Chapitre 3
Boucles

1 Affichages **1**

2 Boucles indexées : boucles `for` **2**
2.1 La commande `range` . . . 2
2.2 Syntaxe des boucles `for` . . . 3
2.3 Applications aux suites récurrentes . . . 5
2.3.1 Calcul des termes d'une suite récurrente . . . 5
2.3.2 La factorielle . . . 6

3 Boucles conditionnelles : boucles `while` **6**
3.1 Syntaxe des boucles `while` . . . 6
3.2 Application aux suites récurrentes . . . 7

4 Comparaison des boucles `for` et `while` **8**

1 Affichages

À mesure que les programmes que l'on écrit deviennent plus compliqués, il devient intéressant, pour bien en comprendre le fonctionnement, de tester chaque étape du programme individuellement. L'une des manières de procéder consiste à **afficher**, de manière provisoire, les résultats intermédiaires, à l'aide de la commande **`print`**.

Testons la commande `print` sur un exemple simple.

```
def fonction(x):
    """
    Entrée : un nombre x (entier ou flottant).
    Sortie : 3.2 si x est strictement supérieur à 8, et 4 * x sinon.
    """
    if -1 < x < 10:
        y = x + 1
        print(y) # On affiche la valeur de la variable y.
    if x > 8:
        y = 3.2
        print(y) # On affiche la nouvelle valeur de la variable y.
    else:
        y = 4 * x
        print(y) # On affiche la nouvelle valeur de la variable y.
    return y # On renvoie la valeur de la variable y.
```

```
fonction(7)

fonction(9)

fonction(-21)
```

⚠ On ne confondra pas les commandes `print` et `return` !
La commande `print` ne renvoie rien, elle produit simplement un affichage à l'écran. Contrairement à la sortie d'une fonction, produite par la commande `return`, un affichage ne peut pas être réutilisé par une fonction ou affecté à une variable. Il est uniquement destiné à l'utilisateur ou l'utilisatrice, et l'aide à suivre les différentes étapes du programme.

```
b = fonction(-5.0)

b # La variable b est la sortie de la fonction fonction appliquée à -5.0.

a = print(4)

type(a)

a
```

Une petite remarque sur la syntaxe de la commande `print` : l'expression à afficher doit toujours être entre parenthèses (c'est l'argument de la fonction `print`).

```
print 3
```

Lorsqu'une opération doit être répétée plusieurs fois dans un programme, on utilise une boucle. Les boucles, qui sont de deux types : boucles `for` et boucles `while`, sont des structures de contrôle fondamentales en informatique.

2 Boucles indexées : boucles `for`

2.1 La commande `range`

La commande **`range`** produit des ensembles d'entiers, qui pourront alors être parcourus par l'indice d'une boucle `for`, de la manière suivante : si n, a, b et r sont des entiers, alors

- ⋆ l'objet `range(n)` est la liste[1] de tous les entiers compris entre 0 et $n-1$, c'est-à-dire de l'intervalle d'entiers $[\![0;n[\![$;
- ⋆ l'objet `range(a, b)` est la liste des entiers de $[\![a;b[\![$, c'est-à-dire la liste des entiers allant de a inclus à b exclu (par pas de 1) ;
- ⋆ l'objet `range(a, b, r)` est la liste de tous les termes de la suite arithmétique de premier terme a et de raison r, jusqu'à la dernière valeur *strictement* avant b, c'est-à-dire la liste des entiers allant de a inclus à b exclu par pas de r.

1. En fait, ce n'est pas tout à fait une liste (de type `list`), mais un objet un peu spécial, de type `range`. Pour le convertir en liste, on peut lui appliquer la fonction `list`. Nous étudierons les listes en détail dans la séance 5.

