Programmation avec Python

Introduction à Python pour débutants

Historique : Python est un langage de programmation développé par **Guido van Rossum** au début des années 1990. Il est conçu pour être **facile à lire** et **simple à utiliser**, même pour ceux qui commencent à programmer.

Caractéristiques principales

- **Langage interprété** : Python exécute le code **ligne par ligne**.
- **Langage orienté objet** : En Python, chaque élément est considéré comme un **objet**.
  - Un objet est une entité qui combine :
    - **Des données** (*attributs*) : par exemple, le nom ou la taille d'un objet.
    - **Des fonctions** (*méthodes*) : des actions que l'objet peut effectuer.
- **Typage dynamique** : En Python, vous n'avez pas besoin de déclarer explicitement le type des variables (entiers, chaînes de caractères, etc.). Le type est **déterminé automatiquement** en fonction de la valeur attribuée.
  - Exemple :

```
1 x = 5          # Python comprend que c'est un entier (int).
2 y = "Hello"    # Python comprend que c'est une chaîne de caractères (str).
3 z = 3.14       # Python comprend que c'est un nombre décimal (float).
```

Grâce à ces caractéristiques, Python est largement utilisé dans des domaines variés tels que :

- Développement web,

- Analyse de données,
- Intelligence artificielle (IA),
- Automatisation de tâches.

Résumé des notions Python

1. Afficher du texte avec `print`: La commande `print` permet d'afficher du texte à l'écran.

```
print ("123")
```

Affiche :

```
123
```

2. Sauter des lignes dans un texte : Pour sauter une ligne, utilisez le caractère spécial `\n`.

```
print ("123\n456")
```

Affiche :

```
123
456
```

3. Afficher des guillemets ou apostrophes dans du texte : Pour inclure des guillemets ou une apostrophe dans une chaîne de caractères, utilisez un antislash (\).

```
print ("Texte \"entre guillemets\" et une apostrophe : s'affiche")
```

Affiche :

```
Texte "entre guillemets" et une apostrophe : s'affiche
```

Remarque 1. Vous pouvez utiliser des guillemets simples (') ou doubles (") pour entourer le texte.

```
print ("123")
print ('123')
```

4. Les commentaires : Les commentaires sont des lignes de texte qui ne s'exécutent pas, mais servent à ajouter des notes dans le code. Ils commencent par le caractère #.

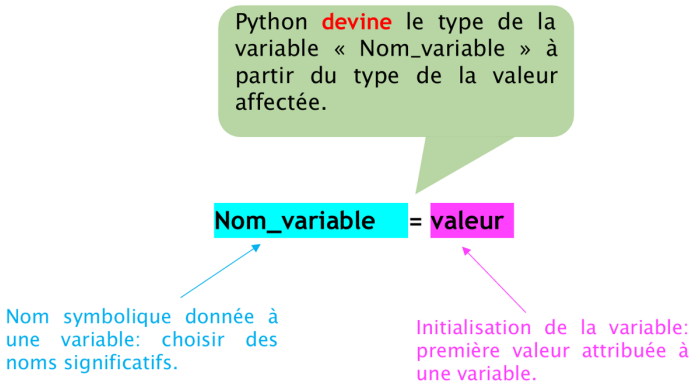
```
# Ceci est un commentaire
print ("phrase")  # Affiche le mot "phrase"
```

Les commentaires ne s'affichent pas lors de l'exécution du programme.

Les variables en Python

1. Définition d'une variable : Une **variable** permet de stocker une valeur (nombre : pour faire des opérations mathématiques (additions, calcul TVA, ..), Mot (chaîne de caractères) : pour former des phrases.) pour l'utiliser dans des calculs ou d'autres opérations.

```
x = 25
y = 17
print(x + y)  # Affiche : 42
```



2. Types de variables

Chaque variable a un **type**, qui définit la nature de sa valeur. Voici les principaux types en Python :

- **Integer (entier):** Utilisé pour les nombres entiers.

```
x = 2
y = 3
print(x * y)  # Affiche : 6
```

- **Float (nombre décimal):** Représente des nombres à virgule.

```
print(15 // 2)    # Affiche : 7 (division entière)
print(15. / 2)   # Affiche : 7.5 (résultat décimal)
```

- **String (texte):** Permet de stocker et manipuler du texte.

```
x = "texte"
y = x
print(y)  # Affiche : texte
```

4. Afficher des variables

Pour afficher plusieurs variables à la suite, utilisez une **virgule** (,) pour les séparer.

```
a = "Python"
b = "est"
c = "génial"
print(a, b, c)  # Affiche : Python est génial
```

Les opérateurs

Les opérateurs arithmétiques :

