
COURS - INTRODUCTION AU LANGAGE PYTHON

Vous avez dit “programme” ?

Un **programme** est la description d'un **algorithme** dans un **langage** compréhensible par un **humain** et par une **machine**, qui l'exécute afin de traiter des **données**.

Il existe de nombreux **langages de programmation**, dont certains sont plus proches du **langage naturel** (on parle de langages de *haut niveau*), tandis que d'autres sont plus proches de celui de la **machine** (on parle de langages de *bas niveau*), on rappelle que la machine ne comprend que le **binaire**, c'est-à-dire une suite de **bits** 0 ou 1 (nous reviendrons sur cela dans le cadre d'un autre chapitre).

On peut citer, parmi les langages :

- de **haut niveau** : **Python**, *JavaScript*, *Java*, *C#*,
- de **bas niveau** : *C*, *Assembleur*, *langage machine* (le plus bas niveau possible).

Le langage de programmation **Python**, que l'on utilisera dans le cadre de cet enseignement, est déjà utilisé au lycée en **mathématiques**, et est également présent dans d'autres disciplines et dans le nouvel enseignement de **sciences numériques et Technologie** (SNT) en seconde.

Au-delà du cadre de l'enseignement, c'est un langage **extrêmement populaire**, car l'un des plus **versatiles** et **généralistes**. Il est utilisé aussi bien par des développeurs débutants que par des développeurs d'applications web et mobile, des ingénieurs logiciels, des Data Scientists...

Le langage Python a été créé par un **ingénieur informaticien néerlandais** du nom de **Guido Van Rossum**.



La première version publique date de *1991*. Van Rossum a ensuite poursuivi son travail sur le projet et a travaillé entre autres pour Google puis Dropbox. La version de Python que nous utiliserons est la **version 3**, disponible depuis *2008* avec des mises à jour régulières. La dernière version en date lors de l'écriture de cet article est la version **3.11.5**.

Le langage Python est **multiplateforme**, vous pouvez donc l'installer sur les systèmes d'exploitation **Linux**, **MacOs**, **Windows**, et même sur des smartphones dotés du système **Android** ou d'**iOS**. Il est **gratuit** et placé sous **licence libre**, la *Python Software Foundation License* (PSFL) .

Les **constructions élémentaires** propres au langage Python sont communes à de nombreux autres langages de programmation.

Un programme est ainsi composé :

- de **séquences**, (des instructions exécutées l'une après l'autre dans l'ordre où elles sont écrites),
- de définitions de **variables** et de **fonctions**,
- d'**affectations**,
- d'**instructions conditionnelles**,
- de **boucles** (bornées et non-bornées),
- d'**expressions** et d'**appels de fonctions**.

Expressions arithmétiques, variables et instructions

Le langage de programmation **Python** permet d'interagir avec la **machine** à l'aide d'un **programme** appelé **interpréteur Python**. On peut l'utiliser de deux manières différentes :

- en **mode interactif**, qui consiste à **dialoguer directement** avec l'**interpréteur**,
- en **mode programme**, qui consiste à **écrire un programme dans un fichier** et à le **faire exécuter par l'interpréteur**.

Mode interactif

Le mode interactif s'apparente à une **calculatrice**.

Les **trois chevrons >>>** indiquent que l'interpréteur **attend vos instructions**.

Par exemple, si vous saisissez `1 + 2` après les **chevrons** et que vous appuyez sur la touche **Entrée**, l'**interpréteur** effectuera le calcul et affichera le résultat :

```
1 >>> 1 + 2
2 3
```

Les types de base

Le langage Python intègre initialement plusieurs **types de données**, que nous allons régulièrement utiliser.

