# Chapitre (S1) 2 Tests conditionnels - Boucle while



**Objectifs**

- Savoir utiliser la structure de test conditionnel.
- Savoir réaliser une boucle à l'aide de l'instruction `while`.

## 1. TESTS CONDITIONNELS

### 1.1. Instruction `if-else`

L'instruction `if`, éventuellement accompagnée de l'instruction `else`, a pour syntaxe générale

```
if condition :
    bloc de commandes 1
else :
    bloc de commandes 2
```

- `condition` est un booléen (ayant pour valeur `True` ou `False`), souvent issu d'un test,
- `bloc de commandes 1` est un ensemble de lignes de code qui sont exécutées si et seulement si `condition` a pour valeur `True`,
- `bloc de commandes 2` est un ensemble de lignes de code qui sont exécutées si et seulement si `condition` a pour valeur `False`

**Exemple 1 (Fonction `positif`)**

```
def positif(a:float)->bool:
    if a >= 0:
        return True
    else:
        return False
```

Cette fonction renvoie `True` si et seulement si la variable `a` passée en entrée est positive ou nulle, et `False` sinon.

**Exemple 2 (Fonction `nb_div`)**

```
def nb_div(n:int)->int:
    N = 0
    for k in range(2,n):
        if n%k == 0: # si k divise n
            N = N+1
    return N
```

Cette fonction renvoie le nombre de diviseurs de l'entier `n` passé en entrée (autres que 1 et `n`).

**Exercice 1** [Sol 1] On donne ci-dessous le code de la fonction `mult_3(n)`, où `n` est supposé être un entier naturel.

```
def mult_3(n:int)->tuple:
    N = 0
    S = 0
    for k in range(1,n):
        if k%3 == 0: # si 3 divise k
            N = N+1
            S = S+k
    return N,S
```

**1)** Que renvoie `mult_3(4)`, `mult_3(10)` ?

**2)** Décrire à l'aide d'une phrase ce que renvoie `mult_3(n)`.

#### PRÉCISIONS SYNTAXIQUES

- La syntaxe du langage impose un symbole « : » après la condition et un autre après le mot `else`.

- Les lignes formant les deux blocs de commandes doivent être indentées (décalées vers la droite) par rapport à celles commençant par **if** et **else**. Dans les éditeurs de texte, cette indentation s'effectue automatiquement par une retour à la ligne après le symbole « : » .
- L'instruction **else** est optionnelle. En l'absence de **else**, si la condition du **if** est fausse, les instructions du `bloc de commandes 1` sont simplement ignorées et le programme poursuit son exécution.

### 1.2. Instruction if-elif-...-else

Considérons le problème suivant : pour un entier $0 \leq n < 10000$, on souhaite écrire une fonction `nb_chiffres(n)` qui renvoie le nombre de chiffres nécessaires pour écrire $n$ (il faut 1 chiffre si $0 \leq n < 10$, deux chiffres si $10 \leq n < 100$ etc.). On peut écrire cette fonction à l'aide d'instructions **if** et **else** successives imbriquées :

```
def nb_chiffres(n:int)->int:
    if n < 10:
        Nb = 1
    else:
        if n < 100:
            Nb = 2
        else :
            if n < 1000:
                Nb = 3
            else:
                Nb = 4
    return Nb
```

On voit ici que l'emploi des tests imbriqués rend peu lisible le code. Une alternative est d'utiliser les instructions **if**, **elif** et **else**, selon la syntaxe générale :

```
if condition 1 :
    bloc de commandes 1
elif condition 2 :
    bloc de commandes 2
elif condition 3 :
    bloc de commandes 3
...
elif condition n :
    bloc de commandes n
else :
    bloc de commandes n+1
```

Lors de l'exécution, les conditions sont examinées successivement (dans l'ordre d 'écriture), et on exécute *uniquement* le bloc de commande relatif à la première condition rencontrées ayant pour valeur **True** (si certaines conditions placées ultérieurement ont pour valeur **True**, les blocs correspondants sont donc ignorés).

Le problème précédent peut donc se traiter également à l'aide de la fonction `nb_chiffres_bis(n)` suivante, plus lisible :

```
def nb_chiffres_bis(n:int)->int:
    if n < 10:
        Nb = 1
    elif n < 100:
        Nb = 2
    elif n < 1000:
        Nb = 3
    else:
        Nb = 4
    return Nb
```

**Exercice 2** [Sol 2] On donne les deux fonctions suivantes, dont le but est de préciser si l'entier $a$ passé en argument est divisible par 2 et/ou par 3.

```
def div(a:int)->tuple:
    d1,d2 = None,None
    if a%2 == 0:
        d1 = 2
    if a%3 == 0:
        d2 = 3
    return d1,d2
```

```
def div_bis(a:int)->tuple:
    d1,d2 = None,None
    if a%2 == 0:
        d1 = 2
    elif a%3 == 0:
        d2 = 3
    return d1,d2
```

Préciser, en justifiant, ce que renvoie chacun des appels `div(6)` et `div_bis(6)`.