En Python, les opérateurs arithmétiques permettent d'effectuer des calculs sur des nombres. Voici les principaux :

Opérateur	Description	Exemple	Résultat
+	Addition	$5+3$	8
-	Soustraction	$5-3$	2
*	Multiplication	5×3	15
/	Division	$5\div 2$	2.5
//	Division entière	5//2	2
%	Modulo (reste de la division)	5%2	1
**	Puissance	5^2	25

Exemple:

```
x = 10
y = 5
print(x + y)  # Affiche : 15
print(x - y)  # Affiche : 5
print(6 * 3)  # Affiche : 18
print(7 / 2)  # Affiche : 3.5
print(7 // 2)  # Affiche : 3 (quotient entier)
print(7 % 2)  # Affiche : 1 (reste de la division)
print(2 ** 3)  # Affiche : 8 (2 au cube)
```

Remarque 2. Ces opérateurs peuvent être combinés dans une seule expression. Python respecte la priorité des opérateurs (ordre des opérations) :

- Parenthèses (( ))
- Exponentiation (**)
- Multiplication, division, division entière, modulo (*, /, //, %)
- Addition et soustraction (+, −)

Ces règles suivent l'ordre standard PEMDAS (Parentheses, Exponents, Multiplication/-Division, Addition/Subtraction).

Exemple :

```
result = (10 + 5) * 2 - 3
print(result)  # Affiche : 27
```

Les opérateurs arithmétiques composés en Python : Les **opérateurs arithmétiques composés** sont des opérateurs qui combinent une opération arithmétique avec une affectation de valeur. Ces opérateurs permettent de simplifier l'écriture de certaines expressions en effectuant une opération et une affectation en une seule étape.

Voici les principaux opérateurs arithmétiques composés en Python :

Opérateur	Description	Exemple	Résultat
+=	Addition avec affectation	$x+=3$	$x=x+3$
-=	Soustraction avec affectation	$x-=2$	$x=x-2$
*=	Multiplication avec affectation	$x*=4$	$x=x\times 4$
/=	Division avec affectation	$x/=2$	$x=x\div 2$
//=	Division entière avec affectation	$x//=3$	$x=x//3$
%=	Modulo avec affectation	$x\%=5$	$x=x\%5$
**=	Puissance avec affectation	$x**=2$	$x=x^2$

Exemple :

```
x = 5
x += 3  # x devient 8
x -= 4  # x devient 1
x *= 2  # x devient 25
x /= 5  # x devient 1
x %= 5  # x devient 0
```

Les opérateurs de comparaison en Python : Les **opérateurs de comparaison** en Python permettent de comparer deux valeurs et de renvoyer un résultat booléen (`True` ou `False`). Ils sont souvent utilisés dans les conditions (`if`, `while`) pour prendre des décisions en fonction des résultats des comparaisons.

Voici les principaux opérateurs de comparaison en Python :

Opérateur	Description	Exemple	Résultat
==	Égal à	$x == y$	`True` si x est égal à y, sinon `False`
!=	Différent de	$x \neq y$	`True` si x est différent de y, sinon `False`
>	Supérieur à	$x > y$	`True` si x est supérieur à y, sinon `False`
<	Inférieur à	$x < y$	`True` si x est inférieur à y, sinon `False`
>=	Supérieur ou égal à	$x \geq y$	`True` si x est supérieur ou égal à y, sinon `False`
<=	Inférieur ou égal à	$x \leq y$	`True` si x est inférieur ou égal à y, sinon `False`

Exemple :

```
x = 5
y = 5
z=3
print(x == y)  # Affiche : True
print(z != y)  # Affiche : True
print(x > y)  # Affiche : False
print(x >= y)  # Affiche : True
```

Les opérateurs logiques en Python : Les **opérateurs logiques** en Python sont utilisés pour combiner plusieurs expressions booléennes.

Opérateur	Description	Exemple	Résultat
`and`	Vrai si les deux conditions sont vraies	$x > 0$ `and` $y < 10$	`True` si $x > 0$ et $y < 10$, sinon `False`
`or`	Vrai si au moins une des conditions est vraie	$x > 0$ `or` $y < 10$	`True` si $x > 0$ ou $y < 10$, sinon `False`
`not`	Inverse la valeur booléenne de l'expression	`not(x > 0)`	`True` si $x \leq 0$, sinon `False`

Exemples :

Exemple 1 : Vérifie si les deux conditions sont vraies.

```
x = 5
y = 3
```

```
3 if x > 0 and y < 10:
4     print("Les deux conditions sont vraies")
```

Résultat : Affiche "Les deux conditions sont vraies" car $x > 0$ et $y < 10$.

