Les structures de contrôle
Les structure itératives

I. Structures de contrôle

En informatique, on appelle **structure de contrôle** une commande qui contrôle l'ordre dans lequel les différentes instructions d'un programme sont exécutées. En **Python**, on peut distinguer plusieurs types de structures de contrôle :

a) les **structures séquentielles** : les différentes commandes sont exécutées les unes à la suite des autres *i.e.* dans un ordre séquentiel.

b) les **structures conditionnelles** : qui permettent d'introduire des branchements conditionnels dans le programme. Un programme peut comporter plusieurs branches. Chacune d'elle est associée à une condition. La branche exécutée est la première dont la condition est réalisée.

c) les **structures itératives** : qui permettent d'effectuer la répétition (*i.e.* l'itération) de commandes. Ces répétitions peuvent s'effectuer un nombre de fois fixé explicitement par le programmeur ou peuvent avoir lieu tant qu'une condition n'est pas réalisée.

Dans ce chapitre, nous nous intéressons aux structures itératives.

II. Les boucles bornées : boucles for

II.1. Syntaxe de la commande

II.1.a) Syntaxe générale

Une boucle for permet de réaliser la répétition d'un bloc d'instructions. La syntaxe de cette commande est la suivante.

```
for var in liste:
    instruction
```

Détaillons les éléments de cette syntaxe.

- *liste* : est une liste de valeurs.
- *var* : est une variable qui prend successivement chacune des valeurs contenues dans *liste*.
- *instruction* : désigne un bloc d'instructions qui va être exécuté, de manière successive, pour toutes les valeurs de *var*.

II.1.b) Syntaxe du cas classique d'utilisation

Généralement, la liste *liste* sera présentée sous la forme `range(n,m,pas)`. Ainsi, la variable *var* prendra successivement les valeurs de `n` à `m` -1 avec une distance de `pas`. Si la quantité `pas` n'est pas précisée, elle prendra par défaut la valeur 1. Si la quantité `n` n'est pas précisée, elle prendre par défaut la valeur 0.

La syntaxe classique d'utilisation est la suivante.

```
for var in range(n, m, p):
    instruction
```

II.2. Ordre d'exécution

Afin de détailler la séquence d'exécution on considère l'exemple consistant à calculer les `m` premiers éléments d'une suite $(u_n)_{n\in\mathbb{N}}$ où `m` est une quantité choisie par l'utilisateur. Considérons ici la suite de terme général $u_n = n^3$.

```
# Calcul des m premiers éléments de la suite u_n = n^3
m = int(input("Entrez la valeur de m : "))
u = []
for i in range(m):
    u.append(i**3)
```

La boucle for s'exécute comme suit :

0) initialement, la variable `i` est affectée à la valeur `0`.
↪ l'instruction `u.append(0**3)` est alors exécutée.

1) la variable `i` est ensuite affectée à la valeur `1` (`0` + le pas de `1`).
↪ l'instruction `u.append(1**3)` est alors exécutée.

2) la variable `i` est ensuite affectée à la valeur `2` (`1 + 1`).
↪ l'instruction `u.append(2**3)` est alors exécutée.

...

m-2) la variable `i` est ensuite affectée à la valeur `m-2`.
↪ l'instruction `u.append((m-2)**3)` est alors exécutée.

m-1) la variable `i` est ensuite affectée à la valeur `m-1`.
↪ l'instruction `u.append((m-2)**3)` est alors exécutée.

En sortie de boucle, la liste `u` contient les `m` premières valeurs de la suite de terme général $u_n = n^3$. La variable `i`, quant à elle, contient alors la valeur `m-1`.

Remarque Dans cet exemple, on peut aussi obtenir un tel vecteur `u` par compréhension de liste :

```
u = [i**3 for i in range(m)]
```

III. Les boucles non bornées : boucles while

III.1. Syntaxe de la commande

Une boucle while permet de réaliser la répétition d'un bloc d'instructions sous réserve qu'une condition est vérifiée. La syntaxe de cette commande est la suivante.

```
while condition:
    instruction
```

Détaillons les éléments de cette syntaxe.

- *condition* : représente une expression **Python** dont l'évaluation est soit `True` soit `False`. Autrement dit, un objet de type `boolean`.
- *instruction* : désigne un bloc d'instructions qui va être exécuté, de manière successive, **tant que** *condition* est vraie. Il est à noter que cette condition peut dépendre de variables qui sont modifiées lors de l'exécution de *instruction*.

