

# **PYTHON**

**Maîtriser les fondamentaux de Python.**

**Utiliser ce langage pour aborder des problématiques avancées de programmation (performance, qualité du code)**

**Cours – partie 1**

**Jean-Christophe RANGON**

**2024**



**email:** [jc.rangon.formateur@gmail.com](mailto:jc.rangon.formateur@gmail.com)

**linkedIn:** [www.linkedin.com/in/jean-christophe-rangon-dev-web](https://www.linkedin.com/in/jean-christophe-rangon-dev-web)

pour :





## **Table des matières**

**I. PRESENTATION**

**II. INSTALLATION DE PYTHON**

**III. INSTALLATION JUPYTER**

**III. CREATION ET EXECUTION D'UN FICHIER PYTHON**

**V. LES VARIABLES**

**VI. LES TYPES**

**VII. LES OPERATEURS**

**VIII. ENTREES ET SORTIES**

**IX. LES STRUCTURES DE CONTROLE**

**X. LES FONCTIONS**

**XI. GESTION DES ENTREES, SORTIES ET DES ERREURS**

**XII. LES CHAINES DE CARACTERES (str)**

**XIII. LES LISTES**

**XIV. LES TUPLES**

**XV. LES DICTIONNAIRES**

**XVI. LES SETs**

**XVII. LES FILES**



## I. PRESENTATION



## Historique

Guido van Rossum

1989

1ere publication:

1991

V0.9.0



# Pourquoi apprendre Python ?

Sep 2024	Sep 2023	Change	Programming Language	Ratings	Change
1	1		 Python	20.17%	+6.91%
2	3	▲	 C++	10.75%	+0.09%
3	4	▲	 Java	9.45%	-0.04%
4	2	▼	 C	8.89%	-2.38%
5	5		 C#	6.06%	-1.22%

<https://www.tiobe.com/tiobe-index/>

## Avantages de Python

- Interprété
- Orienté objet
- Haut niveau
- A syntaxe positionnelle
- Portable
- Lisible
- Extensible
- Logiciel libre

# Inconvénients de Python

- Lent
- Absence de pointeurs
- Typage différent

## Python2 vs Python3

- `print "toto"` ne fonctionnera pas en Python 3 (utiliser `print("toto")`)
- Nommage des paquets debian (`python-*` vs `python3-*`)
- En Python 2, les chaînes de caractères sont par défaut en ASCII, tandis qu'en Python 3, elles sont en Unicode, ce qui facilite le travail avec différentes langues et encodages.
- En Python 2, la division de deux entiers renvoie un entier (ex:  $3/2 = 1$ ), tandis qu'en Python 3, elle renvoie un nombre flottant (ex:  $3/2 = 1.5$ )
- Python 3 n'est pas entièrement compatible avec Python 2, ce qui signifie que certains anciens scripts Python 2 ne fonctionnent pas directement en Python 3 sans modifications.
- Il existe des outils pour automatiser la transition.

## Conseils

PRATIQUEZ....PRATIQUEZ... PRATIQUEZ!!

## II. INSTALLATION DE PYTHON

### 1. Procédures d'installation

#### a. Télécharger Python

Accédez au site officiel de Python : Rendez-vous sur [python.org](https://python.org) pour télécharger la dernière version stable de Python.

Choisissez la version appropriée : Sélectionnez la version compatible avec votre système d'exploitation (Windows, macOS, Linux).

#### b. Installation sur différents systèmes d'exploitation

##### Windows

Exécutez l'installateur : Après avoir téléchargé l'installateur, double-cliquez dessus pour lancer le processus d'installation.

##### Options importantes :

**Ajouter Python au PATH** : Cochez la case "Add Python to PATH" en bas de la première fenêtre de l'installateur. Cela permet d'exécuter Python depuis la ligne de commande.

Installer pour tous les utilisateurs : Vous pouvez choisir cette option si vous souhaitez que Python soit disponible pour tous les comptes sur l'ordinateur.

Suivez les étapes : Cliquez sur "Install Now" et suivez les instructions à l'écran jusqu'à la fin de l'installation.

##### macOS

Utiliser le paquet d'installation : Téléchargez le fichier .pkg depuis le site officiel et ouvrez-le.

Suivez l'assistant d'installation : Suivez les instructions à l'écran pour installer Python.

Vérifier l'installation : Ouvrez le Terminal et tapez :  
`python3 -version`

## Linux

Sur la plupart des distributions Linux, Python est déjà installé. Cependant, vous pouvez installer la dernière version via le gestionnaire de paquets.

### Debian/Ubuntu :

```
sudo apt update  
sudo apt install python3 python3-venv python3-pip
```

### Fedora :

```
sudo dnf install python3 python3-venv python3-pip
```

### Arch Linux :

```
sudo pacman -S python python-pip
```

## 2. Création d'Environnements Isolés

Les environnements isolés permettent de gérer les dépendances spécifiques à chaque projet sans interférer avec les autres projets ou avec les paquets système.

### a. Utiliser venv (module intégré à Python)

Étapes pour créer un environnement virtuel  
Ouvrez votre terminal ou invite de commandes.

Naviguez vers votre projet :

```
cd chemin/vers/votre/projet
```

Créer un environnement virtuel :

```
python3 -m venv nom_env
```

Remplacez nom\_env par le nom que vous souhaitez donner à votre environnement virtuel (par exemple, venv).

Activer l'environnement virtuel :

### Sur Windows :

```
nom_env\Scripts\activate
```

### Sur macOS et Linux :

```
source nom_env/bin/activate
```

Une fois activé, vous verrez le nom de l'environnement virtuel précédant votre invite de commande, par exemple (venv) C:\Projet>.

Installer des paquets : Avec l'environnement activé, utilisez pip pour installer les paquets nécessaires :  
`pip install nom_du_paquet`

Désactiver l'environnement virtuel : Lorsque vous avez terminé, vous pouvez désactiver l'environnement en tapant :  
`deactivate`

### **b. Utiliser virtualenv (alternative à venv)**

virtualenv offre des fonctionnalités similaires à venv et peut être utilisé si vous avez besoin de compatibilité avec des versions antérieures de Python.

Installer virtualenv :  
`pip install virtualenv`

Créer un environnement virtuel :  
`virtualenv nom_env`

Activer et désactiver : Les commandes d'activation et de désactivation sont les mêmes que pour venv.

### **c. Utiliser pipenv (gestionnaire de paquets et d'environnements)**

pipenv combine la gestion des dépendances et des environnements virtuels.

Installer pipenv :  
`pip install pipenv`

Créer et activer un environnement : Dans le répertoire de votre projet, exécutez :  
`pipenv install`

Activer l'environnement :  
`pipenv shell`

Installer des paquets :  
`pipenv install nom_du_paquet`

#### **d. Utiliser Anaconda/Miniconda (option avancée)**

Anaconda et Miniconda sont des distributions Python qui incluent des outils puissants pour la gestion des environnements et des paquets, particulièrement utiles pour le data science.

Télécharger et installer Anaconda ou Miniconda depuis le site officiel ou Miniconda.

Créer un environnement conda :

```
conda create --name nom_env python=3.x
```

Remplacez nom\_env par le nom de votre environnement et 3.x par la version de Python souhaitée.

Activer l'environnement :

```
conda activate nom_env
```

Installer des paquets :

```
conda install nom_du_paquet
```

Désactiver l'environnement :

```
conda deactivate
```

### **3. Conseils Supplémentaires**

Gestion des dépendances : Utilisez un fichier requirements.txt pour lister les paquets nécessaires à votre projet. Vous pouvez générer ce fichier avec :

```
pip freeze > requirements.txt
```

Et installer les dépendances avec :

```
pip install -r requirements.txt
```

### III. INSTALLATION JUPYTER

## Méthode 1 : Installation via pip

**Anaconda** est une distribution Python qui inclut Jupyter ainsi que de nombreuses autres bibliothèques scientifiques. C'est une méthode recommandée, surtout pour les débutants.

### 1. Télécharger Anaconda

- Rendez-vous sur le site officiel d'Anaconda :  
<https://www.anaconda.com/products/distribution>
- Téléchargez le programme d'installation approprié pour votre système d'exploitation (Windows, macOS ou Linux).

### 2. Installer Anaconda

#### Windows

1. Exécutez le fichier .exe téléchargé.
2. Suivez les instructions de l'installateur :
  - Acceptez le contrat de licence.
  - Choisissez l'installation pour "Just Me" ou "All Users".
  - Sélectionnez le répertoire d'installation.
  - **Recommandation** : Cochez l'option pour ajouter Anaconda à la variable d'environnement PATH (si proposée).
3. Terminez l'installation.

#### macOS

1. Ouvrez le fichier .pkg téléchargé.
2. Suivez les instructions de l'installateur :
  - Acceptez le contrat de licence.
  - Sélectionnez le disque d'installation.
3. Terminez l'installation via le Terminal si nécessaire.

#### Linux

1. Ouvrez le Terminal.
2. Naviguez jusqu'au répertoire contenant le fichier .sh téléchargé.
3. Exécutez la commande suivante (remplacez Anaconda3-2023.XX-Linux-x86\_64.sh par le nom exact du fichier) :

```
bash Anaconda3-2023.XX-Linux-x86_64.sh
```

4. Suivez les instructions à l'écran :
  - Acceptez le contrat de licence.
  - Choisissez le répertoire d'installation.
  - Confirmez l'initialisation d'Anaconda dans votre shell.
5. Rechargez votre shell ou redémarrez le Terminal.

### 3. Lancer Jupyter Notebook et JupyterLab

Après l'installation d'Anaconda, Jupyter est déjà inclus.

#### Jupyter Notebook

1. Ouvrez **Anaconda Navigator** depuis le menu Démarrer (Windows) ou depuis le dossier Applications (macOS).
2. Cliquez sur **Launch** sous **Jupyter Notebook**.

#### JupyterLab

1. Ouvrez **Anaconda Navigator**.
2. Cliquez sur **Launch** sous **JupyterLab**.

### Utilisation via le Terminal ou l'Invite de Commandes

Vous pouvez également lancer Jupyter directement depuis le Terminal ou l'Invite de Commandes :

#### Windows

1. Ouvrez l'**Invite de Commandes** ou **Anaconda Prompt**.
2. Tapez :

```
jupyter notebook
```

ou

```
jupyter lab
```

#### macOS et Linux

1. Ouvrez le **Terminal**.
2. Tapez :

```
jupyter notebook
```

ou

```
jupyter lab
```

## Méthode 2 : Installation via pip

Si vous préférez utiliser **pip** avec une installation Python standard, suivez les étapes ci-dessous.

### 1. Installer Python

Assurez-vous que Python est installé sur votre système.

#### Windows

1. Téléchargez Python depuis [python.org](https://python.org).
2. Exécutez le programme d'installation :
  - **Important** : Cochez l'option "Add Python to PATH" avant de cliquer sur "Install Now".
3. Terminez l'installation.

#### macOS

1. Téléchargez Python depuis [python.org](https://python.org).
2. Ouvrez le fichier .pkg téléchargé et suivez les instructions d'installation.

#### Linux

La plupart des distributions Linux ont Python préinstallé. Pour vérifier, ouvrez le Terminal et tapez :

```
python3 --version
```

Si Python n'est pas installé, installez-le via le gestionnaire de paquets de votre distribution. Par exemple, sur Debian/Ubuntu :

```
sudo apt update
sudo apt install python3 python3-pip
```

### 2. Installer Jupyter Notebook et JupyterLab avec pip

1. Ouvrez le **Terminal** (macOS/Linux) ou l'**Invite de Commandes** (Windows).
2. (Optionnel) Créez et activez un environnement virtuel pour isoler vos paquets :

```
python3 -m venv mon_env
```

- **Windows** :

```
mon_env\Scripts\activate
```

- **macOS/Linux** :

```
source mon_env/bin/activate
```

### 3. Installez Jupyter Notebook et JupyterLab :

```
pip install --upgrade pip  
pip install notebook jupyterlab
```

## 3. Lancer Jupyter Notebook et JupyterLab

### Jupyter Notebook

Dans le Terminal ou l'Invite de Commandes, tapez :

```
jupyter notebook
```

### JupyterLab

Dans le Terminal ou l'Invite de Commandes, tapez :

```
jupyter lab
```

Cela ouvrira Jupyter dans votre navigateur par défaut. Vous pouvez maintenant créer et gérer vos notebooks.

## Remarques Finales

- **Mises à jour** : Pour mettre à jour Jupyter, utilisez pip ou conda selon la méthode d'installation choisie.

- Avec pip :

```
pip install --upgrade notebook jupyterlab
```

- Avec conda :

```
conda update notebook jupyterlab
```

### III. CREATION ET EXECUTION D'UN FICHIER PYTHON

#### 1. Choisir un Éditeur de Texte ou un Environnement de Développement Intégré (IDE)

Avant de créer un fichier Python, vous devez choisir un éditeur de texte ou un IDE. Voici quelques options populaires :

**Visual Studio Code (VS Code)** : Gratuit et extensible avec de nombreuses extensions pour Python.

**PyCharm** : Puissant IDE dédié à Python, disponible en versions gratuite (Community) et payante.

**Atom** : Éditeur de texte personnalisable avec de nombreuses extensions.

**Nous travaillerons ici sur VS Code**

#### Installer l'extension Python :

Ouvrez VS Code.

Allez dans le Marketplace (icône des extensions sur la barre latérale gauche).

Recherchez "Python" et installez l'extension officielle de Microsoft.

#### 2. Créer un Fichier Python

##### a. Créer un Nouveau Fichier

Ouvrez votre éditeur de texte ou IDE.

Créer un nouveau fichier :

Dans VS Code, cliquez sur File > New File ou utilisez le raccourci Ctrl + N (Windows/Linux) ou Cmd + N (macOS).

Enregistrer le fichier :

File > Save As.

Nommez votre fichier avec l'extension .py, par exemple mon\_script.py. Choisissez le répertoire où vous souhaitez enregistrer le fichier.

### **b. Écrire du Code Python**

Voici un exemple simple de code Python que vous pouvez écrire dans votre fichier mon\_script.py :

```
# mon_script.py

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

if __name__ == "__main__":
    utilisateur = input("Entrez votre nom : ")
    saluer(utilisateur)
```

## **3. Exécuter le Fichier Python dans Votre Environnement Virtuel**

### **a. Activer votre Environnement Virtuel**

Assurez-vous que votre environnement virtuel est activé avant d'exécuter le script. Si ce n'est pas déjà fait, suivez ces étapes :

Ouvrez votre terminal ou invite de commandes.

Naviguez vers votre projet

```
cd chemin/vers/votre/projet
```

Activer l'environnement virtuel :

Sur Windows :

```
nom_env\Scripts\activate
```

Sur macOS et Linux :

```
source nom_env/bin/activate
```

Vous devriez voir le nom de votre environnement virtuel précédant votre invite de commande, par exemple (venv) C:\Projet>.

### **b. Naviguer vers le Répertoire du Fichier**

Dans le terminal activé, assurez-vous d'être dans le répertoire où se trouve votre fichier Python `mon_script.py`. Si ce n'est pas le cas, utilisez la commande `cd` pour y naviguer :

```
cd chemin/vers/le/repertoire
```

### **c. Exécuter le Script Python**

Une fois dans le bon répertoire et avec l'environnement virtuel activé, exécutez votre script en utilisant la commande suivante :

```
python mon_script.py
```

Exemple d'Exécution

Supposons que votre fichier `mon_script.py` est situé dans `C:\Projet` et que votre environnement virtuel s'appelle `venv`.

Ouvrez le terminal.

Activez l'environnement virtuel :

```
C:\Projet> venv\Scripts\activate
```

(Vous verrez `(venv)` au début de la ligne de commande.)

Exécutez le script :

```
(venv) C:\Projet> python mon_script.py
```

Interaction :

```
Entrez votre nom : Alice
Bonjour, Alice !
```

## **4. Utiliser l'IDE pour Exécuter le Script (Optionnel)**

Si vous utilisez un IDE comme VS Code, vous pouvez exécuter votre script directement depuis l'éditeur :

Ouvrez votre fichier `mon_script.py` dans VS Code.

Assurez-vous que l'environnement virtuel est activé :

VS Code devrait détecter automatiquement l'environnement virtuel si celui-ci est activé dans le terminal intégré.

Sinon, cliquez sur le sélecteur d'interpréteur Python en bas de la fenêtre et choisissez l'interpréteur de votre environnement virtuel.

Exécuter le script :

Vous pouvez cliquer sur l'icône de lecture (▶) en haut à droite de l'éditeur.

Ou utilisez le raccourci Ctrl + F5 (Windows/Linux) ou Cmd + F5 (macOS) pour exécuter sans débogage.

## **5. Conseils Supplémentaires**

Débogage : Utilisez les fonctionnalités de débogage de votre IDE pour mettre des points d'arrêt et inspecter les variables.

