


Cours	Cours	
BTS SNIR	Python	
2 ^{ème} année	Les conditions	

Dans cette nouvelle partie, nous allons étudier et comprendre l'intérêt des structures de contrôle en Python. Une structure de contrôle est un ensemble d'instructions qui permet de contrôler l'exécution du code.

Il existe différents types de structures de contrôle. Les deux types de structures les plus communément utilisées sont les structures de contrôle conditionnelles qui permettent d'exécuter un bloc de code si une certaine condition est vérifiée et les structures de contrôle de boucle qui permettent d'exécuter un bloc de code en boucle tant qu'une condition est vérifiée.

Présentation des conditions Python

Les structures de contrôle conditionnelles (ou plus simplement conditions) vont nous permettre d'exécuter différents blocs de code selon qu'une condition spécifique soit vérifiée ou pas.

Nous allons très souvent utiliser les conditions avec des variables : selon la valeur stockée dans une variable, nous allons pouvoir exécuter un bloc de code plutôt qu'un autre.

Python nous fournit les structures conditionnelles suivantes :

- La condition `if` ("si") ;
- La condition `if...else` ("si...sinon") ;
- La condition `if...elif...else` ("si...sinon si... sinon") .

Nous allons étudier et comprendre l'intérêt de chacune de ces conditions dans la suite de cette leçon. Avant de les étudier, nous allons devoir présenter un nouveau type d'opérateurs : les opérateurs de comparaison qui vont être au centre de nos conditions.

Les opérateurs de comparaison

Comme je l'ai précisé plus haut, nous allons souvent construire nos conditions autour de variables : selon la valeur d'une variable, nous allons exécuter tel bloc de code ou pas.

Pour pouvoir faire cela, nous allons comparer la valeur d'une variable à une certaine autre valeur donnée et selon le résultat de la comparaison exécuter un bloc de code ou pas. Pour comparer des valeurs, nous allons devoir utiliser des opérateurs de comparaison.

Voici ci-dessous les différents opérateurs de comparaison disponibles en Python ainsi que leur signification :

Opérateur Définition

<code>==</code>	Permet de tester l'égalité en valeur et en type
<code>!=</code>	Permet de tester la différence en valeur ou en type
<code><</code>	Permet de tester si une valeur est strictement inférieure à une autre
<code>></code>	Permet de tester si une valeur est strictement supérieure à une autre
<code><=</code>	Permet de tester si une valeur est inférieure ou égale à une autre
<code>>=</code>	Permet de tester si une valeur est supérieure ou égale à une autre

Notez bien ici que ces opérateurs ne servent pas à indiquer à Python que telle valeur est supérieure, égale, inférieur ou différente à telle autre valeur. Lorsqu'on utilise un opérateur de comparaison, on demande au contraire à Python de tester si telle valeur est supérieure, égale, inférieur ou différente à telle autre valeur. Python va donc comparer les deux valeurs et toujours renvoyer un booléen : `True` si la comparaison est vérifiée ou `False` dans le cas contraire.

Notez également que les opérateurs de comparaison d'égalité et de différence testent l'égalité et la différence à la fois sur les valeurs et sur les types. Ainsi, si on demande à Python de tester l'égalité entre la chaîne de caractères "4" et le chiffre 4, celui-ci renverra `False` puisque pour lui ces deux valeurs ne sont pas égales.

Regardez plutôt les exemples suivants pour vous en persuader :

```
>>> 4 < 8
True
>>> 4 > 8
False
>>> 4 == 4
True
>>> 4 == "4"
False
>>> 5 != 10
True
```

Vous pouvez retenir ici que c'est cette valeur booléenne renvoyée par le Python à l'issue de toute comparaison que nous allons utiliser pour faire fonctionner nos conditions.

La condition if en Python

La structure conditionnelle `if` est une structure de base qu'on retrouve dans de nombreux langages de script. Cette condition va nous permettre d'exécuter un code si (et seulement si) une certaine condition est vérifiée.

On va en fait passer une expression à cette condition qui va être évaluée par Python. Cette expression sera souvent une comparaison explicite (une comparaison utilisant les opérateurs de comparaison) mais pas nécessairement.

Si Python évalue l'expression passée à `True`, le code dans la condition `if` sera exécuté. Dans le cas contraire, le code dans `if` sera ignoré.

Prenons immédiatement un premier exemple afin de nous familiariser avec le fonctionnement et la syntaxe de cette condition :

```
>>> x, y = 8, 4
>>> if x > y:
...     print("x contient une valeur strictement supérieure à y")
...
x contient une valeur strictement supérieure à y
>>>
>>> if x == 5:
...     print("x contient l'entier 5")
...
>>>
```

Nous créons ici deux conditions `if`. Comme vous pouvez le voir, la syntaxe générale d'une condition `if` est `if condition : code à exécuter`. Pensez bien à indiquer le `:` et à bien indenter le code qui doit être exécuté si la condition est vérifiée sinon votre condition ne fonctionnera pas.