Type	Explication
<code>int</code>	Permet de définir un nombre entier relatif , par exemple 0, 5, -5, 13402...
<code>float</code>	Permet de définir un nombre à virgule flottante (ou <i>nombre flottant</i>), cela permet de représenter les nombres réels , comme par exemple 12.05, 6.33333, 42.0...
<code>str</code>	Permet de définir une chaîne de caractères , ce qui permet de représenter des caractères (par exemple 'L', 'e', '!', '5', ' '), des mots (par exemple "bonjour", "Salut") ou encore des phrases (par exemple "Salut tout le monde !"). Les chaînes de caractères sont reconnaissables car elles commencent et finissent par un apostrophe ou un guillemet .
<code>bool</code>	Permet de définir un booléen . Il n'y a que deux valeurs booléennes possibles : <code>True</code> (vrai) et <code>False</code> (faux).

Arithmétique

En Python, on peut saisir des **combinaisons d'opérations arithmétiques**.

Par exemple :

```
1 >>> 2 + 5 * (10 - 1 / 2)
2 49.5
```

À noter que les espaces sont **purement décoratifs**, on aurait pu écrire :

```
1 >>> 2+6*(10-1/2)
2 49.5
```

Erreurs de syntaxe

L'interpréteur n'accepte que les **expressions arithmétiques bien formées**. Autrement, une **SyntaxError** indiquant une **erreur de syntaxe** sera levée, par exemple :

```
1 >>> 1 + * 2
2     File "<stdin>", line 1
3     1 + * 2
4         ^
5 SyntaxError: invalid syntax
```

<stdin> signifie *standard input* (entrée standard), on reviendra sur les notions d'entrée standard et de sortie standard plus loin dans ce cours lorsque l'on présentera les fonctions `input` et `print`.

Erreurs : Divisions par zéro

Un autre type d'erreur qui peut être levée par l'interpréteur Python est une **zeroDivisionError**, indiquant que l'expression contient **une division par zéro**. Par exemple :

```
1 >>> 2 / (3 - 3)
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   ZeroDivisionError: division by zero
```

Différentes manières de diviser

Lorsque l'on utilise l'opérateur de division classique de Python `/`, on obtient un **nombre flottant** (type `float`) :

```
1 >>> 7 / 2
2 3.5
3 >>> type(7 / 2)
4 <class 'float'>
```

Ici, on a utilisé une **fonction** appelée `type` afin de voir de **quel type** est l'objet qui résulte de l'opération `7 / 2`. (On a ici la confirmation que l'on obtient bien un **flottant**.)

Il existe **deux autres opérateurs** bien utiles :

- `//` permettant d'obtenir le **quotient** de la **division euclidienne** de deux opérandes,
- `%` permettant d'obtenir le **reste** de la **division euclidienne** de deux opérandes.

Par exemple :

```
1 >>> 7 // 2
2 3
3 >>> 7 % 2
4 1
5 >>> type(7 // 2)
6 <class 'int'>
7 >>> type(7 % 2)
8 <class 'int'>
```

Ces deux opérateurs renvoient donc **des entiers**.

On rappelle que pour deux nombres entiers a et b , $a = q \times b + r$, avec $q = a // b$ le **quotient** et $r = a \% b$ le **reste**.

Variables

Saisir des expressions arithmétiques, c'est bien. Mais une calculatrice sait déjà le faire !

Nous allons maintenant introduire la notion de **variable**. Une **variable** permet de **stocker une donnée** utilisée par un **programme**.

Cela se fait par une **affectation** qui associe une **donnée**, représentée par une **valeur** ou une **expression**, avec un **nom**. Une **expression** stockée dans une variable peut elle-même contenir d'**autres variables**.

Une **variable** peut s'apparenter à une "boîte" sur laquelle est écrit un **nom** et dans laquelle on place des **informations** diverses (même si dans la réalité, ce n'est pas tout à fait ça). Un **nom** peut être n'importe quelle **chaîne alphanumérique**, à l'exception de certains mots réservés, et ne doit pas commencer par un chiffre.

L'opérateur d'affectation est noté `=`.