Gestion des Paquets : Installez les paquets nécessaires dans votre environnement virtuel en utilisant pip :

```
pip install nom_du_paquet
```

Version de Python : Assurez-vous que votre environnement virtuel utilise la version de Python souhaitée. Vous pouvez spécifier la version lors de la création de l'environnement virtuel :

```
python3 -m venv --python=python3.8 nom_env
```

## V. LES VARIABLES

Les **variables** en Python sont des conteneurs utilisés pour stocker des valeurs. Une variable permet de faire référence à une donnée en lui attribuant un nom, facilitant ainsi la manipulation des informations dans un programme. Voici quelques points essentiels à comprendre à propos des variables en Python :

### 1. Déclaration et affectation de variables

En Python, vous pouvez créer une variable simplement en lui assignant une valeur à l'aide de l'opérateur `=`. Vous n'avez pas besoin de spécifier le type de données (comme dans d'autres langages comme Java ou C), car Python déduit automatiquement le type en fonction de la valeur affectée.

```
x = 10 # x est une variable de type entier
nom = "Alice" # nom est une variable de type chaîne de
caractères
prix = 19.99 # prix est une variable de type flottant
```

### 2. Types de variables

Les variables peuvent stocker des données de différents types, et Python les gère dynamiquement. Quelques types de base incluent :

- **Entiers (int)** : Pour stocker des nombres entiers, comme `x = 5`.
- **Flottants (float)** : Pour stocker des nombres décimaux, comme `prix = 19.99`.
- **Chaînes de caractères (str)** : Pour stocker du texte, comme `nom = "Alice"`.
- **Booléens (bool)** : Pour stocker des valeurs de vérité, soit `True` ou `False`, comme `is_active = True`.
- **Listes (list)** : Pour stocker plusieurs valeurs dans un ordre défini, comme `fruits = ["pomme", "banane", "orange"]`.
- **Dictionnaires (dict)** : Pour stocker des paires clé-valeur, comme `personne = {"nom": "Alice", "âge": 25}`.

Exemple de types de données différents assignés à des variables :

```
nombre = 10 # type entier  
prix = 12.5 # type flottant  
nom = "Alice" # type chaîne  
is_active = True # type booléen
```

### 3. Changement de valeur et type dynamique

En Python, une variable peut changer de type en fonction de la valeur qui lui est assignée. Une variable peut être assignée à un entier puis, plus tard, à une chaîne de caractères sans problème.

```
x = 10 # x est un entier  
x = "Bonjour" # x devient une chaîne de caractères
```

Cette flexibilité est l'une des caractéristiques de Python. Cependant, il est recommandé de nommer vos variables de manière descriptive pour maintenir la clarté du code.

### 4. Nommage des variables

Les noms de variables en Python doivent respecter certaines règles :

- Ils doivent commencer par une lettre ou un underscore (\_), mais pas par un chiffre.
- Ils peuvent contenir des lettres, des chiffres et des underscores, mais pas d'espaces ni de symboles spéciaux.
- Les noms de variables sont sensibles à la casse, donc nom et Nom sont deux variables différentes.
- Il est recommandé d'utiliser des noms descriptifs pour vos variables (par exemple, nombre\_étudiants au lieu de x).

Exemples de noms valides :

```
age = 25
_nom = "Alice"
nombre_etudiants = 30
```

## 5. Variables globales et locales

Les variables en Python peuvent être **locales** (déclarées à l'intérieur d'une fonction) ou **globales** (déclarées à l'extérieur de toutes les fonctions).

- **Variables locales** : Elles ne sont accessibles que dans la fonction où elles sont définies.
- **Variables globales** : Elles sont accessibles partout dans le programme.

Exemple de variable locale et globale :

```
x = 10 # Variable globale

def ma_fonction():
    x = 5 # Variable locale
    print(x) # Imprime la variable locale x (5)

ma_fonction()
print(x) # Imprime la variable globale x (10)
```

Dans cet exemple, x à l'intérieur de la fonction est une variable locale, tandis que x à l'extérieur de la fonction est une variable globale.

## 6. Affichage des variables

Pour afficher la valeur d'une variable, vous pouvez utiliser la fonction `print()` :

```
x = 10
nom = "Alice"
print(x)  # Affiche 10
print(nom)  # Affiche Alice
```

## 6. Ecriture formatée

L'écriture formatée est un mécanisme permettant d'afficher des variables avec un format précis, par exemple justifiées à gauche ou à droite, ou encore avec un certain nombre de décimales pour les floats. Depuis la version 3.6, Python a introduit les f-strings pour mettre en place l'écriture formatée.

```
##f-string
a = 32
name = "John"
print(f"{name} a {a} ans")
```

```
John a 32 ans
```

L'écriture formatée permet aussi de spécifier le format des variables à afficher dans la chaîne envoyée au terminal :

```
stat_result = (7802+4752)/12700
print("la stat finale est: ", stat_result)
```

donne :

```
la stat finale est:  0.988503937007874
[nodemon] clean exit - waiting for changes be
```

### Specification du format

L'écriture formatée permet aussi de spécifier le format des variables à afficher dans la chaîne envoyée au terminal :

```
stat_result = (7802+4752)/12700
print("la stat finale est: ", stat_result)

print(f"la stat finale est: {stat_result:.2f}")
print(f"la stat finale est: {stat_result:.3f}")
print(f"la stat finale est: {stat_result:.0f}")
```

```
la stat finale est:  0.988503937007874
la stat finale est: 0.99
la stat finale est: 0.989
la stat finale est: 1
[nodemon] clean exit - waiting for changes
```

## Spécification de l'alignement

```
##alignement
print(f"{10:*>6d}") ; print(f"{1000:*>6d}")
print('---')
print(f"{10:*<6d}") ; print(f"{1000:*<6d}")
print('---')
print(f"{10:*^6d}") ; print(f"{1000:*^6d}")
```

```
****10
**1000
---
10****
1000**
---
**10**
*1000*
[nodemon] clean exit - waiting
```

6d ici signifie le nombre de caractères total

> signifie alignement à droite

< signifie alignement à gauche

^ signifie alignement centré

Le caractère de remplissage est fourni juste avant la directive d'alignement. Le caractère de remplissage par défaut est l'espace.

## Justification

```
## justification
print(f"atom {'HN':>4s}") ; print(f"atom {'HDE1':>4s}")
```

```
atom  HN
atom HDE1
[nodemon] clea
```

il est important de bien comprendre qu'une f-string est indépendante de la fonction `print()`. Si on donne une f-string à la fonction `print()`, Python évalue d'abord la f-string et c'est la chaîne de caractères qui en résulte qui est affichée à l'écran

```
print(type(f"atom {'HN':>4s}"))
```

```
<class 'str'>
[nodemon] clean exit -
```

### Expressions dans les f-string

Une fonctionnalité extrêmement puissante des f-strings est de supporter des expressions Python au sein des accolades. Ainsi, il est possible d'y mettre directement une opération ou encore un appel à une fonction :

```
##Expression dans les f-string
print(f"Résultat d'une opération avec des floats : {(4.1 * 6.7)}")

entier = 2
print(f"Le type de {entier} est {type(entier)}")
```

```
Résultat d'une opération avec des floats : 27.47
Le type de 2 est <class 'int'>
[nodemon] clean exit - waiting for changes before r
□
```

## Ecriture scientifique

Pour les nombres très grands ou très petits, l'écriture formatée permet d'afficher un nombre en notation scientifique (sous forme de puissance de 10) avec la lettre **e**

```
##Ecriture scientifique
print(f"{1_000_000_000:e}")
```

```
Le type de 2 est <class 'int'>
1.000000e+09
[nodemon] clean exit - wait
□
```

## Ecriture brute (r-string)

Les chaînes brutes sont utiles lorsqu'on veut ignorer les caractères d'échappement comme `\n` (nouvelle ligne) ou `\t` (tabulation). Elles sont définies en préfixant la chaîne avec un `r`.

```
chemin = r"C:\nouveau\dossier"  
print(chemin) # Affiche "C:\nouveau\dossier" sans interpréter le \n et le \d
```

## VI. LES TYPES

Python possède plusieurs types de données intégrés qui permettent de manipuler différents types d'informations. Voici les principaux **types de données** en Python :

### 1. Types numériques

- **int** : Représente les entiers, qu'ils soient positifs ou négatifs.

```
x = 10 # entier positif  
y = -5 # entier négatif
```

- **float** : Représente les nombres à virgule flottante (nombres décimaux).

```
pi = 3.14  
temperature = -5.6
```

- **complex** : Représente les nombres complexes avec une partie réelle et une partie imaginaire.

```
z = 1 + 2j # 1 est la partie réelle, 2j est la partie imaginaire
```

### 2. Chaînes de caractères (str)

- **str** : Représente une séquence de caractères (texte). Les chaînes de caractères sont délimitées par des guillemets simples ou doubles.

```
nom = "Alice"  
phrase = 'Bonjour tout le monde'
```

Les chaînes de caractères sont immuables, ce qui signifie que leur contenu ne peut pas être modifié après leur création.

### 3. Types de séquences

Les séquences sont des types de données qui représentent une collection d'éléments ordonnés.

- **list** : Représente une liste mutable, c'est-à-dire une collection ordonnée d'éléments qui peuvent être modifiés (ajoutés, supprimés, etc.).

```
fruits = ["pomme", "banane", "orange"]
```

- **tuple** : Représente une liste immuable, c'est-à-dire une collection ordonnée d'éléments qui ne peuvent pas être modifiés après leur création.

```
coordonnees = (10, 20)
```

- **range** : Génère une séquence d'entiers dans une liste. Souvent utilisé pour parcourir des boucles

```
range(5) # Donne [0, 1, 2, 3, 4]
```

#### 4. Types de mappage

- **dict** : Représente un dictionnaire, qui est une collection non ordonnée de paires clé-valeur. Chaque clé est associée à une valeur, et ces clés doivent être uniques.

```
etudiant = {"nom": "Alice", "âge": 25, "classe": "M2"}
```

#### 5. Types d'ensembles

- **set** : Représente une collection non ordonnée et non indexée d'éléments uniques. Les ensembles ne permettent pas les doublons.

```
ensemble = {1, 2, 3, 4, 5}
```

- **frozenset** : Représente un ensemble immuable. Il a les mêmes propriétés qu'un ensemble, mais ses éléments ne peuvent pas être modifiés après la création.

```
ensemble_immuable = frozenset([1, 2, 3, 4, 5])
```

#### 6. Booléens (bool)

- **bool** : Représente les valeurs de vérité, soit True (vrai) ou False (faux)

```
is_active = True  
est_vide = False
```

## 7. Types binaires

- **bytes** : Représente une séquence immuable d'octets

```
data = b'Hello'
```

- **bytearray** : Représente une séquence mutable d'octets.

```
data = bytearray(5) # Tableau de 5 octets
```

- **memoryview** : Permet d'accéder à une vue en mémoire d'une séquence binaire sans faire de copie.

```
data = memoryview(bytes(5))
```

## 8. NoneType

- **None** : Représente l'absence de valeur ou une valeur nulle. Utilisé principalement pour initialiser des variables ou indiquer une absence de résultat.

```
resultat = None
```

## 9. La fonction native « type() »

Si vous ne vous souvenez plus du type d'une variable, utilisez la fonction `type()` qui vous le rappellera.

```
nom = "Alice"
```

```
print(type(nom))
```

```
<class 'str'>
```

Nous verrons ce que 'class' signifie dans la suite de ce cours.

## 10. Fonctions de conversion de type

Python offre des fonctions intégrées pour convertir d'un type à un autre :

- **int()** : Convertit en entier.
- **float()** : Convertit en flottant.
- **str()** : Convertit en chaîne de caractères.
- **list()** : Convertit en liste.
- **tuple()** : Convertit en tuple.
- **set()** : Convertit en ensemble.

```
a = "123"  
b = int(a) # Convertit la chaîne "123" en entier 123
```

## 11. Objets personnalisés (Classes)

En plus des types de données prédéfinis, vous pouvez créer vos propres types en définissant des **classes** pour structurer des objets avec des attributs et des méthodes spécifiques.

## VII. LES OPERATEURS

### ➤ Arithmétiques

`+`, `-`, `*`, `**` (exposant), `/`, `//` (division entière), `%`  
`+` et `*` opèrent sur des str et sur des listes

### ➤ assignation

`=` (affectation), `+=` (ajouter à), `*=`, `/=`, `-=`, `%=`

### ➤ logiques

**and** (et), **or** (ou), **not** (non)

### ➤ comparaison

`==`, `!=`, `<`, `<=`, `>`, `>=`

## Opérations sur les types numériques (opérateur arithmétiques)

### 1. Type **int** (entier)

Le type `int` en Python représente des nombres entiers qui peuvent être aussi grands ou petits que nécessaire, car la taille des entiers n'est limitée que par la mémoire disponible.

#### Opérations de base avec les entiers (`int`)

- **Addition (+) :**

```
a = 5 + 3 # 8
```

- **Soustraction (-) :**

```
b = 10 - 4 # 6
```

- **Multiplication (\*)** :

```
c = 6 * 7 # 42
```

- **Division entière (//)** : Division qui renvoie uniquement la partie entière du quotient (sans la partie décimale).

```
d = 7 // 2 # 3
```

- **Modulo (%)** : Reste de la division entière.

```
e = 10 % 3 # 1
```

- **Puissance (\*\*)** : Exponentiation.

```
f = 2 ** 3 # 8
```

## Fonctions utiles pour les entiers

- **abs(x)** : Renvoie la valeur absolue de x.

```
abs(-5) # 5
```

- **divmod(a, b)** : Renvoie un tuple contenant le quotient entier et le reste de la division de a par b.

```
divmod(10, 3) # (3, 1)
```

- **pow(x, y)** : Renvoie x à la puissance y, équivalent à x \*\* y.

```
pow(2, 3) # 8
```

## 2. Type **float** (flottant)

Le type float est utilisé pour représenter des nombres réels avec des parties décimales. Il utilise la norme IEEE 754 pour la représentation des nombres flottants, ce qui signifie qu'il a une précision limitée.

### Opérations de base avec les flottants (float)

Les opérations sur les flottants sont similaires à celles sur les entiers, mais elles prennent en compte les décimales :

- **Addition (+)** :

```
a = 5.5 + 3.2 # 8.7
```

- **Soustraction (-)** :

```
b = 10.1 - 4.2 # 5.9
```

- **Multiplication (\*)** :

```
c = 6.0 * 7.0 # 42.0
```

- **Division classique (/)** :

```
d = 7.0 / 2.0 # 3.5
```

- **Division entière (//)** : Renvoie la partie entière du quotient, même si les opérandes sont des flottants.

```
e = 7.0 // 2.0 # 3.0
```

- **Modulo (%)** : Renvoie le reste de la division, prenant en compte les parties décimales.

```
f = 10.5 % 3.2 # 0.8999999999999999 (à cause des erreurs de précision des flottants)
```

- **Puissance (\*\*)** :

```
g = 2.5 ** 3 # 15.625
```

## Fonctions utiles pour les flottants

- **round(x, n)** : Arrondit x à n décimales.

```
round(3.14159, 2) # 3.14
```

- **abs(x)** : Renvoie la valeur absolue.

```
abs(-3.5) # 3.5
```

- **math.ceil(x)** : Renvoie l'entier supérieur le plus proche de x.

```
import math  
math.ceil(4.2) # 5
```

- **math.floor(x)** : Renvoie l'entier inférieur le plus proche de x.

```
math.floor(4.8) # 4
```

- **math.sqrt(x)** : Renvoie la racine carrée de x.

```
math.sqrt(16) # 4.0
```

## 3. Type complex (complexe)

Le type complex est utilisé pour représenter des nombres complexes en Python, avec une partie réelle et une partie imaginaire. En Python, les nombres complexes sont exprimés sous la forme  $a + bj$ , où  $a$  est la partie réelle, et  $b$  est la partie imaginaire.

## Opérations de base avec les nombres complexes (complex)

- **Addition (+) :**

```
z = (1 + 2j) + (2 + 3j) # (3 + 5j)
```

- **Soustraction (-) :**

```
z = (5 + 4j) - (1 + 2j) # (4 + 2j)
```

- **Multiplication (\*) :**

```
z = (2 + 3j) * (4 + 5j) # (-7 + 22j)
```

- **Division (/) :**

```
z = (1 + 2j) / (3 + 4j) # (0.44 + 0.08j)
```

- **Puissance (\*\*) :**

```
z = (1 + 1j) ** 2 # (2j)
```

## Accéder aux parties réelle et imaginaire

Vous pouvez accéder à la partie réelle et à la partie imaginaire d'un nombre complexe à l'aide des attributs `real` et `imag`.

```
z = 3 + 4j
print(z.real) # 3.0
print(z.imag) # 4.0
```

## Fonction utile pour les complexes

- **abs(z) :** Renvoie la magnitude (module) du nombre complexe `z`.

```
abs(3 + 4j) # 5.0 (car sqrt(3^2 + 4^2) = 5)
```

## Opérations numériques générales

Ces opérations peuvent s'appliquer à tous les types numériques (int, float, complex) :

- **Comparaisons** : Vous pouvez utiliser les opérateurs de comparaison entre différents types numériques :
  - == (égalité), != (différent), < (inférieur), > (supérieur), <= (inférieur ou égal), >= (supérieur ou égal).

```
3 == 3.0 # True (car Python considère les nombres int et float égaux s'ils ont la même valeur)
5 > 2   # True
```

## Conversion entre types numériques

Vous pouvez convertir entre les types numériques en Python à l'aide des fonctions suivantes :

- **int(x)** : Convertit x en entier.

```
int(3.14) # 3 (tronque la partie décimale)
```

- **float(x)** : Convertit x en flottant.

```
float(5) # 5.0
```

- **complex(a, b)** : Crée un nombre complexe avec a comme partie réelle et b comme partie imaginaire.

```
complex(3, 4) # (3 + 4j)
```

## Opérateurs de comparaison

Python possède une particularité au niveau des types lorsqu'on essaye de les comparer. Contrairement à d'autres langages qui possèdent deux types d'opérateurs d'égalité afin de distinguer une comparaison superficielle (comparaison des valeurs uniquement après transtypage automatique) ou stricte (comparaison des valeurs et de types sans transtypage), python, lui ne possède qu'un seul opérateur.

