

# Course: Python

---

Ali ZAINOUL <contact@clearcode.fr>

Crystal Clear Code  
June 16, 2025



# 1 Introduction

- Historique
- Versions de Python
- Caractéristiques du langage
- Installation de Python
  - Installation de Python sur Windows
  - Installation de Python sur Linux
  - Installation de Python sur MacOS
- Panorama de la bibliothèque standard
- Modules d'extension et commande pip
- Principe de fonctionnement de l'interpréteur
- Interpréteur officiel CPython et autres interpréteurs
- Interpréteur interactif
- Ressources
- Fonction `help()` et chaînes documentaires
- Principe de l'indentation pour délimiter les blocs d'instruction
- Commentaires

- Mots-clés réservés
- Conventions de nommage
- Programme autonome
- Fonctions intégrées élémentaires

## 2 Types fondamentaux

- Booléen

- Objets True et False

- Nombres

- Les entiers relatifs (int)
  - Les réels (float)
  - Les complexes

- Chaîne de caractères unicode (str)

- Manipulation des Chaînes de Caractères
  - Définition avec simple et double guillemets
  - Chaînes multilignes avec triple simple ou double guillemets
  - Mode raw
  - Constructeur
  - Indicage positif et négatif
  - Tranche de valeurs (slice)

- Opérateurs +, \* et in
- Itération
- Méthodes incontournables de `str`
- L'affichage formaté de variables

### 3 Les Opérateurs

- Opérateurs Arithmétiques
- Opérateurs Relationnels
- Opérateurs Bit à Bit
- Opérateurs d'Assignment
- Opérateur Ternaire
- Autres opérateurs
- Priorité des opérateurs en Python
- Opérateurs et types fondamentaux
  - Opérateurs et les booléens
  - Opérateurs et les entiers
  - Opérateurs et les réels
  - Opérateurs et les complexes

### 4 Notations exponentielle, binaire, octale et hexadécimale

- Systèmes de notation et notion de base N
- Fonctions `hex()`, `oct()`, `bin()`, `chr()`, `ord()`

## 5 Structures conditionnelles et répétitives

- Les structures conditionnelles `if/elif/else`
- Les boucles d'itérations `while` et `for`
- Instructions `break` et `continue`
- Fonction `enumerate()`
- Bloc `else` dans une structure répétitive

## **\*\* Introduction \*\***

# Historique

- **Auteur :** Guido van Rossum, informaticien néerlandais. Il a travaillé sur Python chez CWI aux Pays-Bas, influencé par le langage ABC. Van Rossum a voulu créer Python pour sa simplicité, sa productivité et sa syntaxe claire, aboutissant à un langage convivial, puissant et polyvalent.
- **Date de la première version :** Février 1991

# Version Python 2.x

- Python 2.x est une version héritée.
- Elle n'est plus maintenue depuis janvier 2020.
- Il est fortement recommandé de migrer vers Python 3.



# Version Python 3.x

- Python 3.x est la branche actuelle et en développement continu.
- Elle apporte de nombreuses améliorations par rapport à la branche 2.x.
- Python 3.5 a introduit l'opérateur de décomposition de matrices.
- Python 3.6 a introduit les f-strings pour un formatage de chaîne plus efficace.
- Veille technologique sur les dernières versions de Python.

# Python ?!

## ■ Python en bref :

- Langage de programmation puissant ;
- Facile à utiliser et à apprendre ;
- Open source ;
- Langage à typage dynamique fort.

## ■ Avantages de Python :

- Adapté aux débutants ;
- Utilisé à grande échelle dans une variété de projets industriels.



Figure: Python logo

# Ecosystème



Figure: Ecosystem Python

# Caractéristiques

Python est un langage de programmation qui est :

- Muni d'une syntaxe claire ;
- Interprété ;
- Multi-paradigmes et Multi-Plateformes ;
- qui favorise la :
  - Programmation impérative structurée ;
  - Programmation fonctionnelle ;
  - Programmation Orientée Objet.

# Installation de Python sur Windows

- Téléchargez le dernier programme d'installation de Python depuis le site officiel.
- Ouvrez le programme d'installation .exe.
- Cochez le bouton pour ajouter Python au PATH, et suivez les instructions à l'écran pour terminer l'installation.
- Vérifiez que Python a bien été installé correctement: ouvrez l'invite de commandes **Powershell** et saisissez `python --version` afin de récupérer la version de Python installée.

# Installation de Python sur Linux

- Ouvrez le terminal et saisissez la commande suivante : `sudo apt-get install python3`
- Entrez votre mot de passe lorsque vous y êtes invité.
- Attendez que l'installation soit terminée.
- Pour vérifier que Python a été installé correctement, saisissez `python3` dans le terminal pour lancer l'interpréteur Python.

# Installation de Python sur MacOS

- Ouvrez le terminal et saisissez la commande suivante : `xcode-select --install`
- Suivez les instructions à l'écran pour installer les outils de ligne de commande Xcode.
- Une fois les outils de ligne de commande Xcode installés, saisissez la commande suivante dans le terminal : `brew install python3`
- Attendez que l'installation soit terminée.
- Pour vérifier que Python a été installé correctement, saisissez `python3` dans le terminal pour lancer l'interpréteur Python.

# Panorama de la bibliothèque standard

- La **bibliothèque standard de Python** est une collection de `modules` et de `packages` étendus couvrant divers domaines, tels que l'accès aux fichiers, le traitement de chaînes, la manipulation de données, les communications réseau et bien d'autres encore.
- Ces modules sont inclus dans une installation standard de Python et peuvent être utilisés pour développer une grande variété d'applications sans avoir à installer des bibliothèques tierces.
- Pour plus d'informations sur les modules disponibles dans la bibliothèque standard de Python, consultez la [Documentation bibliothèque standard](#).



# Éléments les plus utilisés

- Built-in Functions
- Built-in Constants
- Built-in Types
- Built-in Exceptions
- Text Processing Services
- Data Types
- Numeric and Mathematical Modules
- Functional Programming Modules
- File and Directory Access
- Data Compression and Archiving
- Generic Operating System Services
- Development Tools
- Debugging and Profiling
- Software Packaging and Distribution
- Python Runtime Services
- Importing Modules
- Concurrent Execution
- Networking and Interprocess Communication
- Internet Data Handling
- Structured Markup Processing Tools
- Internet Protocols and Support
- Graphical User Interfaces with Tk

# Modules d'extension et commande pip - Introduction

- Les **modules d'extension** sont des **bibliothèques externes** que vous pouvez ajouter à votre environnement Python pour étendre ses fonctionnalités.

