

# TP 4

## Boucles for

Dans ce TP nous approfondissons sur les boucles, et les applications mathématiques aux calculs de suites et de sommes.

À partir de maintenant, dans les exercices, les programmes sont la plupart du temps contenus dans des **fonctions** avec éventuellement des paramètres et une valeur de retour. Cela permet de les ré-utiliser autant qu'on veut en faisant varier un paramètre, mais sans utiliser `input()`.

### I Boucles simples

La boucle `for` sert à répéter un bloc d'instructions un certain nombre de fois. Dans sa syntaxe de base, elle prend la forme

```
for i in range(a, b):  
    instructions
```

qui est exactement équivalente à

```
i = a  
while i < b:  
    instructions  
    i = i + 1
```

Autrement dit elle répète en faisant varier l'indice  $i$  avec  $a \leq i < b$ . Nous verrons qu'elle est un peu plus pratique quand on sait à l'avance combien de fois on veut répéter les instructions. Essai :

```
for i in range(1, 5):  
    print(i)
```

produit bien

```
1  
2  
3  
4
```

Un fait remarquable auquel il faudra s'habituer est que **la borne  $b$  est exclue**. Il existe aussi `range(n)` qui est exactement équivalent à `range(0, n)`. Retenir que `range(n)` répète  $n$  fois, avec  $i$  qui varie entre 0 et  $n-1$  inclus.

Éventuellement, un troisième argument `range(a, b, r)` consiste à faire des sauts de taille  $r$  au lieu de 1. Cela correspond, dans la traduction en terme de boucle `while` ci-dessus, à remplacer la ligne `i = i + 1` par `i = i + r`. On ne l'utilisera pas tous les jours.

Éventuellement aussi, le caractère `_` permet de ne pas donner de nom à la variable indice de la boucle. Ainsi `for _ in range(n)` a réellement le sens de « répéter  $n$  fois ».

C'est l'heure de démarrer le TP !

```
nom = "... " # insérez votre prénom  
for _ in range(3):  
    print("Au travail", nom)
```

**Exercice 1.** Vous connaissez la chanson

```
1 kilomètre à pieds, ça use, ça use,  
1 kilomètre à pieds, ça use les souliers.  
...
```

Écrire une fonction `kilometres(n)` qui affiche les paroles pendant  $n$  kilomètres.

Une application plus sérieuse des boucles `for` est pour le calcul des suites et des sommes, que nous avons déjà abordé. L'exemple de base est la fonction suivante qui calcule et renvoie le nombre  $2^n$  :

```
def puissance2(n):
    u = 1
    for i in range(n):
        u = 2 * u
    return u
```

par exemple

```
>>> puissance2(5):
32
```

Il faut remarquer que :

1. La variable `u` représente la valeur de  $2^i$  en entrant dans la boucle, en particulier c'est  $2^0$  avant de rentrer dans la boucle.
2. La variable `u` représente  $2^{i+1}$  à la fin de la boucle.
3. Lors de la dernière itération,  $i = n - 1$  et donc à la fin `u` représente bien  $2^n$ .
4. ... d'ailleurs on répète  $n$  fois l'opération de multiplier `u` par 2, en partant de 1, donc le résultat est bien  $2^n$ .

### Avertissement

Il est **très important** de se poser ce genre de questions quand on utilise des boucles pour calculer des suites et des sommes, notamment ce qui se passe au début et à la fin. Cela sera source de **nombreux** problèmes et pièges. Une possibilité est d'écrire clairement, en commentaire ou sur son brouillon, une phrase telle que « `u` représente  $2^i$  en début de boucle ».

Les exercices suivants sont les plus proches possible des questions posées à l'écrit. Pour tester son programme, on pourra aussi afficher les valeurs de la variable `u` dans la boucle.

**Exercice 2.** Écrire une fonction `suite(n)` qui renvoie le terme  $u_n$  de la suite définie par  $u_0 = 2$  et  $\forall n \in \mathbb{N}$ ,  $u_{n+1} = 1 + \frac{1}{u_n}$ .

**Exercice 3.** La *factorielle* de l'entier  $n \geq 0$  est le produit  $1 \times 2 \times \dots \times n$ . On convient que pour  $n = 0$  c'est 1 (c'est cohérent).

Écrire une fonction `factoriel(n)` qui calcule la factorielle de l'entier  $n$ .

**Exercice 4.** La *suite de Fibonacci* est la suite  $(F_n)_{n \in \mathbb{N}}$  définie par  $F_0 = 0$ ,  $F_1 = 1$  et la relation de récurrence

$$\forall n \in \mathbb{N}, \quad F_{n+2} = F_{n+1} + F_n. \quad (1)$$

Pour calculer les termes d'une telle suite on a besoin d'une boucle `for` avec *deux* variables : `u` qui représente la valeur de  $F_i$  et `v` qui représente  $F_{i+1}$ .

