Introduction à Python

Par Ményssa Cherifa-Luron, PhD

Mail LinkedIn GitHub Livres

Table des matières

1. C’est quoi Python ?

Logo Python

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

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

Caractéristiques

Ce langage de programmation présente de nombreuses caractéristiques intéressantes :

- Il est multiplateforme. C’est-à-dire qu’il fonctionne sur de nombreux systèmes d’exploitation : Windows, Mac OS X, Linux, Android, iOS, depuis les mini-ordinateurs Raspberry Pi jusqu’aux supercalculateurs.
- Il est gratuit. Vous pouvez l’installer sur autant d’ordinateurs que vous voulez (même sur votre téléphone !).
- C’est un langage de haut niveau. Il demande relativement peu de connaissance sur le fonctionnement d’un ordinateur pour être utilisé.
- C’est un langage interprété. Un script Python n’a pas besoin d’être compilé pour être exécuté, contrairement à des langages comme le C ou le C++.
- Il est orienté objet. C’est-à-dire qu’il est possible de concevoir des entités qui miment celles du monde réel (une cellule, une protéine, un atome, etc.) avec un certain nombre de règles de fonctionnement et d’interactions.
- Il est relativement simple à prendre en main

Normes et environnement

Le guide de style Python (https://google.github.io/styleguide/pyguide.html) de Google constitue une ressource précieuse offrant une liste de recommandations sur la mise en forme du code Python, largement suivies dans l'industrie.

Programmer en Python : Jupyter et Colab

Avant d'aborder le langage Python, il est important de discuter brièvement des carnets et de la création d'environnements virtuels.

Si vous envisagez de travailler sur différents projets, il est probable que vous devrez utiliser différentes versions de modules Python. Dans ce cas, il est judicieux de créer plusieurs environnements virtuels pour chaque projet.

Configuration Python : Distante vs. Locale (https://aman.ai/primers/python-setup/) offre une couverture détaillée des différentes options disponibles à distance et localement.

Indentation

En Python, l'indentation est primordiale ! Alors que dans d'autres langages de programmation, l'indentation dans le code vise à améliorer la lisibilité, Python utilise l'indentation pour indiquer un bloc de code.

La convention consiste à utiliser quatre espaces, et non des tabulations.

2. Les Fondamentaux de Python

2.1. Variables (types natifs)

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

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

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

Affectation simple

In [ ]:
```
Ma_variable =  10
print(Ma_variable)
```

Affectation de plusieurs valeurs

In [ ]:
```
x, y, z = "Chocolat", "Vanille", "Fraise"

print(x)
print(y)
print(z)
```

Affectation de la même valeur

In [ ]:
```
x = y = z = "Chocolat"

print(x)
print(y)
print(z)
```

2.1.1. Types de variables

Types natifs : Les types de données courants en Python incluent :

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

In [ ]:
```
type("Hello")
```

In [ ]:
```
# Définition de plusieurs variables
nom = "Alice"
age = 25
taille = 1.70
estVrai = True
estFaux = False
```

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

2.1.2. Conversion de types

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

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

Voici des exemples de conversion entre ces types :

- Conversion en Entier (int)

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

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

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

- Conversion en Chaîne de Caractères (str)

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

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

- Conversion en Booléen (bool)

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

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

- Conversion en Float (float)

In [ ]:
```
## Depuis une chaîne de caractères** :
chaine = "123.45"
```

```
flottant = float(chaine)  # Convertit la chaîne "123.45" en float 123.45

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

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

2.1.3. Opérations arithmétiques

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

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

Modulo (%) retourne le reste de la division Puissance (**) élève un nombre à une certaine puissance.

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

2.1.4. Opérations sur les chaines de caractères

- La concaténation (+) combine des chaînes
- La répétition (*) répète une chaîne un certain nombre de fois.

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

2.1.5. Opérateurs d'affectation composés

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

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

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

In [ ]:
```
# Ajouter et assigner
x += 2  # x est maintenant 7
print(x)
```

In [ ]:
```
# Soustraire et assigner
x -= 3  # x est maintenant 4
print(x)
```

In [ ]:
```
# Multiplier et assigner
x *= 4  # x est maintenant 16
print(x)
```

In [ ]:
```
# Diviser et assigner
x /= 2  # x est maintenant 8
print(x)
```

In [ ]:
```
# Division entière et assigner
x //= 3  # x est maintenant 2
print(x)
```

In [ ]:
```
# Modulo et assigner
x %= 3  # x est maintenant 2
print(x)
```

In [ ]:
```
# Exponentielle et assigner
x **= 2  # x est maintenant 4
print(x)
```

2.2 Commentaires et affichage

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

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

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

In [ ]:
```
# Ceci est un commentaire simple

x = 5  # Ceci est un commentaire suivant une instruction

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

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

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

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

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

# Vous pouvez aussi combiner du texte et des variables :
print("Le nombre est", nombre)  # Affiche "Le nombre est 10"
```

Utilisation de f-string

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

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

f-string avec Plusieurs Variables

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

Utilisation de format

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

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

Calcul et arrondi

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

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

In [ ]:
```
prop = 2543276 / 678
print(prop)
```

In [ ]:
```
print(f"le resultat est {prop:.3f}")
```

2.3. Structures de contrôle

2.3.1 Les opérateurs

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

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

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

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

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

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

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

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

2.3.1 Les opérateurs

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

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

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

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

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

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

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

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

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

In [ ]:
```
# opérateurs logiques
## and
(5 > 3) and (10 < 100)
```

In [ ]:
```
not ((5 > 3) and (10 > 100))
```

In [ ]:
```
## or
(5 > 3) or (10 < 100)
(5 > 3) or (10 > 100)
```

In [ ]:
```
# operateurs d'appartenance
# in et not in
x = [5, 10, 24]
10 in x
10 not in x
```

Chainer les comparateurs

Il est également possible de chainer les comparateurs:

In [ ]:
```
a, b, c = 1, 10, 100
a < b < c
```

In [ ]:
```
a > b < c
```

2.3.2 Les conditions

In [ ]:
```
# condition
x = 40