## 1.3. Écriture des conditions

### 1.3.1. Tests élémentaires

La condition qui suit les instructions **if** ou **elif** est très souvent exprimée sous forme de test. Il peut s'agir d'un test élémentaire, dont on peut rappeler les différentes syntaxes :

| math | python |
|---|---|
| $<$ | < |
| $\leq$ | <= |
| $>$ | > |
| $\geq$ | >= |
| $=$ | == |
| $\neq$ | != |

### 1.3.2. Tests multiples

Il est également possible d'utiliser des opérateurs logiques suivants pour écrire des tests plus complexes. La syntaxe de ces opérateurs est la suivante :

| math | python |
|---|---|
| et | **and** |
| ou | **or** |
| non | **not** |
| non ou | nor |

et suivent les règles de logique et de priorité usuelles (vues en cours de mathématiques).

Ainsi pour savoir si un entier $n$ est positif et divisible par 3, on écrira `n >= 0 and n%3 == 0`.

**Remarque 1** On peut signaler une particularité de l'évaluation d'un test du type `a and b`, où `a` et `b` sont des tests élémentaires : si `a` a pour valeur **False**, alors le résultat du test `a and b` est **False**, quelque soit le résultat de `b`. Il n'est donc pas nécessaire d'évaluer le test `b`, et c'est ce que fait Python. Cette propriété, parfois nommée évaluation paresseuse, peut être utile dans certains cas pour éviter des erreurs, comme nous le verrons un plus tard.

**Remarque 2** Dans l'écriture des scripts Python, il est (fortement) recommandé de laisser un espace avant et après chaque opération de test (= =, ! =, etc. ), et ce pour plus de lisibilité (il en est de même pour l'instruction d'affectation =). Ainsi on préfèrera écrire `unecertainevariable = = uneautrevariable` (et pour l'affectation `unecertainevariable = unecertainevaleur`) plutôt que `unecertainevariable= =uneautrevariable` (et `unecertainevariable=unecertainevaleur`).

### 1.3.3. Généralisation

Dans la présentation précédente

```
if condition :
    bloc de commandes 1
else :
    bloc de commandes 2
```

on a vu que `condition` est un booléen (ayant pour valeur **True** ou **False**). Il n'est pas obligatoire d'écrire `condition` sous forme de test. En réalisé, toute variable booléenne peut convenir, ainsi que toute fonction renvoyant un booléen.

Ainsi, supposons que l'on dispose d'une fonction `premier(n)` qui renvoie **True** si `n` est un nombre entier premier, et **False** sinon. Alors, la fonction suivante :

```
def somme_prem(n:int)->int:
    S = 0
    for k in range(n):
        if premier(k):
            S = S+k
    return S
```

renvoie la somme des nombres premiers strictement inférieurs à `n`.

# 2. BOUCLE WHILE

## 2.1. Présentation

L'instruction **while** permet la répétition d'un bloc de commandes *tant qu*'une certaine condition est vérifiée.

La syntaxe de l'instruction **while** est :

```
while condition :
    bloc de commandes
```

où

- `condition` est un booléen (ayant pour valeur **`True`** ou **`False`**), souvent issu d'un test,
- `bloc de commandes` est un ensemble de lignes de code.

Lors de l'exécution, `condition` est évalué, puis :

- si `condition` a pour valeur **`False`**, les instructions de `bloc de commandes` ne sont pas exécutées, et on passe à la suite du programme (on dit qu'on sort de la boucle **`while`**),
- si `condition` a pour valeur **`True`**, les instructions de `bloc de commandes` sont exécutées, puis **on retourne évaluer le booléen `condition`**, et ainsi de suite jusqu'à ce que la condition devienne fausse pour la première fois, auquel cas l'ordinateur sort immédiatement de l'instruction **`while`**.

On peut préciser quelques points importants :

- les instructions de `bloc de commandes` seront donc généralement exécutées plusieurs fois : c'est la différence fondamentale avec une commande **`if`** sans partie **`else`**,
- pour que le programme ne boucle pas indéfiniment, il faut que le bloc de commandes influe sur l'expression formant la condition pour finir par la rendre fausse (sinon on a une boucle infinie, qui est une erreur fréquemment rencontrée),
- la répétition d'instructions permet de réaliser une boucle, qui comme nous le verrons possède d'autres possibilités que celles d'une boucle **`for`**.

### 2.2. Premier exemple : calcul de somme

Considérons la fonction `somme(n)` qui permet de calculer la somme des entiers strictement inférieurs à l'entier `n` :

```
def somme(n:int)->int:
    S = 0
    for k in range(n):
        S = S+k
    return S
```

Il est possible de programmer cette fonction à l'aide d'une boucle **`while`** :

```
def somme(n:int)->int:
    S = 0
    k = 0
    while k < n:
        S = S+k
        k = k+1
    return S
```

Il y a deux modifications relatives à la variable `k` qui parcours les entiers qui apparaissent dans cette deuxième fonction :

- il est nécessaire d'initialiser la variable `k` (instruction `k = 0`),
- il est nécessaire d'incrémenter la variable `k` (instruction `k = k+1`). En l'absence de cette dernière instruction, on aurait une boucle infinie.