Son utilisation repose sur le concept python de **COMPATIBILITE des types**. Si les types ne sont pas compatibles alors il faudra transtyper explicitement au moyen d'une fonction de conversion de type.

Types numériques (**int**, **float**, **complex**)

Les types numériques sont souvent compatibles entre eux dans les opérations. Python convertira automatiquement un type en un autre, comme vu précédemment avec la conversion automatique d'un entier en flottant ou en nombre complexe si nécessaire.

```
x = 5
y = 2.0
z = x + y # x est automatiquement converti en 5.0
print(z) # 7.0 (float)

a = 1 + 2j
b = 4
result = a + b # b est converti en 4 + 0j
print(result) # (5+2j)
```

## Chaînes de caractères (str) et autres types

Les chaînes de caractères ne peuvent pas être directement combinées avec des nombres dans des opérations mathématiques. Vous devrez utiliser une **conversion explicite** avec `str()` ou `int()/float()`, selon la direction de la conversion.

```
x = 5
y = "10"
result = x + int(y) # Conversion de "10" en entier
print(result) # 15

# Conversion inverse : concaténation d'une chaîne et d'un nombre
message = "Il y a " + str(x) + " pommes"
print(message) # "Il y a 5 pommes"
```

## Séquences (list, tuple, set, dict)

- **Listes** et **tuples** : Bien que ces deux types soient des séquences, ils ne sont pas directement compatibles entre eux. Cependant, vous pouvez les **convertir explicitement** pour rendre certaines opérations possibles.

Exemple de conversion :

```
liste = [1, 2, 3]
tuple_ = tuple(liste) # Conversion de liste en tuple
print(tuple_) # (1, 2, 3)
```

- **Ensembles (set)** et **listes** : Vous pouvez **convertir une liste en ensemble et vice versa**, mais un ensemble n'accepte pas les doublons, contrairement aux listes.

```
liste = [1, 2, 2, 3, 4]
ensemble = set(liste)
print(ensemble) # {1, 2, 3, 4} (les doublons sont supprimés)
```

- **Dictionnaires** (dict) : Les dictionnaires ne peuvent pas être combinés avec des listes ou des tuples directement. Cependant, vous pouvez **convertir les clés ou les valeurs** d'un dictionnaire en une liste ou un autre type.

```
d = {"a": 1, "b": 2}
cles = list(d.keys()) # Conversion des clés du dictionnaire en liste
print(cles) # ['a', 'b']
```

## Comparaisons entre types différents

Lors de la comparaison entre types différents, Python renverra généralement False, sauf dans certains cas où la conversion automatique s'applique.

```
# Comparaison d'un entier et d'un flottant
print(10 == 10.0) # True (les deux représentent la même valeur)

# Comparaison d'un entier et d'une chaîne
print(5 == "5") # False (un entier et une chaîne ne sont pas équivalents)

# Comparaison d'un tuple et d'une liste
print((1, 2, 3) == [1, 2, 3]) # False (tuple et liste sont des types différents)
```

## Cas particulier du type None

- `None == None` est vrai
- `None != None` est faux
- `None != 0` est vrai (idem pour n'importe quel nombre )
- `None == 0` est faux (idem pour n'importe quel nombre)
- `None != 'a'` est vrai (idem pour n'importe quelle chaîne)
- `None != []` est vrai, `None != {}` est vrai (pareil pour structure non vide)
- `None < 0` et `None > 0` renvoient des exceptions

## Les opérateurs en détails

### 1. Opérateurs arithmétiques

- **`==` (égal à)** : Vérifie si deux valeurs sont égales.

```
print(5 == 5) # True
print(5 == 3) # False
```

- **`!=` (différent de)** : Vérifie si deux valeurs sont différentes.

```
print(5 != 3) # True
print(5 != 5) # False
```

- **`>` (supérieur à)** : Vérifie si une valeur est strictement supérieure à une autre.

```
print(5 > 3) # True
print(3 > 5) # False
```

- **< (inférieur à)** : Vérifie si une valeur est strictement inférieure à une autre.

```
print(3 < 5) # True
print(5 < 3) # False
```

- **>= (supérieur ou égal à)** : Vérifie si une valeur est supérieure ou égale à une autre.

```
print(5 >= 3) # True
print(5 >= 5) # True
```

- **<= (inférieur ou égal à)** : Vérifie si une valeur est inférieure ou égale à une autre.

```
print(3 <= 5) # True
print(5 <= 5) # True
```

### Comparaison des chaînes de caractères :

Les chaînes de caractères peuvent également être comparées à l'aide de ces opérateurs. Python compare les chaînes selon l'ordre lexicographique (ordre alphabétique basé sur le code ASCII).

```
print("apple" == "apple") # True
print("apple" < "banana") # True (car 'a' est avant 'b' dans l'ordre ASCII)
```

### Comparaison des types mixtes :

En général, les comparaisons entre des types non compatibles (comme un entier et une chaîne) produisent une erreur.

```
print(5 == "5") # False (types différents, pas une erreur mais False)
```

## 2. Opérateurs logiques

Les opérateurs logiques permettent de combiner plusieurs expressions booléennes (ou conditions) et renvoient également un booléen (True ou False).

### Liste des opérateurs logiques :

- **and** : Renvoie True si **toutes** les conditions sont vraies. Si une condition est fausse, il renvoie False.

```
print(5 > 3 and 10 > 5) # True (car les deux conditions sont vraies)
print(5 > 3 and 10 < 5) # False (car la deuxième condition est fausse)
```

- **or** : Renvoie True si **au moins une** des conditions est vraie. Si toutes les conditions sont fausses, il renvoie False.

```
print(5 > 3 or 10 < 5) # True (car la première condition est vraie)
print(3 > 5 or 10 < 5) # False (car les deux conditions sont fausses)
```

- **not** : Inverse la valeur d'une condition. Si la condition est vraie, not renvoie False, et inversement.

```
print(not (5 > 3)) # False (car 5 > 3 est vrai, donc not True = False)
print(not (3 > 5)) # True (car 3 > 5 est faux, donc not False = True)
```

### Priorité des opérateurs logiques :

- L'opérateur **not** a la priorité la plus haute, suivi de and, puis de or. Il est recommandé d'utiliser des **parenthèses** pour clarifier l'ordre d'évaluation dans des expressions complexes.

```
result = not (5 > 3 and 10 > 5) # False, car la condition (5 > 3
and 10 > 5) est True, donc not True = False
print(result)
```

### 3. Opérateurs d'identité

Les opérateurs d'identité permettent de vérifier si deux objets pointent vers la **même** adresse en mémoire (c'est-à-dire si ce sont vraiment le **même objet**, et non juste des objets ayant les mêmes valeurs).

#### Liste des opérateurs d'identité :

- **is** : Renvoie True si les deux objets comparés sont le même objet en mémoire.

```
x = [1, 2, 3]
y = x
print(x is y)  # True (car `x` et `y` pointent vers le même objet)

z = [1, 2, 3]
print(x is z)  # False (car `x` et `z` sont deux objets différents
               # même si leurs valeurs sont les mêmes)
```

- **is not** : Renvoie True si les deux objets comparés ne sont pas le même objet en mémoire.

```
a = [1, 2, 3]
b = [1, 2, 3]
print(a is not b)  # True (car `a` et `b` ne pointent pas vers le même objet)
```

#### Différence entre == et is :

- **==** compare les **valeurs** des objets (si les valeurs sont identiques).
- **is** compare les **références** d'objets (si les objets sont les mêmes en mémoire).

```
a = [1, 2, 3]
b = [1, 2, 3]

# Comparaison avec `==` (comparer les valeurs)
print(a == b)  # True (les valeurs des deux listes sont identiques)

# Comparaison avec `is` (comparer les objets en mémoire)
print(a is b)  # False (car `a` et `b` sont des objets différents
                 en mémoire)
```

## 4. Opérateurs d'appartenance

Les opérateurs d'appartenance sont utilisés pour vérifier si un élément appartient ou non à une séquence (comme une liste, un tuple, un dictionnaire, une chaîne de caractères, etc.).

### Liste des opérateurs d'appartenance :

- **in** : Renvoie True si un élément se trouve dans une séquence.

```
liste = [1, 2, 3, 4]
print(3 in liste) # True (3 est dans la liste)

texte = "Python"
print("P" in texte) # True (la lettre 'P' est dans la chaîne)
```

- **not in** : Renvoie True si un élément ne se trouve pas dans une séquence.

```
liste = [1, 2, 3, 4]
print(5 not in liste) # True (5 n'est pas dans la liste)

texte = "Python"
print("z" not in texte) # True (la lettre 'z' n'est pas dans la chaîne)
```

### Exemple avec dictionnaire :

Lorsque vous utilisez **in** ou **not in** avec un dictionnaire, la recherche se fait sur les **clés** du dictionnaire et non sur les valeurs.

```
d = {"nom": "Alice", "âge": 25}
print("nom" in d) # True (la clé "nom" est dans le dictionnaire)
print("Alice" in d) # False (la valeur "Alice" n'est pas une clé)
```

Python propose plusieurs types d'opérateurs logiques et de comparaison qui permettent de réaliser des tests conditionnels :

- **Opérateurs de comparaison** : Pour comparer des valeurs (==, !=, >, <, >=, <=).
- **Opérateurs logiques** : Pour combiner des conditions booléennes (and, or, not).
- **Opérateurs d'identité** : Pour vérifier si deux objets pointent vers le même espace mémoire (is, is not).
- **Opérateurs d'appartenance** : Pour vérifier si un élément fait partie d'une séquence (in, not in).

Ces opérateurs sont très utilisés dans les structures de contrôle telles que les instructions conditionnelles (if, while, etc.) pour guider l'exécution du programme en fonction de certaines conditions.

## VIII.ENTREES ET SORTIES

La fonction « `print()` » permet d'afficher des valeurs dans le terminal. La signature complète de `print()` est la suivante :

```
print(valeur(s), sep='', end='\n', file=sys.stdout,  
flush=False)
```

Lors de son utilisation, nous pouvons lui passer une liste de valeurs séparées par des virgules,

```
print(5) # Affiche 5  
print("Bonjour", "tout", "le", "monde") # Affiche 'Bonjour tout le monde'
```

redéfinir le séparateur,

```
print("Alice", "Bob", "Charlie", sep=", ") # Affiche 'Alice, Bob, Charlie'  
print("1", "2", "3", sep="-") # Affiche '1-2-3'
```

le retour chariot,

```
print("Alice", end=", ")  
print("Bob") # Affiche 'Alice, Bob' (les deux sur la même ligne)
```

la sortie standard

```
import sys  
  
# Imprimer dans un fichier  
with open('output.txt', 'w') as f:  
    print("Bonjour tout le monde", file=f) # La sortie est écrite dans 'output.txt'
```

et L'affichage immédiat par vidange du tampon d'affichage.

```
print("Chargement...", end="", flush=True) # Cela permet de
forcer l'affichage immédiat de 'Chargement...' sans attendre
que le tampon se remplisse ou que le programme se termine.
```

```
<class 'str'>
Chargement...[nodemon] clean exit - waiting for changes before restart
█
```

Un programme peut aussi interagir avec l'utilisateur. La fonction native « **input()** » permet à l'utilisateur de saisir des données et de les transmettre au programme.

```
age= int(input("Entrez votre âge: "))
taille = float(input("Entrez votre taille: "))
print("vous avez ", age, " ans et vous mesurez ", taille, "m")
```

```
Entrez votre âge: 25
Entrez votre taille: 1.8
vous avez  25  ans et vous mesurez  1.8 m
[nodemon] clean exit - waiting for changes before restart
█
```

## IX. LES STRUCTURES DE CONTROLE

Les **structures de contrôle** en Python permettent de définir le flux d'exécution d'un programme, c'est-à-dire la manière dont certaines sections de code seront exécutées en fonction de conditions ou de boucles répétitives. Les principales structures de contrôle en Python sont les instructions conditionnelles, les boucles, et les instructions de contrôle de boucle.

### 1. Instructions conditionnelles

Les instructions conditionnelles permettent de prendre des décisions dans le code en fonction de certaines conditions. En Python, les principales instructions conditionnelles sont `if`, `elif`, et `else`.

#### 1.1. L'instruction `if`

L'instruction `if` permet d'exécuter un bloc de code uniquement si une condition est **vraie**.

Syntaxe :

```
if condition:
    # Bloc de code exécuté si la condition est vraie
```

```
x = 10
if x > 5:
    print("x est supérieur à 5")
```

#### 1.2. L'instruction `else`

L'instruction `else` est utilisée pour exécuter un bloc de code lorsque la condition du `if` est **fausse**.

Syntaxe :

```
if condition:
    # Bloc de code exécuté si la condition est vraie
else:
    # Bloc de code exécuté si la condition est fausse
```

```
x = 3
if x > 5:
    print("x est supérieur à 5")
else:
    print("x est inférieur ou égal à 5")
```

### 1.3. L'instruction **elif**

L'instruction **elif** (contraction de "else if") permet d'ajouter plusieurs conditions alternatives à un **if**. Elle est exécutée si la condition **if** est **fausse** et que la condition **elif** est **vraie**.

Syntaxe :

```
if condition1:
    # Bloc de code exécuté si condition1 est vraie
elif condition2:
    # Bloc de code exécuté si condition1 est fausse et condition2 est vraie
else:
    # Bloc de code exécuté si toutes les conditions précédentes sont fausses
```

```
x = 7
if x > 10:
    print("x est supérieur à 10")
elif x == 7:
    print("x est égal à 7")
else:
    print("x est inférieur à 10")
```

## 1.4. Conditions imbriquées

Il est possible d'impliquer des structures if, elif, et else les unes dans les autres, permettant ainsi des prises de décisions plus complexes.

```
x = 10
y = 20
if x == 10:
    if y == 20:
        print("x est 10 et y est 20")
```

## 1.5. Expressions conditionnelles (ternaires)

Python permet également d'écrire des instructions conditionnelles simples sous forme d'expressions ternaires.

Syntaxe :

```
valeur = valeur_si_vrai if condition else valeur_si_faux
```

```
x = 5
resultat = "Grand" if x > 10 else "Petit"
print(resultat) # Affiche "Petit"
```

## 2. Boucles

Les boucles permettent de répéter un bloc de code un certain nombre de fois, soit de manière déterminée (avec une boucle for), soit de manière indéterminée (avec une boucle while).

### 2.1. La boucle for

La boucle for est utilisée pour itérer sur une **séquence** (liste, tuple, chaîne de caractères, ou tout autre objet itérable). La boucle exécute un bloc de code pour chaque élément de la séquence.

Syntaxe :

```
for variable in sequence:
```

```
# Bloc de code exécuté pour chaque élément dans la séquence
```

```
fruits = ["pomme", "banane", "orange"]  
for fruit in fruits:  
    print(fruit)
```

Dans cet exemple, fruit prend successivement la valeur de chaque élément de la liste fruits.

## 2.2. La fonction range() dans les boucles for

**range()** est une fonction couramment utilisée dans les boucles for pour générer une séquence de nombres.

```
for i in range(5):  
    print(i) # Affiche 0, 1, 2, 3, 4
```

Vous pouvez également spécifier des arguments pour contrôler le début, la fin, et le pas de l'itération :

```
for i in range(2, 10, 2):  
    print(i) # Affiche 2, 4, 6, 8
```

### 2.3. La boucle while

La boucle while permet d'exécuter un bloc de code **tant qu'une condition est vraie**. La condition est évaluée avant chaque itération, et la boucle continue tant que la condition est vraie.

```
while condition:  
    # Bloc de code exécuté tant que la condition est vraie
```

```
x = 0  
while x < 5:  
    print(x)  
    x += 1 # Incrémentation de x pour éviter une boucle infinie
```

Dans cet exemple, la boucle s'arrête lorsque x devient égal à 5.

### 3. Instructions de contrôle de boucle

Python propose plusieurs instructions qui permettent de modifier le comportement des boucles : break, continue, et else.

#### 3.1. L'instruction break

L'instruction break permet de **sortir immédiatement** d'une boucle, même si la condition n'est pas remplie.

Exemple avec for :

```
for i in range(10):  
    if i == 5:  
        break # Sortie de la boucle lorsque i atteint 5  
    print(i)
```

Exemple avec while :

```
x = 0
while x < 10:
    if x == 5:
        break # Sortie de la boucle lorsque x atteint 5
    print(x)
    x += 1
```

### 3.2. L'instruction continue

L'instruction continue permet de **passer à l'itération suivante** de la boucle, sans exécuter le reste du bloc de code.

```
for i in range(5):
    if i == 3:
        continue # Ignore le reste du bloc et passe à l'itération suivante
    print(i)
```

### 3.3. L'instruction else dans les boucles

En Python, vous pouvez utiliser une clause else avec les boucles for et while. Le bloc else est exécuté lorsque la boucle se termine **normalement**, c'est-à-dire sans rencontre d'un break.

```
for i in range(5):  
    print(i)  
else:  
    print("Boucle terminée normalement")
```

Exemple avec break :

```
for i in range(5):  
    if i == 3:  
        break  
    print(i)  
else:  
    print("Ceci ne sera pas affiché car la boucle a été interrompue par break")
```

## 4. Instructions de contrôle exceptionnelles

Python dispose également d'instructions de contrôle spéciales comme pass et return qui peuvent être utilisées dans différents contextes.