# Première condition : vérifie si x est inférieur à 10
```

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

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

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

2.3.3 Les boucles

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

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

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

In [ ]:
```
liste = list(range(5))
liste
```

In [ ]:
```
# Exemple de boucle for
for i in liste:
    print(i)
```

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

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

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

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

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

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

In [ ]:
```
# Boucle while avec continue
print("\nBoucle while avec continue:")
```

```
i = 0
while i < 5:
    i += 1
    if i == 3:
        continue
    print(i)
```

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

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

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

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

In [ ]:
```
saisi = int(input("Saisissez un chiffre: "))
while True:
    if saisi == 5:
        print("Vous avez saisi la valeur 5")
        break
    else:
        saisi = int(input("Saisissez un chiffre: "))
```

In [ ]:
```
saisi_mot_de_passe = input("Saisissez votre mdp: ")

while True:
    if saisi_mot_de_passe == "azerty":
        print("Login success")
        break
    else:
        print("Login fail")
        saisi_mot_de_passe = input("Re-Saisissez votre mdp: ")
```

In [ ]:
```
import random

nombre_aleatoire = random.randint(1, 10)
nombre_aleatoire
```

2.4 Types de Données (Structures Séquentielles)

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

2.4.1 Les chaines de caractères

Les chaînes de caractères (`str`) sont utilisées pour stocker et manipuler du texte. Elles sont définies en plaçant le texte entre guillemets simples (`'...'`), doubles (`"..."`) ou triples (`'''...'''` ou `"""..."""` pour les chaînes sur plusieurs lignes).

- Séquences de caractères, indexées à partir de 0.
- Peuvent être délimitées par des guillemets simples, doubles ou triples.
- Exemple : `"Python"` , `'Data'` .
- Caractères d'Échappement : pour inclure des caractères spéciaux comme des guillemets ou des retours à la ligne dans une chaîne, utilisez des caractères d'échappement (`\`).

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

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

In [ ]:
```
# Définition chaines de caractères
chaine_simple = 'Bonjour'
```

```
longueur = len(chaine_simple) # 7
print(f"{chaine_simple}, 'la longueur est {longueur}")
```

In [ ]:
```
chaine_simple[3]
```

In [ ]:
```
chaine_double = "Python"
chaine_multiligne = """Ceci est une
chaîne sur plusieurs
lignes."""
print(chaine_double,"\n", chaine_multiligne)

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

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

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

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

In [ ]:
```
# Méthodes str.isupper() et str.islower()
majuscules = "PYTHON"
minuscules = "python"

is_upper = majuscules.isupper()
is_lower = minuscules.islower()
print(is_upper)  # Résultat : True
print(is_lower)  # Résultat : True

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

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

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

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

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

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

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

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

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

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

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

2.4.2 Les listes

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

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

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

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

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

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

- **Le Slicing** : En plus d'accéder aux éléments d'une chaine un par un, Python fournit une syntaxe concise pour accéder aux sous-listes; cela est connu sous le nom de **slicing** :

In [ ]:
```
nums = list(range(5)) # range est une fonction intégrée qui crée une liste d'entiers
nums               # Renvoie "[0, 1, 2, 3, 4]"
nums[2:4]           # Obtenir une tranche de l'index 2 à 4 (exclusif); renvoie "[2, 3]"
nums[2:]           # Obtenir une tranche de l'index 2 jusqu'à la fin; renvoie "[2, 3, 4]"
nums[:2]           # Obtenir une tranche du début jusqu'à l'index 2 (exclusif); renvoie "[0, 1]"
nums[:]            # Obtenir une tranche de toute la liste; renvoie "[0, 1, 2, 3, 4]"
nums[:-1]          # Les indices de tranche peuvent être négatifs; renvoie "[0, 1, 2, 3]"
nums[::-1]         # Les indices de tranche peuvent être négatifs pour inverser l'ordre ; renvoie "[4, 3, 2, 1, 0]"
```

Attribuer à une tranche (même avec une source de longueur différente) est possible car les listes sont mutables :

In [ ]:
```
# Cas 1: source de même longueur
nums1 = [1, 2, 3]
nums1[1:] = [4, 5]    # Assigner une nouvelle sous-liste à une tranche
nums1              # Renvoie "[1, 4, 5]"

# Cas 2: source de longueur différente
nums2 = nums1
nums2[1:] = [6]       # Assigner une nouvelle sous-liste à une tranche
nums2              # Renvoie "[1, 6]"
id(nums1) == id(nums2) # Renvoie True car les listes sont mutables, c'est-à-dire qu'elles peuvent être modifiées sur place
```

Similaire aux tuples, lors de l'évaluation d'une plage sur les indices de liste (quelque chose de la forme [x:y] où x et y sont des indices dans la liste), si notre valeur de droite dépasse la longueur de la liste, Python renvoie simplement les éléments de la liste jusqu'à ce que la valeur dépasse la plage d'index.

In [ ]:
```
a = [1, 2, 3] # Index maximal adressable: 2
a[:3]        # NE renvoie PAS d'erreur, renvoie plutôt [1, 2, 3]
```

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

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

In [ ]:
```
# Création d'une liste de base
ma_liste = [1, 2, 3]
print(ma_liste)

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

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

In [ ]:
```
# Utilisation de insert()
ma_liste.insert(1, 'elephant')
print("Après insert(1, 'elephant'):", ma_liste)
```

In [ ]:
```
# Utilisation de remove()
ma_liste.remove('elephant')
print("Après remove('elephant'):", ma_liste)
```

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

In [ ]:
```
# Utilisation de clear()
ma_liste.clear()
print("Après clear():", ma_liste)
```