# Modules d'extension et commande pip - Utilisation de pip

- La commande `pip` est l'outil recommandé pour installer et gérer ces modules d'extension. Elle vous permet de rechercher, télécharger et installer des packages Python à partir du Python Package Index (PyPI).
- Utilisez la commande `pip install` pour installer un module spécifique. Par exemple, pour installer le module `requests`, vous pouvez exécuter : `pip install requests`.

# Principe de fonctionnement de l'interpréteur - Introduction

- **L'interpréteur Python** est un programme qui **lit et exécute le code source Python**. Il **traduit le code source** en **instructions exécutables** que l'ordinateur peut **comprendre et exécuter**.

# Principe de fonctionnement de l'interpréteur - Bytecode PYC

- Le **bytecode PYC** est un code intermédiaire généré par l'interpréteur Python lors de la compilation du code source. Il est stocké dans des fichiers avec l'extension `.pyc` et est utilisé pour accélérer le chargement et l'exécution du code Python en évitant la recompilation du code source à chaque exécution.

# Interpréteur officiel CPython

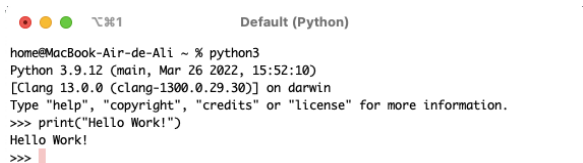
- **CPython** : L'interpréteur officiel de Python, écrit en C, est largement utilisé et prend en charge les extensions C. Il est principalement utilisé pour exécuter des applications Python standard.

# Autres interpréteurs

- **Micropython** : Une implémentation de Python destinée aux microcontrôleurs et aux systèmes embarqués, offrant une empreinte mémoire réduite et un support pour les plateformes à ressources limitées.
- **Brython** : Un projet visant à implémenter un interpréteur Python dans le navigateur Web, permettant l'exécution de code Python côté client.
- **Pypy** : Une implémentation alternative de Python, écrite en Python, visant à améliorer les performances par rapport à CPython en utilisant la compilation JIT.
- **Numba** : Un compilateur JIT pour Python qui vise à accélérer l'exécution de code numérique en générant du code machine optimisé pour les calculs scientifiques et numériques.

# Interpréteur interactif

- Voici un exemple d'utilisation de l'interpréteur interactif:

A screenshot of a macOS terminal window. The title bar at the top shows three colored window control buttons (red, yellow, green) and the text "Default (Python)". The terminal content shows a user at a Mac running the command 'python3'. The output displays the Python version (3.9.12), the main branch, the date and time (Mar 26 2022, 15:52:10), the compiler (Clang 13.0.0), and the operating system (darwin). It then prompts for help information. The user enters '>>> print("Hello Work!")' and the terminal outputs 'Hello Work!'. The prompt '>>>' is followed by a red cursor bar.

```
home@MacBook-Air-de-Ali ~ % python3
Python 3.9.12 (main, Mar 26 2022, 15:52:10)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello Work!")
Hello Work!
>>>
```

Figure: Premier programme en Python



# Ressources - Site Officiel python.org

- Site Internet python.org : La page officielle de Python.org est une ressource précieuse pour les utilisateurs de Python. Elle fournit des informations sur les dernières nouvelles, les événements, les ressources d'apprentissage et bien plus encore.
- Documentation officielle de Python : La documentation officielle de Python est une ressource essentielle pour tout développeur Python. Elle contient des informations détaillées sur la syntaxe du langage, la bibliothèque standard, les modules et bien plus encore.

# Ressources - Autres ressources pour apprendre Python

- LearnPython.org : LearnPython.org offre des tutoriels interactifs et des exercices pratiques pour apprendre Python à votre propre rythme.
- Real Python : Real Python propose des tutoriels, des articles et des cours pour les programmeurs de tous niveaux, du débutant à l'expert.
- Codecademy : Codecademy propose des cours interactifs en ligne pour apprendre Python, avec des exercices pratiques et des projets.

# Fonction `help()`

- La fonction `help()` est utilisée pour obtenir de l'aide et des informations sur les objets Python.
- Voici un exemple d'utilisation de la fonction `help()` pour obtenir de l'aide sur la fonction `print()` :  
`help(print)`

# Chaînes documentaires (docstrings)

- Les **chaînes documentaires**, également appelées `docstrings`, sont des chaînes de caractères utilisées pour documenter les modules, les fonctions, les classes et les méthodes Python.
- Avec l'IDE Visual Studio Code, une personne peut installer l'extension `autodocstring`.

# Chaînes documentaires (docstrings) - Suite

- Voici un exemple d'une docstring pour une fonction simple :

```
1 def add(x: int, y: int) -> int:
2     """
3     Add two numbers.
4
5     Parameters:
6     x (int): The first number.
7     y (int): The second number.
8
9     Returns:
10    int: The sum of the two numbers.
11    """
12    return x + y
```

- L'accès à la documentation d'une telle fonction par exemple peut se faire grâce à l'appel suivant: `print(add_module.add.__doc__)` ou plus simplement: `help(add_module.add)`.

# Indentation en Python

- Python utilise l'indentation pour définir des blocs de code. Cela diffère de nombreux autres langages de programmation tels que C, C++ ou Java, qui utilisent des accolades pour définir des blocs. Voici un exemple :

```
1 number = 123
2 if number > 0:
3     print("The number is positive.")
4 else:
5     print("The number is negative or equals to zero.")
```

- L'indentation indique à Python quelles lignes de code appartiennent à chaque bloc.
- Utilisez la touche **tabulation** pour indenter votre code.

# Fin de ligne en Python

- En Python, une ligne de code se termine par un caractère de nouvelle ligne. Vous pouvez également utiliser un backslash `\` pour indiquer qu'une instruction se poursuit à la ligne suivante. Voici un exemple :

```
1 x = 5 + \  
2 10  
3 print(x) # Displays : 15
```

- Dans cet exemple, le backslash indique à Python que l'opérateur `+` fait partie de la ligne suivante, et la valeur de `x` est 15.

# Commentaires en Python

- Vous pouvez ajouter des commentaires à votre code Python pour expliquer ce que fait le code. Les commentaires commencent par un caractère dièse # et se poursuivent jusqu'à la fin de la ligne. Voici un exemple :

```
1 # This is a comment and will be ignored
2 print("Hello, world!")
```

- Dans cet exemple, la première ligne est un commentaire, et Python l'ignore lors de l'exécution du code. La deuxième ligne affiche le message "Hello, world!" dans la console. Les commentaires peuvent être utilisés pour rendre votre code plus lisible et plus facile à comprendre.