Exemple 2 : Vérifie si au moins une des conditions est vraie.

```
1 x = 5
2 y = 15
3 if x > 0 or y < 10:
4     print("Au moins une condition est vraie")
```

Résultat : Affiche "Au moins une condition est vraie" car $x > 0$ est vrai, même si y n'est pas inférieur à 10.

Exemple 3 : Inverse la valeur booléenne de l'expression.

```
1 x = -5
2 if not(x > 0):
3     print("x n'est pas positif")
```

Résultat : Affiche "x n'est pas positif" car x n'est pas supérieur à 0, donc l'opérateur `not` inverse la condition.

Les opérateurs d'appartenance : Les **opérateurs d'appartenance** en Python sont utilisés pour tester si un élément appartient ou non à une séquence (comme une liste, une chaîne de caractères, un tuple, etc.). Les deux opérateurs principaux sont :

- **in** : Vérifie si un élément existe dans une séquence (liste, chaîne, tuple, etc.). Renvoie `True` si l'élément est présent, sinon `False`.
- **not in** : Vérifie si un élément n'existe pas dans une séquence. Renvoie `True` si l'élément est absent, sinon `False`.

Exemples :

Exemple 1 : Vérifiez si un élément appartient à une séquence.

```python
fruits = ["pomme", "banane", "cerise"]
if "banane" in fruits:
    print("La banane est dans la liste des fruits.")
```

Résultat : Affiche "La banane est dans la liste des fruits" car "banane" est un élément de la liste `fruits`.

Exemple 2 : Vérifiez si un élément n'appartient pas à une séquence. appartient à une séquence.

```python
fruits = ["pomme", "banane", "cerise"]
if "orange" not in fruits:
    print("L'orange n'est pas dans la liste des fruits.")
```

Résultat : Affiche "L'orange n'est pas dans la liste des fruits" car "orange" n'est pas un élément de la liste `fruits`.

Les opérateurs d'identité : Les **opérateurs d'identité** en Python sont utilisés pour comparer deux objets afin de vérifier s'ils pointent vers la même zone mémoire, c'est-à-dire s'ils sont identiques en termes d'identité (et non seulement en termes de valeur). Les deux opérateurs principaux sont :

- `is` : Vérifie si deux objets font référence au même objet (même identité). Renvoie `True` s'ils sont identiques, sinon `False`.
- `is not` : Vérifie si deux objets ne font pas référence au même objet (identités différentes). Renvoie `True` s'ils sont différents, sinon `False`.

Exemples :

Exemple 1 : Vérifiez si deux variables pointent vers le même objet.

```python
a = [1, 2, 3]
b = a
c = [1, 2, 3]
print(a is b)  # True, car b est une référence à a
print(a is c)  # False, car c est une nouvelle liste avec les mêmes valeurs
```

Exemple 2 : vérifiez si deux variables pointent vers des objets différents.

```
x = 10
y = 20
print(x is not y)  # True, car x et y pointent vers des objets différents.
```

Remarque 3.

- L'opérateur `==` compare les valeurs des objets.
- Les opérateurs `is` et `is not` comparent les identités des objets (leur emplacement en mémoire).

Exemple :

```
a = [1, 2, 3]
b = [1, 2, 3]
print(a == b)  # True, car les valeurs sont égales
print(a is b)  # False, car ce sont deux objets différents en mémoire
```

La concaténation

La **concaténation** en Python est une opération qui permet d'assembler des chaînes de caractères (*strings*) en une seule chaîne. Cette opération est courante lorsqu'on travaille avec des données textuelles.

Opérations principales de concaténation :

1. **Utilisation de l'opérateur +** : Permet d'assembler deux ou plusieurs chaînes de caractères.

```
str1 = "Bonjour"
str2 = "le monde"
result = str1 + " " + str2
print(result)  # Affiche : Bonjour le monde
```

2. **Utilisation de l'opérateur *** : Permet de répéter une chaîne plusieurs fois.

```python
 str1 = "Python"
result = str1 * 3
print(result)  # Affiche : PythonPythonPython
```

Points importants :

- Les chaînes doivent être du même type pour être concaténées avec +. Si vous essayez de concaténer une chaîne et un autre type (par exemple, un entier), une erreur sera levée. Pour éviter cela, vous pouvez convertir l'autre type en chaîne à l'aide de `str()`.

```python
age = 25
message = "J'ai " + str(age) + " ans."
print(message)  # Affiche : J'ai 25 ans.
```

- La concaténation ne modifie pas les chaînes existantes, car elles sont immuables. Elle crée une nouvelle chaîne.

