



Chapitre : les structures de controle

- *Author* : Ibrahima SY
- *Email* : syibrahima31@gmail.com
- *Github* : [Cliquez](#)
- *Linkdin* : [Cliquez](#)
- *School* : Institut Supérieur Informatique (ISI)
- *Spécialité* : Licence 2 **GL**

PLAN

1. Sommaire
2. Conditionnelles
3. Répétition
4. Introduction aux liste de compréhension

Sommaire

- Les structures de contrôles sont des regroupements de lignes délimités par un niveau d'indentation et dont le contenu est exécuté en fonction d'une ou plusieurs conditions.
- On dénombre trois structures conditionnelles en Python qui permettent d'organiser le code, définies par les instructions :

```
if  
while  
for
```

Chacune de ces structures est de la forme :

```
instruction condition:  
    <bloc de lignes>  
else:  
    <bloc de lignes>
```

Conditionnelles

L'instruction if else

Dans cette section, nous allons parler de l'instruction `if else` qui permet de faire de l'exécution conditionnelle. C'est à-dire qu'un morceau de code va s'exécuter en fonction du fait qu'un test soit ou faux

```
note = 8

if note > 10 :
    print('reçu')
    print('bravo')
else:
    print('recalé')
```

- `if` est l'instruction `note>10` est l'expression `print('reçu')` et `print('bravo')` forment le bloc d'instructions

- Le `:` est systématique avant un bloc d'instruction. Un bloc d'instructions est un ensemble d'instructions qui sont toutes indentées du même nombre de caractères vers la droite.
- La convention étant d'indenter tous les blocs d'instructions de 4 caractères vers la droite. Si le test if est vrai, Python exécute les instructions du bloc d'instructions.

Conditionnelles

L'instruction if / elif / else et opérateurs booléens

Le bloc d'instruction ne sera exécuté que si le test est vrai. Ensuite, on peut On peut ajouter autant de clause `elif` (contraction de `else if`) que de conditions à tester si les précédentes ont renvoyé `False` .

```
if test1:
    <bloc instructions 1>
elif test2:
    <bloc intructions 2>
elif test3:
    <bloc instructions 3>
-----

else:
    <bloc intructions n>
```

L'intérêt de ces `elif` c'est de faire des tests supplémentaires. Le fonctionnement de cette suite de tests est simple. Il fonctionne comme suit si le test1 est `True` on exécute le bloc d'instruction correspondant. Si `False` on passe à `test2`, si `False` on passe à `test3`. Et si aucun des tests n'est `True`, on finit dans `else` et on exécute le bloc d'instruction de `else`.

Conditionnelles

Ce que peut contenir un test

Dans un test d'un if ou d'un elif, on peut avoir n'importe quelle expression. Le test va appeler la fonction built-in bool sur le résultat de l'évaluation de l'expression. Donc on a une expression qui va être exécutée. Elle va produire un objet et on va appeler bool sur cet objet.

`bool(objet)` va appeler :

- soit la méthode `objet.__bool__()` qui est une méthode spéciale sur laquelle nous reviendrons dans les leçons sur les classes. Cette méthode `.__bool__()` va retourner True ou False.

- soit la méthode `objet.__len__()`. Si la méthode `.__len__()` retourne `0` ce sera `False`, si la méthode `.__len__()` retourne quelque chose d'autre, ce sera `True`.
L'intuition derrière ça c'est qu'un objet vide est considéré comme `False`. Un objet qui n'est pas vide est considéré comme `True`.

Conditionnelles

Exemples d'expressions

Un type built-in :

- sera considéré comme `False` s'il est `False`, `0`, `None` ou n'importe quel type `liste`, `tuple`, `dictionnaire`, `chaîne de caractères` vides.
- tout le reste est `True`.

```
L = ["marc", 10]
```

```
if L:  
    print(L)
```

Conditionnelles

Une comparaison

On peut aussi mettre des comparaisons : supérieur `>` , supérieur ou égal `>=` , inférieur `<` , inférieur ou égal `<=` , égal `==` ou différent `!=` .

```
a=1; b=2  
  
if a!=b:  
    print("faux")
```

Le test d'appartenance : Le test d'appartenance `in`

```
L = ["marc", 10]  
  
if "marc" in L :  
    print("ok")
```

Conditionnelles

les opérateurs de test booléen

Opérateur unaire not

a	not(a)
False	True
True	False

Opérateurs binaires or et and

a	b	a or b	a and b
False	False	False	False
False	True	True	False
True	False	True	False
True	True	True	True

```
s = "123"
```

```
if "1" in s and s.isdecimal():  
    print(int(s) + 10 )
```

les opérateurs de test booléen

Operator	Meaning
== (double equal to)	Equal to
<	Less than
>	Greater than
!=	Not equal to
<=	Less than or equal to
>=	Greater than or equal to

Répétitions

En programmation, on est souvent amené à répéter plusieurs fois une instruction. Incontournables à tout langage de programmation, les boucles vont nous aider à réaliser cette tâche de manière compacte et efficace.

L' instruction **for in**

L'instruction for permet d'exécuter un bloc de lignes en fonction d'une séquence. Elle est de la forme :

```
for variable in sequence:  
    <bloc de lignes>  
  
else:  
    <bloc de lignes>
```

Si sequence possède `n` éléments, le bloc sera exécuté `n` fois, et variable référencera l'élément `sequence[n-1]` qui sera accessible dans le bloc. Lorsque l'exécution est achevée, un bloc de lignes optionnel présenté par `else` est à son tour exécuté.

```
for in "Bonjour":  
    print(i)  
  
else:  
    print("un bloc de code optionel")
```

Répétitions

Utilisation de `continue` et `break`

`continue` : interrompt l'exécution de la boucle pour l'élément de la boucle pour l'élément en cours et passe à l'élément suivant

```
for i in range(5):  
    if i % 2:  
        continue  
    print(i)  
else:  
    print(i)
```

`break` : interrompt définitivement l'exécution de la boucle et n'exécute pas l'instruction `else` . Cette instruction est utile lorsqu'on cherche à appliquer un traitement à un seul élément d'une liste , ou que cet élément est une condition de sortie


```
for i in range(5):  
    if i==4:  
        print("4 a ete trouve")  
        break  
    print("On continue")
```

Lorsque l'exécution est terminée, le dernier élément de la séquence reste toujours accessible par la variable de boucle

Répétitions

L'instruction `while`

L'instruction `while` permet d'exécuter un bloc de lignes tant qu'une expression est vérifiée en renvoyant `True`. Lorsque l'expression n'est plus vraie, l'instruction `else` est exécutée si elle existe et la boucle s'arrête. `continue` et `break` peuvent être utilisés de la même manière que pour l'instruction `for`.

```
i = 0

while i<4:
    print(str(i))
    i +=1
else:
    print("end")
```

```
i = 0
```

```
while i<5:  
    i +=1  
    if i == 2:  
        continue  
    print(str(i))
```

Introduction aux liste de compréhension

- Les compréhension de liste qui permet de manière extrêmement simple et intuitive d'appliquer une opération à chaque élément d'une liste et éventuellement d'ajouter une condition de filtre

Supposons que l'on souhaite prendre les logarithmes d'une liste d'entiers `a = [1, 4, 18, 29, 13]` .

Premier Technique : utilisation d'une boucle

```
import math

L = [1, 4, 18, 29, 13]

liste = []

for i in L:
    liste.append(math.log(i))
```

Deuxième Technique : Utilisation d'une liste de compréhension

```
[math.log(i) for i in L]
```