# Commentaires sur plusieurs lignes en Python

```
1 # This is a long comment that spans
2 # multiple lines in the Python code.
3 # It provides additional explanations or context.
4
5 def example_function():
6     """
7     This is another way to create a multiline comment
8     in Python, using triple double-quotes. This is often
9     used for docstrings, which provide documentation
10    for functions, modules, or classes.
11    """
12    print("Hello, World!")
13
14 # Call the example function
15 example_function()
```

# Mots-clés réservés en Python

- Les **mots-clés réservés** en Python sont des **identificateurs** qui ont une **signification spéciale** pour l'**interpréteur Python**. Ils ne peuvent **pas** être utilisés comme noms de variables, de fonctions ou d'autres identificateurs.
- Il est important de ne pas utiliser les mots-clés réservés comme noms pour vos variables, fonctions ou classes, car cela peut entraîner des erreurs ou un comportement inattendu de votre programme.

# Mots-clés réservés en Python - Liste des mots-clés

<code>False</code>	<code>None</code>	<code>True</code>	<code>and</code>	<code>as</code>
<code>assert</code>	<code>async</code>	<code>await</code>	<code>break</code>	<code>class</code>
<code>continue</code>	<code>def</code>	<code>del</code>	<code>elif</code>	<code>else</code>
<code>except</code>	<code>finally</code>	<code>for</code>	<code>from</code>	<code>global</code>
<code>if</code>	<code>import</code>	<code>in</code>	<code>is</code>	<code>lambda</code>
<code>nonlocal</code>	<code>not</code>	<code>or</code>	<code>pass</code>	<code>raise</code>
<code>return</code>	<code>try</code>	<code>while</code>	<code>with</code>	<code>yield</code>

# Conventions et règles de nommage

- Utilisez des noms de variables et de fonctions significatifs pour améliorer la lisibilité du code.
- Suivez une convention de nommage cohérente telle que `camelCase` ou `snake_case`.
- Évitez d'utiliser des noms de variables d'une seule lettre sauf dans des contextes bien définis (par exemple, pour des compteurs de boucle).
- Utilisez des commentaires pour expliquer des sections de code complexes ou pour fournir du contexte aux autres.

# Conventions et règles de nommage - Suite

- Suivez le guide de style PEP 8 Coding Convention pour le code Python.
- Maintenez une indentation cohérente pour une meilleure structure du code.

```
1 # Examples of camelCase and snake_case
2 variableCamelCase = "A variable in camelCase"
3 variable_snake_case = "A variable in snake_case"
4
5 # Avoid using variables with a single letter
6 number = 13
7 def square(number):
8     return (number*number)
9
10 print(square(13)) # Displays : 169
```

# Outils nécessaires

- Environnement de développement intégré (IDE)  
Choix entre : PyDev, Microsoft Visual Studio Code, Sublime Text...



Figure: Meilleur IDE

- Éditeur de texte & interprétation / exécution *via* le terminal

# Notion de Programme autonome

- Un **programme autonome**, également connu sous le nom de programme indépendant, est un **programme informatique** qui peut fonctionner de **manière autonome**, sans nécessiter l'exécution dans un environnement spécifique ou l'interaction avec d'autres programmes. Ces programmes sont conçus pour être **auto-suffisants** et peuvent être exécutés **indépendamment du système d'exploitation** ou des autres logiciels installés sur l'ordinateur.

# Exemple d'un programme autonome

Listing 1: Python code example

```
1 # Define a simple function
2 def add(x, y):
3     return x + y
4
5 # Call the function and print the result
6 result = add(3, 5)
7 print("The result of addition is:", result)
```

■ Exécution: `python add.py`



# Fonction print()

```
1 # Print a simple message
2 print("Hello, world!")
3
4 # Print multiple values
5 x = 10
6 y = 20
7 print("The value of x is:", x, "and the value of y is:", y)
8
9 # Print with formatting
10 name = "Alice"
11 age = 30
12 print("Name:", name, ", Age:", age)
```

# Fonction type()

```
1 # Determine the type of a variable
2 x = 10
3 y = "Hello"
4 print(type(x))    # Output: <class 'int'>
5 print(type(y))    # Output: <class 'str'>
```

# Fonction input()

```
1 # Take input from the user
2 name = input("Enter your name: ")
3 print("Hello,", name)
```

# Fonction len()

```
1 # Determine the length of a list
2 my_list = [1, 2, 3, 4, 5]
3 print("Length of the list:", len(my_list)) # Output: 5
4
5 # Determine the length of a string
6 my_string = "Hello, world!"
7 print("Length of the string:", len(my_string)) # Output: 13
```

## **\*\* Les fondamentaux \*\***

# Les booléens (Booleans)

- Les **booléens** peuvent prendre uniquement deux valeurs:
  - soit **Vrai (True)** ;
  - soit **Faux (False)**.
- Ils sont souvent utilisés dans les instructions conditionnelles et les boucles.

# Les booléens (Booleans) - Exemple

```
1 # Example of manipulating booleans
2
3 bigger = 10
4 smaller = 5
5 comparison_boolean = (bigger>smaller)
6 print(f"The result of comparing ({bigger}>{smaller}) is:
7 {comparison_boolean}")
8 # Displays : The result of comparing (10 > 5) is: True
```

# Les entiers (Integers)

- Les **entiers** sont des nombres entiers tels que -3, -2, -1, 0, 1, 2, 3, etc. Vous pouvez effectuer des opérations arithmétiques de base sur les entiers, telles que:
  - l'addition + ;
  - la soustraction - ;
  - la multiplication \* ;
  - la division / ;
  - la division entière // ;
  - le reste modulo % ;
- Dans l'exemple qui suit, on rappelle que :

$$7 = (3 \times 2) + 1$$

Où: 7 étant le **dividende**, 3 le **diviseur**, 2 le **quotient** et 1 le **reste** de la **division euclidienne** de 7 par 3.