Écrire une fonction `fibonacci(n)` qui calcule le terme  $F_n$ .

Dans le cas des sommes, la variable qui représente le terme de la suite s'appelle plutôt `s` et s'appelle **variable accumulatrice**, car elle accumule la somme de plus en plus de termes. Elle est toujours initialisée à 0 (si aucune somme ne se produit, la valeur de retour doit être 0) et dans la boucle elle évolue selon une formule du type `s = s + ...` (le nouveau terme à sommer).

**Exercice 5.** Écrire une fonction `somme_cubes(n)` qui calcule et renvoie le résultat de la somme  $1^3 + 2^3 + \dots + n^3$ . Bonus : vérifier sur les 10 premières valeurs de  $n$  que la somme est bien égale à  $\frac{n^2(n+1)^2}{4}$ .

**Exercice 6.** Écrire une fonction `somme_inverse_factoriel(n)` qui calcule et renvoie le résultat de la somme des  $\frac{1}{k!}$  pour  $0 \leq k \leq n$ , c'est à dire  $1 + \frac{1}{2} + \frac{1}{6} + \dots + \frac{1}{n!}$ . Tester la fonction pour des valeurs de plus en plus grandes. Pouvez-vous l'écrire *sans* faire appel à la fonction `factoriel` précédente ?

*Remarque 1.* Parfois, on veut calculer les termes de la suite et arrêter la boucle quand une certaine condition est vérifiée, par exemple quand  $u_n$  dépasse une certaine valeur fixée à l'avance. Dans ce cas il faut revenir à une boucle `while` — la partie qui calcule les termes de la suite reste la même, et on rajoute à la main l'initialisation de `i` avant la boucle et l'incréméntation `i = i + 1` dedans (pour être cohérent, l'incréméntation *en fin de boucle*). Revoir pour cela les TP précédents et leur correction.

## II Boucles doubles

Pour parcourir l'ensemble de toutes les valeurs possibles pour deux indices indépendants  $i$  et  $j$ , il est nécessaire d'utiliser une **double boucle**, ce qui n'est rien de plus qu'une boucle située dans le corps d'une autre boucle.

Étudions l'exemple suivant :

```
for i in range(3):
    for j in range(3):
        print("i =", i, "j =", j)
```

Il produit le résultat :

```
i = 0 j = 0
i = 0 j = 1
i = 0 j = 2
i = 1 j = 0
i = 1 j = 1
i = 1 j = 2
i = 2 j = 0
i = 2 j = 1
i = 2 j = 2
```

Ainsi il énumère toutes les combinaisons possibles où  $i \in \llbracket 0, 2 \rrbracket$  et  $j \in \llbracket 0, 2 \rrbracket$ . Mais il est important de bien comprendre le mécanisme. Que se passe-t-il si on inverse l'ordre des boucles (mais qu'on garde le même `print`)?

Le code

```
for j in range(3):
    for i in range(3):
        print("i =", i, "j =", j)
```

produit

```
i = 0 j = 0
i = 1 j = 0
i = 2 j = 0
i = 0 j = 1
i = 1 j = 1
i = 2 j = 1
i = 0 j = 2
i = 1 j = 2
i = 2 j = 2
```

Ce sont en fait deux ordres naturels pour énumérer le carré d'indices  $(i, j) \in \llbracket 0, 2 \rrbracket \times \llbracket 0, 2 \rrbracket$ .

(dessin au tableau)

**Exercice 7.** On souhaite écrire une fonction (sans arguments ni valeur de retour) qui affiche la table de multiplication, la plus classique, des nombres de 1 à 10.

1. Écrire une fonction `table_multiplication()` qui affiche la table, produisant le résultat suivant :

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
...
1 * 10 = 10
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
...
10 * 10 = 100
```

2. On souhaite maintenant formater la table en carré, dans le but de produire le résultat suivant :

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 ...
3 6 9 12 ...
...
10 20 30 40 50 60 70 80 90 100
```

Pour cela on a besoin de savoir que

— Dans la commande `print()`, rajouter l'argument final `end=""` désactive le saut de ligne automatique (en fait, cela remplace le caractère de saut de ligne ajouté automatiquement par la chaîne vide, donc il ne se passe rien) : `print(x, end="")`.

— Et `print()` sans argument effectue un simple saut de ligne.

Écrire cette fonction, qu'on appellera `table_multiplication_carré()`.