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

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

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

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

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

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

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

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

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

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

for f in fruits:
  if "a" in f:
    newlist.append(f)

print(newlist)
```

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

In [ ]:
```
nombre = [1, 10, 100]
nombre_carres = [x**2 for x in nombre]
nombre_carres
```

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

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

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

D'autre part, les compréhensions de liste peuvent être écrites de manière équivalente en utilisant une combinaison du constructeur de liste, et/ou map et/ou filter :

In [ ]:
```
list(map(lambda x: x + 10, [1, 2, 3]))         # Retourne [11, 12, 13]
list(map(max, [1, 2, 3], [4, 2, 1]))          # Retourne [4, 2, 3]
list(filter(lambda x: x > 5, [3, 4, 5, 6, 7]))  # Retourne [6, 7]
```

Avec des boucles imbriquées

Voici une boucle for qui aplatit une matrice (une liste de listes) :

In [ ]:
```
flattened = []
for row in matrix:
  for n in row:
    flattened.append(n)
```

Et voici une compréhension de liste qui fait la même chose :

In [ ]:
```
flattened = [n for row in matrix for n in row]
```

Les boucles imbriquées dans les compréhensions de liste ne se lisent pas comme du texte. Une erreur commune est de lire cette compréhension de liste comme suit :

In [ ]:
```
flattened = [n for n in row for row in matrix]
```

Mais ce n'est pas correct ! Nous avons inversé les boucles for ici par erreur. La version correcte est celle ci-dessus. Lorsque vous travaillez avec des boucles imbriquées dans les compréhensions de liste, rappelez-vous que les clauses for restent dans le même ordre que dans nos boucles for originales.

2.4.3 Les tuples

Les tuples sont :

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

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

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

In [ ]:
```
un_autre_tuple = 2, "b", 4.28
un_autre_tuple
```

In [ ]:
```
mon_tuple[1]
```

In [ ]:
```
# Tuple à Un Élément
tuple_singleton = 5,
tuple_singleton
```

In [ ]:
```
# Tuple à Un Élément
tuple_singleton = (5,)

# Imbrication
tuple_imbrique = (1, (2, 3), (4, 5))

# Méthodes Utiles
```

In [ ]:
```
longueur = len(mon_tuple)  # Retourne la longueur de mon_tuple, ici 3
longueur
```

In [ ]:
```
compteur = mon_tuple.count("a")  # Compte le nombre de fois que 1 apparaît dans mon_tuple, ici 1
compteur
```

In [ ]:
```
indice = mon_tuple.index("a")  # Trouve l'indice de "a" dans mon_tuple, ici 1
indice
```

In [ ]:
```
# L'affectation n'est pas possible
# mon_tuple[1] = "b"
```

2.4.4 Les ensembles

Les ensembles sont :

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

Opérations d'Ensemble

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

In [ ]:
```
# Définition des ensembles
```

```
ensemble_a = {1, 2, 3, 4}
ensemble_b = {3, 4, 5, 6}
print(ensemble_a, ensemble_b)

# Seule une valeur est représentée
ensemble_c = {1, 1, 1}
print(ensemble_c)
```

In [ ]:
```
# Opérations d'ensemble
union = ensemble_a | ensemble_b
intersection = ensemble_a & ensemble_b
difference_a = ensemble_a - ensemble_b
difference_b = ensemble_b - ensemble_a
difference_symetrique = ensemble_a ^ ensemble_b

# Affichage des résultats
print("Union :", union)
print("Intersection :", intersection)
print("Différence a :", difference_a)
print("Différence b :", difference_b)
print("Différence Symétrique :", difference_symetrique)
```

In [ ]:
```
print(ensemble_a)

# Ajout et suppression
ensemble_a.add(7)
print("Après ajout :", ensemble_a)
```

In [ ]:
```
ensemble_a.remove(1)
print("Après suppression :", ensemble_a)
```

In [ ]:
```
ensemble_a.discard(2)
ensemble_a
```

In [ ]:
```
for i in range(1,5):
    for j in range(1,5):
        "*".join(j)
        print("*" * i * j)
```

2.4.5 Les dictionnaires

Les dictionnaires sont :

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

In [ ]:
```
# Définition d'un dictionnaire
personne = {"nom": "Alice", "age": 30}
autre_dict = dict(nom="Bob", age=25)
```

In [ ]:
```
personne
```

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

In [ ]:
```
# Mise à Jour de plusieurs clés
personne.update([('nom', "Romain"), ('profession', "Data Scientist")])
personne
```

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

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

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

In [ ]:
```
# Utilisation de get()
nom_personne = personne.get("nom")
age_personne = personne.get("age")

nom_personne, age_personne
```

In [ ]:
```
ville = personne.get("ville")
ville
```

In [ ]:
```
ville = personne.get("ville", "Non spécifiée")  # Retourne 'Non spécifiée' car 'ville' n'existe pas
print("Ville:", ville)
```

In [ ]:
```
# Parcours des clés
print("\nParcours des clés:")
for cle in personne.keys():
    print(cle)

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

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

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

In [ ]:
```
personne
```

In [ ]:
```
# Suppression
del personne["age"]
```

In [ ]:
```
personne
```

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

In [ ]:
```
profession
```

In [ ]:
```
personne
```

Dictionary Comprehensions

Ces dernières sont similaires aux compréhensions de listes, mais vous permettent de construire facilement des dictionnaires.

Par exemple, considérez une boucle for qui crée un nouveau dictionnaire en échangeant les clés et les valeurs de l'original :

In [ ]:
```
flipped = {}
original = {"a": 0, "b": 5, "c": 6, "d": 7, "e": 11, "f": 19}
for key, value in original.items():
    flipped[value] = key

flipped
```

Le même code écrit sous forme d'une **Dictionary Comprehensions** :

In [ ]:
```
flipped = {value: key for key, value in original.items()}
```

Autre exemple :

In [ ]:
```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)
```

Autre exemple :

In [ ]:
```
{x: x**2 for x in range(5)}
```

2.5 Les fonctions : définition et rappel

Qu'est-ce qu'une Fonction ?