### 4.1. L'instruction pass

pass est une instruction qui ne fait rien. Elle est souvent utilisée comme un **espace réservé** dans le code lorsqu'une structure est nécessaire mais qu'il n'y a pas encore de code à exécuter.

```
for i in range(5):  
    if i == 3:  
        pass # Ne fait rien pour l'itération 3  
    print(i)
```

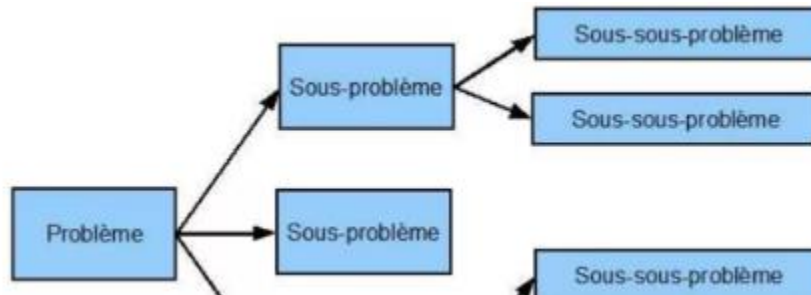
## 4.2. L'instruction return

L'instruction return est utilisée dans les fonctions pour **renvoyer une valeur** et **terminer l'exécution** de la fonction.

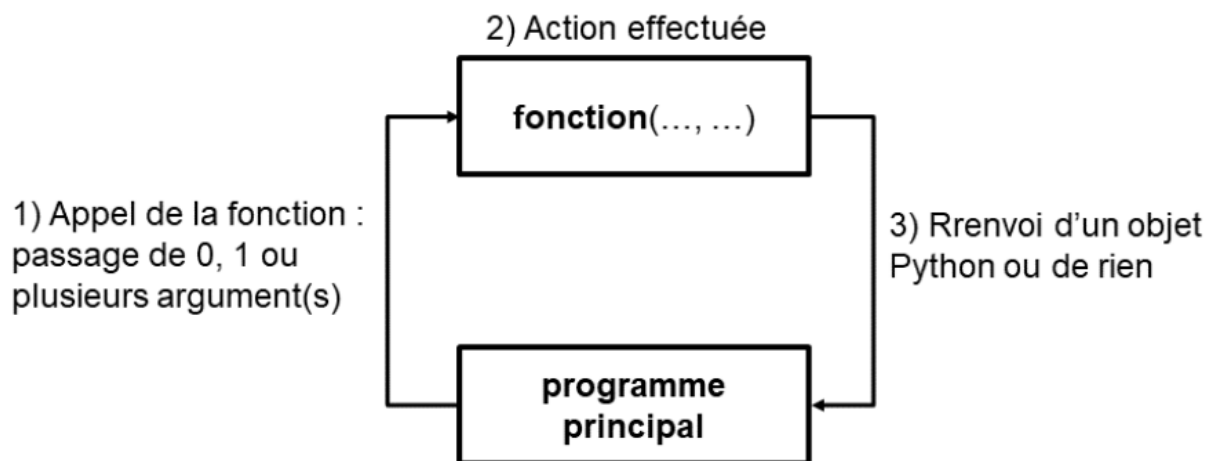
```
def addition(a, b):  
    return a + b  
  
resultat = addition(3, 4)  
print(resultat) # Affiche 7
```

## X. LES FONCTIONS

Les **fonctions** en Python sont des blocs de code réutilisables qui exécutent une tâche spécifique.



Elles permettent de diviser un programme en sous-unités logiques, facilitant ainsi la lisibilité, la maintenance, et la réutilisation du code. Python fournit des **fonctions intégrées** (comme `print()`, `len()`, etc.), mais il est également possible de définir ses propres **fonctions personnalisées**.



L'acronyme **DRY** signifie « Don't Repeat Yourself ». Les fonctions permettent de satisfaire ce principe en évitant la duplication de code. En effet, plus un code est dupliqué plusieurs fois dans un programme, plus il sera source d'erreurs, notamment lorsqu'il faudra le faire évoluer.

## 1. Définition d'une fonction

Une fonction en Python est définie à l'aide du mot-clé « **def** », suivi du nom de la fonction, de parenthèses (éventuellement avec des paramètres) et d'un deux-points « : ». Le corps de la fonction doit être indenté.

Syntaxe :

```
def nom_de_la_fonction(paramètre1, paramètre2, ...):  
    # Bloc de code de la fonction  
    return valeur # (optionnel)
```

Exemple simple :

```
def dire_bonjour():  
    print("Bonjour tout le monde!")  
  
dire_bonjour() # Appel de la fonction
```

## 2. Paramètres et arguments

Les **paramètres** sont des variables spécifiées entre les parenthèses lors de la définition de la fonction. Les **arguments** sont les valeurs passées aux paramètres lorsque la fonction est appelée.

Exemple avec paramètres :

```
def addition(a, b):  
    return a + b  
  
resultat = addition(3, 5)  
print(resultat) # Affiche 8
```

Ici, a et b sont des **paramètres**, tandis que 3 et 5 sont des **arguments** passés lors de l'appel de la fonction.

### 3. Paramètres par défaut

Il est possible de définir des **valeurs par défaut** pour les paramètres. Si l'argument correspondant n'est pas fourni lors de l'appel, la valeur par défaut sera utilisée.

```
def saluer(nom, message="Bonjour"):  
    print(f"{message}, {nom}!")  
  
saluer("Alice") # Affiche "Bonjour, Alice!" (utilise la valeur par défaut de message)  
saluer("Bob", "Salut") # Affiche "Salut, Bob!" (message personnalisé)
```

### 4. Arguments arbitraires (\*args)

Vous pouvez définir une fonction capable de recevoir un **nombre arbitraire d'arguments** en utilisant `*args`. Dans ce cas, les arguments supplémentaires sont regroupés dans un **tuple**.

```
def addition_multiple(*nombres):  
    total = 0  
    for nombre in nombres:  
        total += nombre  
    return total  
  
resultat = addition_multiple(1, 2, 3, 4)  
print(resultat) # Affiche 10
```

Ici, « **\*nombres** » permet à la fonction d'accepter un nombre variable d'arguments, que la fonction traite ensuite dans une boucle.

### 5. Arguments nommés arbitraires (\*\*kwargs)

Les **arguments nommés arbitraires** permettent de passer un nombre indéterminé d'arguments sous la forme de **paires clé-valeur**. Ces arguments sont regroupés dans un dictionnaire (dict).

```
def afficher_infos(**infos):  
    for clé, valeur in infos.items():  
        print(f"{clé}: {valeur}")  
  
afficher_infos(nom="Alice", âge=25, ville="Paris")
```

Cela affichera :

```
nom: Alice  
âge: 25  
ville: Paris
```

## 6. Retourner une valeur avec return

Une fonction peut renvoyer une valeur à l'aide de l'instruction `return`. Si `return` est omis, la fonction renvoie automatiquement `None`.

```
def multiplication(a, b):  
    return a * b  
  
resultat = multiplication(4, 5)  
print(resultat) # Affiche 20
```

Il est possible de renvoyer plusieurs valeurs sous forme de tuple :

```
def calculer(a, b):  
    somme = a + b  
    produit = a * b  
    return somme, produit  
  
somme, produit = calculer(3, 4)  
print(somme) # Affiche 7  
print(produit) # Affiche 12
```

## 7. Portée des variables (scope)

Les variables définies à l'intérieur d'une fonction sont des **variables locales** et ne sont accessibles qu'à l'intérieur de la fonction. En revanche, les **variables globales** sont accessibles depuis n'importe quel endroit du code.

- **Variable locale** : définie et utilisée uniquement à l'intérieur de la fonction.
- **Variable globale** : définie en dehors de la fonction et accessible à l'intérieur et à l'extérieur de la fonction.

```
def ma_fonction():  
    x = 10 # Variable locale  
    print(x)  
  
ma_fonction()  
print(x) # Erreur, x n'est pas définie en dehors de la fonction
```

### Utilisation de « global » :

Si vous souhaitez modifier une **variable globale** à l'intérieur d'une fonction, vous devez utiliser le mot-clé `global`.

```
x = 5  
  
def modifier_x():  
    global x  
    x = 10  
  
modifier_x()  
print(x) # Affiche 10
```

## 8. Fonctions anonymes (lambda)

Les **fonctions lambda** sont des fonctions anonymes en Python, définies en une seule ligne. Elles peuvent avoir plusieurs paramètres mais ne peuvent contenir qu'une seule expression.

```
lambda arguments: expression
```

Exemple :

```
addition = lambda a, b: a + b
print(addition(3, 4)) # Affiche 7
```

Les fonctions lambda sont souvent utilisées comme arguments pour d'autres fonctions.

Exemple avec « **sorted()** » :

```
nombres = [3, 1, 4, 2]
nombres_triés = sorted(nombres, key=lambda x: -x)
print(nombres_triés) # Affiche [4, 3, 2, 1]
```

## 9. Fonctions imbriquées

Python permet de définir des fonctions **à l'intérieur d'autres fonctions**. Ces fonctions internes ne sont accessibles qu'à l'intérieur de la fonction qui les contient.

```
def exterieur(a, b):
    def interieur(x):
        return x * 2
    return interieur(a) + interieur(b)

print(exterieur(3, 4)) # Affiche 14
```

## 10. Fonctions comme objets de première classe

En Python, les fonctions sont des **objets de première classe**. Cela signifie que vous pouvez les stocker dans des variables, les passer en arguments à d'autres fonctions, ou les renvoyer à partir d'une fonction.

Passer une fonction en argument :

```
def appliquer_fonction(fonction, valeur):  
    return fonction(valeur)  
  
resultat = appliquer_fonction(lambda x: x**2, 5)  
print(resultat)  # Affiche 25
```

## 11. Décorateurs

Les **décorateurs** sont un moyen puissant de modifier ou d'étendre le comportement des fonctions. Un décorateur est une fonction qui prend une fonction en entrée et renvoie une nouvelle fonction avec un comportement modifié.

Exemple simple de décorateur :

```
def decorateur(fonction):  
    def nouvelle_fonction():  
        print("Avant l'exécution de la fonction")  
        fonction()  
        print("Après l'exécution de la fonction")  
    return nouvelle_fonction  
  
@decorateur  
def dire_bonjour():  
    print("Bonjour!")  
  
dire_bonjour()
```

Cela affichera :

```
Avant l'exécution de la fonction  
Bonjour!  
Après l'exécution de la fonction
```

Le symbole « **@decorateur** » est utilisé pour appliquer le décorateur à la fonction **dire\_bonjour()**.

## 12. Fonctions génératrices (générateurs)

Les **générateurs** sont des fonctions spéciales qui produisent une séquence de valeurs au lieu de les renvoyer toutes en une seule fois. Ils sont définis à l'aide de l'instruction « **yield** ».

Exemple de générateur :

```
def generateur():  
    yield 1  
    yield 2  
    yield 3  
  
for valeur in generateur():  
    print(valeur)
```

Cela affichera :

```
1  
2  
3
```

Les générateurs permettent d'itérer sur de grandes séquences sans utiliser beaucoup de mémoire, car ils produisent les valeurs à la demande.

## 14. Fonctions récursives

Les fonctions récursives en Python sont des fonctions qui s'appellent elles-mêmes pour résoudre un problème. La récursion est une technique de programmation où un problème est résolu en le divisant en sous-problèmes plus simples, jusqu'à atteindre une condition de base où la solution est évidente et immédiate.

La récursion est souvent utilisée pour résoudre des problèmes complexes qui peuvent être naturellement divisés en sous-problèmes de structure similaire, comme le calcul du factoriel d'un nombre, le parcours d'un arbre, ou la suite de Fibonacci.

### 1. Composants d'une fonction récursive

Une fonction récursive comporte deux éléments essentiels :

**Condition de base** : C'est une condition qui permet de sortir de la récursion. Sans une telle condition, la fonction continuerait à s'appeler indéfiniment, provoquant une erreur.

**Appel récursif** : C'est l'appel de la fonction à elle-même avec des arguments modifiés qui progressent vers la condition de base.

### 2. Exemple simple : le calcul du factoriel

Le factoriel d'un nombre entier positif  $n$ , noté  $n!$ , est le produit de tous les nombres entiers de 1 à  $n$ .

Par exemple,  $5! = 5 * 4 * 3 * 2 * 1 = 120$ .

Le factoriel peut être défini de manière récursive :

**Condition de base** : Le factoriel de 1 est 1.

**Appel récursif** : Le factoriel de  $n$  est  $n * \text{factoriel}(n-1)$ .

```
def factoriel(n):
    if n == 1: # Condition de base
        return 1
    else:
        return n * factoriel(n - 1) # Appel récursif

# Appel de la fonction
print(factoriel(5)) # Affiche 120
```

Si  $n = 5$ , la fonction retourne  $5 * \text{factoriel}(4)$ .

Si  $n = 4$ , elle retourne  $4 * \text{factoriel}(3)$ .

...

Et ainsi de suite, jusqu'à ce que  $n = 1$ , où elle retourne 1 (la condition de base).

### 3. Récursivité et pile d'appels

Chaque fois qu'une fonction récursive est appelée, un nouvel appel est ajouté à la pile d'appels. La pile d'appels garde une trace des fonctions qui ont été appelées mais qui n'ont pas encore renvoyé de résultat. Chaque appel de fonction attend que l'appel récursif suivant renvoie une valeur avant de continuer.

Dans l'exemple du factoriel, la pile d'appels ressemblerait à ceci :

```
factoriel(5)
  factoriel(4)
    factoriel(3)
      factoriel(2)
        factoriel(1) -> renvoie 1
      factoriel(2) -> renvoie 2 * 1 = 2
    factoriel(3) -> renvoie 3 * 2 = 6
  factoriel(4) -> renvoie 4 * 6 = 24
factoriel(5) -> renvoie 5 * 24 = 120
```

Lorsque la condition de base est atteinte ( $n == 1$ ), la pile d'appels se vide progressivement et les résultats sont calculés en remontant.

#### 4. Récursion et itération

Toute fonction récursive peut être transformée en une fonction itérative (à l'aide d'une boucle). La récursion est souvent plus intuitive dans certains types de problèmes (comme le parcours d'arbres ou le calcul de la suite de Fibonacci), mais l'itération est souvent plus efficace en termes de mémoire et de performance car elle n'implique pas l'utilisation de la pile d'appels.

```
def factoriel_iteratif(n):  
    resultat = 1  
    for i in range(1, n + 1):  
        resultat *= i  
    return resultat  
  
print(factoriel_iteratif(5)) # Affiche 120
```

#### 5. Problème de la récursivité excessive

Une fonction récursive peut être inefficace si elle effectue des appels récursifs redondants. Par exemple, dans le cas du calcul de la suite de Fibonacci, une approche récursive naïve est inefficace car elle recalculerait plusieurs fois les mêmes termes.

Exemple de fonction récursive pour la suite de Fibonacci :

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(5)) # Affiche 5
```

Bien que cette fonction fonctionne, elle devient inefficace pour des valeurs plus grandes de  $n$ , car elle recalculera plusieurs fois les mêmes valeurs de Fibonacci.

Une solution plus efficace consisterait à utiliser la **mémoïsation** ou une approche itérative.

```
memo = {}

def fibonacci_memo(n):
    if n in memo:
        return memo[n]
    if n == 0:
        result = 0
    elif n == 1:
        result = 1
    else:
        result = fibonacci_memo(n - 1) + fibonacci_memo(n - 2)

    memo[n] = result
    return result

print(fibonacci_memo(50)) # Calcul rapide pour des valeurs élevées
```

## 6. Récursion directe et indirecte

**Récursion directe** : Une fonction s'appelle elle-même directement. C'est le cas dans les exemples précédents.

**Récursion indirecte** : Une fonction appelle une autre fonction, qui finit par rappeler la première fonction. Cela crée un cycle de récursion indirecte. Exemple de récursion indirecte :

```
def fonction_a():  
    print("Fonction A")  
    fonction_b()  
  
def fonction_b():  
    print("Fonction B")  
    fonction_a()  
  
# fonction_a() ou fonction_b() provoquerait une récursion infinie ici
```

## 14. Annotations de types

Python permet d'ajouter des **annotations de types** pour indiquer les types attendus pour les paramètres et le type de retour d'une fonction. Cependant, ces annotations ne sont pas vérifiées à l'exécution ; elles sont simplement là à titre informatif.

```
def addition(a: int, b: int) -> int:  
    return a + b
```

Dans cet exemple, on indique que a et b doivent être des entiers, et que la fonction retourne également un entier.

## 15. Docstrings

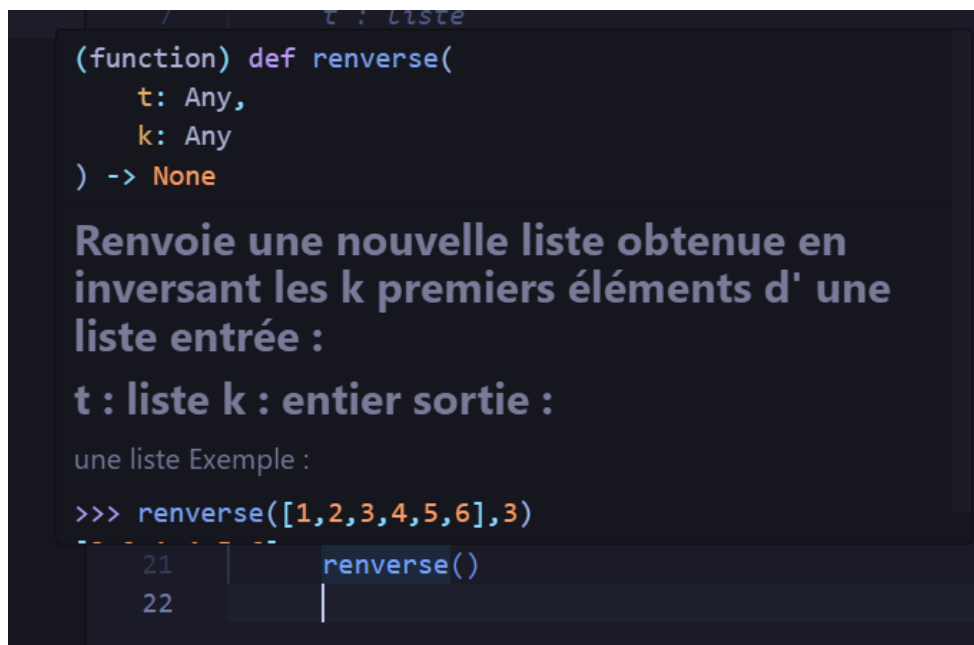
Les **docstrings** sont des chaînes de documentation utilisées pour décrire le comportement d'une fonction. Elles sont placées immédiatement après la définition de la fonction et permettent de documenter la fonction.

