cours

December 4, 2023

1 Introduction à Python

1.1 Table des matières

1.2 1. C'est quoi Python ?

Le langage de programmation Python a été créé en 1989 par Guido van Rossum, aux Pays-Bas.

Le nom Python vient d'un hommage à la série télévisée **Monty Python's Flying Circus** dont G. van Rossum est fan. La première version publique de ce langage a été publiée en 1991.

Ce langage de programmation présente de nombreuses caractéristiques intéressantes :

— Il est multiplateforme. C'est-à-dire qu'il fonctionne sur de nombreux systèmes d'exploitation : Windows, Mac OS X, Linux, Android, iOS, depuis les mini-ordinateurs Raspberry Pi jusqu'aux supercalculateurs.

— Il est gratuit. Vous pouvez l'installer sur autant d'ordinateurs que vous voulez (même sur votre téléphone !).

— C'est un langage de haut niveau. Il demande relativement peu de connaissance sur le fonctionnement d'un ordinateur pour être utilisé.

— C'est un langage interprété. Un script Python n'a pas besoin d'être compilé pour être exécuté, contrairement à des langages comme le C ou le C++.

— Il est orienté objet. C'est-à-dire qu'il est possible de concevoir en Python des entités qui miment celles du monde réel (une cellule, une protéine, un atome, etc.) avec un certain nombre de règles de fonctionnement et d'interactions.

— Il est relativement simple à prendre en main

1.3 2. Les Fondamentaux de Python

- Variables (types natifs) et opérations.
- Commentaires et affichage
- Structures de contrôle : opérateurs, conditions, boucles
- Types de Données (structures séquentielles)
- Fonctions : définition et appel.

1.3.1 2.1. Variables (types natifs)

Variables : Les variables sont des conteneurs permettant de stocker des valeurs de données, telles que des nombres ou des chaînes de caractères.

Par exemple, `x = 10` crée une variable nommée x avec la valeur 10. Python est un langage à typage dynamique, ce qui signifie que le type de la variable est déterminé au moment de l'exécution.

- Affectation (=).
- Python détermine automatiquement le type de données en fonction de la valeur attribuée.
- Ecraser les valeurs des variables en les réaffectant.
- Sensible à la casse : "X" et "x" sont deux variables différentes
- Imprimer plusieurs variables dans une seule instruction print avec (,)
- Plusieurs valeurs à plusieurs variables ou plusieurs variables à une seule valeur sur une seule ligne.
- Dénomination des variables incluent :
  - la casse camel : *testVariableCase*
  - la casse Pascal : *TestVariableCase*
  - la casse serpent : *test_variable_case*
- Evitez de:
  - commencer les noms de variables par des chiffres
  - utiliser des symboles tels que des tirets ou des espaces
  - mélanger des chaînes de caractères et des nombres entiers directement dans la concaténation.

Types natifs : Les types de données courants en Python incluent : - **Entiers** (`int`) : Nombres sans partie décimale. Exemple : `5`, `-3`. - **Nombres Complexes** (`complex`) : Nombres avec une partie réelle et une partie imaginaire. Exemple : `3 + 4j`. - **Flottants** (`float`) : Nombres avec une partie décimale. Exemple : `3.14`, `-0.001`. - **Chaîne de caractères** (`str`) : Une séquence de caractères, entourée de guillemets simples ou doubles. - **Booléens** (bool) : Représentent deux valeurs : `True` (Vrai) et `False` (Faux).. Le type booléen a seulement deux valeurs : True et False. Ils sont souvent le résultat d'expressions de comparaison ou de conditions logiques.

[1]:
```
nom = "Alice"
age = 25
taille = 1.70
```

```
estVrai = True
estFaux = False

print("la variable nom est de type :", type(nom))
print("la variable age est de type :",type(age))
print("la variable taille est de type :",type(taille))
print("la variable estVrai est de type :",type(estVrai))
print("la variable estFaux est de type :",type(estFaux))
```

```
la variable nom est de type : <class 'str'>
la variable age est de type : <class 'int'>
la variable taille est de type : <class 'float'>
la variable estVrai est de type : <class 'bool'>
la variable estFaux est de type : <class 'bool'>
```

Conversion de types

La conversion de types, souvent appelée **"casting"** en programmation, est un moyen de convertir une valeur d'un type donné en un autre type.

En Python, il existe plusieurs fonctions intégrées pour réaliser ces conversions entre les types de base comme les entiers (int), les chaînes de caractères (str), les booléens (bool), et les nombres à virgule flottante (float).