Dans le premier `if`, nous demandons à Python dévaluer la comparaison `x > y`. Comme notre variable `x` stocke 8 et que notre variable `y` stocke 4, Python valide cette comparaison et renvoie `True`. La condition `if` reçoit `True` et le code qu'elle contient est exécuté.

Dans notre deuxième `if`, on demande cette fois-ci à Python de nous dire si le contenu de `x` est égal au chiffre 5. Ce n'est pas le cas et donc Python renvoie `False` et le code dans ce `if` n'est donc pas exécuté.

Au final, vous pouvez retenir que toute expression qui suit un `if` va être évaluée par Python et que Python renverra toujours soit `True`, soit `False`. Nous n'avons donc pas nécessairement besoin d'une comparaison explicite pour faire fonctionner un `if`.

Pour comprendre cela vous devez savoir qu'en dehors des comparaisons Python évaluera à `True` toute valeur passée après `if` à l'exception des valeurs suivantes qui seront évaluées à `False` :

- La valeur 0 (et 0.0) ;
- La valeur `None` ;
- Les valeurs chaîne de caractères vide `""`, liste vide `[]`, dictionnaire vide `{}` et tuple vide `()`.

```
>>> x, y, z = 0, 1, ""
>>> if x:
...     print("x a été évaluée à True")
...
>>> if y:
...     print("y a été évaluée à True")
...
y a été évaluée à True
>>> if z:
...     print("z a été évaluée à True")
...
>>>
```

La condition `if... else` en Python

Avec la condition `if`, nous restons relativement limités puisque cette condition nous permet seulement d'exécuter un bloc de code si le résultat d'un test soit évalué à `True`.

La structure conditionnelle `if...else` (« si... sinon » en français) est plus complète que la condition `if` puisqu'elle nous permet d'exécuter un premier bloc de code si un test renvoie `True` ou un autre bloc de code dans le cas contraire.

La syntaxe d'une condition `if...else` va être la suivante :

```
>>> x, y = 5, 10
>>> if x == 5:
...     print("x contient l'entier 5")
... else:
...     print("x ne contient pas 5")
...
x contient l'entier 5
>>>
>>> if y == 5:
...     print("y contient l'entier 5")
... else:
...     print("y ne contient pas 5")
...
y ne contient pas 5
```

Ici, on demande dans notre première condition à Python d'évaluer si la valeur de `x` est différente du chiffre 5 ou pas. Si c'est le cas, Python renverra `True` (puisque'on lui demande ici de tester la différence et non pas l'égalité) et le code du `if` sera exécuté. Dans le cas contraire, c'est le code du `else` qui sera exécuté.

Notre deuxième condition fait exactement le même travail mais cette fois si on compare la valeur de `y` à 5.

Notez bien ici qu'on n'effectuera jamais de test dans un `else` car le `else` est par définition censé prendre en charge tous les cas non pris en charge par le `if`.

La condition `if... elif... else` en Python

La condition `if...elif...else` (« si...sinon si...sinon ») est une structure conditionnelle encore plus complète que la condition `if...else` qui va nous permettre cette fois-ci d'effectuer autant de tests que l'on souhaite et ainsi de prendre en compte le nombre de cas souhaité.

En effet, nous allons pouvoir ajouter autant de `elif` que l'on souhaite entre le `if` de départ et le `else` de fin et chaque `elif` va pouvoir posséder son propre test ce qui va nous permettre d'apporter des réponses très précises à différentes situations.

```

>>> x = 5
>>> if x < 0:
...     print("x contient une valeur négative")
... elif x < 10:
...     print("x contient une valeur comprise entre 0 et 10 exclu")
... elif x < 100:
...     print("x contient une valeur comprise entre 10 et 100 exclu")
... else:
...     print("x contient une valeur supérieure à 100")
...
x contient une valeur comprise entre 0 et 10 exclu

```

Il faut cependant faire attention à un point en particulier lorsqu'on utilise une structure Python `if... elif... else` : le cas où plusieurs `elif` possèdent un test évalué à `True` par Python. Dans ce cas là, vous devez savoir que seul le code du premier `elif` (ou du `if` si celui-ci est évalué à `True`) va être exécuté. En effet, Python sort de la structure conditionnelle dans son ensemble sans même lire ni tester la fin de celle-ci dès qu'un cas de réussite a été rencontré et que son code a été exécuté.

Imbriquer des conditions

Souvent, nous allons vouloir comparer plusieurs valeurs au sein d'une même condition, c'est-à-dire n'exécuter son code que si plusieurs conditions sont vérifiées.

Pour faire cela, nous allons pouvoir soit utiliser plusieurs opérateurs de comparaison, soit les opérateurs logiques, soit imbriquer plusieurs conditions les unes dans les autres.