# Les entiers (Integers) - Exemple

```
1 # Example of basic arithmetic operations in Python
2
3 # Definition of two variables x and y
4 x = 7
5 y = 3
6 result_addition = x + y
7 print(f"Addition: {x} + {y} = {result_addition}")
8 # Displays: Addition: 7 + 3 = 10
9
10 # Subtraction
11 result_subtraction = x - y
12 print(f"Subtraction: {x} - {y} = {result_subtraction}")
13 # Displays: Subtraction: 7 - 3 = 4
```

```
1 # Multiplication
2 result_multiplication = x * y
3 print(f"Multiplication: {x} * {y} = {result_multiplication}")
4 # Displays: Multiplication: 7 * 3 = 21
5
6 # Division
7 result_division = x / y
8 print(f"Division: {x} / {y} = {result_division}")
9 # Displays: Division: 7 / 3 = 2.3333333333333335
```

```
1 # Entire Division (Euclidian Division)
2 result_entire_division2 = x // 2
3 print(f"Entire Division: {x} // {2} = {result_entire_division2}")
4 # Displays: Entire Division: 7 // 2 = 3
5
6 # Entire Division (Euclidian Division)
7 result_entire_division = x // y
8 print(f"Entire Division: {x} // {y} = {result_entire_division}")
9 # Displays: Entire Division: 7 // 3 = 2
10
11 # Modulo
12 result_modulo = x % y
13 print(f"Modulo: {x} % {y} = {result_modulo}")
14 # Displays: Modulo: 7 % 3 = 1
```

# Constructeurs pour les Entiers (Integers)

- Les entiers peuvent être créés en utilisant le constructeur `int()`.
- Il peut prendre divers arguments comme un autre entier, une chaîne de caractères contenant un nombre ou même un flottant.
- Exemple :

```
1 x = int(5)
2 y = int("10")
3 z = int(3.14)
```

- Documentation officielle : [int](#)

# Les flottants (Floats)

- Les **flottants** sont des nombres décimaux (nombres à virgule), et qui ont un nombre fini de chiffres après la virgule, tels que: 3.14; 2.0 et  $-1.2345$ .
- Vous pouvez également effectuer des opérations arithmétiques de base sur les flottants.

# Les flottants (Floats) - Exemple

```
1 # Example of manipulating floats in Python
2
3 x = 3.14
4 y = 2
5 z = 3.0
6
7 result_product = x * y
8 result_addition = result_product + z
9 print(f"Result of: (({x} * {y}) + {z}) = {result_addition}")
10 # Displays: Result of: ((3.14 * 2) + 3.0) = 9.280000000000001
```

# Constructeurs pour les Flottants (Floats)

- Les flottants peuvent être créés en utilisant le constructeur `float()`.
- Il peut prendre divers arguments comme un entier, une chaîne de caractères contenant un nombre ou même un autre flottant.
- Exemple :

```
1 x = float(5)
2 y = float("10.5")
3 z = float(3.14)
```

- Documentation officielle : [float](#)

# Les nombres complexes (Complex Numbers)

- Les **nombres complexes** sont des nombres ayant à la fois une partie réelle et une partie imaginaire. Ils sont souvent utilisés en mathématiques et en physique pour représenter des quantités oscillantes telles que les ondes.
- En Python, les nombres complexes sont créés en utilisant la lettre `j` ou `J` pour représenter la partie imaginaire.
- Exemples de nombres complexes:  $3 + 2j$ ,  $-1.5 + 0.5j$ ,  $2J$ .



# Les nombres complexes (Complex Numbers) - Exemple

```
1 # Example of manipulation of complex numbers in Python
2
3 import cmath
4 z1 = 3 + 2j
5 z2 = -1.5 + 0.5j
6
7 result_sum = z1 + z2
8 result_product = z1 * z2
9
10 print(f"Result of addition: {z1} + {z2} = {result_sum}")
11 # Displays: Result of addition: (3+2j) + (-1.5+0.5j) = (1.5+2.5j)
12
13 print(f"Result of product: {z1} * {z2} = {result_product}")
14 # Displays: Result of product: (3+2j) * (-1.5+0.5j) = (-5.5-3.5j)
```

# Constructeurs pour les Nombres Complexes (Complex Numbers)

- Les nombres complexes peuvent être créés en utilisant le constructeur `complex()`.
- Il peut prendre deux arguments, la partie réelle et imaginaire, ou un seul argument sous forme de chaîne de caractères contenant les deux parties.
- Exemple :

```
1 import cmath
2 x = complex(3, 2)
3 y = complex("-1.5+0.5j")
```

- Documentation officielle : [complex](#)

# Manipulation des Chaînes de Caractères

- En Python, les chaînes de caractères (`str`) offrent une variété de méthodes intégrées pour effectuer des opérations sur les chaînes. Vous pouvez consulter la documentation officielle pour en savoir plus : [Documentation Python - String Methods](#).

# Manipulation des Chaînes de Caractères - Méthodes

```
1 # String methods examples
2 message = "Hello, World!"
3
4 # String methods
5 string_methods = dir(message)
6 print("String methods:", string_methods)
```

# Manipulation de Chaînes de Caractères - Exemples

```
1 # Concatenation
2 message = "Hello, World!"
3 new_message = message + " Welcome to Python!"
4 print("Concatenation:", new_message)
5
6 # String length
7 length = len(message)
8 print("String length:", length)
9
10 # Accessing a specific character
11 first_char = message[0]
12 print("First character:", first_char)
13
14 # Conversion to uppercase
15 print("Uppercase:", message.upper())
```

# Définition avec guillemets simples

- Les guillemets simples ( ' ' ) sont utilisés pour définir une chaîne de caractères.
- Exemple :

```
1 single_quote_string = 'Hello, world!'
```

# Définition avec guillemets doubles

- Les guillemets doubles (") sont également utilisés pour définir une chaîne de caractères.
- Exemple :

```
1 double_quote_string = "Hello, world!"
```

# Définition avec guillemets simples et doubles

- Les guillemets simples ( ' ' ) ou doubles ( " " ) peuvent être utilisés pour définir une chaîne de caractères.
- Exemple :

```
1 sentence = "He said: 'Hello, world!'"
```



# Chaînes multilignes avec triple guillemets simples

- Les triple guillemets simples (''' ''') sont utilisés pour définir des chaînes de caractères multilignes.
- Exemple :

```
1 multiline_text_single_quote_string =  
2 '''Hello,  
3 This is a  
4 multiline string.'''
```

# Chaînes multilignes avec triple guillemets doubles

- Les triple guillemets doubles (""" """) sont également utilisés pour définir des chaînes de caractères multilignes.
- Exemple :

```
1 multiline_text_double_quote_string =  
2 """Hello,  
3 This is another  
4 multiline string."""
```