Quelques remarques :

- ⋆ en informatique, comme pour les étages d'un immeuble, on commence à compter à zéro ;
- ⋆ lorsque deux valeurs sont spécifiées dans la fonction `range`, la première est incluse et la deuxième est exclue ;
- ⋆ les arguments de la fonction `range` doivent être entiers ;
- ⋆ si n est un entier, alors `range(n)` contient n éléments (les entiers de 0 à $n-1$) ;
- ⋆ si a et b sont des entiers tels que $a \leqslant b$, alors les éléments de `range(a, b)` vont de a à $b-1$, donc `range(a, b)` contient $(b-1)-a+1 = b-a$ éléments.

```
list(range(10))

list(range(3, 13))

list(range(13, 3)) # Cette liste est vide.

list(range(3, 13, 4))

list(range(3, -1, -1)) # C'est parti pour les boucles !
```

2.2 Syntaxe des boucles `for`

La boucle **`for`** permet de répéter un bloc d'instructions un certain nombre de fois, ce nombre étant connu. Elle s'écrit de la manière suivante.

```
for k in range(n):
    bloc indenté d'instructions
    dépendant éventuellement de la variable k
```

La variable `k` va alors parcourir la liste `range(n)`, c'est-à-dire qu'elle va prendre successivement la valeur 0, la valeur 1, et ainsi de suite jusqu'à la valeur $n-1$. Pour chacune de ces valeurs, le bloc d'instructions sera exécuté. Il y aura donc n tours de boucle.

Comme d'habitude, on prendra garde

- ⋆ aux discrets deux points à la fin de la première ligne ;
- ⋆ à l'indentation, qui délimite le bloc d'instructions à répéter. Ce qui n'est pas indenté d'un cran n'est pas dans la boucle `for` et n'est donc pas répété.

Comparer les deux scripts suivants :

```
for k in range(10):
    a = 2**k
    print(a)
```

et

```
for k in range(10):
    a = 2**k
print(a)
```

Bien entendu, on peut écrire des boucles `for` dans lesquelles l'indice de boucle parcourt une autre version de liste `range` :

```
for k in range(a, b):
    bloc indenté d'instructions
    dépendant éventuellement de la variable k
```

ou

```
for k in range(a, b, r):
    bloc indenté d'instructions
    dépendant éventuellement de la variable k
```

Voici quelques remarques sur l'indice d'une boucle `for`.

⋆ Le bloc d'instructions de la boucle ne dépend pas nécessairement de l'indice de la boucle. Quand ce n'est pas le cas, l'indice de la boucle sert simplement à comptabiliser le nombre de fois où le bloc d'instructions est répété.

```
for k in range(7): # k va de 0 à 6, donc il y a 7 tours de boucle.
    print("Je me répète.")
```

```
def puissance(x, n):
    """
    Entrées : un nombre x entier ou flottant,
              un entier n positif.
    Sortie : le nombre x élevé à la puissance n.
    """
    produit = 1 # On initialise la variable produit à 1 (produit vide).
    for k in range(1, n + 1): # k va de 1 à n,
                              # donc il y a n tours de boucle.
        produit = produit * x
        # À chaque tour, la variable produit est multipliée par x.
    return produit
    # À la fin de la boucle, on renvoie la valeur de produit.

puissance(3, 2)

puissance(3, 0)
```

⋆ L'indice de la boucle n'a pas obligation à s'appeler `k` : il peut prendre n'importe quel nom que ne porte pas déjà une autre variable. Comme pour toute autre variable, il est pertinent de donner un nom significatif à l'indice de boucle : par exemple, si l'indice parcourt une liste d'années, on choisira de l'appeler `an` ou `year` (parce qu'on parle drôlement bien anglais).

⋆ Contrairement à ce qui se passe en mathématiques pour un indice de sommation, qui est muet et n'est donc pas défini à l'extérieur de la somme, l'indice de la boucle garde après la boucle la dernière valeur qu'il a prise.

```
compteur_de_tours = 0
for k in range(1, 14, 3):
    compteur_de_tours += 1 # On incrémente le compteur.
```

```
compteur_de_tours

k
```

⋆ On peut bien sûr imbriquer plusieurs boucles les unes dans les autres, à condition de donner des noms différents aux indices des différentes boucles (comme en mathématiques).

```
for i in range(4):
    for j in range(6):
        print(i * j)
```

2.3 Applications aux suites récurrentes

Un exemple important d'utilisation de boucles `for` est le calcul des termes d'une suite récurrente.