Une fonction est une suite d'instructions que l'on peut appeler avec un nom. En programmation, une fonction est comme une petite "machine" qui prend des entrées (arguments), effectue certaines opérations, et parfois retourne une sortie (valeur de retour).

- **Explication de ce qu'est une fonction en programmation**

, une fonction est définie en utilisant le mot-clé `def` , suivi du nom de la fonction, des parenthèses `()` contenant des paramètres (si nécessaires), et un bloc d'instructions indenté.

```
def ma_fonction():
    # Ici, nous mettons le code que la fonction exécutera.
```

- **L'importance des fonctions pour réduire la répétition de code**

Les fonctions sont essentielles pour éviter la répétition de code. Au lieu d'écrire le même code plusieurs fois, une fonction nous permet de regrouper ce code et de l'appeler quand nécessaire. Cela rend le code plus court, plus lisible et plus facile à maintenir.

Pourquoi Utiliser des Fonctions ?

Les fonctions sont un outil fondamental pour plusieurs raisons :

- **Simplification du code**

En regroupant des opérations complexes en fonctions, on simplifie notre code principal. Cela le rend plus lisible et plus facile à comprendre.

- **Réutilisabilité**

Une fois que vous avez écrit une fonction, vous pouvez l'utiliser autant de fois que nécessaire. Cela élimine le besoin de réécrire le même code, économisant du temps et réduisant les erreurs.

- **Meilleure organisation du code**

Les fonctions aident à structurer le code en unités logiques. Chaque fonction a un objectif spécifique, ce qui facilite la compréhension du programme dans son ensemble. De plus, cela facilite le débogage et le test de parties spécifiques du code.

Voici le développement de la section "Syntaxe de Base des Fonctions" avec des explications et des exemples, rédigé en Markdown :

Définition d'une Fonction

, la définition d'une fonction commence par le mot-clé `def`, suivi du nom de la fonction, des parenthèses `()`, et d'un bloc d'instructions indenté.

```
def ma_fonction():
    print("Ceci est une fonction.")
```

Dans cet exemple, `ma_fonction` est le nom de notre fonction. Le bloc d'instructions qui suit définit ce que la fonction fait lorsqu'elle est appelée. Ici, la fonction affiche simplement un message.

La définition d'une fonction permet de regrouper un ensemble d'instructions que vous souhaitez exécuter à plusieurs reprises. Elle n'est exécutée que lorsque la fonction est appelée.

- **Structure de base**

La structure de base d'une fonction comprend :

- Le mot-clé `def`.
- Le nom de la fonction.
- Les parenthèses `()` (avec des paramètres si nécessaire).
- Un bloc d'instructions indenté.

- **Appel d'une Fonction**

Une fois qu'une fonction est définie, elle peut être appelée en utilisant son nom suivi de parenthèses.

```
ma_fonction()
```

Cet appel exécute les instructions définies dans `ma_fonction`.

- **Paramètres et Arguments**

Les fonctions peuvent être plus flexibles lorsqu'elles prennent des paramètres. Les paramètres agissent comme des variables dans la définition de la fonction.

```
def ma_fonction(param1, param2):
    print(f"Paramètre 1 : {param1}, Paramètre 2 : {param2}")
def saluer(nom):
    print(f"Bonjour, {nom}!")
```

Ici, `nom` est un paramètre de la fonction `saluer`. Lorsque la fonction est appelée, elle s'attend à recevoir une valeur pour ce paramètre.

- **Appel d'une Fonction avec Arguments**

Lorsque vous appelez une fonction qui a des paramètres, vous devez fournir des arguments correspondants.

```
saluer("Alice")
```

Dans cet appel, `"Alice"` est un argument qui est passé à la fonction `saluer`. La fonction utilisera la valeur de cet argument lors de son exécution.

Exemples :

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

In [ ]:
```
saluer(nom = "Paul")
saluer(nom = "Ményssa")
saluer(nom = "Kévin")
saluer(nom = "Axel")
```

In [ ]:
```
def affiche_message(message):
    print(f"Message reçu : {message}")

affiche_message("Bonjour, ca  va ?")
affiche_message("Python c'est cool!")
```

In [ ]:
```
def additionner(a, b):
    somme = a + b
    print(f"La somme de {a} et {b} est {somme}")
```

In [ ]:
```
additionner(a = 5, b = 3)
additionner(5, 3)
```

In [ ]:
```
additionner(b = 10, a = 67)
```

In [ ]:
```
def division(a, b):
    quotient = a / b
    print(f"Le quotient de {a} et {b} est {quotient}")

division(a = 5, b = 3)
division(b = 3, a = 5)

division(3, 5)
division(5, 3)
```

In [ ]:
```
def puissance(base, exposant):
    resultat = base ** exposant
    print(f"{base} élevé à la puissance {exposant} est égal {resultat}")

puissance(exposant = 2, base =  3)
```

In [ ]:
```
import math

def perimetre_cercle(rayon):
    perimetre = 2 * math.pi * rayon
    print(f"Le périmètre d'un cercle de rayon {rayon} est {perimetre:.2f}")

perimetre_cercle(5)
```

2.5.1 Les paramètres et arguments

Les paramètres de fonction sont des variables spécifiées dans la définition d'une fonction. Ils agissent comme des placeholders pour les valeurs (arguments) que la fonction recevra lorsqu'elle est appelée.

