné (ici, si la lettre est contenue dans « AEIOUYaeiouy », c'est-à-dire si `lettre` est une voyelle). On retrouvera plus

loin cette fonctionnalité.

## Un petit bonus : les mots-clés `break` et `continue`

Je vais ici vous montrer deux nouveaux mots-clés, `break` et `continue`. Vous ne les utiliserez peut-être pas beaucoup mais vous devez au moins savoir qu'ils existent... et à quoi ils servent.

### Le mot-clé `break`

Le mot-clé `break` permet tout simplement d'interrompre une boucle. Il est souvent utilisé dans une forme de boucle que je n'approuve pas trop :

```
1 while 1: # 1 est toujours vrai -> boucle infinie
2     lettre = input("Tapez 'Q' pour quitter : ")
3     if lettre == "Q":
4         print("Fin de la boucle")
5         break
```

La boucle `while` a pour condition `1`, c'est-à-dire une condition qui sera *toujours* vraie. Autrement dit, en regardant la ligne du `while`, on pense à une boucle infinie. En pratique, on demande à l'utilisateur de taper une lettre (un 'Q' pour quitter). Tant que l'utilisateur ne saisit pas cette lettre, le programme lui redemande de taper une lettre. Quand il tape 'Q', le programme affiche `Fin de la boucle` et la boucle s'arrête grâce au mot-clé `break`.

Ce mot-clé permet d'arrêter une boucle quelle que soit la condition de la boucle. Python sort immédiatement de la boucle et exécute le code qui suit la boucle, s'il y en a.

C'est un exemple un peu simpliste mais vous pouvez voir l'idée d'ensemble. Dans ce cas-là et, à mon sens, dans la plupart des cas où `break` est utilisé, on pourrait s'en sortir en précisant une véritable condition à la ligne du `while`. Par exemple, pourquoi ne pas créer un booléen qui sera **vrai** tout au long de la boucle et **faux** quand la boucle doit s'arrêter ? Ou bien tester directement si `lettre != « Q »` dans le `while` ?

Parfois, `break` est véritablement utile et fait gagner du temps. Mais ne l'utilisez pas à outrance, préférez une boucle avec une condition claire plutôt qu'un bloc d'instructions avec un `break`, qui sera plus dur à appréhender d'un seul coup d'œil.

### Le mot-clé `continue`

Le mot-clé `continue` permet de... continuer une boucle, en repartant directement à la ligne du `while` ou `for`. Un petit exemple s'impose, je pense :

```
1 i = 1
2 while i < 20: # Tant que i est inférieure à 20
3     if i % 3 == 0:
4         i += 4 # On ajoute 4 à i
```

```
5           print("On incrémente i de 4. i est maintenant égale à",
                i)
6           continue # On retourne au while sans exécuter les
                autres lignes
7       print("La variable i =", i)
8       i += 1 # Dans le cas classique on ajoute juste 1 à i
```

Voici le résultat :

```
1  La variable i = 1
2  La variable i = 2
3  On incrémente i de 4. i est maintenant égale à 7
4  La variable i = 7
5  La variable i = 8
6  On incrémente i de 4. i est maintenant égale à 13
7  La variable i = 13
8  La variable i = 14
9  On incrémente i de 4. i est maintenant égale à 19
10 La variable i = 19
```

Comme vous le voyez, tous les trois tours de boucle, `i` s'incrémente de 4. Arrivé au mot-clé `continue`, Python n'exécute pas la fin du bloc mais revient au début de la boucle en testant à nouveau la condition du `while`. Autrement dit, quand Python arrive à la ligne 6, il saute à la ligne 2 sans exécuter les lignes 7 et 8. Au nouveau tour de boucle, Python reprend l'exécution normale de la boucle (`continue` n'ignore la fin du bloc que pour le tour de boucle courant).

Mon exemple ne démontre pas de manière éclatante l'utilité de `continue`. Les rares fois où j'utilise ce mot-clé, c'est par exemple pour supprimer des éléments d'une liste, mais nous n'avons pas encore vu les listes. L'essentiel, pour l'instant, c'est que vous vous souveniez de ces deux mots-clés et que vous sachiez ce qu'ils font, si vous les rencontrez au détour d'une instruction. Personnellement, je n'utilise pas très souvent ces mots-clés mais c'est aussi une question de goût.

## En résumé

- Une boucle sert à répéter une portion de code en fonction d'un prédicat.
- On peut créer une boucle grâce au mot-clé `while` suivi d'un prédicat.
- On peut parcourir une séquence grâce à la syntaxe `for element in sequence:`.