Voici des exemples de conversion entre ces types :

- Conversion en Entier (`int`)

[2]:
```
# Depuis une chaîne de caractères** :
chaine = "123"
nombre = int(chaine)  # Convertit la chaîne "123" en entier 123

## Depuis un booléen** :
vrai = True
entier = int(vrai)  # Convertit True en 1

## Depuis un float** :
flottant = 9.99
entier = int(flottant)  # Convertit 9.99 en 9 (troncature, pas d'arrondi)
```

- Conversion en Chaîne de Caractères (`str`)

[3]:
```
## Depuis un entier ou un float** :
nombre = 123
chaine = str(nombre)  # Convertit l'entier 123 en chaîne "123"

## Depuis un booléen** :
faux = False
chaine = str(faux)  # Convertit False en "False"
```

- Conversion en Booléen (`bool`)

[4]:
```
## Depuis une chaîne de caractères** :
chaine = ""
booleen = bool(chaine)  # Convertit une chaîne vide en False
chaine2 = "Python"
booleen2 = bool(chaine2)  # Convertit "Python" en True

## Depuis un nombre** :
nombre = 0
booleen = bool(nombre)  # Convertit 0 en False
nombre2 = 123
booleen2 = bool(nombre2)  # Convertit 123 en True
```

- Conversion en Float (`float`)

[5]:
```
## Depuis une chaîne de caractères** :
chaine = "123.45"
flottant = float(chaine)  # Convertit la chaîne "123.45" en float 123.45

## Depuis un entier** :
entier = 100
flottant = float(entier)  # Convertit l'entier 100 en float 100.0

## Depuis un booléen** :
vrai = True
flottant = float(vrai)  # Convertit True en 1.0
```

Affectation de valeurs différentes

[6]:
```
x, y, z = "Chocolat", "Vanille", "Fraise"
print(x)
print(y)
print(z)
```

```
Chocolat
Vanille
Fraise
```

Affectation de la même valeur

[7]:
```
x = y = z = "Chocolat"
print(x)
print(y)
print(z)
```

```
Chocolat
Chocolat
Chocolat
```

1.3.2 Opérations arithmétiques

Arithmétique : Addition (+), soustraction (-), multiplication (*), et division (/).

[8]:
```
somme = 10 + 5        # Résultat: 15
difference = 10 - 5   # Résultat: 5
produit = 10 * 5      # Résultat: 50
quotient = 10 / 5     # Résultat: 2.0
```

Modulo (%) retourne le reste de la division

Puissance (**) élève un nombre à une certaine puissance.

[9]:
```
reste = 10 % 3        # Résultat: 1
puissance = 2 ** 3    # Résultat: 8
```

1.3.3 Opérations sur les chaînes

La concaténation (+) combine des chaînes

La répétition (*) répète une chaîne un certain nombre de fois.

[10]:
```
message = "Bonjour" + " le monde"  # Résultat: "Bonjour le monde"
echo = "Ha" * 3                    # Résultat: "HaHaHa"
```

1.3.4 Opérateurs d'affectation composés

Les opérateurs d'affectation composés en Python sont des raccourcis qui **combinent une opération arithmétique ou binaire avec une affectation.**

En gros, ils vous permettent d'effectuer une opération sur une variable et de lui réaffecter le résultat en une seule étape.

[11]:
```
# Initialisation de x
x = 5
print(x)

# Ajouter et assigner
x += 2  # x est maintenant 7
print(x)

# Soustraire et assigner
x -= 3  # x est maintenant 4
print(x)

# Multiplier et assigner
x *= 4  # x est maintenant 16
print(x)

# Diviser et assigner
x /= 2  # x est maintenant 8
```

```python
print(x)

# Division entière et assigner
x //= 3  # x est maintenant 2
print(x)

# Modulo et assigner
x %= 3  # x est maintenant 2
print(x)

# Exponentielle et assigner
x **= 2  # x est maintenant 4
print(x)
```

```
5
7
4
16
8.0
2.0
2.0
4.0
```

1.3.5 2.2 Commentaires et affichage

Les **commentaires** sont utilisés pour expliquer le code en Python.

Ils sont très utiles pour vous-même et pour les autres personnes qui lisent votre code. En Python, les commentaires commencent par un # et s'étendent jusqu'à la fin de la ligne.

Les commentaires sont ignorés lors de l'exécution du code.

[12]:
```python
# Ceci est un commentaire simple en Python

x = 5  # Ceci est un commentaire suivant une instruction

# Vous pouvez utiliser les commentaires pour expliquer
# ce que votre code est censé faire :
y = x + 2  # Ajout de 2 à x et stockage du résultat dans y
```

Pour **l'affichage** en Python, la fonction **print()** est utilisée.

Elle permet d'afficher le texte, les nombres, et d'autres objets sur la console.