# Le Mode Raw pour les Chaînes de Caractères

- Le mode raw est un mode spécial pour les chaînes de caractères en Python.
- En utilisant le mode raw, Python ne traite pas les séquences d'échappement spéciales dans les chaînes de caractères.
- Pour activer le mode raw, préfixez la chaîne de caractères avec un `r` (minuscule ou majuscule).
- Utile lors de la manipulation de chaînes de caractères qui contiennent de nombreux caractères d'échappement et que vous ne souhaitez pas que Python les interprète.
- [Documentation officielle sur les Chaînes de Caractères](#)

## Exemple d'Utilisation du Mode Raw (1/3)

```
1 # Example of using raw strings in Python
2 raw_string1 = r"C:\Users\username\Documents"
3 raw_string2 = r"Line 1\nLine 2\nLine 3"
4 print(raw_string1) # Output: C:\Users\username\Documents
5 print(raw_string2) # Output: Line 1\nLine 2\nLine 3
```

## Exemple d'Utilisation du Mode Raw (2/3)

```
1 # Using raw strings with regular expressions
2 import re
3 pattern = r'\d+'
4 text = 'There are 10 apples and 20 oranges'
5 matches = re.findall(pattern, text)
6 print(matches) # Output: ['10', '20']
```

## Exemple d'Utilisation du Mode Raw (3/3)

```
1 # Using raw strings with file paths
2 file_path = r"C:\Users\username\Documents\example.txt"
3 with open(file_path, 'r') as file:
4     content = file.read()
5     print(content)
```

# Constructeurs de la classe `str`

- La classe `str` offre plusieurs constructeurs pour créer des objets de type chaîne de caractères.
- Voici quelques-uns des constructeurs les plus couramment utilisés :
  1. Constructeur vide : Crée une chaîne de caractères vide.
  2. Constructeur avec une chaîne de caractères : Crée une copie de la chaîne de caractères spécifiée.
  3. Constructeur avec un objet convertible en chaîne : Convertit l'objet spécifié en une chaîne de caractères.
  4. Constructeur avec une séquence : Crée une chaîne de caractères à partir de la séquence spécifiée.
- Documentation officielle des constructeurs de la classe `str`

# Exemples d'Utilisation des Constructeurs de la classe str

```
1 # Empty constructor
2 empty_string = str()
3 print("# Output: Empty string:", empty_string) # Output:
4
5 # Constructor with a string
6 original_string = "Hello"
7 copied_string = str(original_string)
8 print("# Output: Copy of the string:", copied_string) # Output: Hello"
9
10 # Constructor with an object convertible to a string
11 number = 42
12 converted_string = str(number)
13 print("# Output: String from a number:", converted_string) # Output: 42
14
15 # Constructor with a sequence
16 sequence = ['a', 'b', 'c']
17 sequence_string = str(sequence)
18 print("# Output: String from a sequence:", sequence_string) # Output: ['a', 'b', 'c']
```



# Indiçage positif et négatif

- En Python, les chaînes de caractères peuvent être accédées en utilisant des indices positifs et négatifs.
- Les indices positifs commencent à 0 pour le premier caractère et augmentent de 1 pour chaque caractère suivant.
- Les indices négatifs commencent à -1 pour le dernier caractère et diminuent de 1 pour chaque caractère précédent.
- Exemple :

```
1 s = "hello"
2 print("First character:", s[0])
3 # Output : First character: h
4 print("Last character:", s[-1])
5 # Output : Last character: o
```

# Tranche de valeurs (slice)

- La tranche de valeurs en Python vous permet d'accéder à une plage de caractères d'une chaîne de caractères.
- Syntaxe : `string[début:fin:pas]`
- Exemple :

```
1 s = "hello"  
2 print("Slice:", s[1:4])  
3 # Output : Slice: ell
```

# Opérateurs +, \*, et in

- Python propose plusieurs opérateurs pour la manipulation de chaînes de caractères.
- + est utilisé pour la concaténation de chaînes de caractères.
- \* est utilisé pour la répétition de chaînes de caractères OU pour l'unpacking (voir l'exemple dans le frame suivant).
- in est utilisé pour vérifier si une sous-chaîne existe dans une chaîne de caractères.
- Exemple :

```
1 s1 = "hello"
2 s2 = "world"
3 print("Concatenated:", s1 + s2)      # Output: Concatenated: helloworld
4 print("Repeated:", s1 * 3)          # Output: Repeated: hellohellohello
5 print("Contains 'lo '?", 'lo ' in s1) # Output: Contains 'lo '? True
```

# Itération

- Les chaînes de caractères en Python peuvent être itérées à l'aide d'une boucle `for`.
- Chaque caractère de la chaîne est parcouru individuellement.
- Exemple :

```
1 s = "hello"
2 for char in s:
3     print(char)
4 # Or via index:
5 for i in range(len(s)):
6     print(s[i])
7 # Or via * starred expression:
8 print(*s) # Output : h e l l o
```

# Méthodes de manipulation des chaînes de caractères

Voici quelques méthodes de manipulation des chaînes de caractères en Python :

- `split()`: Divise une chaîne en une liste de sous-chaînes en fonction d'un séparateur spécifié.
- `replace()`: Remplace toutes les occurrences d'une sous-chaîne par une autre dans la chaîne.
- `lower()`: Convertit tous les caractères de la chaîne en minuscules.
- `upper()`: Convertit tous les caractères de la chaîne en majuscules.
- `strip()`: Supprime les espaces (ou d'autres caractères spécifiés) au début et à la fin de la chaîne.
- `join()`: Joindre les éléments d'une séquence en une chaîne, en les séparant par une chaîne délimiteur.

# Exemples

Voici quelques exemples d'utilisation des méthodes de manipulation des chaînes de caractères en Python :

```
1 # Example of string manipulation methods in Python
2 s = "Hello, world!"
3 # Splitting the string into a list of words
4 words = s.split(",")
5 print("Split:", words) # Output: Split: ['Hello', ' world!']
6 # Replacing 'world' with 'Python'
7 new_s = s.replace("world", "Python")
8 print("Replace:", new_s) # Output: Replace: Hello, Python!
9 # Converting to lowercase
10 lower_s = s.lower()
11 print("Lowercase:", lower_s) # Output: Lowercase: hello, world!
12 # Converting to uppercase
13 upper_s = s.upper()
14 print("Uppercase:", upper_s) # Output: Uppercase: HELLO, WORLD!
15 # Stripping whitespace
16 stripped_s = s.strip()
17 print("Stripped:", stripped_s) # Output: Stripped: Hello, world!
18 # Joining a list of words into a single string
19 words = ['Hello', 'world']
20 joined_s = ' '.join(words)
21 print("Joined:", joined_s) # Output: Joined: Hello world
```

# Les f-strings en Python

- Les f-strings (formatted string literals) ont été introduites dans Python 3.6 en tant que fonctionnalité puissante de formatage de chaînes.
- Elles permettent l'insertion directe d'expressions Python dans des littéraux de chaînes.
- Les f-strings sont créées en préfixant une chaîne avec le caractère 'f' ou 'F'.
- Les expressions Python sont placées entre accolades à l'intérieur de la chaîne.