Par exemple, l'instruction `x = 4` associe la valeur 4 au **nom** `x` :

```
1 >>> x = 4
2 >>>
```

Si l'on saisit une **instruction d'affectation** dans l'**interpréteur Python**, aucun résultat n'est affiché. Si l'on souhaite accéder à la valeur **mémorisée** dans `a`, il suffit de saisir :

```
1 >>> a
2 4
```

L'instruction `y = 3 + 5` associe la valeur de l'expression située à droite du signe `=`, ici **8**, au nom `y`.

L'instruction `z = x + y` associe la valeur de l'expression située à droite du signe `=`, ici **12** (la somme des valeurs contenues dans les variables `x` et `y`), au nom `z`.

Python permet par ailleurs d'effectuer des **affectations multiples**, par exemple `x, y, z = 1, 3, 5`, qui associe les valeurs **1**, **3** et **5** respectivement aux noms `x`, `y` et `z`. Ceci est équivalent à écrire `x=1` ; `y=3` ; `z=5` sur une ligne ou à effectuer les 3 affectations sur 3 lignes successives.

À retenir !

- Une **variable** est composée d'un **nom** (ou identificateur), d'une **adresse en mémoire** où est enregistrée une **valeur** (ou un **ensemble de valeurs**), et d'un **type** qui **définit ses propriétés**.
- Une **expression** a une **valeur** qui est le **résultat** d'une **combinaison de variables ou d'objets**, de **constantes** et d'**opérateurs**.
- Une **instruction** est une commande qui doit être **exécutée** par la machine.
- Une **affectation** est une **instruction** qui commande à la machine de créer une **variable** en lui précisant son **nom** et la **valeur** qui lui est associée.

Il est important de bien distinguer une **expression**, qui se **calcule** et a une **valeur**, d'une **instruction**, qui est **exécutée** par la machine.

Mode programme

Le **mode programme** de **Python** consiste à écrire une **suite d'instructions** dans un **fichier** et à les faire **exécuter** par l'**interpréteur Python**. Cette suite d'instructions s'appelle un **programme** (ou **code source**).

Le **mode programme** permet :

- de pouvoir **ré-exécuter** une **suite d'instructions** sans avoir à les ressaisir une par une dans l'**interpréteur Python**.
- de faire la **distinction** entre le rôle de **programmeur** et celui d'**utilisateur** d'un programme. Un programme Python peut facilement être exécuté par une personne sans qu'elle n'ait besoin d'en visualiser le code.

Affichage sur la sortie standard

Contrairement au **mode interactif**, en **mode programme**, les résultats des expressions calculées ne sont plus affichés à l'écran. Il faut utiliser pour ceci une instruction explicite d'affichage. En Python, elle s'appelle `print`. Par exemple :

```
1 print(3)
```

On peut également fournir à `print` une **expression**, qui sera **calculée** puis **affichée** :

```
1 4 + 2 * 3
```

`print` est également capable d'**afficher du texte**, qui doit être mis entre guillemets `"` ou apostrophes `'`, par exemple :

```
1 print("Bienvenue à tous !")
```

On peut également afficher **la valeur d'une variable**, exemple :

```
1 a = 34
2 b = 21 + a
3 print(a)    # afficher la valeur de a
4 print(b)    # afficher la valeur de b
```

À retenir !

Si l'on souhaite inclure **la valeur d'une variable** dans un **texte affiché** par `print`, on peut procéder de différentes façons :

- en donnant **plusieurs valeurs** (plusieurs *arguments*) à notre fonction `print` : dans ce cas, `print` affichera chaque **valeur** les unes à la suite des autres en les séparant par un **espace**.
- **avec la concaténation** : on peut inclure notre **variable** dans une **chaîne de caractères** en utilisant la **concaténation de chaînes de caractères**. Pour cela, on transforme notre variable de type `int` en type `str` (c'est-à-dire en *chaîne de caractères*) avec la fonction `str()`, et on effectue la concaténation en utilisant le symbole `+`.
- avec des **f-strings** (pas au programme) : si on ajoute un `f` devant notre chaîne de caractères, on peut inclure nos variables **entre accolades**, et elles seront remplacées par la valeur qu'elles contiennent.

```
1 nombre = 42
2 print("Votre nombre est " + str(nombre) + " !")    # avec une
    concaténation de chaînes de caractères
3
4 print(f"Votre nombre est {nombre} !")    # avec un f-string
5
6 print("Votre nombre est", nombre, "!")    # en donnant plusieurs
    à print
```