[13]:
```python
print("Bonjour, monde !")  # Affiche un message simple

nombre = 10
print(nombre)  # Affiche la valeur de la variable 'nombre'

# Vous pouvez aussi combiner du texte et des variables :
```

```
print("Le nombre est", nombre)  # Affiche "Le nombre est 10"
```

```
Bonjour, monde !
10
Le nombre est 10
```

Utilisation de f-string

[14]:
```
prenom = 'Marie'
age = 15
taille = 207

# Formatage de chaînes avec f-string pour une lisibilité améliorée :
print(f"Le nombre est {nombre}")
print(f"{prenom}, est partie")
```

```
Le nombre est 10
Marie, est partie
```

f-string avec Plusieurs Variables

[15]:
```
print(f"{prenom} mesure {taille} metres et a {age} ans")
```

```
Marie mesure 207 metres et a 15 ans
```

Utilisation de format

[16]:
```
print("{} a {} ans ".format(prenom, age))
```

```
Marie a 15 ans
```

- L'instruction précédente utilise la méthode `.format()` pour insérer les variables dans la chaîne de caractères.
- `{}` sont des placeholders (espaces réservés) qui seront remplacés par les arguments de `.format()` dans l'ordre.
- `ma_chaine` et `age` seront insérés respectivement dans le premier et le second `{}`.

Calcul et arrondi

- Le premier `print(prop)` affiche la valeur de `prop` avec tous ses chiffres après la virgule.
- Le second `print()` utilise une f-string avec un formatage spécifique `{prop:.2f}`.

Cela signifie que la valeur de `prop` sera formatée pour afficher seulement deux chiffres après la virgule.

[17]:
```
prop = (4500 + 2575) / 14800
print(prop)
print(f"La proportion de GC est {prop:.2f}")
```

```
0.4780405405405405
La proportion de GC est 0.48
```

1.3.6 2.3. Structures de contrôle

Les opérateurs Les opérateurs Python sont utilisés pour effectuer des opérations sur les variables et les valeurs.

Les opérateurs de comparaison en Python incluent égal à, différent de, supérieur à, inférieur à, supérieur ou égal à, et inférieur ou égal à.

Opérateur comparaison	Description
Egalité	==
Différent de	!=
Supérieur	>
Inférieur	<
Supérieur ou égale	>=
Inférieur ou égale	<=

- Renvoient soit Vrai, soit Faux, en fonction du résultat de la comparaison.

Les opérateurs logiques et, ou et non sont souvent combinés avec les opérateurs de comparaison.

Opérateur logique	Description	Résultat
And	and	True si les deux propositions sont vraies
Or	or	True si au moins une des propositions est Vraie
Not	not	inverse de l'instruction

Les opérateurs d'appartenance in et not in sont utilisés pour vérifier si une valeur ou une chaîne de caractères se trouve à l'intérieur d'une autre valeur, chaîne de caractères ou séquence.

Opérateur logique	Description	Résultat
In	in	True si la valeur spécifiée est présente dans l'objet.
Not in	not in	True si la valeur spécifiée n'est pas présente dans l'objet.

[18]:
```
# Exemples d'opérateurs de comparaison
print("Opérateurs de comparaison:")
print(5 == 5)  # Égalité, renvoie True
print(5 != 2)  # Différent de, renvoie True
print(5 > 3)   # Supérieur à, renvoie True
print(5 < 8)   # Inférieur à, renvoie True
print(5 >= 3)  # Supérieur ou égal à, renvoie True
print(5 <= 8)  # Inférieur ou égal à, renvoie True

# Exemples d'opérateurs logiques
print("\nOpérateurs logiques:")
print(True and False)  # and, renvoie False
```

```
print(True or False)   # or, renvoie True
print(not True)        # not, renvoie False

# Exemples d'opérateurs d'appartenance
print("\nOpérateurs d'appartenance:")
liste = [1, 2, 3, 4, 5]
print(3 in liste)      # in, renvoie True
print(6 not in liste)  # not in, renvoie True
```

```
Opérateurs de comparaison:
True
True
True
True
True
True

Opérateurs logiques:
False
True
False

Opérateurs d'appartenance:
True
True
```

Conditions

[19]:
```
# condition
x = 100

# Première condition : vérifie si x est inférieur à 10
if x < 10:
    print("x est inférieur à 10")
# Deuxième condition : vérifie si x est supérieur ou égal à 100
elif x >= 100:
    print("x est supérieur ou égal à 100")
# Dernière condition : si toutes les conditions précédentes sont fausses
else:
    print("x est supérieur à 10 mais inférieur à 100")
```

```
x est supérieur ou égal à 100
```