# Sortie formatée avec les f-strings

```
1 # Example of formatted output with f-string
2 name = "Ali"
3 age = 30
4 print(f"Hello, {name}! You are {age} years old.")
5
6 # Explanation
7 # The f-string is a concise way to format strings in Python.
8 # It allows embedding expressions inside string literals,
9 # using curly braces {} to enclose the expressions.
10 # In this example, variables 'name' and 'age' are inserted
11 # into the string to create a personalized greeting.
```



# Sortie formatée avec les f-strings - Suite

```
1 # Example 2
2 decimal = 2
3 amount = 42.75
4 print(f"The total amount is ${amount:.{decimal}f}.")
5
6 # Explanation
7 # The f-string also supports formatting options.
8 # In this example, the variable 'amount' is formatted
9 # as a floating-point number with two decimal places,
10 # creating a string that displays the total amount with proper
    currency formatting.
```

# La méthode format() en Python

- La méthode format() est une autre méthode de formatage de chaînes disponible en Python.
- Elle utilise des espaces réservés, appelés dans la chaîne de format, auxquels des valeurs sont passées en tant qu'arguments à la méthode format().
- La méthode format() peut effectuer un formatage plus avancé, tel que spécifier la largeur et la précision des nombres.

# Sortie formatée avec la méthode format()

```
1 # Example of formatted output with the format() method
2 item = "apple"
3 qty = 5
4 print("You have {} {}{}."
5       .format(qty, item, 's' if qty > 1 else ''))
6
7 # Explanation
8 # The format() method is another way to format strings.
9 # It uses placeholders enclosed in curly braces {} within the
   string.
10 # The values to be inserted are provided as arguments to the
   format() method.
11 # In this example, the quantity and item variables are inserted
   into the string,
12 # and 's' is included conditionally based on the quantity.
```

## Sortie formatée avec la méthode format() - Suite

```
1 # Example 2
2 price = 19.99
3 discount = 0.15
4 final_price = price * (1 - discount)
5 print("The final price after a {}% discount is
    ${:.2f}.".format(int(discount * 100), final_price))
6
7 # Explanation
8 # The format() method allows for more complex formatting.
9 # In this example, the final price after applying a discount
10 # is formatted to display the percentage discount and the final
    price
11 # with two decimal places.
```

# Les Opérateurs

**Définition :** En programmation informatique, un **opérateur** est un symbole ou un caractère qui indique au compilateur d'effectuer des processus mathématiques ou logiques spécifiques.

■ Les opérateurs présentés ici sont communs à de nombreux langages de programmation. Python fournit ces opérateurs intégrés suivants:

- opérateurs **arithmétiques**;
- opérateurs **relationnels**;
- opérateurs **d'assignation**;
- opérateurs **logiques**;
- opérateurs **bit à bit**;
- opérateurs **divers** (Misc).

# La Table de vérité

Un concept important lié à la manipulation des opérateurs est la table de vérité. Supposons que nous avons deux propositions (P) et (Q). Alors :

P	Q	P ET Q	P OU Q	P XOR Q	NON P	NON Q
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

# Opérateurs Arithmétiques

Le tableau suivant explicite les opérateurs arithmétiques intégrés en Python. Cet exemple montre ces opérateurs arithmétiques dans un programme Python.

Opérateur	Signification
+	Ajoute la valeur de l'opérande du côté droit (RHS) à la valeur de l'opérande du côté gauche (LHS) ( $a+b$ )
-	Soustrait la valeur de l'opérande du côté droit (RHS) à la valeur de l'opérande du côté gauche (LHS) ( $a-b$ )
*	Multiplie les valeurs des opérandes du côté gauche (LHS) et du côté droit (RHS) ensemble ( $a*b$ )
/	Divise la valeur de l'opérande du côté gauche (LHS) par la valeur de l'opérande du côté droit (RHS) ( $a/b$ )
//	(Division Entière/Euclidienne) Divise la valeur de l'opérande du côté gauche (LHS) par la valeur de l'opérande du côté droit (RHS) ( $a//b$ )
%	Le modulo renvoie le reste de la division euclidienne de la valeur de l'opérande du côté gauche (LHS) par la valeur de l'opérande du côté droit (RHS) ( $a\%b$ )

# Opérateurs Relationnels

Le tableau suivant explicite les opérateurs relationnels intégrés en Python. Cet exemple montre ces opérateurs relationnels dans un programme Python.

Opérateur	Signification
==	Évalué à vrai seulement si les deux opérandes sont égales
!=	Évalué à vrai seulement si les deux opérandes ne sont pas égales
>	Évalué à vrai seulement si la valeur de l'opérande du côté gauche (LHS) est strictement supérieure à celle du côté droit (RHS)
<	Évalué à vrai seulement si la valeur de l'opérande du côté droit (RHS) est strictement supérieure à celle du côté gauche (LHS)
>=	Évalué à vrai seulement si la valeur de l'opérande du côté gauche (LHS) est strictement supérieure ou égale à celle du côté droit (RHS)
<=	Évalué à vrai seulement si la valeur de l'opérande du côté droit (RHS) est strictement supérieure ou égale à celle du côté gauche (LHS)



# Opérateurs Logiques

Le tableau suivant explicite les opérateurs logiques intégrés en Python.  
Cet exemple montre ces opérateurs logiques dans un programme Python.

Opérateur	Signification
<code>and</code>	Opérateur ET logique. La condition est vraie seulement si les deux opérandes ne sont pas nulles
<code>or</code>	Opérateur OU logique. La condition est vraie seulement si l'une des opérandes n'est pas nulle
<code>not</code>	Opérateur NON logique. Il inverse l'état logique. Si la condition est vraie, alors l'opérateur NON logique la rendra fausse, et vice versa.

# Opérateurs Bit à Bit

Le tableau suivant explicite les opérateurs bit à bit intégrés en Python.  
Cet exemple montre ces opérateurs bit à bit dans un programme Python.