Cela affichera :

```
1 Votre nombre est 42 !
```

La fonction `print` effectue par défaut un **retour à la ligne** après avoir affiché les valeurs que vous lui avez donné. Pour changer ce comportement, on peut ajouter le **paramètre** `end` :

```
1 ''' Utilisation du paramètre end de print. '''
2 print("abc", end="")
3 print("ghi", end=".")
```

Cela affichera :

```
1 abcdef.
```

Un autre **paramètre** que l'on peut utiliser avec la fonction `print` est le paramètre `sep`.

Ce **paramètre** permet d'indiquer un **autre caractère** pour **séparer** plusieurs valeurs données à la fonction `print`, par exemple :

```
1 ''' Utilisation du paramètre sep de print. '''
2 # Remplaçons le séparateur par un tiret :
3 print("Bonjour", "Monsieur", "Demerville", sep="-")
```

Cela affichera :

```
1 Bonjour-Monsieur-Demerville
```

Interaction avec l'utilisateur, lire sur l'entrée standard

Pour demander à l'utilisateur de **saisir une valeur**, de manière à pouvoir la **stocker dans une variable** et en faire quelque chose, on utilise la fonction `input`.

```
1 ''' Programme qui calcule le nombre suivant celui
2 donné par un utilisateur. '''
3
4 s = input()
5 a = int(s)
6 print("le nombre suivant est ", a + 1)
```

Attention aux types !

La valeur renvoyée par la fonction `input` est de type `str` (chaîne de caractères). Si vous voulez utiliser cette valeur dans une **opération arithmétique** par exemple, il faut donc la convertir en `int` (nombre entier), d'où l'utilisation de la fonction `int` ci-dessus.

Un **programme** contenant un appel à `input` ne s'arrête que **lorsque l'utilisateur a saisi une valeur et appuyé sur la touche Entrée**. En attendant, le programme reste en *stand-by*.

On peut également indiquer dans un `input` un **message à afficher**, voici un exemple :

```
1 ''' Programme qui calcule le nombre suivant celui
2 donné par un utilisateur. '''
3
4 age = input("Indiquez votre âge : ")
5 print("Votre âge est de :", age, "ans.")
```

Cela permet d'indiquer à l'utilisateur la nature de la valeur attendue.

Les boucles bornées for

Répéter plusieurs fois les mêmes instructions est assez rébarbatif. C'est pour cela qu'il existe une instruction appelée **boucle bornée**, utilisant le mot-clé `for`, qui permet de répéter plusieurs fois un bloc d'instructions. Par exemple :

```
1 for i in range(10):
2     print("Je ne dois pas bavarder en cours.")
```

Dans la fonction `range`, on indique **le nombre de fois** que l'on souhaite **afficher l'instruction print**. Ici, on l'affiche **10 fois**.

En réalité, ce qu'il se passe, c'est que la boucle va **itérer** de la valeur `i = 0` à la valeur `i = 9` (la valeur indiquée dans le `range` moins 1), la variable `i` fournie à notre boucle étant ce l'on appelle l'**indice de boucle**. On dit que `i` **incrémente** (augmente de 1) à chaque **itération** de la boucle.

En **pseudo-langage**, on pourrait traduire ce programme de la manière suivante :

```
1 POUR i ALLANT DE 0 à 9 :  
2     AFFICHER "Je ne dois pas bavarder en cours"
```

Si l'on affiche ce que contient notre **variable** `i` à chaque fois :

```
1 for i in range(10):  
2     print("i = ", i)
```

On obtient :

```
1 i = 0  
2 i = 1  
3 i = 2  
4 i = 3  
5 i = 4  
6 i = 5  
7 i = 6  
8 i = 7  
9 i = 8  
10 i = 9
```

À retenir !