3. Bonus : on souhaite enfin aligner correctement les colonnes, pour obtenir

```
 1  2  3  4  5  6  7  8  9 10
 2  4  6  8 ...
 3  6  9 12 ...
...
10 20 30 40 50 60 70 80 90 100
```

Pour cela on utilise les chaînes formatées : la syntaxe `print(f"{x:4d}")` affiche le nombre entier `x` en prenant l'espace de 4 caractères, serré à droite. Si `x = 10` la chaîne `f"{x:4d}"` est ainsi remplacée par `" 10"` (le `f` devant signifie que la chaîne doit être formatée, le `x` est le nom de la variable, et `4d` est l'un des nombreux codes de formatage signifiant ici *entier affiché sur 4 caractères*).

Écrire cette fonction, qu'on appellera `table_multiplication_carré_joli()`.

**Exercice 8.** Un *triplet pythagoricien* est la donnée de trois nombres  $(a, b, c) \in \mathbb{Z}^3$  tels que

$$a^2 + b^2 = c^2. \quad (2)$$

On souhaite afficher tous les triplets pythagoriciens. Remarquons que :

1. On peut toujours changer l'un des nombres en son opposé et on obtient toujours un triplet pythagoricien. On se limitera donc aux triplets de nombres positifs.
2. Il y a des solutions évidentes pour  $a = 0$  (avec  $b = c$ ) et pour  $b = 0$  (avec  $a = c$ ). On peut donc même chercher les solutions où  $a$  et  $b$  sont strictement positifs.
3. Il peut y avoir une infinité de solutions. Il faut donc fixer un entier  $N$  et chercher les solutions  $(a, b, c)$  avec  $a \leq N, b \leq N, c \leq N$ .

Écrire une fonction `pythagore(N)` qui affiche tous les triplets pythagoriciens avec  $(a, b, c)$  comme ci-dessus.

Bonus : chaque triplet apparaît deux fois, en échangeant  $a$  et  $b$ . Pouvez-vous les énumérer de telle façon à ce que chaque triplet n'apparaisse qu'une seule fois ?

### III D'autres types d'itération

Un **tuple** est la donnée de deux ou plusieurs objets Python, entre parenthèses et séparés par des virgules. Cela forme un nouveau type qui correspond directement à la notion de produit cartésien en mathématiques. Par exemple la variable

```
>>> t = (4, 12)
>>> type(t)
<class 'tuple'>
```

est construite à partir de deux nombres et est de type `tuple`. Il est alors possible de la **recupérer ses valeurs** avec la syntaxe

```
>>> (x, y) = t
>>> x
4
>>> y
12
```

qui va alors créer d'un coup deux nouvelles variables `x` et `y`. Les composantes d'un tuple peuvent en fait être de tous les types vus précédemment.

Il est possible d'**itérer sur les éléments d'un tuple**. Dans la boucle suivante, la variable `x` prend successivement les valeurs des composantes de `t`. Testez-la !

```
t = ("mathématiques", "physique-chimie", "SVT"):
for x in t:
    print("J'aime le cours de", x, "car je suis en BCPST.")
```

**Exercice 9.** Écrire un programme, le plus court possible, qui affiche côte à côte les valeurs de la fonction `somme_cubes(n)` de l'exercice 5, et les valeurs de  $\frac{n^2(n+1)^2}{4}$ , pour les valeurs de  $n = 1$ ,  $n = 2$ ,  $n = 3$ ,  $n = 10$ ,  $n = 15$ .

Un autre intérêt des tuples est de fournir une syntaxe rapide et pratique pour échanger deux variables : c'est tout simplement

```
(y, x) = (x, y)
```

sans avoir besoin d'écrire une variable intermédiaire.

Les tuples sont faciles à utiliser pour une fonction qui doit renvoyer *deux* valeurs.

**Exercice 10.** Écrire une fonction `minmax(a, b)` qui prend en argument deux nombres et renvoie un tuple formé (dans cet ordre) de leur minimum et de leur maximum.

**Exercice 11.** Écrire une fonction `multiplie(a, t)` qui prend en argument un nombre `a` et un tuple de deux variables `t` et qui renvoie le tuple formé des deux composantes de `t` multipliées par `a`. On doit par exemple avoir

```
>>> multiplie(4, (1, 3))
(4, 12)
```

Attention aux syntaxes et aux types des objets !

Enfin nous verrons au TP suivant qu'une boucle `for` est utile pour obtenir un par un les caractères d'une chaîne. Le code suivant

```
s = "Bonjour"
for x in s:
    print(x)
```

produit le résultat

```
B
o
n
j
o
u
r
```

autrement dit la variable `x` prend successivement pour valeurs les caractères de la chaîne `s`. On parle d'**itération sur une chaîne de caractères**.