> **Remarque 3** Il est toujours possible (mais pas toujours souhaitable) de réécrire une boucle **`for`** à l'aide d'une boucle **`while`**. Dans l'exemple de la fonction `somme` précédente, il n'y a aucun intérêt à utiliser une boucle **`while`** plutôt qu'une boucle **`for`**, et l'emploi de la boucle **`for`** est même à privilégier, car son écriture est plus simple.

### 2.3. Deuxième exemple : suite de Syracuse

Certaines répétitions ne peuvent s'exprimer qu'avec une boucle while, et non par une boucle for. C'est dans ces cas que l'instruction **`while`** prend tout son intérêt. Considérons par exemple la suite de Syracuse définie par :

$$u_0 = p, p \in \mathbb{N}$$
$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

Une conjecture célèbre et non démontrée à ce jour est qu'une telle suite finit toujours par faire apparaître le nombre 1 (puis se poursuivre par une répétition de la séquence $4, 2, 1$) quelque soit l'entier naturel non nul $p$ choisi pour initialiser la suite. La fonction `indice_syracuse` ci-dessous renvoie, pour $p$ donné, le premier indice $k$ tel que $u_k = 1$.

```
def indice_syracuse(p:int)->int:
    u = p
    k = 0
    while u != 1:
```

```
        k = k+1
        if u%2 == 1:
            u = 3*u+1
        else:
            u = u // 2
    return k
```

On peut noter que :

- Cette fonction ne peut pas être programmée à l'aide d'une boucle **for**,
- la boucle termine pour tout entier $p$ uniquement si la conjecture de SYRACUSE est vérifiée,
- l'emploi de la division entière `//` a pour unique rôle de maintenir le type `int` de la variable u.

## 2.4. for ou while ?

Une question que l'on doit forcément se poser lorsqu'on est amené à créer une boucle est la suivante : *doit-on utiliser une boucle* **for** *ou bien une boucle* **while** ?

Pour répondre à cette question, il faut se demander si, pour une entrée connue, mais quelconque, on peut connaitre avant d'exécuter la boucle le nombre d'itérations qu'elle comportera. Si oui, alors, il vaut mieux utiliser une boucle **for** (on peut utiliser une boucle **while**, mais c'est souvent un peu plus difficile à programmer - et à lire), sinon, on doit utiliser une boucle **for**. Par exemple, pour la suite de SYRACUSE précédente :

- Si l'on veut calculer $u_{20}$ à partir d'un entier $u_0 = p$ quelconque, alors on sait qu'on devra faire 20 itérations (une première pour calculer $u_1$, une deuxième pour $u_2$ et ainsi de suite jusqu'à $u_{20}$). Il vaut mieux dans ce cas là utiliser une boucle **for**.
- Si l'on souhaite comme dans l'exemple précédent déterminer le plus petit indice $k$ tel que $u_k$ soit égal à 1, à partir d'un entier $u_0 = p$ quelconque, alors on ne sait pas combien d'itérations seront nécessaires.

**Exercice 3** [Sol 3] On désire écrire une fonction `nb_div2(n)` qui renvoie le plus grand entier $k$ tel que $2^k$ divise $n$. Par exemple, pour $n = 24$, la fonction devra renvoyer 3 car 24 est divisible par $2^3 = 8$, mais pas par $2^4 = 16$.

**1)** Faut-il utiliser une boucle **for** ou une boucle **while** pour écrire cette fonction ?

**2)** Écrire la fonction demandée.

## SOLUTIONS DES EXERCICES

**Solution 1**

**1)** `mult_3(4)` renvoie `(1,3)` et `mult_3(10)` renvoie `(3,18)`.

**2)** Plus généralement, `mult_3(n)` renvoie le tuple `(N,S)` où `N` est le nombre d'entiers $k$ multiples de 3 vérifiant $1 \leq k < n$ et `S` est la somme de ces mêmes entiers.

**Solution 2** Pour la fonction `div`, la condition relative au premier **`if`** est vrai, donc on affecte 2 à `d1`. On examine ensuite le deuxième **`if`**, dont la condition est aussi vraie, et on affecte 3 à `d2`. On a donc pour retour `(2,3)`. Pour la fonction `div_bis`, la condition relative au premier **`if`** est vrai, donc on affecte 2 à `d1`, mais on n'exécute pas le bloc relatif au **`elif`**. la variable `d2` reste inchangée et la fonction renvoie `(2,None)`.

**Solution 3**

**1)** Pour réaliser la fonction, il faut effectuer des divisions par 2 successives. Pour $n$ quelconque, on ne connait pas le nombre de divisions nécessaires avant d'avoir exécuté l'algorithme. Il faut donc utiliser une boucle **`while`**.

**2)** Fonction `nb_div2(n)`

```
def nb_div2(n:int)->int:
    k = 0
    while n%2 == 0:
        n = n//2
        k = k+1
    return k
```