Opérateur	Signification
&	Opérateur ET binaire, évalue à VRAI seulement si les deux bits sont VRAIS
	Opérateur OU binaire, évalue à VRAI seulement si l'un des bits est VRAI
^	Opérateur OU exclusif binaire, évalue à VRAI seulement si un bit est VRAI dans un opérande mais pas dans les deux.
~	Opérateur NON binaire unaire (ou opérateur de complément). L'opérateur de complément inverse simplement les bits (même logique que NON).
<<	Opérateur de décalage binaire à gauche. La valeur des opérandes du côté gauche est déplacée vers la gauche par le nombre de bits spécifié par l'opérande du côté droit.
>>	Opérateur de décalage binaire à droite. La valeur des opérandes du côté gauche est déplacée vers la droite par le nombre de bits spécifié par l'opérande du côté droit.

# Opérateurs d'Assignment

Le tableau suivant explicite les opérateurs d'assignation intégrés en Python. Cet exemple montre ces opérateurs d'assignation dans un programme Python.

Opérateur	Signification
=	Opérateur d'assignation. Il assigne les valeurs des opérandes du côté droit (RHS) aux opérandes du côté gauche (LHS).
+=	Opérateur d'assignation ET d'addition. Il ajoute l'opérande du côté droit (RHS) à l'opérande du côté gauche (LHS) et assigne le résultat à l'opérande du côté gauche (LHS).
-=	Opérateur d'assignation ET de soustraction. Il soustrait l'opérande du côté droit (RHS) à l'opérande du côté gauche (LHS) et assigne le résultat à l'opérande du côté gauche (LHS).
*=	Opérateur d'assignation ET de multiplication. Il multiplie l'opérande du côté droit (RHS) à l'opérande du côté gauche (LHS) et assigne le résultat à l'opérande du côté gauche (LHS).
/=	Opérateur d'assignation ET de division. Il divise l'opérande du côté gauche (LHS) par l'opérande du côté droit (RHS) et assigne le résultat à l'opérande du côté gauche (LHS).
%=	Opérateur d'assignation ET de modulo. Il assigne le reste de la division euclidienne de l'opérande du côté gauche (LHS) par l'opérande du côté droit (RHS) à l'opérande du côté gauche (LHS).

# Opérateurs d'Assignment - Suite

Opérateur	Signification
<<=	Opérateur d'assignation ET de décalage à gauche.
>>=	Opérateur d'assignation ET de décalage à droite.
&=	Opérateur d'assignation ET logique.
^=	Opérateur d'assignation ET de OU exclusif logique.
=	Opérateur d'assignation ET de OU inclusif logique.

# Opérateur Ternaire

- La signature de l'**opérateur Ternaire** est la suivante:

value\_if\_condition\_is\_true if condition else  
value\_if\_condition\_is\_false

```
1 # Ternary Operator
2 x = 7
3 y = 4
4 max_value = x if x >= y else y
5 print(f"Max value between x and y: {max_value}")
```

# Autres opérateurs

■ Voici un exemple d'autres opérateurs que l'on peut trouver en Python:

```
1 # Other operators in Python
2
3 # Membership Operators
4 list_a = [1, 2, 3, 4, 5]
5 print(f"2 in list_a: {2 in list_a}") # True if 2 is in list_a
6 print(f"6 not in list_a: {6 not in list_a}") # True if 6 is not in list_a
7
8 # Identity Operators
9 z = 10
10 print(f"x is z: {x is z}") # True if x and z reference the same object
11 print(f"x is not z: {x is not z}") # True if x and z reference different objects
```

# Priorité des opérateurs en Python

- La priorité des opérateurs en Python détermine l'ordre dans lequel les opérations sont effectuées lorsqu'une expression contient plusieurs opérateurs.
- Les opérateurs avec une plus haute priorité sont évalués en premier.
- Voici un tableau illustrant la priorité des opérateurs du plus prioritaire au moins prioritaire :

# Priorité des opérateurs en Python - Tableau

Priorité	Opérateurs
1	Parenthèses ()
2	Puissance **
3	Multiplication *, Division /, Division entière //, Modulo %
4	Addition +, Soustraction -
5	Opérateurs de comparaison (>, <, >=, <=, ==, !=)
6	Opérateurs logiques (and, or, not)



# Opérateurs pour les Booléens (Booleans)

- Les booléens en Python prennent en charge certains des opérateurs mentionnés précédemment, notamment :
  - `&` : opérateur de bitwise AND
  - `|` : opérateur de bitwise OR

# Opérateurs pour les Booléens (Booleans) - Exemple

## ■ Voici des exemples d'utilisation :

```
1 x = True
2 y = False
3 result_bitwise_and = x & y # False
4 result_bitwise_or = x | y # True
```

# Opérateurs pour les Entiers (Integers)

## ■ Les entiers en Python supportent plusieurs opérateurs, y compris :

- `>>` : décalage à droite
- `<<` : décalage à gauche
- `|` : opérateur de bitwise OR
- `&` : opérateur de bitwise AND
- `^` : opérateur de bitwise XOR
- `/` : division
- `//` : division entière
- `%` : modulo
- `**` : exponentiation

# Opérateurs pour les Entiers (Integers) - Exemple

## ■ Voici des exemples d'utilisation :

```
1 x = 5
2 y = 2
3 result_shift_right = x >> y # 1
4 result_shift_left = x << y # 20
5 result_bitwise_or = x | y # 7
6 result_bitwise_and = x & y # 0
7 result_bitwise_xor = x ^ y # 7
8 result_division = x / y # 2.5
9 result_floor_division = x // y # 2
10 result_modulo = x % y # 1
11 result_exponentiation = x ** y # 25
```

# Opérateurs pour les Flottants (Floats)

- Les flottants en Python prennent en charge certains des opérateurs mentionnés précédemment, notamment :
  - / : division
  - // : division entière
  - % : modulo
  - \*\* : exponentiation

# Opérateurs pour les Flottants (Floats) - Exemple

## ■ Voici des exemples d'utilisation :

```
1 x = 5.5
2 y = 2.0
3 result_division = x / y # 2.75
4 result_floor_division = x // y # 2.0
5 result_modulo = x % y # 1.5
6 result_exponentiation = x ** y # 30.25
```

# Opérateurs pour les Nombres Complexes (Complex Numbers)

- Les nombres complexes en Python prennent en charge certains des opérateurs mentionnés précédemment, notamment :
  - `**` : exponentiation
- Voici un exemple d'utilisation :

```
1 z1 = 3 + 2j
2 z2 = 2 + 1j
3 result_exponentiation = z1 ** z2
```

# Opérateurs pour les Nombres Complexes (Complex Numbers) - Exemple

■ Voici un exemple d'utilisation :

```
1 import cmath
2 z1 = 3 + 2j
3 z2 = 2 + 1j
4 result_exponentiation = z1 ** z2
```



# Systèmes de notation

Une image illustrant l'équivalence entre les systèmes de notation:

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Figure: Number systems

# Notations Exponentielle

- En Python, la notation exponentielle est utilisée pour représenter les nombres en notation scientifique.
- Exemple:
  - `1.23e-4` représente  $1.23 \times 10^{-4}$ .
  - `2.5e2` représente  $2.5 \times 10^2$ .