- **Différence entre Paramètres et Arguments**
  - **Paramètres** sont les noms donnés aux variables dans la définition de la fonction. Ils sont utilisés dans la fonction pour se référer aux valeurs qui seront passées lors de l'appel de la fonction.
  - **Arguments** sont les valeurs réelles passées à la fonction lorsqu'elle est appelée. Ces valeurs sont affectées aux paramètres correspondants de la fonction.
- **Valeurs par Défaut des Paramètres** Les fonctions peuvent avoir des paramètres avec des valeurs par défaut. Ces valeurs sont utilisées si aucun argument correspondant n'est fourni lors de l'appel de la fonction.

```
def ma_fonction(param1, param2=42):
    print(f"Paramètre 1 : {param1}, Paramètre 2 : {param2}")
```

Dans cet exemple, `param2` a une valeur par défaut de `42`. Si `ma_fonction` est appelée sans un second argument, `param2` utilisera cette valeur par défaut.

Les valeurs par défaut rendent les fonctions plus flexibles et permettent d'éviter des erreurs si certains arguments ne sont pas essentiels. Voici un exemple d'appel de la fonction avec et sans le second argument :

```
ma_fonction("test")        # Utilise la valeur par défaut pour param2
ma_fonction("test", 24)    # Remplace la valeur par défaut de param2
```

In [ ]:
```
def ma_fonction(param1, param2=42):
    print(f"Paramètre 1 : {param1}, Paramètre 2 : {param2}")
```

In [ ]:
```
ma_fonction(2)
```

In [ ]:
```
ma_fonction(2, 30)
```

2.5.2 Les types de fonctions

Il existe deux types principaux de fonctions : celles qui retournent une valeur et celles qui ne retournent rien.

- **Fonctions avec Retour** : Une fonction avec retour renvoie une valeur à l'endroit où elle est appelée.
Ce retour est effectué en utilisant le mot-clé `return`.
- **Fonctions sans Retour**: Les fonctions sans retour effectuent des actions mais ne renvoient aucune valeur.
Elles sont souvent utilisées pour effectuer une tâche, comme afficher un message ou modifier des données.

```
def ma_fonction():
    return "valeur retournée"
```

Cette fonction retourne la chaîne de caractères "valeur retournée" . Lorsqu'elle est appelée, on peut récupérer cette valeur :

```
resultat = ma_fonction()
print(resultat)  # Affiche "valeur retournée"
```

```
def ma_fonction():
    print("Fonction exécutée")
```

Cette fonction affiche un message mais ne retourne rien. L'appel de cette fonction se concentre sur l'effet produit (ici, l'affichage) plutôt que sur une valeur retournée.

In [ ]:
```
def divison_par_deux(a, b = 2):
    print( a / b )

resultat = divison_par_deux(5)
print(resultat)
```

In [ ]:
```
def divison_par_deux(a, b = 2):
    return a / b

resultat = divison_par_deux(5)
print(resultat)
```

In [ ]:
```
def carre(nombre):
    return nombre * nombre

resultat = carre(4)
print(resultat)  # Affichera 16

def celsius_en_fahrenheit(celsius):
    return (celsius * 9/5) + 32

temperature_fahrenheit = celsius_en_fahrenheit(0)
print(temperature_fahrenheit)  # Affichera 32

def est_pair(nombre):
    return nombre % 2 == 0

resultat = est_pair(10)
print(resultat)  # Affichera True

def maximum(a, b):
    if a > b:
        return a
    else:
        return b
```

In [ ]:
```
max_nombre = maximum(10, 20)
print(max_nombre)  # Affichera 20
```

2.5.3 La portée des variables

La portée d'une variable détermine où dans le code elle est accessible. En Python, les variables peuvent avoir une portée locale ou globale.

In [ ]:
```
x = "global"  # Variable globale
x
```

In [ ]:
```
def ma_fonction():
    x = "local"  # Variable locale
    print(x)

ma_fonction()
x
```

Les variables globales sont utiles pour partager des données entre différentes parties d'un programme.

Cependant, il faut les utiliser avec prudence pour éviter des problèmes de conception.

In [ ]:
```
# le mot-clé `global` est utilisé pour modifier la variable globale
#  `z` à l'intérieur de la fonction `modifier_globale`.

z = 10

def modifier_globale():
    global z
    z = 20

print(z)  # Affiche 10
modifier_globale()
print(z)  # Affiche 20
```

In [ ]:
```
# Les arguments mot-clé permettent de passer des arguments
#  à une fonction en spécifiant le nom du paramètre.
def ma_fonction(nom, age):
    print(f"Nom : {nom}, Âge : {age}")

ma_fonction(age=25, nom="Alice")
```

2.5.4 Nombre variable d'arguments

Vous pouvez créer des fonctions qui acceptent un nombre variable d'arguments grâce à *args pour les listes et **kwargs pour les dictionnaires de mots-clés.

In [ ]:
```
def multiplication(a, b, c, d, e, f):
    return a * b

multiplication(2, 3)
```

In [ ]:
```
import numpy as np
# arguments arbitraire on ne connait pas le nombre de parametre à l'avance
def multiplication(*nombre):
    res = np.prod(nombre)
    return res
```

In [ ]:
```
multiplication(4, 3, 4, 90)
```

In [ ]:
```
def multiplication(*nombre):
    print(nombre[0]*nombre[1]*nombre[2]*nombre[3])
```

In [ ]:
```
multiplication(4, 3, 4, 90)
```

In [ ]:
```
def param_args(*parametres):
    print(parametres[0]*parametres[1])
    print(f"Message : {parametres[2]}")

param_args(2, 3, "Salut!")
```

In [ ]:
```
args_tuple = (5, 6, 8, 9)
nombre_args(*args_tuple) # mettre etoile pour le tuple
```

In [ ]:
```
# pratique pour définir des clé d'arguments
def nombre_kwarg(**nombre):
    print('Mon age est : ' + nombre['age'] + ' ans' + ' et ma taille est : ' + nombre["taille"] + ' metres')

nombre_kwarg(age  = '15', taille = '1.65', profession = " Dev")
```