2.3.1 Calcul des termes d'une suite récurrente

Pour calculer le terme de rang n d'une suite récurrente $(u_n)_{n\in\mathbb{N}}$, on procède de la manière suivante.

- On commence par créer une variable `terme` qui vaut initialement u_0, le premier terme de la suite.
- Ensuite, on fait une boucle : à chaque tour de la boucle, la variable `terme` prendra la valeur du terme suivant de la suite.
  Pour cela, on applique à la variable `terme` la formule de récurrence qui définit la suite, puis on affecte le résultat à la variable `terme`. La variable `terme` change donc de valeur après chaque tour de boucle.
  La boucle possèdera n tours :
  - ⋆ avant le premier tour de boucle, la variable `terme` vaut u_0 ;
  - ⋆ après un tour de boucle, la variable `terme` vaut u_1 ;
  - ⋆ après deux tours de boucle, la variable `terme` vaut u_2 ;
  - ⋆ …
  - ⋆ après n tours de boucle, la variable `terme` vaut u_n.
- Une fois la boucle terminée, on renvoie la valeur de la variable `terme`, qui vaut u_n.

Illustrons cette méthode sur un exemple.
Considérons la suite $(u_n)_{n\in\mathbb{N}}$ définie par :

$$\begin{cases} u_0 = 4 \\ \forall n \in \mathbb{N}, u_{n+1} = 3u_n^2 - 1. \end{cases}$$

La fonction suivante prend en argument un entier n et calcule le terme u_n de la suite $(u_n)_{n\in\mathbb{N}}$ (c'est-à-dire son $(n+1)$-ième terme).

```
def suite(n):
    """
    Entrée : un entier naturel n.
    Sortie : le terme u_n de rang n.
    """
    terme = 4 # Initialement, la variable terme vaut u_0.
    for iteration in range(n): # L'entier iteration va de 0 à n - 1,
                               # on fait donc n tours de boucle,
                               # car on va calculer n nouveaux termes.
        terme = 3 * terme**2 - 1
        # terme prend la valeur du terme suivant de la suite.
    return terme

suite(0)

suite(3)
```

2.3.2 La factorielle

```
def factorielle(n):
    """
    Entrée : un entier n positif.
    Sortie : la factorielle de n.
    """
    produit = 1
    for k in range(1, n + 1): # L'indice k va de 1 à n.
        produit = produit * k
        # À chaque étape, on multiplie le résultat par k.
    return produit
```

```
factorielle(4)

factorielle(0)
```

3 Boucles conditionnelles : boucles `while`

3.1 Syntaxe des boucles `while`

La boucle **`while`** permet de répéter un bloc d'instructions aussi longtemps qu'une certaine condition reste vérifiée. Elle s'écrit de la manière suivante.

```
while condition :
    bloc indenté
    d'instructions
```

Tant que la condition booléenne `condition` est vraie, Python exécute le bloc d'instructions. Dès que la condition devient fausse, Python sort de la boucle et passe à la suite.

Voici quelques remarques sur les boucles `while`.

- ⋆ Comme d'habitude, on prendra garde aux deux points à la fin de la première ligne, ainsi qu'à l'indentation qui délimite le bloc d'instructions de la boucle.
- ⋆ Pour que la boucle `while` termine, il faut s'assurer qu'une instruction provoque à terme l'arrêt de la boucle.

  Le bloc d'instructions doit donc agir sur la condition de la boucle, de sorte que celle-ci, vraie au départ, devienne fausse au bout d'un certain nombre d'itérations. Afin de s'assurer de la terminaison de la boucle, il est recommandé d'écrire au brouillon la condition d'arrêt de la boucle et sa négation.