```
def addition(a, b):  
    """  
    Ajoute deux nombres et renvoie le résultat.  
  
    :param a: Premier nombre  
    :param b: Deuxième nombre  
    :return: La somme de a et b  
    """  
    return a + b
```

Les docstrings peuvent être consultées avec la fonction **help()** ou en accédant à l'attribut **\_\_doc\_\_** d'une fonction.

```
def renverse(t,k):  
    """  
    Renvoie une nouvelle liste obtenue en  
    inversant les k premiers éléments d'  
    une liste  
    entrée :  
    -----  
    t : liste  
    k : entier  
    sortie :  
    -----  
    une liste  
    Exemple :  
    >>> renverse([1,2,3,4,5,6],3)  
    [3,2,1,4,5,6]  
    """
```

Si on passe la souris sur l'appel de la fonction



## XI. GESTION DES ENTREES, SORTIES ET DES ERREURS

### Assertions

L'un des moyens simples pour pallier ce défaut de contrôle est d'utiliser la fonction « **assert** ».

Continuons à écrire la fonction `renverse(t,k)`

```
def renverse(t,k):  
    [3,2,1,4,5,6]  
    """  
  
    assert(type(t)==list), 'le premier argument doit être une  
    liste'  
    assert(len(t)>0), 'la liste doit être non vide'  
    assert(k>=2), 'on doit inverser au moins 2 éléments'  
    t2 = [] # nouveau tableau  
    n = min(k,len(t)) # nombre d'éléments à vraiment inverser (t  
    trop petit)  
  
    for i in range(n):  
        t2.append(t[n-1-i])  
  
    for i in range(n,len(t)):  
        t2.append(t[i])  
  
    return t2
```

puis appelons-la dans le point d'entrée correctement :

```
if __name__ == "__main__":  
    #Entrées Sorties Erreurs  
    #-----  
    print(renverse([], 3))
```

```

ain.py , line 18, in renverse
    assert(len(t)>0), 'la liste doit être non vide'
           ^^^^^^^
AssertionError: la liste doit être non vide
[nodemon] app crashed - waiting for file changes before starting...

```

Continuons a tester :

```

if __name__ == "__main__":
    #Entrées Sorties Erreurs
    #-----
    # print(renverse([], 3))

    print(renverse("abcdef",4))

    print(renverse(["a", "b", "c", "d", "e", "f"], 4))

```

```

AssertionError: le premier argument doit être une liste
[nodemon] app crashed - waiting for file changes before starting...

```

Puis enfin :

```

if __name__ == "__main__":
    #Entrées Sorties Erreurs
    #-----
    # print(renverse([], 3))

    # print(renverse("abcdef",4))

    print(renverse(["a", "b", "c", "d", "e", "f"], 4))

```

```

['d', 'c', 'b', 'a', 'e', 'f']
[nodemon] clean exit - waiting for changes before restart

```

## Test unitaires de fonctions

L'instruction «`assert()`» peut nous permettre d'effectuer des tests unitaires sur les fonctions définies dans le fichier courant ou les modules importés.

```
def maximum(T):  
    """  
    renvoie la valeur maximale des éléments de T - on suppose que  
    T possède des  
    données comparables  
    entrée : liste T  
    sortie : la valeur maximale ou None si T est vide  
    """  
    n = len(T)  
    if n==0: return None  
    max_temp = T[0]  
    for i in range(1,n):  
        max_temp = max(max_temp ,T[i])  
  
    return max_temp
```

```
#assertions en tant que test unitaires de fonctions  
assert(maximum([])==None), 'test1 fails: maximum([])!=None'  
assert(maximum([3])==3), 'test2 fails: maximum([3])!=3'  
assert(maximum([3,8,2,4,0,1,4])==8), 'test3 fails: maximum([3,8,2,4,0,1,4])!=8'  
assert(maximum([9,4,5,2])==9), 'test4 fails, maximum([9,4,5,2])!=9'  
assert(maximum([4,9,2,5,11])==11), 'test5 fails: maximum([4,9,2,5,11])!=11'  
assert(maximum([3,5,8,2,8,1])==8), 'test6 fails: maximum([3,5,8,2,8,1])!=8'
```

Ici tous les tests passent.

## Le bloc Try...except...else

Il existe un système plus général pour gérer les erreurs : des instructions qui permettent d'interrompre la fonction en cours et de signaler qu'une certaine erreur a été commise (division par zéro, accès à un élément inexistant d'une liste...).

Ce « message » est alors transmis à la séquence de fonctions appelantes jusqu'à, soit être intercepté par une fonction qui va traiter cette erreur, soit arriver au niveau principal du programme et arrêter l'exécution.

Syntaxe :

```
try:
    instructions
except erreur1:
    instruction
except (erreur2,erreur3):
    instructions
else:
    instructions
```

```
#Le bloc Try...except...else
try:
    f = open('mon fichier.txt')
except IOError:
    print("Problème d'ouverture du fichier")
```

```
Problème d'ouverture du fichier
[nodemon] clean exit - waiting for changes before restart
```

Le programme ne plante pas car l'erreur **IOError** a été gérée.

L'erreur générique en python est l'objet '**Exception**'  
L'instruction pour lever manuellement une exception est « **raise** ».

La levée de l'exception interrompt le programme.

```
def moyenne(l):
    """
    entrée : liste d'entiers ou flottants
    sortie : valeur moyenne """
    n = len(l)
    if len(l)==0:
        raise ValueError('La liste est vide')
    S = 0
    for x in l:
        try:
            S += x
        except Exception:
            raise Exception('Problème dans la somme des
            éléments')
    return S/n
```

2.0

[nodemon] clean exit - waiting for changes before restart

Testons avec ceci maintenant :

```
print(moyenne([1,"a",3]))
```

Exception: Problème dans la somme des éléments

[nodemon] app crashed - waiting for file changes before starting...

## XII. LES CHAINES DE CARACTERES (str)

Nous avons déjà abordé les chaînes de caractères dans les chapitres sur les Variables. Ici nous allons un peu plus loin, notamment avec les méthodes associées aux chaînes de caractères

Les **chaînes de caractères** (ou **strings**) en Python sont des séquences immuables d'éléments, chaque élément étant un caractère. Elles sont utilisées pour représenter du texte et constituent un type de données fondamental en Python. Les chaînes de caractères peuvent être définies entre des guillemets simples, des guillemets doubles, ou des guillemets triples (pour les chaînes multilignes).

Voici une explication complète des chaînes de caractères en Python, couvrant leur création, leurs propriétés, les opérations que vous pouvez effectuer dessus, et les fonctions associées.

### 1. Définition et création d'une chaîne de caractères

Les chaînes de caractères en Python peuvent être définies en utilisant des **guillemets simples** ('), **guillemets doubles** ("), ou des **guillemets triples** (''' ou """).

#### 1.1. Chaînes simples

- Avec guillemets simples :

```
#Les chaines de caracteres
#-----
chaine = 'Bonjour'
```

- Avec guillemets doubles :

```
chaine = "Bonjour"
```

- Avec guillemets triples pour des chaînes multilignes :

```
chaîne = """Ceci est une  
chaîne multi-lignes."""
```

## 1.2. Chaînes vides

Vous pouvez créer une **chaîne vide** simplement en mettant des guillemets sans contenu entre eux :

```
chaîne= ""
```

## 2. Propriétés des chaînes de caractères

Les chaînes de caractères en Python ont plusieurs propriétés importantes à connaître :

### 2.1. Immutabilité

Les chaînes de caractères sont **immuables** en Python. Cela signifie que vous ne pouvez **pas modifier** une chaîne une fois qu'elle a été créée. Si vous avez besoin de modifier une chaîne, vous devez en créer une nouvelle.

```
chaîne = "Bonjour"  
chaîne[0] = 'b'  # Erreur : Les chaînes sont immuables
```

```
TypeError: 'str' object does not support item assignment  
[nodemon] app crashed - waiting for file changes before starti
```

Utilisons plutôt un block « **try...except** »

```
chaîne = "Bonjour"

try:
    chaîne[0] = 'b' # Erreur : Les chaînes sont immuables
except TypeError:
    print("TypeError: les chaînes sont immuables")
```

```
TypeError: les chaînes sont immuables
[nodemon] clean exit - waiting for changes before restart
```

## 2.2. Indexation

Les chaînes de caractères sont des séquences **indexées**. Chaque caractère d'une chaîne a un indice, et vous pouvez accéder à ces caractères en utilisant des crochets [] et un index. Les indices commencent à 0.

```
chaîne = "Python"
print(chaîne[0]) # Affiche "P" (Le premier caractère)
print(chaîne[-1]) # Affiche "n" (Le dernier caractère)
```

```
P
n
[nodemon] clean exit - waiting for changes before restart
```

### 2.3. Slicing (tranchage)

Le **tranchage** permet d'extraire une sous-chaîne d'une chaîne en utilisant des indices. La syntaxe est « **chaîne[start:end:step]** », où :

- start est l'index de départ (inclusif).
- end est l'index de fin (exclusif).
- step est la taille du pas (optionnel).

```
chaîne = "Python"
print(chaîne[1:4]) # Affiche "yth"
print(chaîne[:3]) # Affiche "Pyt" (debut implicite 0)
print(chaîne[3:]) # Affiche "hon" (fin implicite jusqu'à la fin)
print(chaîne[::-2]) # Affiche "Pto" (tous les deux caractères)
```

```
yth
Pyt
hon
Pto
[nodemon] clean exit - waiting for cha
█
```

### 2.4. Longueur d'une chaîne

Vous pouvez obtenir la longueur (le nombre de caractères) d'une chaîne en utilisant la fonction **len()**.

```
print(len(chaîne)) # Affiche 6
```

### 3. Opérations sur les chaînes de caractères

Python permet de réaliser de nombreuses **opérations** sur les chaînes de caractères, comme la concaténation, la répétition, et la recherche.

#### 3.1. Concaténation

Vous pouvez **concaténer** (combiner) des chaînes de caractères en utilisant l'opérateur +.

```
# Concaténation
chaine1 = "Bonjour"
chaine2 = " tout le monde"
resultat = chaine1 + chaine2
print(resultat) # Affiche "Bonjour tout le monde"
```

```
Bonjour tout le monde
[nodemon] clean exit - waiting for changes before restart
```

#### 3.2. Répétition

Vous pouvez répéter une chaîne un certain nombre de fois en utilisant l'opérateur \*.

```
# Répétition
chaine = "Python! "
resultat = chaine * 3
print(resultat) # Affiche "Python! Python! Python! "
```

```
Python! Python! Python!
[nodemon] clean exit - waiting for changes before restart
```

### 3.3. Appartenance (in et not in)

Vous pouvez utiliser les opérateurs in et not in pour vérifier si une sous-chaîne est présente dans une chaîne.

```
#Appartenance (in et not in)  
chaîne = "Bonjour tout le monde"  
print("tout" in chaîne)  # True  
print("au revoir" not in chaîne)  # True
```

```
True  
True  
[nodemon] clean exit - waiting for changes before
```

### 3.4. Comparaison de chaînes

Les chaînes de caractères peuvent être comparées en utilisant les opérateurs de comparaison comme ==, !=, <, >, etc. Les comparaisons se basent sur l'ordre lexicographique (ordre alphabétique basé sur le code ASCII).

```
#Comparaison de chaînes  
print("Python" == "Python")  # True  
print("Python" < "python")  # True (car 'P' a un code ASCII  
inférieur à 'p')
```

```
True  
True  
[nodemon] clean exit - waiting for change
```

## 4. Méthodes des chaînes de caractères

Python propose de nombreuses méthodes intégrées pour manipuler les chaînes de caractères.

Voici quelques-unes des plus courantes :

### 4.1. Modification du format des chaînes

- **upper()** : Convertit une chaîne en majuscules.

```
# upper()
chaine = "python"
print(chaine.upper()) # Affiche "PYTHON"
```

- **lower()** : Convertit une chaîne en minuscules.

```
# Lower()
chaine = python
print(chaine.lower()) # Affiche "python"
```

- **capitalize()** : Met en majuscule le premier caractère de la chaîne.

```
#capitalize()
chaine = "python"
print(chaine.capitalize()) # Affiche "Python"
```

```
Python
[nodemon] clean exit - waiting t
```

- **title()** : Met en majuscule la première lettre de chaque mot.

```
#title()
chaine = "bonjour tout le monde"
print(chaine.title()) # Affiche "Bonjour Tout Le Monde"
```

```
Bonjour Tout Le Monde
[nodemon] clean exit - waiting for
```

#### 4.2. Méthodes de recherche et de remplacement

- **find()** : Renvoie l'index de la première occurrence d'une sous-chaîne. Si la sous-chaîne n'est pas trouvée, renvoie -1.

```
# find()
chaine = "bonjour tout le monde"
print(chaine.find("tout")) # Affiche 8
```

- **replace()** : Remplace toutes les occurrences d'une sous-chaîne par une autre.

```
# replace()
chaine = "Bonjour tout le monde"
print(chaine.replace("tout", "à tous")) # Affiche "Bonjour à tous Le monde"
```

```
Bonjour à tous le monde
[nodemon] clean exit - waiting for chan
```

#### 4.3. Suppression d'espaces et de caractères spéciaux

- **strip()** : Supprime les espaces ou caractères spécifiques au début et à la fin d'une chaîne.

```
# Suppression d'espaces et de caractères spéciaux
# strip()
chaîne = " Python "
print(chaîne.strip()) # Affiche "Python" (sans les
espaces)
```

```
Python
[nodemon] clean exit
```

- **lstrip()** et **rstrip()** : Suppriment les espaces à gauche ou à droite uniquement.

```
# lstrip() et rstrip()
chaîne = " Python "
print(chaîne.lstrip()) # Affiche "Python " (espaces
supprimés à gauche)
```

```
Python
[nodemon] clean exit - waiting f
```

#### 4.4. Division et jonction de chaînes

- **split()** : Divise une chaîne en une liste, en utilisant un délimiteur donné.

```
#Division et jonction de chaînes
# split()
chaine = "un,deux,trois"
print(chaine.split(",")) # Affiche ['un', 'deux', 'trois']
```

```
['un', 'deux', 'trois']
[nodemon] clean exit - waiting for
```

Par défaut le délimiteur est l'espace blanc.

```
chaine = "un deux trois"
print(chaine.split()) # Affiche ['un', 'deux', 'trois']
```

```
['un', 'deux', 'trois']
[nodemon] clean exit - waiti
```

- **join()** : Joint les éléments d'une séquence en une chaîne de caractères, en utilisant un délimiteur donné.

```
#join()
liste = ["un", "deux", "trois"]
print(", ".join(liste)) # Affiche "un, deux, trois"
```

```
un, deux, trois
[nodemon] clean exit - waiting for
```

## 5. Autres methodes

### 5.1. Méthode format()

La méthode **format()** permet également d'insérer des valeurs dans une chaîne. C'est une alternative moins puissante que les f-string, mais très utilisée.

```
# Méthode format()
nom = "Alice"
age = 25
print("Je m'appelle {} et j'ai {} ans.".format(nom, age))
```

```
Je m'appelle Alice et j'ai 25 ans.
[nodemon] clean exit - waiting for changes
```

### 5.2. Opérateur %

L'opérateur % est une ancienne méthode de formatage des chaînes, similaire à celle utilisée en C. est encore très utilisée en PHP.

```
#Opérateur %
nom = "Alice"
age = 25
print("Je m'appelle %s et j'ai %d ans." % (nom, age))
```

```
Je m'appelle Alice et j'ai 25 ans.
[nodemon] clean exit - waiting for changes
```

### 5.3. Count()

La méthode `.count()` compte le nombre d'occurrences d'une chaîne de caractères passée en argument :

```
#Count()
animaux = "girafe tigre"
print(animaux.count('i')) # affiche 2
```

```
2
[nodemon] clean exit
```

### 5.4. StartWith()

La méthode `.startswith()` vérifie si une chaîne de caractères commence par une autre chaîne de caractères :

```
#StartWith()
True
chaîne = "[nodemon] clean exit - waiting f!"
print(ch
```

## 7. Extraction de valeurs numériques d'une chaîne de caractères

Une tâche courante en Python est de lire une chaîne de caractères (provenant par exemple d'un fichier), d'en extraire des valeurs pour ensuite les manipuler.

Cela se fait en 2 étapes:

- On utilise la méthode **.split()** pour découper la chaîne,
- Puis on utilise la **déstructuration** pour récupérer les éléments du tableau généré dans des variables:

```
#Extraction de données d'une chaîne:  
chaine1 = "3.4 17.2 john"  
liste1 = chaine1.split()  
print(liste1)  
|  
nb1, nb2, nom = liste1 #déstructuration  
print(nb1)  
print(nb2)  
print(nom)
```

```
['3.4', '17.2', 'john']  
3.4  
17.2  
john  
[nodedmon] clean exit - waiting  
|
```

## 8. Chaînes Unicode

Python gère nativement les **chaînes Unicode**, ce qui signifie qu'il peut représenter des caractères de presque toutes les langues et systèmes d'écriture. Vous pouvez utiliser des caractères Unicode directement dans les chaînes ou les encoder avec des séquences d'échappement.