In [ ]:
```
# `args` est un tuple contenant tous les arguments positionnels
# `kwargs` est un dictionnaire contenant tous les arguments nommés.

def ma_fonction(*args, **kwargs):
    print("args:", args)
    print("kwargs:", kwargs)

ma_fonction(1, 2, 3, nom="Alice", age=25)
```

- **Bonnes Pratiques** Nommer clairement vos fonctions est crucial pour la lisibilité et la maintenabilité de votre code. Les noms de fonctions doivent être descriptifs et refléter ce qu'ils font.
  - Utilisez des noms descriptifs.
  - Privilégiez les verbes pour les fonctions effectuant des actions.
  - Évitez les abréviations non évidentes. Documenter vos fonctions avec des **docstrings**, aide d'autres programmeurs (et vous-même dans le futur) à comprendre rapidement ce que fait votre fonction.

```
def ma_fonction():
    """
    Description de ce que fait la fonction.
    Peut inclure des détails sur les paramètres et la valeur de retour.
    """
    # corps de la fonction
```

Les docstrings sont placées juste sous la définition de la fonction et sont entourées de triples guillemets.

In [ ]:
```
# fonction qui renvoie un nombre à la puissance
def nombre_puissance(nombre, puissance):
    # documenter le code
    """
    cette fonction calcule un nombre à la puissance et retourne le résultat.
    Parameters:
    nombre    (int): le premier nombre
    puissance (int): le deuxième nombre
    Returns   (int): le resultat nombre a la puissance

    """
    return nombre ** puissance

# interaction  avec un utilisateur
num = int(input("Saississez un nombre entier : "))
power = int(input("Saississez une puissance : "))

print("Le résultat est :", nombre_puissance(nombre = num, puissance = power))
```

2.5.5 Les fonctions anonymes : lambda

En Python, le mot-clé **lambda** est utilisé pour définir des fonctions anonymes, également appelées fonctions lambda.

Les fonctions lambda sont des fonctions qui ne sont pas définies avec un nom, mais qui peuvent être utilisées dans des expressions ou des fonctions.

Elles sont souvent utilisées dans des situations où une fonction simple est nécessaire pour une tâche spécifique, comme trier une liste ou filtrer des éléments.

In [ ]:
```
c_to_f = lambda x: (x * 9/5) + 32
c_to_f(0)
```

In [ ]:
```
people = [{'name': 'Alice', 'age': 25},
          {'name': 'Bob', 'age': 30},
          {'name': 'Charlie', 'age': 20}]
sorted(people, key=lambda x: x['age'])
```

In [ ]:
```
ma_liste = ['abcd', 'abc', 'a', 'ab', 'abcde']
sorted(ma_liste, key=lambda x: len(x))
```

In [ ]:
```
carre  = lambda x: x**2
liste_carre = [carre(nombre) for nombre in [5, 9, 10]]
liste_carre

liste_carre = [(lambda x: x**2)(nombre) for nombre in [5, 9, 10]]
liste_carre
```

In [ ]:
```
personnes = [
  {"nom": "Alice", "age": 25},
  {"nom": "Bob", "age": 30},
  {"nom": "Charlie", "age": 20},
]

personnes.sort(key=lambda personne: personne["age"])

print(personnes)
```

In [ ]:
```
produits = [
  {"nom": "Produit 1", "en_stock": True},
  {"nom": "Produit 2", "en_stock": False},
  {"nom": "Produit 3", "en_stock": True},
]

produits_en_stock = list(filter(lambda produit: produit["en_stock"], produits))

print(produits_en_stock)
```

In [ ]:
```
**Lambda avec Map et Filter**

Les fonctions map et filter de Python sont presque toujours associées à des expressions lambda.
Il est courant de voir des questions sur StackOverflow demandant "qu'est-ce que lambda" répondre avec des exemples de code comme celui-ci :
```

In [ ]:
```
numbers = [2, 1, 3, 4, 7, 11, 18]

# Appliquer la fonction map pour obtenir les carrés de chaque élément de numbers
squared_numbers = map(lambda n: n**2, numbers)

# Appliquer la fonction filter pour obtenir les nombres impairs de numbers
odd_numbers = filter(lambda n: n % 2 == 1, numbers)

# Afficher les résultats
print("Squared numbers:", squared_numbers)
print("Odd numbers:", odd_numbers)
```

In [ ]:
```
# Afficher les résultats
print("Numbers:", numbers)
print("Squared numbers:", list(squared_numbers))
print("Odd numbers:", list(odd_numbers))
```

2.5.6 Les fonctions et la récursivité

La **récursivité** est un concept fondamental en programmation, et Python est un langage qui le supporte très bien.

La récursivité consiste en une fonction qui s'appelle elle-même dans sa propre définition. Cela peut sembler étrange au début, mais c'est une technique très puissante pour résoudre certains types de problèmes, en particulier ceux qui peuvent être décomposés en problèmes plus petits et similaires.

Voici un exemple simple en Python pour illustrer la récursivité :

In [ ]:
```
def nom():
    prenom = input("Nom :")
    if prenom == "Ményssa":
        return f'ok'
    else:
        nom()
nom()
```

In [ ]:
```
def countdown(n):
    if n <= 0:
        print("Lancement terminé!")
    else:
        print(n)
        countdown(n - 1)

countdown(5)
```

Dans cet exemple, la fonction `countdown` prend un nombre entier `n` en paramètre. Si `n` est inférieur ou égal à zéro, la fonction affiche "Lancement terminé!". Sinon, elle affiche la valeur actuelle de `n` puis appelle elle-même avec `n - 1`.

La récursivité doit être utilisée avec prudence, car une mauvaise utilisation peut entraîner des erreurs de débordement de pile (stack overflow).

Cela se produit lorsque trop de fonctions récursives sont empilées sur la pile d'appels du programme.