# Notations Binaire

- En Python, la notation binaire est utilisée pour représenter les nombres en base 2.
- Exemple:
  - `0b1010` représente  $10_{10}$  en base 10.
  - `0b1101` représente  $13_{10}$  en base 10.

# Notations Octale

- En Python, la notation octale est utilisée pour représenter les nombres en base 8.
- Exemple:
  - `0o10` représente  $8_{10}$  en base 10.
  - `0o17` représente  $15_{10}$  en base 10.

# Notations Hexadécimale

- En Python, la notation hexadécimale est utilisée pour représenter les nombres en base 16.
- Exemple:
  - 0xA représente  $10_{10}$  en base 10.
  - 0xFF représente  $255_{10}$  en base 10.

# Fonction `hex()`

- La fonction `hex()` convertit un entier en sa représentation hexadécimale sous forme de chaîne.
- **Usage:** `hex(number)`.
- **Exemple:**

```
1 # Example of function hex()
2 x = 255
3 print("Hexadecimal representation of x: ", hex(x))
```

# Fonction oct()

- La fonction `oct()` convertit un entier en sa représentation octale sous forme de chaîne.
- **Usage:** `oct(number)`.
- **Exemple:**

```
1 # Example of function oct()
2 x = 255
3 print("Octal representation of x: ", oct(x))
```

# Fonction bin()

- La fonction `bin()` convertit un entier en sa représentation binaire sous forme de chaîne.
- **Usage:** `bin(number)`.
- **Exemple:**

```
1 # Example of function bin()
2 x = 255
3 print("Binary representation of x: ", bin(x))
```



# Fonction chr()

- La fonction `chr()` renvoie le caractère Unicode correspondant au nombre entier donné.
- **Usage:** `chr(number)`.
- **Exemple:**

```
1 # Example of function chr()
2 x = 65
3 print("Unicode Character of x: ", chr(x))
```

# Fonction ord()

- La fonction `ord()` renvoie l'entier Unicode correspondant au caractère donné.
- **Usage:** `ord(character)`.
- **Exemple:**

```
1 # Example of function ord()
2 x = 'A'
3 print("Unicode integer code of x :", ord(x))
```

## **\*\* Structures conditionnelles et répétitives \*\***

# Structure conditionnelle if ...

- En algorithmique et en programmation, nous appelons **structure conditionnelle** l'ensemble des instructions qui testent si une condition est vraie ou non. Il y en a trois :
  - La structure conditionnelle **if** :

```
if condition:  
    my_instructions
```

# Structure conditionnelle if ...else

## ■ Suite :

- La structure conditionnelle if ...else :

```
if condition:
    my_instructions
else:
    my_other_instructions
```

# Structure conditionnelle if ...elif ...else

## ■ Suite :

- La structure conditionnelle if ...elif ...else :

```
if condition_1:
    my_instructions_1
elif condition_2:
    my_instructions_2
else:
    my_other_instructions
```

# Structure conditionnelle boucle `while`

- De plus, une **structure conditionnelle itérative** permet d'exécuter plusieurs fois (itérations) la même série d'instructions. L'instruction **`while`** exécute les blocs d'instructions tant que la condition de `while` est vraie. Le même principe s'applique à la boucle **`for`** (pour), les deux structures sont détaillées ci-dessous :
  - La structure conditionnelle itérative **`while`** :

```
while condition:  
    mes_instructions
```

# Structure conditionnelle: boucle for

## ■ Suite :

- La structure conditionnelle itérative for :

```
for i in range(begin, end, step):  
    my_instructions
```

- **Remarque :** La différence majeure entre la boucle for et la boucle while est que la boucle for est utilisée lorsque le nombre d'itérations est connu, tandis que l'exécution se fait dans la boucle while jusqu'à ce que l'instruction dans le programme soit prouvée fausse.



# Structure conditionnelle: boucle for

- La structure conditionnelle element-based for :

```
for element in collection:  
    my_instructions
```

# Instructions break et continue

- L'instruction `break` permet de sortir prématurément d'une boucle.
- L'instruction `continue` permet de passer à l'itération suivante dans une boucle.
- Elles sont souvent utilisées pour contrôler le flux d'exécution dans les boucles.

```
1 # Example of break and continue statements
2 for i in range(10):
3     if i == 5:
4         break # Exit the loop when i equals 5
5     print(i, end=' ')
6 print("\n")
7
8 for i in range(10):
9     if i % 2 == 0:
10        continue # Skip the current iteration if i is even
11    print(i, end=' ')
```

# Fonction enumerate()

- La fonction `enumerate()` est utilisée pour énumérer les éléments d'une séquence tout en conservant leur indice.
- Elle renvoie un objet énumérable qui produit des tuples contenant à la fois l'index et la valeur de chaque élément de la séquence.
- Elle est couramment utilisée dans les boucles pour accéder à la fois à l'indice et à la valeur de chaque élément.

```
1 # Example of enumerate function
2 fruits = ['apple', 'banana', 'cherry']
3 for index, fruit in enumerate(fruits):
4     print(index, fruit)
```

# Bloc else dans une structure répétitive

- En Python, les boucles `for` et `while` peuvent avoir un bloc `else` qui est exécuté lorsque la boucle se termine normalement (sans être interrompue par un `break`).
- Le bloc `else` est exécuté après que la condition de la boucle devienne fausse.

```
1 # Examples of else block on loop
2 for i in range(5):
3     print(i)
4 else:
5     print("Loop completed without break")
```

```
1 for i in range(5):
2     if i == 3:
3         break
4 else:
5     print("Loop completed without break")
```

- Cela peut être utile pour exécuter du code après la boucle si aucune instruction `break` n'a été rencontrée.