[20]:
```
# Initialisation des variables
x = 100
y = 30

# Première condition : vérifie si x est inférieur à 10 ET y est supérieur à 20
if (x < 10) and (y > 20):
```

```
    print("x est inférieur à 10 et y est supérieur à 20")
# Deuxième condition : vérifie si x est supérieur ou égal à 100 OU y est␣
 ↪supérieur ou égal à 30
elif x >= 100 or y >= 30:
    print("x est supérieur ou égal à 100 ou y est supérieur ou égal à 30")
# Dernière condition : si toutes les conditions précédentes sont fausses
else:
    print("x est supérieur à 10")
```

```
x est supérieur ou égal à 100 ou y est supérieur ou égal à 30
```

Les boucles **La boucle for** est utilisée pour parcourir des structures de données : liste, tuple, tableau, chaîne de caractères ou un dictionnaire.

> La boucle commence par examiner le premier élément de la séquence, effectue des actions dans son corps, puis passe à l'élément suivant jusqu'à ce que la séquence soit terminée.

- Boucle avec mot-clé **"for"**, une **variable temporaire** pour contenir chaque élément, le mot-clé **"in"** et la séquence à parcourir, suivis de **deux points**. Jusqu'au bout de l'élément
- Appliquer des **opérations sur la variable temporaire** dans le corps de la boucle pour effectuer diverses opérations.
- **Imbriquer des for"**
- Sur les *dictionnaires*, on peut **boucler sur les clés et les valeurs à l'aide de la méthode "items()"**.

[21]:
```
# Exemple de boucle for
for i in range(5):
    print(i)

fruits = ["pomme", "banane", "mangue", "cerise"]
for fruit in fruits:
    print(fruit)

print("Boucle for sur une liste:")
ma_liste = [1, 2, 3, 4, 5]
for element in ma_liste:
    print(element)

nombres = [1, 2, 3, 4, 5]
for nombre in nombres:
    if nombre % 2 == 0:
        print(f"{nombre} est pair")
    else:
        print(f"{nombre} est impair")
```

```
0
1
2
3
```

```
4
pomme
banane
mangue
cerise
Boucle for sur une liste:
1
2
3
4
5
1 est impair
2 est pair
3 est impair
4 est pair
5 est impair
```

La boucle while itère sur un bloc de code tant qu'une condition spécifiée est vraie. Contrairement aux boucles "for", les boucles "while" continuent l'itération tant que la condition reste vraie.

- **"break"** pour quitter prématurément une boucle "while", même si la condition est toujours vraie.
- **"else"** peut être utilisée avec une boucle "while" pour spécifier un bloc de code qui s'exécutera lorsque la condition de la boucle ne sera plus vraie.
- **"continue"** permet d'ignorer le code restant dans l'itération actuelle de la boucle et de passer à l'itération suivante.
- !! **Soyez prudent lorsque vous utilisez "continue" afin d'éviter de créer des boucles infinies.**

[22]:
```python
# Boucle while avec break
print("\nBoucle while avec break:")
i = 1
while i <= 5:
    print(i)
    if i == 3:
        print("Break à 3")
        break
    i += 1

# Boucle while avec continue
print("\nBoucle while avec continue:")
i = 0
while i < 5:
    i += 1
    if i == 3:
        continue
    print(i)
```

```python
# Boucle while avec else
print("\nBoucle while avec else:")
i = 1
while i <= 5:
    print(i)
    i += 1
else:
    print("La condition n'est plus vraie (i > 5)")

# Exemple de boucle while True
print("Boucle while True avec un mécanisme de sortie :")

compteur = 0
while True:
    compteur += 1
    print(compteur)

    # Interrompre la boucle si compteur atteint 5
    if compteur == 5:
        print("Compteur a atteint 5, sortir de la boucle.")
        break
```

```
Boucle while avec break:
1
2
3
Break à 3

Boucle while avec continue:
1
2
4
5

Boucle while avec else:
1
2
3
4
5
La condition n'est plus vraie (i > 5)
Boucle while True avec un mécanisme de sortie :
1
2
3
4
5
```

```
Compteur a atteint 5, sortir de la boucle.
```

1.3.7 2.4. Types de Données en Python (Structures Séquentielles)

Chaque type de données séquentielles a ses propres caractéristiques et utilisations spécifiques, offrant une grande flexibilité pour le traitement des données en Python. Les listes et les dictionnaires sont particulièrement populaires pour leur flexibilité et leurs capacités de stockage dynamique.