On peut utiliser le `range` de plusieurs manière différentes :

- `range(valeur)` : avec **une seule valeur** entière, la boucle va itérer de 0 à `valeur - 1`,
- `range(min, max)` : avec **deux valeurs** entières, la boucle va itérer de `min` à `max - 1`,
- `range(min, max, pas)` : avec **trois valeurs** entières, la boucle va itérer de `min` à `max - 1` avec un **pas** de `pas`. Si l'on n'indique pas ce **pas**, il est de 1 par défaut.

Par exemple, si l'on souhaite afficher tous les nombres **pairs** de 2 à 98 :

```
1 ''' Affichage des nombres pairs de 2 à 98 '''  
2  
3 for nb in range(2, 99, 2):  
4     print(nb)
```

Ici, on a appelé l'**indice de boucle** `nb`. On peut l'appeler comme on veut, mais on utilise souvent des noms à une lettre comme `i`, `j` et `k`.

On peut également passer la valeur de retour d'un `input` à l'intérieur d'un `range`, par exemple :

```
1 ''' Un compte à rebours '''  
2  
3 nb = input("Valeur initiale du compte à rebours : ")  
4 for i in range(int(nb), -1, -1):  
5     print(i)
```

Attention aux types !

Le `range` ne prend que des **entiers**. Si vous souhaitez lui passer la valeur de retour d'un `input`, il faut donc convertir cette valeur en valeur entière avec la fonction `int`.

Les boucles non bornées while

Nous avons vu qu'il était possible de créer des **boucles bornées** à l'aide d'instructions utilisant le **mot-clé** `for`. Nous allons voir ici que l'on peut également créer des boucles **non bornées** à l'aide du **mot-clé** `while` (qui signifie *TANT QUE* en français).

Une **boucle non bornée** permet d'exécuter un **bloc d'instructions** plusieurs fois, et de continuer **TANT QUE** une **condition donnée** est **vérifiée**.

La **structure** d'une boucle `while` en Python est la suivante :

```
1 while condition:
2     # Bloc d'instructions à répéter tant que la condition est vraie
```

La boucle `while` commence par **évaluer la condition** :

- Si la **condition** est **vraie**, le **bloc d'instructions** à l'intérieur de la boucle est **exécuté**.
- Après chaque exécution du bloc, la **condition** est à nouveau **évaluée** :
 - **Tant que** la **condition** reste **vraie**, la boucle **continue de s'exécuter**.
 - Dès que la **condition** devient **fausse**, la boucle **s'arrête**.

Voici un premier exemple d'utilisation d'une boucle `while`:

```
1 i = 1
2 while i <= 5:
3     print(i)
4     i += 1
```

Dans cet exemple, la boucle `while` est utilisée pour **afficher les nombres de 1 à 5**.

La variable `i` est initialisée à 1, et la boucle continue **TANT QUE** `i` est **inférieur ou égal** à 5. À **chaque itération de la boucle**, la valeur de `i` est ici **affichée** (à l'aide du `print`), puis est **incrémentée de 1** à l'aide de l'instruction `i += 1`. La boucle **s'arrête** lorsque `i` atteint 6, car la **condition devient fausse**.

Voici un autre exemple un peu plus concret de l'utilisation d'une boucle `while` :

```
1 mot_de_passe = "secret"
2 essais_restants = 3
3
4 while essais_restants > 0:
5     tentative = input("Entrez le mot de passe : ")
6
7     if tentative == mot_de_passe:
8         print("Accès autorisé !")
9         break # Sortir de la boucle
10    else:
11        essais_restants -= 1
```

```
12     print(f"Mot de passe incorrect. Il vous reste {essais_restants}  
13         essais.")  
14 if essais_restants == 0:  
15     print("Nombre maximal d'essais atteint. Accès refusé.")
```

Dans cet exemple, une boucle `while` est utilisée pour **demander à l'utilisateur** (à l'aide de la fonction `input`) de **saisir un mot de passe** jusqu'à **trois essais**.