Les opérateurs logiques vont nous permettre de créer des conditions plus puissantes mais dans certains cas il sera plus intéressant et plus rapide d'imbriquer des conditions.

```

>>> x, y = 1, 2
>>> if x < 5:
...     if y < 10:
...         print("x contient une valeur inférieure à 5 et y une valeur inférieure à 10")
...     else:
...         print("x contient une valeur < 5 et y une valeur >= 10")
... else:
...     print("x contient une valeur supérieure à 5")
...
x contient une valeur inférieure à 5 et y une valeur inférieure à 10

```

Dans cet exemple, on imbrique deux structures `if...else` l'une dans l'autre. La première structure demande à Python de tester si notre variable `x` contient un nombre strictement inférieur à 5. Si c'est le cas, on rentre dans le `if` et on teste donc la condition du deuxième `if`. Dans le cas contraire, on va directement au `else` de fin.

Notre deuxième condition teste si `y` contient une valeur strictement inférieure à 10. Si c'est le cas, on exécute le code dans la condition. Sinon, on va directement au `else` de cette condition imbriquée.

Utiliser les opérateurs logiques avec les conditions

Les opérateurs logiques vont être principalement utilisés avec les conditions puisqu'ils vont nous permettre d'écrire plusieurs comparaisons au sein d'une même condition ou encore d'inverser la valeur logique d'un test.

Opérateur Définition

and Renvoie `True` si toutes les deux expressions sont évaluées à `True`

or Renvoie `True` si une des comparaisons vaut `True`

not Renvoie `True` si la comparaison vaut `False` (et inversement)

Les opérateurs logiques `and` et `or` vont nous permettre de passer plusieurs tests pour évaluation à Python. On va par exemple pour tester si une variable `x` contient une valeur inférieure à 5 et / ou si `y` contient une valeur inférieure à 10 au sein d'une même condition.

Dans le cas où on utilise `and`, chaque expression devra être évaluée à `True` par Python pour que le code dans la condition soit exécuté.

```
>>> x, y = 1, 5
>>> if x < 5 and y < 10:
...     print("x contient une valeur inférieure à 5 et y une valeur inférieure à 10")
...
x contient une valeur inférieure à 5 et y une valeur inférieure à 10
```

Dans le cas où on utilise `or`, il suffit qu'une expression soit évaluée à `True` par Python pour que le code dans la condition soit exécuté.

```
>>> x, y = 1, 5
>>> if x < 5 or y < 5:
...     print("x ou y (ou les deux) stocke(nt) une valeur stric. inférieure à 5")
...
x ou y (ou les deux) stocke(nt) une valeur stric. inférieure à 5
```

Ici, vous pouvez noter que Python a l'inverse de la plupart des autres langages possède une syntaxe très logique et très intuitive qui va nous permettre d'effectuer plusieurs tests dans une condition "comme si" on utilisait un opérateur logique `and` en utilisant tout simplement plusieurs opérateurs de comparaison à la suite.

Je vous recommande cependant plutôt d'utiliser des opérateurs logiques dans cette situation afin de rendre votre code plus clair.

Finalement, l'opérateur logique `not` est très particulier puisqu'il nous permet d'inverser la valeur logique d'un test : si Python renvoie `False` à l'issue d'une évaluation par exemple et qu'on utilise l'opérateur `not` sur cette expression l'opérateur inversera la valeur renvoyée par Python et la valeur finale passée à la condition sera `True`.

```
>>> x, y = 1, 5
>>> if not x >= 5:
...     print("x contient une valeur stric. inférieure à 5")
... else:
...     print("x contient une valeur supérieure ou égale à 5")
...
x contient une valeur stric. inférieure à 5
```

Présentation des opérateurs d'appartenance ou d'adhésion

Python met à notre disposition deux opérateurs d'appartenance qui vont nous permettre de tester si une certaine séquence de caractères ou de valeurs est présente ou pas dans une valeur d'origine.

Ces opérateurs ne vont fonctionner qu'avec des séquences (chaines de caractères, listes, etc.) et ne vont donc pas marcher avec des valeurs de type numérique par exemple.

L'opérateur `in` permet de tester si une certaine séquence de caractères ou de valeurs est présente dans une valeur d'origine et renvoie `True` si c'est le cas.

```
>>> prenom = ["Pierre", "Mathilde", "Florian", "Thomas"]
>>> if "Pierre" in prenom:
...     print("Pierre est dans la liste")
...
Pierre est dans la liste
```

L'opérateur `not in` permet au contraire de tester si une certaine séquence de caractères ou de valeurs n'est pas présente dans une valeur d'origine et renvoie `True` si c'est le cas.

```
>>> ages = [29, 27, 30, 29]
>>> if 31 not in ages:
...     print("Personne n'a 31 ans")
...
Personne n'a 31 ans
```

Comme des valeurs booléennes sont renvoyées, on va tout à fait pouvoir utiliser ce type d'opérateurs au sein de nos conditions même s'ils sont communément plus utilisés au sein de boucles qu'on étudiera dans la prochaine leçon.