- **Chaînes de Caractères** (`str`) : Les chaînes de caractères (strings) en Python sont utilisées pour stocker et manipuler du texte. Elles sont définies en plaçant le texte entre guillemets simples (`'...'`), doubles (`"..."`) ou triples (`'''...'''` ou `"""..."""` pour les chaînes sur plusieurs lignes).
  - Séquences de caractères, indexées à partir de 0.
  - Peuvent être délimitées par des guillemets simples, doubles ou triples.
  - Exemple : `"Python"`, `'Data'`.
  - Caractères d'Échappement : pour inclure des caractères spéciaux comme des guillemets ou des retours à la ligne dans une chaîne, utilisez des caractères d'échappement (\).

  ```
  citation = "Elle a dit \"Bonjour!\""
  retour_ligne = "Ligne 1\nLigne 2"
  ```

  Les chaînes de caractères en Python sont extrêmement polyvalentes et fournissent une gamme complète de méthodes pour le traitement de texte. Que ce soit pour la manipulation de base ou des opérations complexes, les chaînes sont un outil indispensable en Python.

[23]:
```python
# Définition chaines de caractères
chaine_simple = 'Bonjour'
longueur = len(chaine_simple) # 7
print(f"{chaine_simple}, 'la longueur est {longueur}")

chaine_double = "Python"
chaine_multiligne = """Ceci est une
chaîne sur plusieurs
lignes."""
print(chaine_double,"\n", chaine_multiligne)

# Acces aux éléments
premier_char = chaine_simple[0]  # 'B'
sous_chaine = chaine_double[1:4] # 'yth'
print(premier_char, "et",  sous_chaine)

# Concaténation
salutation = chaine_simple + ", " + chaine_double # 'Bonjour, Python'
print(salutation)

# Répétition
echo = "echo " * 3  # 'echo echo echo '
print(echo)
```

```
Bonjour, 'la longueur est 7
Python
 Ceci est une
chaîne sur plusieurs
lignes.
B et yth
Bonjour, Python
echo echo echo
```

- **Méthodes pour les Chaînes de Caractères** : Les méthodes sont des fonctions qui sont associées à des objets spécifiques. Elles permettent d'exécuter des actions sur ces objets ou de manipuler leurs données internes. Voici quelques caractéristiques clés des méthodes :
  - **Association avec des Objets** : Contrairement aux fonctions indépendantes, les méthodes sont liées à des objets. Par exemple, les méthodes de chaînes de caractères (`str`) en Python sont conçues pour effectuer des opérations sur des chaînes de caractères spécifiques.
  - **Syntaxe** : En Python, une méthode est appelée en utilisant la notation pointée. Par exemple, `objet.methode()`.
  - **Méthodes de Chaînes de Caractères** : les chaînes de caractères disposent de méthodes telles que `upper()`, `lower()`, `replace()` pour convertir en majuscules, en minuscules ou remplacer une partie de la chaîne.
  - **Appel** : Pour appeler une méthode, utilisez généralement la syntaxe `objet.methode(arguments)`.

[24]:
```python
# Méthodes str.isupper() et str.islower()
majuscules = "PYTHON"
minuscules = "python"
is_upper = majuscules.isupper()
is_lower = minuscules.islower()
print(is_upper)  # Résultat : True
print(is_lower)  # Résultat : True

# Méthode str.upper()
texte_upper = "python est génial".upper()
print(texte_upper)  # Résultat : "PYTHON EST GÉNIAL"

# Méthode str.lower()
texte_lower = "Python est GÉNIAL".lower()
print(texte_lower)  # Résultat : "python est génial"

# Méthode str.capitalize()
texte_capitalize = "python est génial".capitalize()
print(texte_capitalize)  # Résultat : "Python est génial"

# Méthode str.title()
texte_title = "python est génial".title()
print(texte_title)  # Résultat : "Python Est Génial"
```

```python
# Méthode str.find(substring)
texte = "Python est génial"
position = texte.find("est")
print(position)  # Résultat : 7

# Méthode str.replace(old, new)
texte_replace = "Les pommes sont rouges.".replace("pommes", "bananes")
print(texte_replace)  # Résultat : "Les bananes sont rouges."

# Méthode str.split(separator)
texte_split = "apple,orange,banana".split(",")
print(texte_split)  # Résultat : ["apple", "orange", "banana"]

# Méthode str.join(iterable)
elements = ["apple", "orange", "banana"]
texte_join = ", ".join(elements)
print(texte_join)  # Résultat : "apple, orange, banana"

# Méthode str.strip()
texte_strip = "   Bonjour   ".strip()
print(texte_strip)  # Résultat : "Bonjour"

# Méthode str.startswith(prefix)
texte_startswith = "Bonjour, comment ça va ?".startswith("Bonjour")
print(texte_startswith)  # Résultat : True

# Méthode str.endswith(suffix)
texte_endswith = "Leçon terminée.".endswith("terminée.")
print(texte_endswith)  # Résultat : TrueS
```

```
True
True
PYTHON EST GÉNIAL
python est génial
Python est génial
Python Est Génial
7
Les bananes sont rouges.
['apple', 'orange', 'banana']
apple, orange, banana
Bonjour
True
True
```

Les listes Les listes en Python sont des structures de données qui permettent de stocker une série d'éléments. Elles sont flexibles, peuvent contenir des éléments de différents types et sont **mutables** (modifiables).