```
#Chaîne Unicode  
chaîne = u"\u00E9cole" # Caractère é (école)  
print(chaîne) # Affiche "école"
```

```
école  
[nodemon] clean exit - waiting fo
```

Reference complète des méthodes

[https://www.w3schools.com/python/python\\_ref\\_string.asp](https://www.w3schools.com/python/python_ref_string.asp)

## XIII. LES LISTES

### 1. Qu'est-ce qu'une Liste en Python ?

Une **liste** en Python est une collection **ordonnée** et **mutable** (modifiable) d'éléments. Les éléments d'une liste peuvent être de **n'importe quel type** : nombres, chaînes de caractères, booléens, voire d'autres listes (listes imbriquées).

Exemple :

```
ma_liste = [1, "Bonjour", 3.14, True]
```

### 2. Création de Listes

- Liste vide :

```
liste_vide = []
```

- Liste avec des éléments :

```
fruits = ["pomme", "banane", "cerise"]
```

- À partir d'un itérable :

```
ma_liste = list(range(5)) # [0, 1, 2, 3, 4]
```

### 3. Accès aux Éléments

Les éléments sont indexés à partir de 0.

- Accéder au premier élément :

```
print(fruits[0]) # Affiche "pomme"
```

- Accéder au dernier élément :

```
print(fruits[-1]) # Affiche "cerise"
```

#### 4. Modification des Éléments

Tu peux modifier les éléments en utilisant leur index.

```
fruits[1] = "orange"  
print(fruits) # ["pomme", "orange", "cerise"]
```

#### 5. Opérations sur les Listes

##### 5.1. Ajouter des Éléments

- **append()** : Ajoute un élément à la fin de la liste.

```
fruits.append("kiwi")  
# ["pomme", "orange", "cerise", "kiwi"]
```

- **insert()** : Insère un élément à une position spécifique.

```
fruits.insert(1, "mangue")  
# ["pomme", "mangue", "orange", "cerise", "kiwi"]
```

- **extend()** : Ajoute les éléments d'une autre collection à la liste.

```
fruits.extend(["ananas", "raisin"])  
# ["pomme", "mangue", "orange", "cerise", "kiwi", "ananas", "raisin"]
```

## 5.2. Supprimer des Éléments

- **remove()** : Supprime la première occurrence d'une valeur.

```
fruits.remove("orange")  
# ["pomme", "mangue", "cerise", "kiwi", "ananas", "raisin"]
```

- **pop()** : Supprime l'élément à un index donné (par défaut le dernier) et le renvoie.

```
dernier_fruit = fruits.pop()  
# dernier_fruit = "raisin", fruits = ["pomme", "mangue", "cerise", "kiwi", "ananas"]
```

- **del** : Supprime un élément ou une tranche d'éléments.

```
del fruits[0] # Supprime "pomme"  
# ["mangue", "cerise", "kiwi", "ananas"]
```

- **clear()** : Vide la liste.

```
fruits.clear()  
# []
```

## 5.3. Opérations de Fusion et Répétition

- **Concaténation (+)** :

```
liste1 = [1, 2, 3]  
liste2 = [4, 5, 6]  
liste3 = liste1 + liste2  
# [1, 2, 3, 4, 5, 6]
```

- **Répétition (\*) :**

```
liste_repetee = [0] * 5  
# [0, 0, 0, 0, 0]
```

#### 5.4. Slicing (Découpe de Liste)

Permet d'extraire une sous-liste.

```
nombres = [0, 1, 2, 3, 4, 5, 6]  
sous_liste = nombres[2:5] # [2, 3, 4]
```

- **Syntaxe du slicing :**

- **liste[start:stop:step]**
- **start** : Index de départ (inclus).
- **stop** : Index de fin (exclu).
- **step** : Pas de l'intervalle.

Exemples :

- `nombres[:3]` : Du début jusqu'à l'index 2.
- `nombres[3:]` : De l'index 3 jusqu'à la fin.
- `nombres[::2]` : Tous les éléments avec un pas de 2.

### 6. Méthodes des Listes

Les listes en Python possèdent de nombreuses méthodes intégrées pour les manipuler.

#### 6.1. count()

Compte le nombre d'occurrences d'une valeur dans la liste.

```
nombres = [1, 2, 3, 2, 2, 4]  
occurrences = nombres.count(2) # 3
```

### 6.2. index()

Retourne l'index de la première occurrence d'une valeur. Soulève une exception si la valeur n'est pas trouvée.

```
position = nombres.index(3) # 2
```

### 6.3. sort()

Trie les éléments de la liste en place.

- Tri par défaut (ordre croissant) :

```
nombres.sort()  
# [1, 2, 2, 2, 3, 4]
```

- Tri décroissant :

```
nombres.sort(reverse=True)  
# [4, 3, 2, 2, 2, 1]
```

- Tri avec une fonction clé :

```
mots = ["apple", "banana", "cherry"]  
mots.sort(key=len)  
# ["apple", "banana", "cherry"]
```

### 6.4. reverse()

Inverse l'ordre des éléments de la liste en place.

```
nombres.reverse()  
# [1, 2, 2, 2, 3, 4] devient [4, 3, 2, 2, 2, 1]
```

### 6.5. copy()

Retourne une copie superficielle de la liste.

```
nouv_liste = nombres.copy()
```

### 6.6. clear()

Supprime tous les éléments de la liste.

```
nombres.clear()  
# []
```

## 7. List Comprehensions (Compréhensions de Liste)

Une syntaxe concise pour créer des listes basées sur des itérables existants.

**Syntaxe :**

```
[expression for élément in iterable if condition]
```

Exemples :

- Liste des carrés de 0 à 9 :

```
carres = [x**2 for x in range(10)]  
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Filtrer les nombres pairs :

```
pairs = [x for x in range(20) if x % 2 == 0]  
# [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

- Convertir en majuscules :

```
fruits = ["pomme", "banane", "cerise"]
fruits_majuscules = [fruit.upper() for fruit in fruits]
# ["POMME", "BANANE", "CERISE"]
```

## 8. Listes Imbriquées

Les listes peuvent contenir d'autres listes en tant qu'éléments.

Exemple :

```
matrice = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

Accéder à un élément spécifique :

```
print(matrice[1][2]) # Affiche 6
```

## 9. Fonctions Utiles avec les Listes

- `len()` : Retourne le nombre d'éléments dans la liste.

```
taille = len(fruits)
```

- `max()` : Retourne l'élément maximal (pour les listes de nombres ou de chaînes).

```
max_valeur = max(nombres)
```

- **min()** : Retourne l'élément minimal.

```
min_valeur = min(nombres)
```

- **sum()** : Calcule la somme des éléments (pour les listes de nombres).

```
total = sum(nombres)
```

- **any()** : Retourne True si au moins un élément est vrai.

```
result = any([0, False, '', None, 5]) # True
```

- **all()** : Retourne True si tous les éléments sont vrais.

```
result = all([1, True, 'test']) # True
```

## 10. Itération sur les Listes

Tu peux parcourir les éléments d'une liste à l'aide de boucles.

### 10.1. Boucle for

```
for fruit in fruits:  
    print(fruit)
```

### 10.2. Boucle while

```
i = 0  
while i < len(fruits):  
    print(fruits[i])  
    i += 1
```

### 10.3. Itération avec index

```
for index, fruit in enumerate(fruits):  
    print(f"À l'index {index}, on a {fruit}")
```

## 11. Copies de Listes et Références

Il est important de comprendre la différence entre une copie par référence et une copie réelle.

- Copie par référence :

```
liste1 = [1, 2, 3]  
liste2 = liste1  
liste2.append(4)  
# liste1 et liste2 sont toutes deux [1, 2, 3, 4]
```

- Copie réelle :

- Avec slicing :

```
liste2 = liste1[:]  
liste2.append(5)  
# liste1 = [1, 2, 3, 4], liste2 = [1, 2, 3, 4, 5]
```

- Avec la méthode `copy()` :

```
liste2 = liste1.copy()
```

- Pour des copies profondes (listes imbriquées) :

```
import copy
liste2 = copy.deepcopy(liste1)
```

## 12. Les Listes sont Mutables

Tu peux modifier les listes après leur création, contrairement aux tuples qui sont immuables.

```
ma_liste = [1, 2, 3]
ma_liste[0] = 10
# ma_liste est maintenant [10, 2, 3]
```

## 13. Comparaison de Listes

Les listes peuvent être comparées élément par élément.

```
liste_a = [1, 2, 3]
liste_b = [1, 2, 3]
print(liste_a == liste_b) # True
```

## 14. Conversion en Liste

Tu peux convertir d'autres structures de données en listes.

- À partir d'une chaîne de caractères :

```
texte = "Python"
lettres = list(texte)
# ['P', 'y', 't', 'h', 'o', 'n']
```

- À partir d'un tuple :

```
mon_tuple = (1, 2, 3)
ma_liste = list(mon_tuple)
# [1, 2, 3]
```

- À partir d'un ensemble :

```
mon_ensemble = {1, 2, 3}
ma_liste = list(mon_ensemble)
# [1, 2, 3] (l'ordre peut varier)
```

## 15. Précautions avec les Listes

- Modification lors de l'itération :

Il est déconseillé de modifier une liste (ajouter ou supprimer des éléments) pendant que tu l'itères, car cela peut entraîner des comportements imprévisibles.

**Mauvais exemple :**

```
for x in ma_liste:
    if condition:
        ma_liste.remove(x)
```

**Solution :**

```
ma_liste = [x for x in ma_liste if not condition]
```

## 16. Exemples Pratiques

### 16.1. Filtrer une Liste

Retirer les nombres négatifs d'une liste.

```
nombres = [-5, 3, -1, 7, -2]
positifs = [x for x in nombres if x >= 0]
# [3, 7]
```

### 16.2. Aplatir une Liste Imbriquée

Transformer `[[1, 2], [3, 4], [5, 6]]` en `[1, 2, 3, 4, 5, 6]`.

```
liste_imbriquee = [[1, 2], [3, 4], [5, 6]]
aplatie = [element for sous_liste in liste_imbriquee for element in sous_liste]
```

### 16.3. Transposer une Matrice

Inverser les lignes et colonnes d'une matrice.

```
matrice = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
transposée = [[ligne[i] for ligne in matrice] for i in range(len(matrice[0]))]
```

Reference des méthodes de Listes :

[https://www.w3schools.com/python/python\\_ref\\_list.asp](https://www.w3schools.com/python/python_ref_list.asp)

## XIV. LES TUPLES

### 1. Qu'est-ce qu'un Tuple en Python ?

Un **tuple** est une structure de données en Python qui est une collection **ordonnée** d'éléments. Contrairement aux listes, les tuples sont **immuables**, ce qui signifie que **leurs éléments ne peuvent pas être modifiés** après leur création.

```
mon_tuple = (1, 2, 3)
```

### 2. Création de Tuples

#### 2.1. À l'aide de parenthèses

La manière la plus courante de créer un tuple est d'utiliser des parenthèses ().

```
vide = ()  
singleton = (1,) # Tuple avec un seul élément (note la virgule)  
mon_tuple = (1, 2, 3)
```

#### Attention !! :

Pour créer un tuple avec un seul élément, tu dois inclure une **virgule** après l'élément, sinon Python le considérera comme une simple expression entre parenthèses.

```
type((1))      # <class 'int'>  
type((1,))     # <class 'tuple'>
```

#### 2.2. Sans parenthèses (Emballage par défaut)

Python permet de créer des tuples sans parenthèses, en séparant simplement les éléments par des virgules.

```
mon_tuple = 1, 2, 3
```

### 2.3. À partir de la fonction tuple()

Tu peux convertir un itérable en tuple en utilisant la fonction tuple().

```
liste = [1, 2, 3]  
mon_tuple = tuple(liste) # (1, 2, 3)
```

### 3. Caractéristiques des Tuples

- **Ordonnés** : Les éléments ont un ordre défini et peuvent être accédés via leur index.
- **Immuables** : Une fois créés, les tuples ne peuvent pas être modifiés (ajoutés, supprimés ou modifiés).
- **Hétérogènes** : Les tuples peuvent contenir des éléments de différents types.

```
mon_tuple = (1, "Bonjour", 3.14, True)
```

### 4. Accès aux Éléments

Les éléments d'un tuple sont indexés à partir de 0.

```
mon_tuple = ('a', 'b', 'c', 'd')  
  
print(mon_tuple[0]) # 'a'  
print(mon_tuple[2]) # 'c'  
print(mon_tuple[-1]) # 'd' (dernier élément)
```

## 5. Immutabilité des Tuples

Une fois un tuple créé, tu ne peux pas modifier ses éléments.

```
mon_tuple = (1, 2, 3)
mon_tuple[0] = 10 # Erreur : 'tuple' object does not support item assignment
```

## 6. Opérations sur les Tuples

### 6.1. Concatenation (+)

Tu peux combiner des tuples en utilisant l'opérateur +.

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
tuple3 = tuple1 + tuple2 # (1, 2, 3, 4, 5, 6)
```

### 6.2. Répétition (\*)

Tu peux répéter un tuple en utilisant l'opérateur \*.

```
tuple_repeté = ('A',) * 3 # ('A', 'A', 'A')
```

### 6.3. Slicing (Découpe de Tuple)

Comme avec les listes, tu peux extraire une sous-partie d'un tuple.

```
mon_tuple = (0, 1, 2, 3, 4, 5)

sous_tuple = mon_tuple[1:4] # (1, 2, 3)
sous_tuple = mon_tuple[:3] # (0, 1, 2)
sous_tuple = mon_tuple[::2] # (0, 2, 4)
```

## 6.4. Longueur d'un Tuple

Utilise la fonction `len()` pour obtenir le nombre d'éléments.

```
longueur = len(mon_tuple) # 6
```

## 6.5. Appartenance (`in`, `not in`)

Vérifie si un élément est dans le tuple.

```
mon_tuple = (1, 2, 3)

print(2 in mon_tuple)      # True
print(4 not in mon_tuple)  # True
```

## 7. Méthodes des Tuples

Les tuples ont moins de méthodes que les listes en raison de leur immutabilité.

### 7.1. `count()`

Compte le nombre d'occurrences d'une valeur.

```
mon_tuple = (1, 2, 2, 3, 2)
nombre_de_2 = mon_tuple.count(2) # 3
```

### 7.2. `index()`

Retourne l'index de la première occurrence d'une valeur.

```
mon_tuple = ('a', 'b', 'c', 'b')
index_b = mon_tuple.index('b') # 1
```

## 8. Déballage (Unpacking) des Tuples

Tu peux assigner les éléments d'un tuple à des variables individuelles.

```
mon_tuple = (1, 2, 3)
a, b, c = mon_tuple

print(a) # 1
print(b) # 2
print(c) # 3
```

### 8.1. Déballage avec l'Opérateur \* (Asterisk)

Si tu ne connais pas le nombre exact d'éléments, tu peux utiliser \* pour capturer les éléments restants.

```
mon_tuple = (1, 2, 3, 4, 5)
a, *b, c = mon_tuple

print(a) # 1
print(b) # [2, 3, 4]
print(c) # 5
```

## 9. Tuples Imbriqués

Les tuples peuvent contenir d'autres tuples ou structures de données.

```
tuple_imbriqué = (1, (2, 3), [4, 5])

print(tuple_imbriqué[1]) # (2, 3)
print(tuple_imbriqué[2][1]) # 5
```

## 10. Tuples vs Listes

Similarités :

- **Ordre** : Les deux sont ordonnés.
- **Indexation** : Les deux supportent l'indexation et le slicing.
- **Hétérogénéité** : Les deux peuvent contenir des éléments de types différents.

Différences :

- **Mutabilité** :
  - **Liste** : Mutable (modifiable).
  - **Tuple** : Immuable (non modifiable).
- **Méthodes** :
  - Les listes ont plus de méthodes (comme `append()`, `remove()`, etc.).
- **Performance** :
  - Les tuples peuvent être légèrement plus rapides que les listes en raison de leur immuabilité.
- **Usage** :
  - **Listes** : Utilisées lorsque tu as besoin d'une collection modifiable.
  - **Tuples** : Utilisés pour des collections immuables, souvent pour représenter des enregistrements.

## 11. Utilisation des Tuples

### 11.1. Stockage de Données Immuables

Lorsque tu as une collection de données qui ne doit pas être modifiée, les tuples sont appropriés.

### 11.2. Clés de Dictionnaires

Les tuples peuvent être utilisés comme clés dans les dictionnaires (s'ils contiennent des éléments immuables), contrairement aux listes.

```
coordonnées = (45.4215, -75.6972)
lieux = {coordonnées: "Ottawa"}

print(lieux[(45.4215, -75.6972)]) # "Ottawa"
```

### 11.3. Retour Multiple de Fonctions

Les fonctions peuvent retourner plusieurs valeurs sous forme de tuple.

```
def div_mod(a, b):
    return a // b, a % b

quotient, reste = div_mod(10, 3)
```

## 12. Conversion entre Listes et Tuples

### 12.1. Tuple vers Liste

```
mon_tuple = (1, 2, 3)
ma_liste = list(mon_tuple)
```

### 12.2. Liste vers Tuple

```
ma_liste = [1, 2, 3]
mon_tuple = tuple(ma_liste)
```

### 13. Comparaison de Tuples

Les tuples sont comparés élément par élément.