Alternatives modernes : Pour assembler des chaînes de caractères plus efficacement ou avec des variables, on peut utiliser :

- **Les f-strings** (Python 3.6 et plus) :

```python
nom = "Alice"
age = 30
print(f"Je m'appelle {nom} et j'ai {age} ans.")
# Affiche : Je m'appelle Alice et j'ai 30 ans.
```

- **La méthode `join`** pour concaténer une liste de chaînes :

```python
mots = ["Bonjour", "le", "monde"]
result = " ".join(mots)
print(result)  # Affiche : Bonjour le monde
```

Remarque 4.

- L'opérateur $*$ répète la chaîne exactement comme elle est, sans ajouter de séparateurs.
- La méthode `join()` permet de spécifier un séparateur (comme un espace `" "`) entre les éléments répétés.

Exemple :

```
mot = "Python"
result = " ".join([mot] * 3)
print(result)#Cela affichera : Python Python Python
```

Le casting en Python

En Python, le **casting** est une opération qui permet de convertir une variable d'un type à un autre. Cela est utile pour manipuler des données de différents types dans des opérations ou des fonctions qui nécessitent un type spécifique.

Principales fonctions de casting

1. `int()` : Convertit en un entier (*integer*).

```
x = "10"
y = int(x)  # Convertit "10" en 10
print(y + 5)  # Affiche : 15
```

2. `float()` : Convertit en un nombre à virgule flottante (*float*).

```
x = "3.14"
y = float(x)  # Convertit "3.14" en 3.14
print(y + 1.86)  # Affiche : 5.0
```

3. `str()` : Convertit en une chaîne de caractères (*string*).

```python
x = 42
y = str(x)  # Convertit 42 en "42"
print("Le nombre est : " + y)  # Affiche : Le nombre est : 42
```

4. `bool()` : Convertit en une valeur booléenne (`True` ou `False`).

```python
print(bool(0))  # Affiche : False
print(bool(1))  # Affiche : True
print(bool(""))  # Affiche : False
print(bool("Hello")) # True : chaîne non vide est true
print(bool([]))     # False : liste vide est false
print(bool([1, 2])) # True : liste non vide est true
```

Entrées / Sorties en Python

En Python, les entrées et sorties (I/O) sont des opérations essentielles permettant d'interagir avec l'utilisateur et d'afficher ou de récupérer des données. Voici un résumé des principales méthodes d'entrée et de sortie en Python.

1. Entrée de données : `input()` La fonction `input()` permet de récupérer une entrée de l'utilisateur sous forme de chaîne de caractères. L'utilisateur tape sa réponse au clavier, et cette réponse est ensuite renvoyée sous forme de texte.

Syntaxe :

```python
variable = input("Message d'invite : ")
```

Exemple :

```python
nom = input("Quel est votre nom ? ")
print("Bonjour, " + nom)
```

Par défaut, `input()` retourne une chaîne de caractères. Si l'on souhaite récupérer des données numériques, il faut convertir la chaîne obtenue avec `int()` ou `float()`.

Exemple avec conversion :

```
age = int(input("Quel âge avez-vous ? "))
print("Vous avez " + str(age) + " ans.")#str() est utilisée pour convertir l'entier age
#en une chaîne de caractères, car print() attend des chaînes de caractères
#pour pouvoir les afficher.
```

2. Sortie de données : `print()` : La fonction `print()` est utilisée pour afficher des informations à l'utilisateur. Elle peut afficher des chaînes de caractères, des résultats de calculs, ou des valeurs de variables.

Syntaxe :

```
print(valeur)
```

Exemple :

```
print("Bonjour tout le monde!")
```

Affichage de plusieurs valeurs :

```
x = 5
y = 10
print("La somme de", x, "et", y, "est", x + y)
```

Par défaut, `print()` sépare les éléments avec un espace. On peut personnaliser ce séparateur avec le paramètre `sep` :

```
print("a", "b", "c", sep="-")  # Affiche a-b-c
```

Affichage avec saut de ligne : Le `print()` ajoute un saut de ligne après chaque appel. Si l'on ne souhaite pas cela, on peut utiliser l'argument `end` :

```
print("Bonjour", end=" ")
print("tout le monde!")
```

Structures de contrôle

Les structures de contrôle permettent de modifier l'exécution d'un programme en fonction de certaines conditions ou de répéter des actions plusieurs fois. En Python, on trouve principalement des structures conditionnelles, des boucles et des instructions de contrôle de flux.

1. Les structures conditionnelles : Les structures conditionnelles permettent de choisir parmi plusieurs options en fonction de conditions.

1.1. `if`

La structure `if` permet d'exécuter un bloc de code si une condition est vraie.

Syntaxe :

```
if condition:
    # instructions si la condition est vraie
```

1.2. `else`

La structure `else` permet d'exécuter un bloc de code lorsque la condition de l'`if` est fausse.