- **Listes** (`list`) :
  - Collections ordonnées de valeurs, pouvant contenir divers types de données.
  - Les éléments sont séparés par des virgules et entourés de crochets.
  - Mutables (modifiables).
  - Pour créer une liste, placez une série d'éléments séparés par des virgules entre crochets `[]`. Exemple : `[1, "a", 3.14]`.
  - L'accès aux éléments se fait par leur index, en commençant par `0`.
  - Les listes étant mutables, vous pouvez modifier leurs éléments.

[25]:
```python
# Définition de liste
ma_liste = [1, 2, 3, "Python", 3.14]
print(ma_liste)

premier_element = ma_liste[0]  # Accède au premier élément (1)
dernier_element = ma_liste[-1] # Accède au dernier élément (3.14)
print(premier_element,"\n", dernier_element)

 # Modifie le deuxième élément
ma_liste[1] = "deux"
print(ma_liste)

# Concaténation et Répétition
combinee = ma_liste + ["autre", "liste"]
repetee = [1, 2, 3] * 3
```

```
[1, 2, 3, 'Python', 3.14]
1
 3.14
[1, 'deux', 3, 'Python', 3.14]
```

- Les méthodes des listes : Chaque méthode offre une fonctionnalité unique pour manipuler des listes, rendant ces structures de données extrêmement flexibles et puissantes pour une variété de tâches en programmation Python.

Méthode	Description	Exemple d'Utilisation
`append()`	Ajoute un élément à la fin de la liste.	`liste.append(5)`
`extend()`	Étend la liste en ajoutant tous les éléments d'une autre liste.	`liste.extend([6, 7])`
`insert()`	Insère un élément à une position donnée.	`liste.insert(1, 'a')`
`remove()`	Supprime la première occurrence d'un élément.	`liste.remove('a')`
`pop()`	Supprime et renvoie un élément à une position donnée (par défaut, le dernier).	`liste.pop()`
`clear()`	Supprime tous les éléments de la liste.	`liste.clear()`
`index()`	Retourne l'indice du premier élément correspondant.	`liste.index('a')`
`count()`	Compte le nombre d'occurrences d'un élément spécifique.	`liste.count(5)`

Méthode	Description	Exemple d'Utilisation
`sort()`	Trie les éléments de la liste (dans un ordre spécifique).	`liste.sort()`
`reverse()`	Inverse l'ordre des éléments de la liste.	`liste.reverse()`
`copy()`	Retourne une copie superficielle de la liste.	`nouvelle_liste = liste.copy()`

[26]:
```python
# Création d'une liste de base
ma_liste = [1, 2, 3]

# Utilisation de append()
ma_liste.append(4)
print("Après append(4):", ma_liste)

# Utilisation de extend()
ma_liste.extend([5, 6])
print("Après extend([5, 6]):", ma_liste)

# Utilisation de insert()
ma_liste.insert(1, 'a')
print("Après insert(1, 'a'):", ma_liste)

# Utilisation de remove()
ma_liste.remove('a')
print("Après remove('a'):", ma_liste)

# Utilisation de pop()
element_supprime = ma_liste.pop()
print("Après pop():", ma_liste, ", Élément supprimé:", element_supprime)

# Utilisation de clear()
ma_liste.clear()
print("Après clear():", ma_liste)

# Recréation de la liste pour les autres méthodes
ma_liste = [3, 1, 4, 2, 2]

# Utilisation de index()
index_de_4 = ma_liste.index(4)
print("Index de 4:", index_de_4)

# Utilisation de count()
compte_de_2 = ma_liste.count(2)
print("Nombre d'occurrences de 2:", compte_de_2)

# Utilisation de sort()
```

```
ma_liste.sort()
print("Après sort():", ma_liste)

# Utilisation de reverse()
ma_liste.reverse()
print("Après reverse():", ma_liste)

# Utilisation de copy()
copie_de_ma_liste = ma_liste.copy()
print("Copie de la liste:", copie_de_ma_liste)
```

```
Après append(4): [1, 2, 3, 4]
Après extend([5, 6]): [1, 2, 3, 4, 5, 6]
Après insert(1, 'a'): [1, 'a', 2, 3, 4, 5, 6]
Après remove('a'): [1, 2, 3, 4, 5, 6]
Après pop(): [1, 2, 3, 4, 5] , Élément supprimé: 6
Après clear(): []
Index de 4: 2
Nombre d'occurrences de 2: 2
Après sort(): [1, 2, 2, 3, 4]
Après reverse(): [4, 3, 2, 2, 1]
Copie de la liste: [4, 3, 2, 2, 1]
```