La boucle **continue tant qu'il reste des essais** (tant que `essais_restants > 0`) :

- Si l'utilisateur saisit le **mot de passe correct** (c'est-à-dire lorsque `tentative == mot_de_passe`), la boucle **s'arrête** à l'aide de l'instruction `break`. L'instruction `break` permet de **sortir directement** de la boucle, et évite donc de ré-évaluer sa condition.
- Sinon, le **nombre d'essais restants** est **décrémenté** (cela signifie que l'on **soustrait 1** à `essais_restants`), et un message est **affiché** (à l'aide de la fonction `print`) pour informer l'utilisateur du **nombre d'essais restants**.

Une fois que l'on est **sorti de la boucle** (ce qui se produit soit si on a rencontré l'instruction `break`, soit si la **condition** de la boucle est **fausse**), on **affiche** un message d'accès refusé si le nombre maximal d'essais a été atteint.

Dans une boucle non bornée `while`, contrairement à une boucle bornée `for`, il y a un risque de créer **boucle infinie**, c'est-à-dire une boucle dont la **condition n'est jamais évaluée à False**. Par exemple :

```
1 i = 1  
2 while i < 5:  
3     print(i)
```

Ici, on n'a pas écrit d'instruction permettant d'**incrémenter** la variable `i`, donc la **condition** `i < 5` sera **toujours vraie**. Le programme ne s'arrêtera donc **jamais**.

Comparaisons, booléens, tests

Une part importante de la conception d'un programme consiste à imaginer les différents **cas de figure possibles**, notamment selon les **entrées fournies** par l'utilisateur ou les **valeurs** des différentes **variables**, de manière à adapter le programme à chacun des cas.

Pour traduire cela dans l'écriture du programme, on peut utiliser des **instructions de branchement** (`if`, `elif`, `else`) qui rassemblent **plusieurs blocs de code alternatifs**, chacun associé à une **condition logique**, et qui à chaque exécution sélectionne au plus l'un de ces blocs.

Les instructions de branchement

Les instructions `if` (*si*), `elif` (*sinon si*) et `else` (*sinon*) permettent d'exécuter des blocs de code uniquement lorsqu'une **condition** est remplie. Par exemple, voici un programme qui demande à l'utilisateur de saisir un nombre et affiche un message différent selon le nombre saisi :

```
1 nb = int(input("Saisissez votre âge : "))  
2 if nb < 5:  
3     print("Vous êtes un bébé.")
```

```
4 elif nb < 12:
5     print("Vous êtes un enfant.")
6 elif nb < 18:
7     print("Vous êtes un ado")
8 elif nb < 40:
9     print("Vous êtes un adulte")
10 elif nb < 60:
11     print("Vous êtes un vieil adulte")
12 else:
13     print("Vous êtes un vieillard.")
```

Le programme est exécuté **séquentiellement**, autrement dit, on vérifie d'abord la première condition, si elle est vraie, alors on affiche le message indiqué, sinon on passe à la deuxième condition, et ainsi de suite...

Le comportement de ce programme est donc le suivant :

- **si** `nb` est inférieur à 5 (donc compris entre 0 et 4 inclus), afficher "`Vous êtes un bébé.`",
- **sinon si** `nb` est inférieur à 12 (donc compris entre 5 et 11 inclus), afficher "`Vous êtes un enfant.`",
- **sinon si** `nb` est inférieur à 18 (donc compris entre 12 et 17 inclus), afficher "`Vous êtes un ado.`",
- **sinon si** `nb` est inférieur à 40 (donc compris entre 18 et 39 inclus), afficher "`Vous êtes un adulte.`",
- **sinon si** `nb` est inférieur à 60 (donc compris entre 40 et 59 inclus), afficher "`Vous êtes un vieil adulte.`",
- **sinon**, (donc si l'âge est au moins de 60), afficher "`Vous êtes un vieillard.`".