Syntaxe :

```
if condition:
    # instructions si la condition est vraie
else:
    # instructions si la condition est fausse
```

1.3. `elif`

La structure `elif` permet de tester plusieurs conditions si l'`if` initial est faux.

Syntaxe :

```
if condition1:
    # instructions si condition1 est vraie
elif condition2:
    # instructions si condition2 est vraie
```

```
else:
    # instructions si aucune des conditions n'est vraie
```

Exemple :

```
x = 10
if x > 0:
    print("x est positif")
elif x == 0:
    print("x est nul")
else:
    print("x est négatif")
```

2. Les boucles : Les boucles permettent d'exécuter un bloc de code plusieurs fois.

2.1. for

La structure `for` permet d'itérer sur une séquence (comme une liste, une chaîne de caractères, etc.).

Syntaxe :

```
for element in sequence:
    # instructions à répéter pour chaque élément
```

Exemple :

```
for i in range(5):
    print(i)
```

Cela affiche les nombres de 0 à 4.

Utilisation avec des chaînes de caractères : Une boucle `for` peut aussi être utilisée pour itérer sur chaque caractère d'une chaîne de caractères.

```
for char in "Python":
    print(char)
```

Cela affiche :

```
P
y
t
h
o
n
```

Différentes formes de `range()` :

La fonction `range()` est utilisée pour générer des séquences d'entiers dans les boucles `for`. Elle peut être utilisée sous plusieurs formes selon les besoins du programme.

1. Forme de base : `range(n)`: Cette forme génère une séquence d'entiers allant de 0 à $n-1$.

```
1 for i in range(5):
2     print(i)
```

Cela affiche :

```
0
1
2
3
4
```

2. `range(start, stop)`: Cette forme génère une séquence allant de `start` à $stop-1$.

```
1 for i in range(2, 6):
2     print(i)
```

Cela affiche :

```
2
3
4
5
```

3. `range(start, stop, step)`: Cette forme permet de spécifier un pas (`step`) entre les éléments de la séquence.

```
for i in range(0, 10, 2):
    print(i)
```

Cela affiche :

```
0
2
4
6
8
```

4. `range(start, stop, -step)`: Si un pas négatif est utilisé, la séquence devient décroissante.

```
for i in range(10, 0, -2):
    print(i)
```

Cela affiche :

```
10
8
6
4
2
```

2.2. `while`

La structure `while` répète un bloc de code tant qu'une condition est vraie.

Syntaxe :

```
while condition:
    # instructions à répéter tant que la condition est vraie
```

Exemple :

```
i = 0
while i < 5:
    print(i)
    i += 1
```

Cela affiche les nombres de 0 à 4.

3. Les instructions de contrôle de flux

Ces instructions permettent de manipuler le déroulement de l'exécution du programme.

3.1. `break`

L'instruction `break` permet d'interrompre une boucle et d'arrêter son exécution, même si la condition n'est pas remplie. **Exemple** :

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

3.2. `continue`

L'instruction `continue` permet de sauter une itération dans une boucle et de passer directement à la suivante. **Exemple** :

```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

3.3. pass

L'instruction pass permet de définir un bloc vide, qui ne fait rien mais qui est nécessaire pour la syntaxe (par exemple, dans des structures conditionnelles ou des boucles vides).
Exemple :

```
if x < 0:
    pass  # Le code n'exécute rien ici
```

Les Listes en Python

Les **listes** en Python sont des structures de données très utilisées permettant de stocker des éléments dans un ordre spécifique. Voici un résumé des principales caractéristiques et opérations liées aux listes en Python.

1. Création d'une Liste: Une liste est créée en utilisant des crochets `[ ]` et en séparant les éléments par des virgules. Les éléments peuvent être de différents types (entiers, chaînes de caractères, booléens, etc.).

- Liste d'entiers : `ma_liste = [1, 2, 3, 4]`
- Liste mixte : `ma_liste_mixte = [1, "Python", True, 3.14]`

2. Accès aux Éléments : Les éléments d'une liste sont indexés, avec l'index 0 pour le premier élément. Vous pouvez accéder à un élément en utilisant son index.

- Accéder au premier élément : `ma_liste[0]` → 1
- Accéder au dernier élément : `ma_liste[-1]` → 4

3. Modification des Éléments : Vous pouvez modifier un élément en accédant à son index et en lui attribuant une nouvelle valeur.