- **Parcours de Liste** Le parcours de liste en Python consiste à accéder séquentiellement à chaque élément d'une liste. Cela peut être fait de différentes manières, mais les plus courantes sont les boucles `for` et `while`.
  - 1. *Boucle `for`* : La boucle `for` est la méthode la plus couramment utilisée pour parcourir une liste. Elle permet de traiter chaque élément individuellement.
  - 2. *`List Comprehension`* : La compréhension de liste est une méthode concise pour créer des listes en Python. Elle permet de transformer une liste en une autre liste, en filtrant les éléments pour former une liste des résultats d'une expression donnée.

La syntaxe de base d'une compréhension de liste est :

```
[nouvelle_expression for item in iterable if condition]
```

- `nouvelle_expression` est l'expression qui définit comment mapper les éléments de l'`iterable` (par exemple, une liste).
- `item` est la variable qui prend la valeur de chaque élément de l'`iterable` pendant chaque itération.
- `condition` est une condition optionnelle pour filtrer les éléments de l'`iterable`.

[27]:
```
# récupérer tous les fruits avec un "a"
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for f in fruits:
  if "a" in f:
```

```
    newlist.append(f)

print(newlist)
```

```
['apple', 'banana', 'mango']
```

[28]:
```
newlist = []
newlist = [f for f in fruits if "a" in f ]
print(newlist)
```

```
['apple', 'banana', 'mango']
```

[29]:
```
# Créer une liste des carrés des nombres de 0 à 9 :
carres = [x**2 for x in range(10)]
```

[30]:
```
# Filtrer les nombres pairs dans une liste :
nombres_pairs = [x for x in range(10) if x % 2 == 0]
```

[31]:
```
# Appliquer une fonction à tous les éléments :
noms_majuscules = [nom.upper() for nom in ["alice", "bob", "charlie"]]
```

- **Tuples** (`tuple`) :
  - Similaires aux listes, mais **immuables** (non modifiables après création).
  - Les éléments sont séparés par des virgules et entourés de parenthèses.
  - Exemple : `(1, "a", 3.14)`.

Avantage	Description
Sécurité des Données	Parfait pour protéger les données contre les modifications.
Performance	Plus rapides à parcourir que les listes.
Utilisation en tant que Clés de Dictionnaire	Peuvent être utilisés comme clés dans les dictionnaires, contrairement aux listes.
Retour de Plusieurs Valeurs de Fonction	Utilisés pour retourner plusieurs valeurs depuis une fonction.
Stockage de Données Constantes	Idéaux pour stocker des données qui ne doivent pas être modifiées.

[32]:
```
# Définition d'un tuple
mon_tuple = (1, "a", 3.14)

un_autre_tuple = 2, "b", 4.28

# Accès aux Éléments
premier_element = mon_tuple[0]  # 1

# Tuple à Un Élément
tuple_singleton = (5,)

# Imbrication
```

```
tuple_imbrique = (1, (2, 3), (4, 5))

# Méthodes Utiles
longueur = len(mon_tuple)  # Retourne la longueur de mon_tuple, ici 3
compteur = mon_tuple.count(1)  # Compte le nombre de fois que 1 apparaît dans␣
 ↪mon_tuple, ici 1
indice = mon_tuple.index("a")  # Trouve l'indice de "a" dans mon_tuple, ici 1
```

- **Ensembles** (`set`) :
  - Collections non ordonnées de valeurs uniques.
  - Utiles pour les opérations d'ensemble et la recherche de valeurs uniques.
  - Mutables, mais chaque élément doit être unique.
  - Non-Ordonnés : Les ensembles ne maintiennent pas l'ordre des éléments. Vous ne pouvez donc pas accéder aux éléments par un index.
  - Exemple : `{1, 2, 3}`.