Une seule **branche** est donc choisie en fonction de la valeur de `nb`, et un seul message sera donc affiché.

À retenir !

Voici les **opérateurs** permettant d'effectuer des **comparaisons** :

>	plus grand que	>=	supérieur ou égal à	==	égal à
<	plus petit que	<=	inférieur ou égal à	!=	différent de

Homogénéité des valeurs comparées

Lorsque vous effectuez une **comparaison** (utilisant l'un des symboles ci-dessus), il faut vous assurer que vous compariez bien ce qui est comparable. Vous pouvez comparer des entiers entre eux, des flottants entre eux, des chaînes de caractères entre eux, ou encore comparer un entier avec un flottant... Mais vous ne pouvez pas, par exemple, comparer un **entier** avec une **chaîne de caractères**, ou vous obtiendrez une erreur.

```
1 >>> 1 < "123"
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: '<' not supported between instances of 'int' and 'str'
```

Le type *bool*

Lorsque vous effectuez une **expression conditionnelle** (qu'on appellera parfois **test**), cette expression est **évaluée** par **Python** soit à la valeur **True**, soit à la valeur **False**. Voici un exemple (exécuter le programme pour voir le résultat) :

```
1 x = 10
2 test1 = x > 5
3 test2 = x > 15
4
5 print("test1 =", test1) # renvoie True
6 print("test2 =", test2) # renvoie False
```

Les valeurs **True** et **False** sont des valeurs **booléennes** (de type **bool**), ce sont d'ailleurs les **deux seules valeurs** que peut prendre une variable de ce type.

Lorsque vous utilisez une **instruction de branchement**, comme **if <condition>**, la **condition** est une **expression** qui est **évaluée** à **True** ou **False**. Les instructions du **bloc if** ne sont exécutées que si la **condition** est évaluée à **True**.

Par exemple :

```
1 age = 30
2 majeur = age >= 18
3 if majeur:
4     print("Vous êtes majeur")
```

Ici, l'expression **age >= 18** est évaluée à **True**, et la valeur **True** est stockée dans la variable **majeur**. Ainsi, on rentre bien dans le bloc **if**.

Cela revient au même résultat que d'écrire :

```
1 age = 30
2 if age >= 18:
3     print("Vous êtes majeur")
```

Les opérateurs logiques

En *Python*, des **opérateurs logiques** permettent de combiner plusieurs **conditions** et ainsi réduire le nombre d'instructions de branchement nécessaires.

L'opérateur **and**

Si l'on souhaite par exemple vérifier **deux conditions** à la fois et afficher le message **"OK"** lorsque les deux conditions sont **vérifiées**, on peut procéder ainsi :

```
1 if <condition 1>:
2     if <condition 2>:
3         print("OK !")
```

Plutôt que d'utiliser deux instructions `if`, on peut obtenir le même comportement en utilisant l'**opérateur logique** `and` de la manière suivante :

```
1 if <condition 1> and <condition 2>:  
2     print("OK")
```

En effet, ici, on entre dans le bloc `if` seulement si le test `<condition 1> and <condition 2>` est évalué à `True`, c'est-à-dire si la **condition 1 ET la condition 2** sont toutes les deux **vraies** (évaluées au booléen `True`).

On peut bien sûr utiliser autant de fois l'opérateur `and` que l'on souhaite à l'intérieur d'un **test**, voici un exemple :

```
1 taille = 175  
2 poids = 70  
3 if taille > 170 and taille < 180 and poids > 60 and poids < 80:  
4     print("Vous pouvez entrer")
```

Ici, le message `"Vous pouvez entrer"` est affiché **uniquement** si les **4 conditions** `taille > 170`, `taille < 180`, `poids > 60`, `poids < 80` sont **vérifiées**.

L'opérateur or

Autre situation : on souhaite maintenant tester **deux conditions**, et vérifier si seulement **l'une** ou **l'autre** est **vraie**, ou si **les deux** sont **vraies** à la fois.

