2 - Boucles : principes et exemples

I- Introduction

La boucle est une structure qui n'a pas vraiment d'équivalent, elle n'existe que dans les langages de programmation. On peut aussi parler de **structures répétitives** ou de **structures itératives**. Le but des boucles étant de répéter un nombre fini connu ou inconnu de le fois un bloc de code.

Grâce à la puissance de calcul des processeurs, ces structures de boucles permettent de réaliser des blocs d'instructions un grand nombre de fois en très peu de temps. C'est utile pour des simulations, parcours d'élément, saisie au clavier, …

On distingue deux principales boucles :

- la boucle « while »
- la boucle « for»

II- La boucle « while »

a- Principe et structure

C'est une boucle à condition, c'est à dire que le bloc d'instructions à l'intérieur de la boucle de sera exécuté que si la condition est vrai.

Syntaxe :

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

Les différentes conditions possibles :

Condition	Notation en python
a est égale à b	a == b
a est différent de b	a != b
a est inférieur à b	a < b
a est inférieur ou égal à b	a <= b
a est supérieur à b	a > b
a est supérieur ou égal à b	a >= b
a et b référence le même objet	a is b
a et b ne référence pas le même objet	a is not b
a in liste1	a est inclus dans liste1
a not in liste1	a n'est pas inclus dans liste1

Il est aussi possible de combiner plusieurs conditions à l'aide des connecteurs logiques « and », « or » et « not »

```python
liste = [1,2,3,4,5,6,7]
i=0
while i<len(liste):
    print(liste[i])
    i+=1
```

Ci-contre, un exemple de parcours de liste avec une boucle « while », la condition deviendra fausse dès que « i » sera supérieur à « taille de la liste (7) ». Il n'y a pas de risque de boucle infinie que i est incrémenté à chaque tour de boucle donc i croit vers len(liste).

Les itérations s'arrêteront que si la condition est fausse, il existe quelques cas d'erreur lié avec cette condition.

b- Cas d'erreurs

Le premier cas d'erreur est le fait que **la condition soit toujours fausse**, les instructions de la boucle de seront donc jamais exécutées, on appelle ça donc une **boucle superflue** ou du **code mort**.

Le deuxième cas d'erreur est que **la condition est toujours vraie**, on est donc dans un cas de **boucle infinie**. Malheureusement, il n'est pas possible de prouver qu'un algorithme se termine ou non. C'est un problème qu'on appelle le **problème de l'arrêt** et il est considéré indécidable (Turing-1936, voir leçon 13). Le programme ne détectera donc pas la boucle infini avant d'être bloqué à l'intérieur.

Ce genre de boucle infini prend beaucoup de ressources processeur et sur un système mono tâche, l'utilisateur sera incapable d'effectuer tout autre action. Les systèmes d'aujourd'hui sont pratiquement tous multitâche se qui permet donc à l'utilisateur d'interrompre le programme. C'est pour cela que les boucles infinies sont majoritairement considérées comme des bugs. Néanmoins, il existe des cas ou ce type de boucle trouve sont utilité comme en programmation évènementielle. Il est possible d'arrêter cette boucle avec une commande que l'on verra par la suite.

c- Boucle « do … while »

Il existe une variante de cette boucle while, la boucle « do… while ». Elle permet d'exécuter au moins une fois le code avant de vérifier la condition. Cette boucle n'existe pas en python mais voici la syntaxe en JAVA :

```java
1 do{
2   //Instructions
3 }while(a < b);
```

Il est quand même possible d'émulé une boucle « do…while » en python :

```python
# Instruction 1
# Instruction 2
# ...
# Instruction N
while condition:
    # Instruction 1
    # Instruction 2
    # ...
    # Instruction N
```

III- La boucle « for »

a- Principe et structure

Cette boucle permet d'exécuter un nombre de fois fini un bloc d'instruction. En effet, avant d'exécuter cette boucle, on connaît ou on peut connaître sont nombre d'itération.

La boucle « for » de python, au contraire d'autre langage, travail sur ce qu'on l'on appelle des séquences. Les séquences peuvent être des séquences de caractères (string), des listes, des séquences de nombre (range, linspace),…

Syntaxe de la boucle « for » en python avec un exemple de parcours de liste :

```
liste = [1,2,3,4,5,6,7]
for i in liste:
    print(i)
```

Comment ça se passe sous le capot ? Dès que le compilateur tombe sur une structure « for » comme ci-dessus, il va appeler l'itérateur de l'objet *liste*, l'itérateur va se charger de parcourir la liste. Il est créé à l'aide de la méthode spéciale « __iter__ » de l'objet *liste.* À chaque tour de boucle, Python appelle la méthode spéciale « __next__ » de l'itérateur, qui doit renvoyer l'élément suivant du parcours ou lever l'exception « StopIteration » si le parcours touche à sa fin.

b- Boucles de parcours avec range() et xrange()

Il est possible qu'on ne veuille pas parcourir de liste ou de chaine de caractère, il est donc possible de générer des séquences de nombre avec les fonctions *range* et *xrange*. Chacune de deux fonctions s'écrivent de la même façon :

```
range(borne_inf,borne_sup, optionnel : pas)
xrange(borne_inf,borne_sup, optionnel : pas)
```

La différence ? Elle se situe dans la manière de générer cette séquence de nombre.

La fonction *range* va retourner **une liste python** à N éléments, qui sera donc stockée en mémoire. Plus il y aura d'éléments, plus la liste prendra de l'espace mémoire. L'avantage reste dans l'accès aux éléments, car on possède une liste python et tous ces avantages.

La fonction *xrange* est quand à elle un peu différente, elle ne va pas retourner une liste mais **un objet *xrange***. Une seule valeur va être stockée en mémoire et la fonction va créer les valeurs au fur et à mesure qu'on en a besoin grâce à une technique appelée **yielding**. Ce type de fonction est un **générateur**.

c- Créer notre propre générateur

Développement : Créer notre propre générateur

```
def intervalle(borne_inf, borne_sup):

    while borne_inf < borne_sup:
        yield borne_inf
        borne_inf += 1


generateur = intervalle(5, 20)
for nombre in generateur:
    print(nombre)
    if nombre>17 :
        # equivalent du break
        generateur.close()
```

IV- Structures de boucles particulières

a- break, continue, else

L'instruction break permet de couper une boucle *for* ou *while*.
L'instruction continue permet de passer à l'itération suivante de la boucle.

Les boucles peuvent également disposer d'une instruction else ; celle-ci est exécutée lorsqu'une boucle se termine alors que tous ses éléments ont été traités (dans le cas d'un for) ou que la condition devient fausse (dans le cas d'un while), mais pas lorsque la boucle est interrompue par une instruction break.

Développement : Utilisation de break, else et continue

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         # loop fell through without finding a factor
...         print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

b- Boucles imbriquées

c- Boucles cachés

Condition : a in b
List[::-1]