```
ma_liste[1] = 10
```

Cela modifie le deuxième élément (index 1). La liste devient `[1, 10, 3, 4]`.

4. Ajout d'Éléments : Il existe plusieurs méthodes pour ajouter des éléments dans une liste :

- `append()` : Ajoute un élément à la fin de la liste.
- `insert()` : Insère un élément à une position spécifique.
- `extend()` : Ajoute les éléments d'une autre liste à la fin de la liste actuelle.

Exemple :

```
ma_liste.append(5)    # Ajouter un élément à la fin
ma_liste.insert(1, 8) # Insérer un élément à l'index 1
ma_liste.extend([6, 7]) # Ajouter plusieurs éléments
```

La liste devient `[1, 8, 10, 3, 4, 5, 6, 7]`.

5. Suppression d'Éléments : Il existe plusieurs méthodes pour supprimer des éléments :

- `remove()` : Supprime la première occurrence d'un élément spécifique.
- `pop()` : Supprime et retourne l'élément à une position donnée (par défaut, le dernier élément).
- `clear()` : Supprime tous les éléments de la liste.

Exemple :

```
ma_liste.remove(10)  # Supprimer la première occurrence de 10
element = ma_liste.pop(2) # Supprimer et retourner l'élément à l'index 2
ma_liste.clear()   #Supprimer tous les éléments
```

6. Longueur d'une Liste : La fonction `len()` permet de connaître la taille d'une liste, c'est-à-dire le nombre d'éléments qu'elle contient.

Exemple :

```
len(ma_liste)  #0
```

7. Tris et Inversions : Vous pouvez trier ou inverser les éléments d'une liste :

- `sort()` : Trie la liste par ordre croissant (modifie la liste originale).
- `reverse()` : Inverse l'ordre des éléments (modifie la liste originale).
- `sorted()` : Renvoie une nouvelle liste triée sans modifier l'originale.

Exemple :

```
ma_liste = [4, 1, 3, 2]
ma_liste.sort()   # Trie la liste par ordre croissant
ma_liste.reverse() # Inverse l'ordre des éléments
```

La liste devient `[4, 3, 2, 1]`.

8. Listes comprises : Les **listes comprises** sont une manière concise de créer des listes en appliquant une expression à chaque élément d'un itérable.

Exemple :

```
carrés = [x**2 for x in range(1, 6)]
```

Cela crée une liste des carrés des nombres de 1 à 5 : `[1, 4, 9, 16, 25]`.

9. Slicing (Découpage) de Listes : Vous pouvez extraire une sous-liste (slice) en utilisant la syntaxe `[start:end]`, où `start` est l'index de début et `end` est l'index de fin (non inclus).

Exemple :

```
ma_liste = [1, 2, 3, 4, 5]
sous_liste = ma_liste[1:4]  # Obtenir une sous-liste des éléments de l'index 1 à 3
#(non inclus)
```

Cela donne `[2, 3, 4]`. Voici plusieurs exemples d'utilisation des slices:

1. Slice avec des indices spécifiques `[n : p]`: Cela retourne une sous-liste contenant les éléments de `nom_liste` de l'indice `n` à l'indice `p-1`.

2. Slice jusqu'à un indice `[: p]` : Cela retourne tous les éléments de la liste depuis l'indice 0 jusqu'à l'indice `p-1`.

3. Slice à partir d'un indice `[p :]` : Cela retourne tous les éléments de la liste à partir de l'indice `p` jusqu'à la fin de la liste.

10. Opérations sur les Listes :

- **Concatenation** : Vous pouvez concaténer des listes en utilisant l'opérateur +.
- **Multiplication** : Vous pouvez multiplier une liste pour la répéter plusieurs fois en utilisant l'opérateur *.

Exemple :

```
liste1 = [1, 2]
liste2 = [3, 4]
liste_concatenee = liste1 + liste2  # [1, 2, 3, 4]
liste_repeatee = [1, 2] * 3  # [1, 2, 1, 2, 1, 2]
```

Remarque 5. En Python, les indices négatifs sont utilisés pour accéder aux éléments d'une liste, d'une chaîne de caractères ou d'autres séquences en comptant à partir de la fin plutôt que du début. Les indices négatifs sont particulièrement utiles lorsque vous souhaitez accéder rapidement aux éléments à la fin d'une séquence sans avoir à connaître sa longueur exacte.

Exemple d'utilisation :

- **Exemple 1**

```
ma_liste = [10, 20, 30, 40, 50]

# Accéder aux éléments par indices négatifs
print(ma_liste[-1])  # 50
print(ma_liste[-2])  # 40
print(ma_liste[-3])  # 30
```

- **Exemple 2**

```
fruits = ["pomme", "banane", "orange", "raisin"]
# Accéder aux éléments par indices négatifs
print(fruits[-1])  # Affiche 'raisin', dernier fruit
print(fruits[-2])  # Affiche 'orange', avant-dernier fruit
print(fruits[-3])  # Affiche 'banane'
print(fruits[-4])  # Affiche 'pomme'
```

11. Parcourir une liste via les indices : Il existe plusieurs façons de parcourir une liste en utilisant ses indices. Voici deux méthodes courantes :

1. Utiliser une boucle `for` avec `range()`:

```
nom_liste = ["pomme", "banane", "orange", "raisin"]

# Parcourir la liste via les indices
for indice in range(len(nom_liste)):
print(f"Indice {indice} : {nom_liste[indice]}")
```

2. Utiliser l'opérateur `in` pour vérifier la présence d'un élément dans la liste :

```
nom_liste = ["pomme", "banane", "orange", "raisin"]
recherche = "orange"
if recherche in nom_liste:
    indice = nom_liste.index(recherche)  # Trouver l'indice de l'élément
    print(f"L'élément {recherche} est à l'indice {indice}")
else:
    print(f"L'élément {recherche} n'est pas dans la liste")
```

Les Fonctions en Python

Les fonctions en Python sont un moyen d'encapsuler du code afin de le réutiliser, de rendre le programme plus lisible et plus modulaire. Voici un résumé des principaux concepts liés aux fonctions.

1. Définir une fonction : Une fonction est définie avec le mot-clé `def`, suivi du nom de la fonction et de paramètres éventuels entre parenthèses.

```
def ma_fonction():
    print("Bonjour, c'est ma fonction !")
```

2. Appeler une fonction : Une fonction est appelée en utilisant son nom suivi de parenthèses.

```
ma_fonction()  # Affiche : Bonjour, c'est ma fonction !
```

3. Fonction avec des paramètres : Les fonctions peuvent accepter des paramètres qui permettent de passer des valeurs à la fonction.

```
def saluer(nom):
    print(f"Bonjour, {nom}!")

saluer("Alice")  # Affiche : Bonjour, Alice!
```

4. Valeur de retour avec `return` : Une fonction peut retourner une valeur avec `return`.

```
def addition(a, b):
    return a + b

resultat = addition(3, 4)
print(resultat)  # Affiche : 7
```

5. Valeurs par défaut pour les paramètres : Une fonction peut avoir des valeurs par défaut pour certains paramètres.

```
def saluer(nom="Inconnu"):
    print(f"Bonjour, {nom}!")

```

```
saluer()        # Affiche : Bonjour, Inconnu!
saluer("Bob")   # Affiche : Bonjour, Bob!
```

6. Arguments positionnels et nommés : Les arguments peuvent être positionnels ou nommés.

```
def afficher_info(nom, age):
    print(f"Nom: {nom}, Âge: {age}")

afficher_info(age=30, nom="Alice")  # Arguments nommés
```

7. Fonctions lambda : Les fonctions lambda sont des fonctions anonymes (c'est-à-dire une fonction sans nom.), définies sans `def`.

```
addition = lambda a, b: a + b
print(addition(5, 7))  # Affiche : 12
```

8. Fonctions récursives : Les fonctions récursives s'appellent elles-mêmes.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))  # Affiche : 120
```

9. Fonctions avec un nombre variable d'arguments : On peut définir des fonctions qui acceptent un nombre variable d'arguments à l'aide de `*args`.

```
def somme(*args):
    return sum(args)

print(somme(1, 2, 3))  # Affiche : 6
```

Structure générale d'un programme Python

Un programme Python type a la structure ci-contre :

```
#####################################
#####  Programme Python type ########
#####  Auteur : .....        ########
#####  Version : .....       ########
#####################################

#####################################
# importation des fonctions externes

from math import sqrt

#####################################
# Déclaration des fonctions utilisateur

def racine (x):
    res = sqrt(x)
    return res

#####################################
# Corps du programme principal

if __name__ == '__main__':
    print("Saisir une valeur :")
    val = int(input())
    racine_val = racine(val)
    print("La racine de {} est {}".format(val, racine_val))
```

Pour une meilleure organisation, il est également possible de regrouper les fonctions par thèmes dans des modules (fichiers) séparés, et les importer pour les utiliser dans le programme principal.

Programmation fonctionnelle

Portée des variables :

1. Lorsqu'une fonction est définie, ses paramètres ainsi que toutes les variables qu'elle utilise sont locaux. Analysons l'exemple suivant :

Main.py

```
#####################################
# importation des fonctions externes

from math import sqrt
from monfichier import *

#####################################
# Corps du programme principal

if __name__ == '__main__':
    print("Saisir une valeur :")
    val = int(input())
    racine_val = racine(val)
    print("La racine de {} est {}".format(val, racine_val))
```

monfichier.py

```
#####################################
# Déclaration des fonctions utilisateur

def racine (x):
    res = sqrt(x)
    return res
```

```
1  # Initialisation d'une variable a
2  a = 5
3
4  # Définition d'une fonction après la déclaration de a
5  def print_a():
6      print(" a = ", a)
7
8  # 1er appel de la fonction
9  print_a() # Affiche " a =  5"
10
11 # Modification de la valeur de a
12 a = 8
13
14 # 2ème appel de la fonction
15 print_a()# Affiche " a =  8"
```

En Python :

- Les variables définies à l'intérieur de la fonction sont **locales** et ne peuvent pas être utilisées à l'extérieur de la fonction, sauf si elles sont explicitement renvoyées ou modifiées via des paramètres.
- Une fonction peut accéder à une variable **globale** si elle n'est pas redéfinie localement dans la fonction.