En utilisant des instructions de branchement, on pourrait écrire :

```
1 if <condition 1>:  
2     print("OK !")  
3 elif <condition 2>:  
4     print("OK !")  
5 else:  
6     print("Pas ok.")
```

Ici encore, on peut réduire le nombre d'instructions de branchement avec l'**opérateur logique** `or` :

```
1 if <condition 1> or <condition 2>:  
2     print("OK !")  
3 else:  
4     print("Pas ok.")
```

Un opérateur paresseux

L'opérateur `or` possède une caractéristique intéressante. En effet, si vous utilisez un `or` entre **deux conditions**, **Python** n'évaluera pas la **deuxième condition** si la **première** est évaluée à `True`. Par exemple :

```
1 x = 35
2 y = 15
3 if x >= 30 or y <= 20:
4     print("OK")
```

Ici, l'expression `y <= 20` ne sera **pas évaluée** car la première expression `x >= 30` est évaluée à `True`. Étant donné que seule l'une ou l'autre des conditions doit être vraie pour que le test `x >= 30 or y <= 20` soit vérifié, on a pas besoin de tester la deuxième condition.

Tout comme pour le `and`, on peut accumuler autant de fois l'opérateur `or` qu'on le souhaite dans un **test**, on peut également combiner des `and` et des `or`, par exemple :

```
1 age = 15
2 accompagne = True
3 if (age >= 18 and age < 70) or accompagne:
4     print("Vous pouvez entrer")
```

Priorité des opérations :

En *Python*, la **priorité des opérateurs logiques** est la suivante :

- `not` – **plus haute** priorité
- `and` – priorité **intermédiaire**
- `or` – **plus basse** priorité

Cela signifie que dans une expression contenant **plusieurs opérateurs**, les termes de l'expression utilisant le `not` sont évalués **en premier**, puis viennent ensuite les termes de l'expression utilisant le `and`, et enfin les termes de l'expression utilisant le `or`.

Après avoir appliqué les priorités, l'évaluation s'effectue de **gauche à droite**.

On peut toutefois obtenir des différences dans le résultat booléen obtenu en ajoutant des **parenthèses** pour **forcer un ordre d'évaluation** différent, par *exemple* :

```
1 >>> x = 5
2 >>> y = 10
3 >>> z = 15
4 >>> x > 0 or y > 10 and z > 20  # True or False and False => True or
    False => True
5 True
6 >>> (x > 0 or y > 10) and z > 20  # (True or False) and False => True
    and False => False
7 False
8 >>> x > 0 or y < 0 and not z == 15  # True or False and not True =>
    True or False and False => True or False => True
```

```
9 True
10 >>> (x > 0 or y < 0) and not z == 15  # (True or False) and not True =>
    True and False => False
11 False
```

L'opérateur not

L'**opérateur logique** `not` utilisée sur une expression booléenne renvoie `True` si l'expression est évaluée à `False`, et `False` si l'expression est évaluée `True`.

Voici un exemple :

```
1 x = 10
2 test1 = x > 5
3 test2 = not (x > 5)
4
5 print("test1 =", test1)  # affichera "test1 = True"
6 print("test2 =", test2)  # affichera "test2 = False"
```

Tables de vérité

Les **opérateurs logiques** `and`, `or` et `not` permettent de combiner des **expressions booléennes** pour créer des **conditions plus complexes**.

À retenir !

Voici les **tables de vérité** résumant le fonctionnement de ces opérateurs.

La table du `and` :

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

La table du `or` :

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

À retenir !

La table du `not` :

A	<code>not A</code>
True	False
False	True

Ces tables montrent que l'opérateur `and` ne renvoie `True` que si **les deux opérandes** sont **vraies**, tandis que l'opérateur `or` renvoie `True` si **au moins une** des opérandes est **vraie**. L'opérateur `not`, quant à lui, inverse la valeur booléenne.