Il est important de s'assurer qu'il y ait un cas de base dans la récursivité, c'est-à-dire une condition qui arrête les appels récursifs.

Sinon, la fonction continuera à s'appeler elle-même indéfiniment.

Voici un exemple plus avancé qui utilise la récursivité pour calculer la somme des éléments d'une liste :

In [ ]:
```
def recursive_sum(arr):
    if len(arr) == 0:
        return 0
    else:
        return arr[0] + recursive_sum(arr[1:])

my_list = [1, 2, 3, 4, 5]
print("La somme des éléments de la liste est:", recursive_sum(my_list))
```

Dans cet exemple, la fonction `recursive_sum` prend une liste en paramètre. Si la liste est vide, elle retourne 0.

Sinon, elle retourne le premier élément de la liste ajouté à la somme des éléments restants, obtenue en appelant récursivement la fonction avec une sous-liste de `arr` qui exclut le premier élément.

2.5.7 Générateurs et décorateurs

Les **générateurs** sont des structures puissantes en Python permettant de créer des itérables de manière efficace et sans consommer beaucoup de mémoire.

Contrairement aux listes, les générateurs produisent des valeurs au fur et à mesure de leur utilisation, ce qui les rend idéaux pour traiter de grandes quantités de données de manière efficace.

- Un générateur est une fonction qui utilise le mot-clé `yield` pour renvoyer des valeurs.
- Lorsqu'une fonction contient `yield`, elle devient un générateur.
- Chaque fois que le générateur est appelé, il exécute son code jusqu'à ce qu'il rencontre l'instruction `yield`.
- À ce moment-là, il renvoie la valeur spécifiée et se met en pause, en conservant son état.
- Lorsqu'il est rappelé, il reprend à partir de l'instruction `yield` précédente.

In [ ]:
```
def mon_generateur():
    yield 1
    yield 2
```

```
    yield 3

# Utilisation du générateur
gen = mon_generateur()
print(next(gen)) # Affiche : 1
print(next(gen)) # Affiche : 2
print(next(gen)) # Affiche : 3
print(next(gen)) # `StopIteration`
```

Dans cet exemple, chaque yield renvoie une valeur successive à chaque appel de next() .

Lorsque toutes les valeurs ont été renvoyées, le générateur lève une exception StopIteration .

In [ ]:
```
def pairs_infinis():
    n = 0
    while True:
        yield n
        n += 2

# Utilisation du générateur
gen = pairs_infinis()

for _ in range(10):
    print(next(gen)) # Affiche : 0, 2, 4, 6, 8
```

In [ ]:
```
def fibonacci(n):
    sequence = [0, 1]
    while len(sequence) < n:
        sequence.append(sequence[-1] + sequence[-2])
    return sequence

# Utilisation de la fonction
print(fibonacci(10))  # Affiche les 10 premiers nombres de la séquence Fibonacci
```

In [ ]:
```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Utilisation du générateur
gen = fibonacci()
for _ in range(10):
    print(next(gen)) # Affiche les 10 premiers nombres de la séquence Fibonacci
```

In [ ]:
```
import random
import string

def chaine_aleatoire(longueur):
    while True:
        yield ''.join(random.choices(string.ascii_letters, k=longueur))

# Utilisation du générateur
gen = chaine_aleatoire(5)
for _ in range(3):
    print(next(gen)) # Affiche 3 chaînes de 5 caractères aléatoires
```

En Python, les **décorateurs** sont des fonctions qui prennent une autre fonction comme argument, ajoutent un comportement supplémentaire à cette fonction, puis la renvoient sans la modifier.

Les décorateurs utilisent généralement la fonction @decorateur juste avant la définition de la fonction à décorer.

In [ ]:
```
def decorateur(fonction):
    def fonction_decoration():
        print("Début de la fonction décorée")
        fonction()
        print("Fin de la fonction décorée")
    return fonction_decoration

@decorateur
def ma_fonction():
    print("Corps de ma fonction")

# Utilisation de la fonction décorée
ma_fonction()
```

Dans cet exemple, le décorateur decorateur ajoute un comportement d'impression avant et après l'exécution de la fonction ma_fonction.

Les décorateurs peuvent également prendre des arguments. Pour cela, il faut ajouter une couche de fonctionnalité supplémentaire pour accepter ces arguments.

In [ ]:
```
def decorateur_args(argument):
    def decorateur(fonction):
        def fonction_decoration():
            print("Début de la fonction décorée avec l'argument :", argument)
```

```
            fonction()
            print("Fin de la fonction décorée avec l'argument :", argument)
        return fonction_decoration
    return decorateur

@decorateur_args("mon_argument")
def ma_fonction():
    print("Corps de ma fonction")

# Utilisation de la fonction décorée
ma_fonction()
```

Les décorateurs sont un aspect avancé de Python, mais ils offrent une puissante fonctionnalité pour étendre et modifier le comportement des fonctions et des méthodes de manière flexible et élégante. En comprenant les principes de base des décorateurs et leur utilisation avancée, vous pouvez améliorer la lisibilité, la réutilisabilité et la modularité de votre code Python.

1. Les décorateurs peuvent retourner une fonction au lieu de la fonction décorée elle-même, ce qui permet de contrôler davantage le comportement de la fonction.
2. Les décorateurs peuvent également être appliqués aux méthodes de classe pour étendre ou modifier le comportement des méthodes.
3. Il est possible d'empiler plusieurs décorateurs sur une même fonction pour ajouter plusieurs couches de fonctionnalités.

3. Gestion des exceptions ⚠

3.1 Définition

Une exception est une erreur qui se produit pendant l'exécution du programme, contrairement aux erreurs de syntaxe qui sont détectées avant l'exécution. Les exceptions peuvent être dues à diverses raisons, telles que des entrées invalides, des opérations mathématiques impossibles, des manipulations de fichiers incorrectes, etc.