Dans l'exemple ci-dessus, la fonction `print_a()` utilise la variable globale a qui peut être modifiée en dehors de la fonction. Cette modification affecte les appels sub-

séquents à la fonction.

2. Essayons de modifier la valeur de *a* dans la fonction `print_a()`.

```
# Initialisation d'une variable a
a = 5

# Définition d'une fonction après la déclaration de a
def print_a():
    a = a * 2
    print(" a = ", a)

# Appel de la fonction
print_a()
```

L'exécution de ce code Python renverra une erreur `UnboundLocalError`, car Python considère que a est une variable locale à la fonction (en raison de la tentative de modification de sa valeur). Cela montre que Python ne permet pas de modifier une variable globale à l'intérieur d'une fonction sans utiliser le mot-clé `global`.

```
# Initialisation d'une variable a
a = 5

# Définition d'une fonction après la déclaration de a
def print_a():
    global a #Grâce au mot clé « global », la fonction comprend que « a » est une
#variable globale
    a = a * 2
    print(" a = ", a)

# Appel de la fonction
print_a()
```

3. Essayons de modifier la valeur de *a* dans la fonction `interne()` définie dans la fonction `externe()`. Pour que la fonction puisse modifier la variable, il faut utiliser le mot-clé `nonlocal` devant la variable `a`.

```python
def externe():
    a = 5  # Variable locale de la fonction externe

    # Définition d'une fonction interne
    def interne():
        nonlocal a  # Indique que la variable 'a' est celle de la fonction externe
        a = a * 2
        print(" a = ", a)

    interne()
    print(" a = ", a)

externe()
```

Les clés de dictionnaire en Python

En Python, un **dictionnaire** (`dict`) est une structure de données qui permet de stocker des paires de valeurs sous la forme de **clé** et **valeur**.

Clé et valeur

- **Clé** : Une clé est un identifiant unique qui est utilisé pour accéder à la valeur associée dans le dictionnaire.
- **Valeur** : La valeur est l'information qui est associée à la clé dans le dictionnaire.

Exemple :

```python
actions = {
    'Apple': {'prix': 150, 'volume': 1000000, 'rendement': 0.02},
    'Tesla': {'prix': 750, 'volume': 900000, 'rendement': 0.015},
    'Amazon': {'prix': 3500, 'volume': 1300000, 'rendement': 0.01},
    'Microsoft': {'prix': 300, 'volume': 1200000, 'rendement': 0.025},
    'Google': {'prix': 2800, 'volume': 1500000, 'rendement': 0.03}
}
```

Dans ce dictionnaire, chaque entreprise est une **clé** (par exemple, ''Apple'', ''Tesla'', etc.), et les valeurs associées sont des sous-dictionnaires contenant :

- **prix** : Le prix de l'action de l'entreprise.
- **volume** : Le nombre d'actions échangées.
- **rendement** : Le rendement journalier de l'action, exprimé en pourcentage.

Accéder aux informations : Voici comment accéder aux informations d'une entreprise spécifique, par exemple `Apple` :

```
# Accéder au prix de l'action de Apple
print(actions['Apple']['prix'])  # Affiche 150

# Accéder au volume d'actions échangées pour Tesla
print(actions['Tesla']['volume'])  # Affiche 900000

# Accéder au rendement journalier de Microsoft
print(actions['Microsoft']['rendement'])  # Affiche 0.025
```

La méthode `data.keys()` dans les dictionnaires

En Python, la méthode `data.keys()` est utilisée pour obtenir toutes les **clés** d'un dictionnaire. Elle renvoie un objet de type `dict_keys` qui contient toutes les clés du dictionnaire, sans renvoyer les valeurs associées. Prenons l'exemple d'un dictionnaire contenant des informations financières ci-dessus:

```
# Afficher toutes les clés du dictionnaire 'actions'
cles = actions.keys()

# Afficher les clés
print(cles)
```

Résultat : Le résultat sera un objet de type `dict_keys` contenant toutes les clés du dictionnaire :

```
dict_keys(['Apple', 'Tesla', 'Amazon', 'Microsoft', 'Google'])
```