```
tuple1 = (1, 2, 3)
tuple2 = (1, 2, 4)

print(tuple1 == tuple2)  # False
print(tuple1 < tuple2)   # True (car 3 < 4)
```

### 14. Fonctions Utiles avec les Tuples

- `len()` : Nombre d'éléments.
- `max()` : Valeur maximale.
- `min()` : Valeur minimale.
- `sum()` : Somme des éléments (si numériques).
- `any()` : True si au moins un élément est vrai.
- `all()` : True si tous les éléments sont vrais.

### 15. Tuples Nommes (`namedtuple`)

Le module `collections` fournit une classe `namedtuple`, qui permet de créer des tuples avec des champs nommés, améliorant la lisibilité.

```
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
p = Point(10, 20)

print(p.x)  # 10
print(p.y)  # 20
```

## 16. Méthodes de Compte et Index

### 16.1. count()

Compte le nombre de fois qu'un élément apparaît.

```
mon_tuple = (1, 2, 3, 2, 2)
print(mon_tuple.count(2))  # 3
```

### 16.2. index()

Trouve l'index de la première occurrence d'un élément.

```
print(mon_tuple.index(3))  # 2
```

## 17. Itération sur les Tuples

Tu peux parcourir les tuples de la même manière que les listes.

```
mon_tuple = ('pomme', 'banane', 'cerise')

for fruit in mon_tuple:
    print(fruit)
```

## 18. Tuples comme Clés de Dictionnaire

Les tuples peuvent être utilisés comme clés de dictionnaire si tous leurs éléments sont immuables.

```
coordonnées = (45.4215, -75.6972)
lieux = {
    coordonnées: "Ottawa",
    (51.5074, -0.1278): "Londres"
}

print(lieux[coordonnées])  # "Ottawa"
```

## 19. Immuabilité des Tuples et Éléments Mutables

Bien que le tuple lui-même soit immuable, il peut contenir des éléments mutables comme des listes.

```
mon_tuple = (1, [2, 3], 4)

# Modifier un élément mutable à l'intérieur du tuple
mon_tuple[1][0] = 'a'

print(mon_tuple)  # (1, ['a', 3], 4)
```

**Note :** La structure du tuple ne change pas, mais les éléments mutables à l'intérieur peuvent être modifiés.

## 20. Cas Pratiques

### 20.1. Échanger des Variables

Les tuples facilitent l'échange de valeurs entre variables.

```
a = 5
b = 10

a, b = b, a

print(a)  # 10
print(b)  # 5
```

### 20.2. Itération Simultanée sur Plusieurs Listes

Utilise la fonction `zip()` pour combiner plusieurs itérables.

```
noms = ('Alice', 'Bob', 'Charlie')
ages = (25, 30, 35)

for nom, age in zip(noms, ages):
    print(f"{nom} a {age} ans")
```

Reference des méthodes de Tuples:

[https://www.w3schools.com/python/python\\_ref\\_tuple.asp](https://www.w3schools.com/python/python_ref_tuple.asp)

## XV. LES DICTIONNAIRES

### 1. Qu'est-ce qu'un Dictionnaire en Python ?

Un **dictionnaire** en Python est une collection **non ordonnée**, **modifiable** et **indexée** de paires **clé-valeur**. Les dictionnaires sont utilisés pour stocker des données telles que des annuaires téléphoniques, des informations sur des objets, etc.

```
mon_dictionnaire = {  
    "nom": "Alice",  
    "âge": 25,  
    "ville": "Paris"  
}
```

### 2. Caractéristiques des Dictionnaires

- **Non ordonnés** (avant Python 3.7) : Les éléments ne sont pas stockés dans un ordre particulier. Cependant, depuis Python 3.7, les dictionnaires conservent l'ordre d'insertion.
- **Modifiables** : Tu peux ajouter, supprimer ou modifier des éléments après la création du dictionnaire.
- **Indexés par des clés** : Les clés peuvent être de n'importe quel type immuable (chaînes, nombres, tuples immuables).
- **Clés uniques** : Chaque clé doit être unique au sein d'un dictionnaire.

### 3. Création de Dictionnaires

#### 3.1. Avec des accolades {}

```
mon_dictionnaire = {  
    "clé1": "valeur1",  
    "clé2": "valeur2"  
}
```

### 3.2. Avec la fonction dict()

```
mon_dictionnaire = dict(clé1="valeur1", clé2="valeur2")
```

**Note :** Les clés doivent être des chaînes de caractères lorsqu'on utilise cette méthode.

### 3.3. À partir d'une liste de tuples

```
liste_de_tuples = [("clé1", "valeur1"), ("clé2", "valeur2")]  
mon_dictionnaire = dict(liste_de_tuples)
```

### 3.4. Dictionnaire vide

```
dictionnaire_vide = {}
```

## 4. Accès aux Valeurs

### 4.1. Avec des clés

```
mon_dictionnaire = {"nom": "Alice", "âge": 25}  
print(mon_dictionnaire["nom"]) # Affiche "Alice"
```

### 4.2. Avec la méthode get()

Permet d'éviter une erreur si la clé n'existe pas.

```
âge = mon_dictionnaire.get("âge") # Retourne 25  
pays = mon_dictionnaire.get("pays", "Inconnu") # Retourne "Inconnu"
```

## 5. Modification des Valeurs

### 5.1. Ajouter ou Modifier une Entrée

```
mon_dictionnaire["ville"] = "Paris" # Ajoute une nouvelle entrée  
mon_dictionnaire["âge"] = 26       # Modifie une entrée existante
```

### 5.2. Mise à Jour avec update()

```
mon_dictionnaire.update({"pays": "France", "âge": 27})
```

## 6. Suppression d'Éléments

### 6.1. Avec del

```
del mon_dictionnaire["âge"]
```

### 6.2. Avec la méthode pop()

Supprime l'élément et retourne sa valeur.

```
âge = mon_dictionnaire.pop("âge", None) # Retourne la valeur associée à "âge"
```

### 6.3. Avec la méthode popitem()

Supprime et retourne le dernier élément inséré (utile pour implémenter des piles LIFO).

```
clé, valeur = mon_dictionnaire.popitem()
```

## 6.4. Vider le Dictionnaire

```
mon_dictionnaire.clear()
```

## 7. Itération sur les Dictionnaires

### 7.1. Parcourir les Clés

```
for clé in mon_dictionnaire:  
    print(clé)
```

### 7.2. Parcourir les Valeurs

```
for valeur in mon_dictionnaire.values():  
    print(valeur)
```

### 7.3. Parcourir les Paires Clé-Valeur

```
for clé, valeur in mon_dictionnaire.items():  
    print(f"{clé}: {valeur}")
```

## 8. Méthodes Utiles des Dictionnaires

### 8.1. keys()

Retourne une vue des clés du dictionnaire.

```
clés = mon_dictionnaire.keys()
```

### 8.2. values()

Retourne une vue des valeurs du dictionnaire.

```
valeurs = mon_dictionnaire.values()
```

### 8.3. items()

Retourne une vue des paires clé-valeur.

```
items = mon_dictionnaire.items()
```

### 8.4. copy()

Retourne une copie superficielle du dictionnaire.

```
nouveau_dictionnaire = mon_dictionnaire.copy()
```

### 8.5. fromkeys()

Crée un nouveau dictionnaire avec des clés données et une valeur par défaut.

```
clés = ["nom", "âge", "ville"]  
valeur_par_défaut = None  
nouveau_dictionnaire = dict.fromkeys(clés, valeur_par_défaut)
```

### 8.6. setdefault()

Retourne la valeur associée à une clé, et si la clé n'existe pas, l'ajoute avec une valeur par défaut.

```
âge = mon_dictionnaire.setdefault("âge", 30)
```

## 9. Vérification de l'Existence d'une Clé

```
if "nom" in mon_dictionnaire:  
    print("La clé 'nom' existe.")
```

## 10. Dictionnaires Imbriqués

Les dictionnaires peuvent contenir d'autres dictionnaires.

```
étudiants = {  
    "Alice": {"âge": 25, "ville": "Paris"},  
    "Bob": {"âge": 22, "ville": "Lyon"},  
}
```

Accès aux éléments imbriqués :

```
âge_alice = étudiants["Alice"]["âge"]
```

## 11. Exemples Pratiques

### 11.1. Compter les Occurrences

Compter le nombre de fois que chaque élément apparaît dans une liste.

```
liste = ['pomme', 'banane', 'pomme', 'orange', 'banane', 'pomme']  
compteur = {}  
  
for fruit in liste:  
    compteur[fruit] = compteur.get(fruit, 0) + 1  
  
print(compteur) # {'pomme': 3, 'banane': 2, 'orange': 1}
```

## 11.2. Traduction Simple

Utiliser un dictionnaire comme un petit dictionnaire de traduction.

```
traduction = {  
    "bonjour": "hello",  
    "monde": "world"  
}  
  
phrase = "bonjour le monde"  
traduite = " ".join([traduction.get(mot, mot) for mot in phrase.split()])  
print(traduite) # "hello le world"
```

## 12. Tri des Dictionnaires

### 12.1. Trier les Clés

```
dictionnaire_trié = dict(sorted(mon_dictionnaire.items()))
```

### 12.2. Trier par Valeurs

```
dictionnaire_trié = dict(sorted(mon_dictionnaire.items(), key=lambda item: item[1]))
```

## 13. Fusion de Dictionnaires

### 13.1. Avec update()

```
dictionnaire1 = {"a": 1, "b": 2}  
dictionnaire2 = {"b": 3, "c": 4}  
  
dictionnaire1.update(dictionnaire2)  
# dictionnaire1 est maintenant {'a': 1, 'b': 3, 'c': 4}
```

### 13.2. Opérateur \*\* (Depuis Python 3.5)

```
fusionné = {**dictionnaire1, **dictionnaire2}
```

## 14. Copie de Dictionnaires

### 14.1. Copie Superficielle

```
copie = mon_dictionnaire.copy()
```

### 14.2. Copie Profonde

Utilise le module `copy` pour copier les objets imbriqués.

```
import copy

copie_profonde = copy.deepcopy(mon_dictionnaire)
```

## 15. Types de Clés

Les clés doivent être des objets immuables (hashables). Les types couramment utilisés sont :

- Chaînes de caractères
- Nombres (int, float)
- Tuples (contenant uniquement des éléments immuables)

Exemple avec un tuple comme clé :

```
coordonnées = {(0, 0): "Origine", (1, 2): "Point A"}
```

## 16. Compréhensions de Dictionnaires

Crée un dictionnaire de manière concise.

```
carrés = {x: x**2 for x in range(5)}  
# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

## 17. Fonctions Utiles avec les Dictionnaires

- **len()** : Retourne le nombre de paires clé-valeur.

```
taille = len(mon_dictionnaire)
```

- **any()** : Retourne True si au moins une clé est vraie.

```
any(mon_dictionnaire)
```

- **all()** : Retourne True si toutes les clés sont vraies.

```
all(mon_dictionnaire)
```

## 18. Dictionnaires et JSON

Les dictionnaires peuvent être utilisés pour représenter des données JSON.

### 18.1. Conversion en JSON

```
import json  
  
données_json = json.dumps(mon_dictionnaire)
```

## 18.2. Chargement depuis JSON

```
mon_dictionnaire = json.loads(données_json)
```

## 19. Exemples Avancés

### 19.1. Compter les Caractères d'une Chaîne

```
texte = "hello world"
compteur = {}

for caractère in texte:
    compteur[caractère] = compteur.get(caractère, 0) + 1

print(compteur)
```

### 19.2. Inverser un Dictionnaire

Attention aux valeurs non uniques.

```
dictionnaire = {'a': 1, 'b': 2, 'c': 3}
inverse = {valeur: clé for clé, valeur in dictionnaire.items()}
```

## 20. Modules Utiles pour les Dictionnaires

### 20.1. collections.OrderedDict

Avant Python 3.7, les dictionnaires n'étaient pas ordonnés. OrderedDict conservait l'ordre d'insertion.

```
from collections import OrderedDict

d = OrderedDict()
d['a'] = 1
d['b'] = 2
```

### 20.2. collections.defaultdict

Permet de définir une valeur par défaut pour les clés manquantes.

```
from collections import defaultdict

d = defaultdict(int)
d['a'] += 1 # Pas d'erreur même si 'a' n'existait pas
```

### 20.3. collections.Counter

Spécialement conçu pour compter les occurrences.

```
from collections import Counter

compteur = Counter("hello world")
print(compteur)
```

## 21. Gestion des Exceptions

Lorsqu'une clé n'existe pas, accéder à cette clé avec `mon_dictionnaire[clé]` soulève une **KeyError**.

```
try:
    valeur = mon_dictionnaire["clé_inexistante"]
except KeyError:
    print("La clé n'existe pas.")
```

## 22. Trucs et Astuces

### 22.1. Fusionner des Dictionnaires (Depuis Python 3.9)

Utiliser l'opérateur `|` pour fusionner.

```
dictionnaire1 = {"a": 1}
dictionnaire2 = {"b": 2}
fusionné = dictionnaire1 | dictionnaire2
```

### 22.2. Supprimer des Clés avec une Compréhension

```
mon_dictionnaire = {k: v for k, v in mon_dictionnaire.items() if condition}
```

## 23. Limites des Dictionnaires

- **Clés Immuables** : Les clés doivent être des objets immuables.
- **Clés Uniques** : Pas de doublons de clés.
- **Non Ordonnés (avant Python 3.7)** : Pas adapté si l'ordre est important (utiliser **OrderedDict** si nécessaire).

Reference des méthodes de Dictionnaires:

[https://www.w3schools.com/python/python\\_ref\\_dictionary.asp](https://www.w3schools.com/python/python_ref_dictionary.asp)

## XVI. LES SETs

### 1. Qu'est-ce qu'un Ensemble (Set) en Python ?

Un **ensemble** (set) en Python est une collection **non ordonnée** d'éléments **uniques** et **mutables**. Les ensembles sont utilisés pour stocker des éléments sans doublons et pour effectuer des tests d'appartenance et des opérations mathématiques telles que l'union, l'intersection, la différence, etc.

```
mon_ensemble = {1, 2, 3}
```

### 2. Caractéristiques des Ensembles

- **Non ordonnés** : Les éléments ne sont pas stockés dans un ordre particulier, et cet ordre peut changer.
- **Éléments uniques** : Un ensemble ne contient pas de doublons. Chaque élément est unique.
- **Mutable** : Tu peux ajouter ou supprimer des éléments après la création de l'ensemble.
- **Éléments immuables** : Les éléments d'un ensemble doivent être immuables (entiers, chaînes de caractères, tuples immuables, etc.).

### 3. Création d'Ensembles

#### 3.1. Avec des accolades {}

```
mon_ensemble = {1, 2, 3}
```

**Attention** : Pour créer un ensemble vide, vous **ne pouvez pas** utiliser {}, car cela crée un dictionnaire vide.

### 3.2. Avec la fonction `set()`

```
ensemble_vide = set()
```

### 3.3. À partir d'un itérable

```
liste = [1, 2, 2, 3, 4]  
mon_ensemble = set(liste)  # {1, 2, 3, 4}
```

## 4. Accès aux Éléments

Étant donné que les ensembles sont non ordonnés, tu ne peux pas accéder aux éléments via un index. Pour parcourir les éléments, utilise une boucle `for`.

```
for élément in mon_ensemble:  
    print(élément)
```

## 5. Opérations sur les Ensembles

### 5.1. Ajouter des Éléments

- **`add()`** : Ajoute un élément à l'ensemble. Si l'élément existe déjà, l'ensemble ne change pas.

### 5.2. Supprimer des Éléments

- **`remove()`** : Supprime un élément. Soulève une erreur **`KeyError`** si l'élément n'existe pas.

```
mon_ensemble.remove(3)
```

- **discard()** : Supprime un élément si présent. Ne fait rien si l'élément n'existe pas.

```
mon_ensemble.discard(6)
```

- **pop()** : Supprime et retourne un élément arbitraire. Soulève une erreur `KeyError` si l'ensemble est vide.

```
élément = mon_ensemble.pop()
```

- **clear()** : Vide l'ensemble.

```
mon_ensemble.clear()
```

## 6. Opérations Mathématiques sur les Ensembles

### 6.1. Union

Combine tous les éléments de deux ensembles.

- **Opérateur |**

```
ensemble1 = {1, 2, 3}
ensemble2 = {3, 4, 5}
union = ensemble1 | ensemble2  # {1, 2, 3, 4, 5}
```

- **Méthode union()**

```
union = ensemble1.union(ensemble2)
```

## 6.2. Intersection

Éléments communs aux deux ensembles.

- **Opérateur** &

```
intersection = ensemble1 & ensemble2 # {3}
```

- **Méthode** intersection()

```
intersection = ensemble1.intersection(ensemble2)
```

## 6.3. Différence

Éléments présents dans le premier ensemble mais pas dans le second.

- **Opérateur** -

```
différence = ensemble1 - ensemble2 # {1, 2}
```

- **Méthode** difference()

```
différence = ensemble1.difference(ensemble2)
```

## 6.4. Différence Symétrique

Éléments présents dans l'un ou l'autre des ensembles, mais pas dans les deux.

