Chapitre 3 : boucles

Ce que les ordinateurs savent très bien faire : répéter des instructions beaucoup de fois !

En programmation, on appelle *boucle* un système d'instructions qui permet de répéter un certain nombre de fois (voire indéfiniment, ce qui pose problème) toute une série d'opérations.

1 Boucles conditionnelles : `while`

Supposons qu'on veuille chercher, pour un nombre N déjà présent dans la mémoire sous le nom `N`, le plus petit entier n tel que $2^n > N$. On peut utiliser le programme qui suit :

```
n=0   # initialisation nécessaire
while 2**n <=N:
    n=n+1   # itération
```

Dans le programme :

— la variable `n` sert *de compteur*, qui vaut `0` avant le début de la boucle.

— l'instruction d'affectation `n=n+1` dit qu'on *incrémente* i.e. *augmente* de `n` de 1 à chaque étape, *tant que* (le *while*) la condition `2**n<=N` est réalisée.

— A l'arrêt de la boucle : notons n_f est le plus grand entier tel que $2^{n_f} \leq N$. Quand le *compteur* `n` arrive à la valeur n_f, la condition sur laquelle porte le `while` est encore réalisée, donc le programme exécute encore la commande `n=n+1`. A ce moment-là, on a $n = n_f + 1$. Quand la variable `n` est de nouveau testée pour la condition `2**n<=N` , la condition n'est plus vérifié : *on sort de la boucle.*
La valeur de `n` obtenue est bien $n_f + 1$, c'est-à-dire le plus petit entier n tel que $2^n > N$.

Pour les boucles `while` : il faut être sûr que la boucle s'arrête.

Attention à ne pas confondre le `while` avec le `if` :

Certains essaient parfois de faire la boucle précédente avec les instructions :

```
if  2**n <=N:
    n=n+1
```

La condition `if` entraînera *une seule* exécution de ce qui suit, **PAS** une boucle !

2 Boucles inconditionnelles ou boucle `for`

Lorsqu'on souhaite répéter un bloc d'instructions un nombre *déterminé* de fois, on dispose d'une structure qui économise d'une part *l'initialisation* du compteur et d'autre part son *incrémentation* : la boucle `for`.

2.1 Un premier exemple : s'habituer aux `range`

Si l'on veut afficher les entiers de 1 à 10 :

```
for i in range(1,11):
    print(i)
```

• Dans quoi varie le compteur `i` ? Ici dans un `range`.

Le type `range` a été brièvement présenté au chapitre précédent. Disons pour compléter ici que la fonction `range` produit un *itérateur*, qui est un objet qui, au lieu de garder en mémoire une liste d'entiers consécutifs, les fabrique au fur et à mesure, toujours dans un souci d'optimisation de la mémoire.

La gestion des bornes est la même que pour les autres types séquentiels autrement dit `range(0,11)` produira les entiers de 0 à 10 (attention) !

• Que peut-on faire avec un `range` ? De un à trois arguments

— `range(n)` fabriquera les entiers de 0 à $n-1$.
— `range(a,b)` fabriquera les entiers de l'ensemble $[\![a, b-1]\!]$
— `range(a,b,p)` fabriquera les entiers de la forme $a+kp$ pour $k \in \mathbb{N}$ qui sont inférieurs à b. Par exemple `range(3,14,2)` fabriquera `3,5,7,9,11,13`. Le troisième argument `p` s'appelle le *pas*.

Dans une boucle `for i in range()`, c'est PYTHON qui s'occupe de la variable `i` de lui faire prendre les valeurs successives donc :
Ne pas faire `i =i+1` dans une telle boucle par exemple ou encore `i=0` !
TOUCHE PAS A MA VARIABLE `i` de boucle `for` !!

2.2 Les boucles `for` permettent par exemple de calculer un $\sum$:

La méthode suivante de calcul de somme a été présentée en cours de maths :

```
s=0
for i in range(0,101):
    s=s+i # a chaque étape, on ajoute i à s.
print(s)   # ceci n'est pas indenté, donc est en dehors de la boucle for
```

Remarques : • Ici, on a une initialisation nécessaire, non pas pour le compteur `i` mais pour la variable `s` qui va contenir la valeur de la somme.

• En PYTHON, c'est *l'indentation* qui délimite le bloc d'instructions qui va être exécuté à chaque étape de la boucle.

Exercice : Quelle est la différence de résultat entre le code précédent et le suivant ?

```
s=0
for i in range(0,101):
    s=s+i
    print(s)
```

2.3 Le compteur d'une boucle `for` peut parcourir n'importe quel type séquentiel !

C'est une spécificité de PYTHON par rapport à d'autres langages

Les *types séquentiels* ont été introduits au chapitre précédents : on a vu le type *string*, *tuple*, et *list*.

• Un exemple où le compteur varie dans une *liste* :

Supposons qu'on ait une liste L formées de nombres et qu'on veuille ajouter tous les éléments de la liste. On ne sait pas a priori, la taille de la liste.

Dans un langage de programmation usuel, on écrira quelque chose qui avec la syntaxe PYTHON donne :

```
s=0
n=len(L) # donne le nombre d'éléments de L
for i in range(n):
    s=s+L[i] # i est l'INDICE
print(s)
```

Mais en PYTHON, on peut faire plus économique :

```
s=0
for valeur in L:
    s=s+valeur # N.B. valeur n'est pas un mot-clef, c'est juste un nom parlant.
print(s)
```

• **Un exemple où le compteur varie dans une *tuple* :** *le même, en remplaçant* `L` *par un* tuple *!*

• **Un exemple où le compteur varie dans une *chaîne de caractères* :**
Pour manipuler les chaînes de caractères, mentionnons deux options de la commande `print` :

— **Choix du séparateur :** normalement, si on donne plusieurs arguments à `print`, ils sont séparés par un espace à l'exécution de `print`. Par exemple :
```
age=18
print('votre âge est',age)
```
Mais on peut choisir un autre *séparateur* via l'argument (optionnel) `sep` :
```
a=2
b=3
print(a,b,sep='<')
```

— **Remplacement du saut à la ligne en fin de `print` par une autre caractère via l'argument `end`** : Par défaut, entre deux `print` il y a un saut à la ligne. Par exemple, avec les variables précédentes
```
print(a)
print(b)
```
donnera
```
2
3
```
Mais on peut déclarer plutôt
```
print(a,end=' et ')
print(b)
```

Revenons maintenant aux boucles `for` utilisant une chaîne de caractères :

```
classe='MPSI 1'
for i in classe:
    print(i,end='*')
print() # juste pour aller à la ligne à la fin.
```

Parce qu'on peut s'en servir pour *itérer* dans une boucle, les types séquentiels sont aussi appelés *types itérables.*