⚠ **Terminaison** : dans le cas d'une boucle `while`, l'exécution se termine dès que *condition* n'est plus vérifiée. Si *condition* est toujours vérifiée (comme `0==0` ou `True`) le programme admet une exécution infinie. On parle de **boucle infinie**.

III.2. Ordre d'exécution

Afin de bien comprendre l'ordre d'exécution d'une boucle while, on va l'illustrer à l'aide d'un exemple consistant à trouver, dans une liste, le premier élément vérifiant une condition donnée. Commençons par générer une liste de valeurs aléatoires en important la fonction `random`.

```
--> r = [random() for i in range(8)]
    r  =
        [0.49,   0.18,   0.75,   0.94,   0.21,   0.58,   0.26]
```

On demande alors à l'utilisateur d'entrer un réel `t` entre `0` et `1` dans le but de déterminer le premier élément de la liste `r` plus grand que `t`.

```
1  # Le 1er élément de r plus grand que le réel t demandé
2  t = float(input("Choisissez une valeur pour t : "))
3  i = 0
4  while r[i] < t:
5      i = i + 1
6  print("Dans r, le 1er élément >= à t est ", r[i])
```

Si l'utilisateur choisit la valeur `0.5`, on obtient le retour console suivant.

```
--> runfile('/Info/Exemple_cours/exempleWhile.py',
wdir='/Info/Exemple_cours')
Choisissez une valeur pour t : 0.5

  Dans r, le 1er élément >= à t est 0.75
```

Dans cet exemple, la variable `i` est initialement affecté à la valeur `0`. La boucle while s'exécute alors comme suit :

0) on teste si `r[0] < 0.5`.
↪ c'est le cas (`r[0] = 0.49`). Les instructions de la boucle sont donc exécutées. La variable `i` prend alors la valeur `1` (`0 + 1`).

1) on teste si `r[1] < 0.5`.
↪ c'est le cas (`r[1] = 0.18`). Les instructions de la boucle sont donc exécutées. La variable `i` prend alors la valeur `2` (`1 + 1`).

2) on teste si `r[2] < 0.5`.
↪ c'est le cas (`r[2] = 0.75`). Les instructions de la boucle sont donc exécutées. La variable `i` prend alors la valeur `3` (`2 + 1`).

3) on teste si `r[3] < 0.5`.
↪ ce n'est pas le cas (`r[3] = 0.94`). Les instructions de la boucle ne sont pas exécutées et il y a sortie de la boucle.

4) L'instruction d'affichage `print` est alors exécutée.

Si l'utilisateur choisit la valeur `0.95`, on obtient le retour console suivant.

```
--> runfile('/Info/Exemple_cours/exempleWhile.py',
wdir='/Info/Exemple_cours')
Choisissez une valeur pour t : 0.95

  IndexError list index out of range
```

Pour comprendre ce message d'erreur, étudions l'exécution de la boucle while correspondante. La variable `i` est initialement affectée à la valeur `0`. Puis, on teste si les éléments successifs de la liste `r` sont strictement plus petits que `0.95`. Revenons sur le dernier test.

6) on teste si `r[6] < 0.95`.
↪ c'est le cas (`r[6] = 0.26`). Les instructions de la boucle sont exécutées. La variable `i` prend alors la valeur `7` (`7 + 1`).

7) on teste si `r[7] < 0.95`.
↪ ce test provoque l'affichage d'un message d'erreur. En effet, on tente un accès hors index puisqu'il n'y a que 7 éléments dans le vecteur `r`.

Il faut donc modifier le script du programme de sorte à ne pas réaliser d'accès hors index. Pour ce faire, il suffit de vérifier que `i` ne dépasse pas la taille de la liste avant de réaliser le test `r[i] < t`.

```
1  # Le 1er élément de r plus grand que le réel t demandé
2  t = float(input("Choisissez une valeur pour t : "))
3  n = len(r)
4  i = 0
5  while (i <= n-1) and (r[i] < t):
6      i = i + 1
7  if (i > n-1):
8      print("La liste r ne possède pas d'éléments >= à t")
9  else:
10     print("Dans r, le 1er élément >= à t est ", r[i])
```

On teste ce nouveau script. Si l'utilisateur choisit la valeur `0.95`, on obtient maintenant le retour console suivant.

```
--> runfile('/Info/Exemple_cours/exempleWhile.py',
wdir='/Info/Exemple_cours')
Choisissez une valeur pour t : 0.95

  La liste r ne possède pas d'éléments >= à t
```

La modification a bien empêché l'accès hors index. Ceci est notamment dû au mécanisme d'évaluation de l'opérateur `and`. À chaque tour de boucle, la condition `(i <= n-1) & (r[i] < t)` est testée. L'évaluation de ce test se déroule comme suit.