- **Opérateur** ^

```
diff_sym = ensemble1 ^ ensemble2 # {1, 2, 4, 5}
```

- **Méthode** `symmetric_difference()`

```
diff_sym = ensemble1.symmetric_difference(ensemble2)
```

## 7. Méthodes des Ensembles

### 7.1. `add(element)`

Ajoute un élément à l'ensemble.

```
mon_ensemble.add(6)
```

### 7.2. `update(*others)`

Ajoute tous les éléments d'un autre itérable (ensembles, listes, etc.) à l'ensemble.

```
mon_ensemble.update([7, 8], {9, 10})
```

### 7.3. `remove(element)`

Supprime un élément. Soulève une **KeyError** si l'élément n'existe pas.

```
mon_ensemble.remove(6)
```

### 7.4. `discard(element)`

Supprime un élément si présent.

```
mon_ensemble.discard(11)
```

### 7.5. pop()

Supprime et retourne un élément arbitraire.

```
élément = mon_ensemble.pop()
```

### 7.6. clear()

Vide l'ensemble.

```
mon_ensemble.clear()
```

### 7.7. union(\*others)

Retourne un nouvel ensemble contenant l'union.

```
nouvel_ensemble = mon_ensemble.union(autre_ensemble)
```

### 7.8. intersection(\*others)

Retourne un nouvel ensemble contenant l'intersection.

```
nouvel_ensemble = mon_ensemble.intersection(autre_ensemble)
```

### 7.9. difference(\*others)

Retourne un nouvel ensemble contenant la différence.

```
nouvel_ensemble = mon_ensemble.difference(autre_ensemble)
```

### 7.10. `symmetric_difference(other)`

Retourne un nouvel ensemble contenant la différence symétrique.

```
nouvel_ensemble = mon_ensemble.symmetric_difference-autre_ensemble)
```

### 7.11. `issubset(other)`

Vérifie si l'ensemble est un sous-ensemble de other.

```
mon_ensemble.issubset-autre_ensemble)
```

### 7.12. `issuperset(other)`

Vérifie si l'ensemble est un sur-ensemble de other.

```
mon_ensemble.issuperset-autre_ensemble)
```

### 7.13. `isdisjoint(other)`

Vérifie si les ensembles n'ont aucun élément en commun.

```
mon_ensemble.isdisjoint-autre_ensemble)
```

## 8. Exemples Pratiques

### 8.1. Suppression de Doublons dans une Liste

```
liste = [1, 2, 2, 3, 4, 4, 5]  
ensemble_unique = set(liste)  
liste_unique = list(ensemble_unique) # [1, 2, 3, 4, 5]
```

## 8.2. Tests d'Appartenance Rapides

Les ensembles offrent des performances élevées pour les tests d'appartenance.

```
if élément in mon_ensemble:  
    print("L'élément est présent.")
```

## 8.3. Opérations Mathématiques

Calculer les éléments communs, les différences entre ensembles, etc.

## 9. Compréhensions d'Ensembles

Crée un ensemble de manière concise.

```
ensemble_carres = {x**2 for x in range(10)}  
# {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

## 10. Ensembles Immuables : frozenset

Un frozenset est un ensemble **immuable**. Une fois créé, il ne peut pas être modifié.

### 10.1. Création d'un frozenset

```
mon_frozenset = frozenset([1, 2, 3])
```

### 10.2. Utilisation

Les **frozensets** peuvent être utilisés comme clés de dictionnaire ou éléments d'un autre ensemble.

```
ensemble_de_frozensets = {frozenset({1, 2}), frozenset({3, 4})}
```

## 11. Fonctions Utiles avec les Ensembles

- **len()** : Retourne le nombre d'éléments.

```
taille = len(mon_ensemble)
```

- **min()** et **max()** : Retourne l'élément minimal ou maximal.

```
min_val = min(mon_ensemble)  
max_val = max(mon_ensemble)
```

- **sum()** : Calcule la somme des éléments (si numériques).

```
total = sum(mon_ensemble)
```

- **any()** : Retourne True si au moins un élément est vrai.

```
result = any(mon_ensemble)
```

- **all()** : Retourne True si tous les éléments sont vrais.

```
result = all(mon_ensemble)
```

## 12. Limites des Ensembles

- **Éléments Immuables** : Les éléments doivent être immuables. Tu ne peux pas ajouter des listes ou des dictionnaires à un ensemble.

```
mon_ensemble = {1, 2, [3, 4]} # Erreur : TypeError
```

- **Non Indexés** : Tu ne peux pas accéder aux éléments par index.

## 13. Exemples Avancés

### 13.1. Éliminer les Mots en Double dans une Phrase

```
phrase = "bonjour le monde bonjour"  
mots_uniques = set(phrase.split())  
print(mots_uniques) # {'bonjour', 'le', 'monde'}
```

### 13.2. Trouver les Caractères Communs entre Deux Chaînes

```
chaine1 = "python"  
chaine2 = "typhon"  
ensemble1 = set(chaine1)  
ensemble2 = set(chaine2)  
communs = ensemble1 & ensemble2 # {'p', 'y', 't', 'h', 'o', 'n'}
```

## 14. Copie d'Ensembles

### 14.1. Copie Superficielle

```
nouvel_ensemble = mon_ensemble.copy()
```

### 14.2. Copie avec le Constructeur set()

```
nouvel_ensemble = set(mon_ensemble)
```

## 15. Exemples d'Utilisation des Opérateurs

### 15.1. Vérifier si un Ensemble est un Sous-ensemble

```
ensemble_a = {1, 2}
ensemble_b = {1, 2, 3, 4}

print(ensemble_a <= ensemble_b)  # True
print(ensemble_b >= ensemble_a)  # True
```

### 15.2. Vérifier si les Ensembles sont Égaux

```
ensemble_c = {3, 4, 1, 2}
print(ensemble_b == ensemble_c)  # True
```

## 16. Méthodes Avancées

### 16.1. `difference_update(*others)`

Met à jour l'ensemble en supprimant les éléments présents dans les autres ensembles.

```
ensemble_a.difference_update(ensemble_b)
```

### 16.2. `intersection_update(*others)`

Met à jour l'ensemble en ne conservant que les éléments communs.

```
ensemble_a.intersection_update(ensemble_b)
```

### 16.3. `symmetric_difference_update(other)`

Met à jour l'ensemble avec la différence symétrique.

```
ensemble_a.symmetric_difference_update(ensemble_b)
```

## 17. Bonnes Pratiques

- **Utiliser des Ensembles pour Tester l'Appartenance** : Les tests in sont plus rapides avec les ensembles qu'avec les listes.
- **Utiliser les Méthodes Plutôt que les Opérateurs pour la Lisibilité** : Les méthodes comme `union()`, `intersection()` sont plus explicites.

## 18. Cas d'Utilisation des Ensembles

- **Éliminer les Doublons** : Convertir une liste en ensemble pour supprimer les doublons.
- **Opérations en Théorie des Ensembles** : Calculer des unions, intersections, différences entre groupes de données.
- **Tests d'Appartenance Rapides** : Vérifier si un élément appartient à un ensemble de valeurs.

## 19. Limitations

- **Non Ordonnés** : Si l'ordre des éléments est important, les ensembles ne sont pas adaptés.
- **Éléments Immuables Seulement** : Tu ne peux pas ajouter des objets mutables comme des listes ou des dictionnaires.

Reference des méthodes de Sets:

[https://www.w3schools.com/python/python\\_ref\\_set.asp](https://www.w3schools.com/python/python_ref_set.asp)

## **XVII. LES FILES**

### **1. Qu'est-ce qu'une File en Python ?**

Une **file** (queue en anglais) est une structure de données de type **FIFO** (First-In, First-Out), c'est-à-dire que le premier élément ajouté est le premier élément à être retiré. Les files sont utilisées lorsqu'un ordre de traitement doit être respecté, comme dans la gestion des tâches, le traitement des données en temps réel, ou le parcours en largeur d'un graphe.

### **2. Implémentation des Files en Python**

Python n'a pas de type de file intégré comme les listes ou les dictionnaires, mais il existe plusieurs manières d'implémenter des files :

- **Utiliser une liste** (list)
- **Utiliser la classe** deque **du module** collections
- **Utiliser le module** queue
- **Utiliser le module** asyncio **pour les files asynchrones**

### **3. Utilisation des Listes pour les Files**

Bien que les listes puissent être utilisées pour implémenter des files, elles ne sont pas optimisées pour cela, surtout pour de grandes files.

#### **3.1. Création d'une File avec une Liste**

```
file = []
```

#### **3.2. Enfiler (ajouter) un élément**

Utilise la méthode **append()** pour ajouter un élément à la fin de la liste.

```
file.append('a')
file.append('b')
file.append('c')
# file = ['a', 'b', 'c']
```

### 3.3. Défiler (retirer) un élément

Utilise la méthode `pop(0)` pour retirer le premier élément.

```
element = file.pop(0)
# element = 'a', file = ['b', 'c']
```

**Attention :** L'opération `pop(0)` est coûteuse en termes de performance car tous les éléments suivants doivent être décalés (complexité  $O(n)$ ).

## 4. Utilisation de deque du module collections

La classe **deque** (double-ended queue) est optimisée pour ajouter et retirer des éléments aux deux extrémités avec des performances constantes en temps  $O(1)$ .

### 4.1. Importer deque

```
from collections import deque
```

### 4.2. Création d'une File avec deque

```
file = deque()
```

### 4.3. Enfiler un élément

Utilise la méthode **append()** pour ajouter à la fin.

```
file.append('a')
file.append('b')
file.append('c')
# file = deque(['a', 'b', 'c'])
```

### 4.4. Défiler un élément

Utilise la méthode **popleft()** pour retirer du début.

```
element = file.popleft()
# element = 'a', file = deque(['b', 'c'])
```

### 4.5. Avantages de deque

- **Performances constantes** pour les opérations d'ajout et de suppression aux deux extrémités.
- **Flexibilité** : Peut être utilisée comme file (FIFO) ou pile (LIFO).
- **Méthodes supplémentaires** pour la manipulation efficace des données.

## 5. Utilisation du Module queue

Le module queue fournit des classes pour implémenter des files **sécurisées pour les threads**.

### 5.1. Importer Queue

```
from queue import Queue
```

## 5.2. Création d'une File

```
file = Queue()
```

## 5.3. Enfiler un élément

Utilise la méthode `put()`.

```
file.put('a')  
file.put('b')  
file.put('c')
```

## 5.4. Défiler un élément

Utilise la méthode `get()`.

```
element = file.get()  
# element = 'a'
```

## 5.5. Caractéristiques du Module queue

- **Sûreté pour les threads** : Adapté pour les applications multithreads.
- **Blocs et timeouts** : Les méthodes `get()` et `put()` peuvent bloquer jusqu'à ce qu'un élément soit disponible ou que de l'espace soit disponible.
- **Files de taille limitée** : Possibilité de définir une taille maximale pour la file.

## 6. Comparaison des Méthodes d'Implémentation

Méthode	Performance	Thread-safe	Notes
Liste ( <code>list</code> )	Mauvaise	Non	<code>pop(0)</code> est coûteux en temps $O(n)$
<code>deque</code>	Excellente	Non	Opérations en temps $O(1)$
<code>queue.Queue</code>	Bonne	Oui	Conçu pour les threads, méthodes bloquantes

## 7. Méthodes Associées à deque

### 7.1. `append(x)`

Ajoute l'élément `x` à la **fin** de la file.

```
file.append(x)
```

### 7.2. `appendleft(x)`

Ajoute l'élément `x` au **début** de la file.

```
file.appendleft(x)
```

### 7.3. `popleft()`

Retire et retourne l'élément au **début** de la file.

```
x = file.popleft()
```

### 7.4. `pop()`

Retire et retourne l'élément à la **fin** de la file.

```
x = file.pop()
```

## 7.5. Autres Méthodes

- **clear()** : Vide la file.
- **extend(iterable)** : Ajoute les éléments de l'itérable à la fin.
- **extendleft(iterable)** : Ajoute les éléments de l'itérable au début (l'ordre des éléments est inversé).
- **rotate(n)** : Fait une rotation de n pas. Les éléments qui sortent d'une extrémité sont ajoutés à l'autre extrémité.

## 8. Exemples Pratiques

### 8.1. Traitement d'une File de Tâches

```
from collections import deque

tâches = deque()

# Ajouter des tâches
tâches.append("tâche1")
tâches.append("tâche2")
tâches.append("tâche3")

# Traiter les tâches
while tâches:
    tâche = tâches.popleft()
    print(f"Traitement de {tâche}")
```

Sortie :

```
Traitement de tâche1
Traitement de tâche2
Traitement de tâche3
```

## 8.2. Implémentation d'une File Circulaire

```
from collections import deque

file_circulaire = deque(maxlen=5) # Limite la taille à 5 éléments

for i in range(7):
    file_circulaire.append(i)
    print(f"File : {file_circulaire}")
```

Sortie :

```
File : deque([0], maxlen=5)
File : deque([0, 1], maxlen=5)
File : deque([0, 1, 2], maxlen=5)
File : deque([0, 1, 2, 3], maxlen=5)
File : deque([0, 1, 2, 3, 4], maxlen=5)
File : deque([1, 2, 3, 4, 5], maxlen=5)
File : deque([2, 3, 4, 5, 6], maxlen=5)
```

## 9. Files de Priorité

Pour implémenter une file où les éléments avec la priorité la plus élevée sont servis en premier, utilise le module `heapq`.

### 9.1. Importer `heapq`

```
import heapq
```

## 9.2. Création d'une File de Priorité

```
file_priorité = []  
heapq.heappush(file_priorité, (priorité, élément))
```

## 9.3. Récupérer l'Élément avec la Plus Haute Priorité

```
élément = heapq.heappop(file_priorité)
```

Exemple :

```
import heapq  
  
file_priorité = []  
  
heapq.heappush(file_priorité, (2, 'tâche moyenne'))  
heapq.heappush(file_priorité, (1, 'tâche urgente'))  
heapq.heappush(file_priorité, (3, 'tâche moins urgente'))  
  
while file_priorité:  
    priorité, tâche = heapq.heappop(file_priorité)  
    print(f"Traitement de {tâche} avec priorité {priorité}")
```

Sortie :

```
Traitement de tâche urgente avec priorité 1  
Traitement de tâche moyenne avec priorité 2  
Traitement de tâche moins urgente avec priorité 3
```

## 10. Files Sécurisées pour les Threads avec queue

Le module queue est utile dans les contextes multi-threads.

### 10.1. Exemple avec Threads

```
from queue import Queue
from threading import Thread
import time

def producteur(file):
    for i in range(5):
        file.put(i)
        print(f"Produit {i}")
        time.sleep(1)

def consommateur(file):
    while True:
        item = file.get()
        if item is None:
            break
        print(f"Consommé {item}")
        file.task_done()

file = Queue()
t1 = Thread(target=producteur, args=(file,))
t2 = Thread(target=consommateur, args=(file,))

t1.start()
t2.start()

t1.join()
file.put(None) # Signal pour arrêter le consommateur
t2.join()
```

Sortie (l'ordre peut varier) :

```
Produit 0
Consommé 0
Produit 1
Consommé 1
Produit 2
Consommé 2
Produit 3
Consommé 3
Produit 4
Consommé 4
```

## 11. Files Asynchrones avec asyncio

En Python 3.5+, le module asyncio fournit une classe Queue pour les programmes asynchrones.

### 11.1. Exemple Simple

```
import asyncio

async def producteur(file):
    for i in range(5):
        await file.put(i)
        print(f"Produit {i}")
        await asyncio.sleep(1)

async def consommateur(file):
    while True:
        item = await file.get()
        print(f"Consommé {item}")
        file.task_done()
        if item == 4:
            break

async def main():
    file = asyncio.Queue()
    await asyncio.gather(producteur(file), consommateur(file))
    await file.join()

asyncio.run(main())
```

**Sortie :**

```
Produit 0
Consommé 0
Produit 1
Consommé 1
Produit 2
Consommé 2
Produit 3
Consommé 3
Produit 4
Consommé 4
```

## **12. Cas d'Utilisation des Files**

- **Gestion des Tâches** : Planification et exécution séquentielle des tâches en respectant l'ordre d'arrivée.
- **Communication entre Threads** : Partage de données entre threads de manière sécurisée sans avoir à gérer les verrous manuellement.
- **Algorithmes de Parcours** : Parcours en largeur (BFS) dans les structures de données comme les graphes ou les arbres.
- **Traitement des Données en Temps Réel** : Gestion des flux de données entrants, tels que les messages entrants dans un serveur.

## **13. Bonnes Pratiques**

- **Choisir la Bonne Structure** : Utilise **deque** pour les files simples, **queue.Queue** pour les threads, **asyncio.Queue** pour les programmes asynchrones.
- **Éviter les Listes pour les Grandes Files** : Les performances se dégradent avec les listes pour les opérations de file, surtout lors de la suppression d'éléments au début.

- **Gérer les Blocs** : Avec `queue.Queue`, sois conscient que les méthodes peuvent bloquer si la file est vide ou pleine. Utilise les paramètres `block=False` ou `timeout` si nécessaire.
- **Utiliser les Files de Priorité pour les Tâches Priorisées** : Si certaines tâches doivent être traitées avant d'autres, utilise une file de priorité.

#### 14. Limitations et Considérations

- **Sûreté pour les Threads** : Les structures comme `deque` ne sont pas sûres pour les threads. Utilise `queue.Queue` dans les contextes multithreads.
- **Non-Indexation** : Les files n'étant pas destinées à un accès aléatoire, il n'est pas recommandé d'accéder aux éléments par index.
- **Gestion des Exceptions** : Toujours gérer les exceptions lors du défilage pour éviter les erreurs lorsque la file est vide.
- **Blocs Indésirables** : Dans `queue.Queue`, les opérations peuvent bloquer. Sois prudent dans les applications où le blocage peut causer des problèmes.

FIN DE 1ere PARTIE