Opérations d'Ensemble

Opération	Syntaxe	Description
Union	`a | b`	Retourne un nouvel ensemble contenant tous les éléments uniques des ensembles `a` et `b`.
Intersection	`a & b`	Retourne un nouvel ensemble contenant uniquement les éléments communs aux ensembles `a` et `b`.
Différence	`a - b`	Retourne un nouvel ensemble contenant les éléments de `a` qui ne sont pas dans `b`.
Différence Symétrique	`a ^ b`	Retourne un nouvel ensemble contenant tous les éléments qui sont dans `a` ou `b`, mais pas dans les deux.
Ajout	`a.add(x)`	Ajoute l'élément `x` à l'ensemble `a`.
Suppression	`a.remove(x)`	Supprime l'élément `x` de l'ensemble `a` ; lève une erreur si `x` n'est pas présent.
Suppression (sans erreur)	`a.discard(x)`	Supprime l'élément `x` de l'ensemble `a` si `x` est présent ; ne fait rien sinon.

[33]:
```
# Définition des ensembles
ensemble_a = {1, 2, 3, 4}
ensemble_b = {3, 4, 5, 6}

# Opérations d'ensemble
union = ensemble_a | ensemble_b
intersection = ensemble_a & ensemble_b
difference = ensemble_a - ensemble_b
difference_symetrique = ensemble_a ^ ensemble_b

# Affichage des résultats
print("Union :", union)
print("Intersection :", intersection)
print("Différence :", difference)
```

```
print("Différence Symétrique :", difference_symetrique)

# Ajout et suppression
ensemble_a.add(7)
print("Après ajout :", ensemble_a)
ensemble_a.remove(1)
print("Après suppression :", ensemble_a)
ensemble_a.discard(8)  # Ne fait rien car 8 n'est pas dans l'ensemble
```

```
Union : {1, 2, 3, 4, 5, 6}
Intersection : {3, 4}
Différence : {1, 2}
Différence Symétrique : {1, 2, 5, 6}
Après ajout : {1, 2, 3, 4, 7}
Après suppression : {2, 3, 4, 7}
```

- **Dictionnaires** (`dict`) :
  - Basés sur des paires clé-valeur.
  - Les valeurs sont accessibles via les clés.
  - Mutables, permettant les mises à jour et les suppressions.
  - Exemple : `{"nom": "Alice", "âge": 30}`.
  - Utilisation :
    - **Stockage d'Informations Structurées** : Parfait pour stocker des informations complexes comme des données utilisateur.
    - **Recherche Rapide** : Utilisez des dictionnaires pour des recherches rapides basées sur des clés uniques.

[34]:
```
# Définition d'un dictionnaire
personne = {"nom": "Alice", "age": 30}

autre_dict = dict(nom="Bob", age=25)

# Ajout et Mise à Jour
personne["profession"] = "Développeur"
```

[35]:
```
personne
```

[35]:
```
{'nom': 'Alice', 'age': 30, 'profession': 'Développeur'}
```

[36]:
```
# Utilisation de keys()
cles = personne.keys()
print("Clés:", cles)

# Utilisation de values()
valeurs = personne.values()
print("Valeurs:", valeurs)

# Utilisation de items()
```

```python
paires = personne.items()
print("Paires clé-valeur:", paires)

# Utilisation de get()
nom = personne.get("nom")  # Retourne 'Alice'
ville = personne.get("ville", "Non spécifiée")  # Retourne 'Non spécifiée' car␣
↪'ville' n'existe pas
print("Nom:", nom)
print("Ville:", ville)

# Parcours des clés
print("\nParcours des clés:")
for cle in personne:
    print(cle)

# Parcours des valeurs
print("\nParcours des valeurs:")
for valeur in personne.values():
    print(valeur)

# Parcours des paires clé-valeur
print("\nParcours des paires clé-valeur:")
for cle, valeur in personne.items():
    print(cle, ":", valeur)

print("\nParcours des paires clé-valeur avec enumerate:")
for index, (cle, valeur) in enumerate(personne.items()):
    print(f"Index {index}: Clé = {cle}, Valeur = {valeur}")
```

```
Clés: dict_keys(['nom', 'age', 'profession'])
Valeurs: dict_values(['Alice', 30, 'Développeur'])
Paires clé-valeur: dict_items([('nom', 'Alice'), ('age', 30), ('profession',
'Développeur')])
Nom: Alice
Ville: Non spécifiée

Parcours des clés:
nom
age
profession

Parcours des valeurs:
Alice
30
Développeur

Parcours des paires clé-valeur:
nom : Alice
```

```
age : 30
profession : Développeur

Parcours des paires clé-valeur avec enumerate:
Index 0: Clé = nom, Valeur = Alice
Index 1: Clé = age, Valeur = 30
Index 2: Clé = profession, Valeur = Développeur
```

[37]:
```python
profession = personne.pop("profession")
profession
```

[37]:
```
'Développeur'
```

[38]:
```python
# Suppression
del personne["age"]
```

[ ]:
```python

```