Tout d'abord, l'interpréteur teste si `(i <= n-1)`.

- ***(i)*** Si c'est le cas, `(r[i] < t)` est alors testé.
  - ***a)*** Si c'est le cas, la condition est vérifiée et les instructions de la boucle sont exécutées.
  - ***b)*** Sinon, la condition n'est pas réalisée et on sort de la boucle.
- ***(ii)*** Sinon, la condition n'est pas réalisée et on sort de la boucle.

IV. Les différences entre les boucles for et while

IV.1. Simuler une boucle for à l'aide d'un while

On peut simuler un for à l'aide d'un while en utilisant un compteur que l'on incrémente à chaque tour de boucle.
Sur l'exemple de la Partie **II**, ceci se fait de la manière suivante.

```
1  # Calcul des m premiers éléments de la suite u_n=1/n
2  m = int(input("Entrez la valeur de m : "))
3  u = []
4  i = 0
5  while i <= m-1:
6      u.append(i**3)
7      i = i + 1
```

Dans cet exemple, la variable `i` est initialement affectée à la valeur `0`. La boucle while s'exécute alors comme suit :

- ***0)*** on teste si `0 <= m-1`.
  ↪ c'est le cas. Les instructions de la boucle sont donc exécutées. Ainsi `u[0]` prend la valeur `0` (`0**3`) et `i` prend la valeur `1` (`0 + 1`).
- ***1)*** on teste si `1 <= m-1`.
  ↪ c'est le cas. Les instructions de la boucle sont donc exécutées. Ainsi `u[1]` prend la valeur `1` (`1**3`) et `i` prend la valeur `2` (`1 + 1`).

  ...
- ***m-1)*** on teste si `m-1 <= m-1`.
  ↪ c'est le cas. Les instructions de la boucle sont donc exécutées. Ainsi `u[m-1]` prend la valeur `(m-1)**3` et `i` prend la valeur `m`.
- ***m)*** on teste si `m <= m-1`.
  ↪ ce n'est pas le cas. Les instructions de la boucle ne sont pas exécutées et il y a sortie de la boucle.

Finalement, on obtient bien le même résultat que le précédent : la liste `u` contient à la fin du script les `m` premiers éléments de la suite de teme général $u_n = n^3$.

IV.2. Boucle for ou boucle while : que choisir ?

Le choix de la structure itérative est fonction du problème que l'on a à résoudre. Plus précisément :

1) si l'on connaît a priori le nombre d'itérations à effectuer, on se tournera plutôt vers une boucle for.

- ***a)*** Modifier tous les coefficients d'une liste.
- ***b)*** Calculer les n premiers éléments d'une suite.
- ***c)*** Calculer une somme $\sum_{k=0}^{n}$ ou un produit $\prod_{k=0}^{n}$.

2) si l'on ne connaît pas a priori le nombre d'itérations à effectuer, on se tournera plutôt vers une boucle while.

- ***a)*** Trouver l'indice du premier coefficient non nul d'une liste.
- ***b)*** Pour une suite donnée, trouver l'indice du premier élément vérifiant une condition donnée.
- ***c)*** Connaissant le chiffre d'affaire \$ d'une entreprise et une estimation de son pourcentage de progression pour les 10 années à venir, calculer en combien de temps \$ sera doublé.

V. Mise en oeuvre

Exercice 1

1) Écrire une fonction `facto(n)` permettant le calcul de $n!$ à l'aide d'une structure itérative.

2) Écrire une fonction `binom(n, p)` permettant le calcul de $\binom{n}{p}$. On fera appel à la fonction précédente.

Exercice 2

Écrire une fonction `sommeInv(n,r)` calculant $\sum_{k=1}^{n} \frac{1}{k^r}$.