Tester la fonction suivante.

```
def franklin(k):
    """
    Entrée : un nombre k, entier ou flottant.
    Sortie : le nombre de fois qu'il faut ajouter 2 au nombre k
             pour dépasser 12 (largement).
    """
    compteur_de_tours = 0
    while k < 12:
        k = k + 2
        print(k) # On affiche la valeur de k
                 # afin de voir évoluer la boucle.
        compteur_de_tours += 1
    return compteur_de_tours

franklin(-5.7)

franklin(0)

franklin(13)
```

Dans cet exemple, à chaque tour de boucle, le nombre k est augmenté de 2, nous rapprochant ainsi de la sortie de la boucle, c'est-à-dire du moment où la condition $k < 12$ sera fausse.

3.2 Application aux suites récurrentes

Un exemple important d'utilisation des boucles `while` est le calcul du premier rang à partir duquel une suite récurrente dépasse un seuil donné.

Considérons à nouveau la suite $(u_n)_{n\in\mathbb{N}}$ définie par :

$$\begin{cases} u_0 = 4 \\ \forall n \in \mathbb{N}, u_{n+1} = 3u_n^2 - 1. \end{cases}$$

Ce sera un très bon exercice du cours de maths de montrer que la suite $(u_n)_{n\in\mathbb{N}}$ tend vers $+\infty$. Admettons-le ici. Alors quel que soit le seuil choisi, il existera un rang à partir duquel tous les termes de la suite sont supérieurs à ce seuil.

Le calcul du rang de premier dépassement d'un seuil donné peut s'effectuer de deux manières différentes : soit en utilisant la fonction `suite` qui donne accès directement aux termes de la suite, soit, plus efficacement, en calculant les termes de la suite au fur et à mesure.

```
def depassement(seuil):
    """
    Entrée : un nombre seuil, entier ou flottant.
    Sortie : le premier rang auquel la suite (u_n)_(n in N)
             dépasse (au sens large) le seuil seuil.
    """
    n = 0 # n sera le premier rang auquel la suite dépasse le seuil.
    while suite(n) < seuil:
    # Tant que le terme de rang n est inférieur au seuil,
    # on incrémente n de 1, c'est-à-dire qu'on passe au rang suivant.
        n += 1
    return n

depassement(1.4)

depassement(1000)

depassement(10**1000)
```

```
def depassement_version_efficace(seuil):
    """
    Entrée : un nombre seuil, entier ou flottant.
    Sortie : le premier rang auquel la suite (u_n)_(n in N)
             dépasse le seuil seuil.
    """
    n = 0
    terme = 4 # terme sera à chaque étape le terme de rang n.
    while terme < seuil:
        terme = 3 * terme**2 - 1 # On passe au terme suivant.
        n += 1 # Et au rang suivant.
    return n

depassement_version_efficace(10**1000)
```

4 Comparaison des boucles `for` et `while`

Il peut arriver que pour un même algorithme, le choix s'offre à nous d'utiliser une boucle `for` ou une boucle `while`.

Il est en effet très facile de transformer une boucle `for` en boucle `while`.
Par exemple, la boucle `for` suivante

```
for k in range(n):
    bloc
    d'instructions
```

s'écrit ainsi avec une boucle `while` :

```
k = 0 # On initialise l'indice k à 0.
while k < n:
    bloc
    d'instructions
    k = k + 1 # On passe à l'indice suivant.
```

En revanche, comme le nombre d'itérations d'une boucle `while` n'est pas toujours connu à l'avance, la transformation d'une boucle `while` en boucle `for` n'est pas toujours possible.

Lorsque le nombre de tours de boucles est connu à l'avance (ou borné), on choisira en général la boucle `for`, qui est plus simple :

- ⋆ la boucle `for` gère toute seule l'incrémentation de l'indice de la boucle, alors qu'il est nécessaire de le faire à la main dans une boucle `while` ;
- ⋆ l'écriture de la boucle `for` est plus simple que celle de la boucle `while`, dans laquelle l'écriture de la condition de sortie de boucle peut s'avérer délicate.

Lorsque le nombre de tours de boucles n'est pas connu à l'avance (comme c'est le cas dans le calcul du rang de premier dépassement d'un seuil), on choisira une boucle `while`, qui est plus souple.