Voici quelques **exemples d'exceptions** fréquemment rencontrées :

- IndexError : Se produit lorsqu'on tente d'accéder à un index qui n'existe pas dans une séquence (comme une liste ou un tuple).

In [ ]:
```
ma_liste = ["A", 3, "serpent"]
element = ma_liste[5]  # IndexError car l'index 5 n'existe pas dans une liste de taille 3
```

- ValueError : Apparaît lorsqu'une fonction reçoit un argument de type correct mais d'une valeur inappropriée.

In [ ]:
```
nombre = int("123373829173487472474")
nombre
```

In [ ]:
```
nombre = int("abc")  # ValueError car "abc" ne peut pas être converti en entier
```

- TypeError : Se produit lorsqu'une opération ou fonction est appliquée à un objet d'un type inapproprié.

In [ ]:
```
somme = 10 + 5
somme
```

In [ ]:
```
"message" + " : voici mon texte"
```

In [ ]:
```
somme = "texte" + 5  # TypeError car on ne peut pas ajouter une chaîne de caractères et un entier
```

In [ ]:
```
somme = 5  + "5"
```

- FileNotFoundError : Levée lorsqu'une tentative de lecture d'un fichier échoue parce que le fichier n'existe pas.

In [ ]:
```
# with open("../data/Orgueil_et_Prejuges.txt", "r", encoding = 'utf-8') as fichier:
#     contenu = fichier.read()

# print(contenu)
```

In [ ]:
```
with open("fichier_inexistant.txt", "r") as fichier:
    contenu = fichier.read()  # FileNotFoundError si "fichier_inexistant.txt" n'existe pas
```

- ZeroDivisionError : Se produit lorsqu'une division par zéro est tentée.

In [ ]:
```
resultat = 10 / 0  # ZeroDivisionError car on ne peut pas diviser par zéro
```

- NameError : Se produit lorsqu'une variable n'est pas définie dans le contexte actuel.

In [ ]:
```
def ma_fonction():
    variable_a_afficher = "Bonjour"
    print(variable_a_afficher)

ma_fonction()
```

```
ma_fonction()
ma_fonction()
```

In [ ]:
```
print(variable_a_afficher)  # NameError car 'variable_a_afficher' n'est pas définie en dehors de la fonction
```

3.2 Traitement des exceptions

La **gestion des exceptions** est un aspect crucial de la programmation pour plusieurs raisons :

- **Robustesse** : Elle permet à un programme de gérer des situations d'erreur de manière gracieuse sans s'arrêter brusquement.
- **Contrôle du Flux** : Elle offre un moyen de diriger le flux d'un programme lorsque des situations exceptionnelles se produisent.
- **Débogage Facilité** : Elle aide à identifier les causes des erreurs et à les gérer efficacement.
- **Sécurité** : Elle permet de gérer des situations potentiellement dangereuses, comme la lecture de fichiers corrompus ou la gestion de données utilisateur erronées.

3.2.1 Try-except

La syntaxe de base pour la gestion des exceptions en Python utilise les instructions `try` et `except`.

Voici un exemple simple :

```
try:
    # Bloc de code à essayer
    result = 10 / 0
except ZeroDivisionError:
    # Ce bloc est exécuté en cas d'erreur ZeroDivisionError
    print("Division par zéro !")
```

Dans cet exemple, si le code dans le bloc `try` cause une `ZeroDivisionError`, le programme ne s'arrêtera pas. Au lieu de cela, il exécutera le code dans le bloc `except`.

In [ ]:
```
result = 10 / 0
```

In [ ]:
```
try:
    # Bloc de code à essayer
    result = 10 / 0
except ZeroDivisionError:
    # Ce bloc est exécuté en cas d'erreur ZeroDivisionError
    print("Division par zéro !")
```

In [ ]:
```
def gestion_index_error(liste, index):
    try:
        return liste[index]
    except IndexError as e:
        return f"Ce n'est pas possible, Erreur d'index  : {e}"

# Test
ma_liste = ["A", 3, "serpent"]
print(gestion_index_error(ma_liste, 1))
print(gestion_index_error(ma_liste, 5))
```

In [ ]:
```
def conversion_en_entier(entree):
    try:
        return int(entree)
    except ValueError as e:
        return f"La conversion n'est pas possible, Erreur de valeur : {e}"

# Test
val_entree = "190"
print(conversion_en_entier(val_entree))
val_entree = "Robert"
print(conversion_en_entier(val_entree))
```

In [ ]:
```
def additionner(a, b):
    try:
        return a + b
    except TypeError as e:
        return f"Erreur de type : {e}"

# Test
print(additionner(2, 5))
print(additionner("texte", 5))
```

In [ ]:
```
def lire_fichier(nom_fichier):
    try:
        with open(nom_fichier, "r", encoding = 'utf-8') as fichier:
            return fichier.read()
    except FileNotFoundError as e:
        return f"Erreur de fichier non trouvé : {e}"
```

```
    # Test
    # data = lire_fichier("../data/Orgueil_et_Prejuges.txt")
    # print(data)

    data = lire_fichier("../data/fichier_inexistant.txt")
    print(data)
```

In [ ]:
```
def diviser(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        return f"Erreur de division par zéro : {e}"

# Test
print(diviser(10, 0))
```

In [ ]:
```
def acceder_variable(nom_variable):
    try:
        return eval(nom_variable) # renvoie contenu de la variable
    except NameError as e:
        return f"Erreur de nom : {e}"

# Test
print(acceder_variable("val_entree"))
print(acceder_variable("variable_inexistante"))
```

In [ ]:
```
def conversion_en_entier(chaine):
    try:
        return int(chaine)
    except ValueError as e:
        return f"Erreur de valeur : {e}"

def additionner(a, b):
    try:
        return a + b
    except TypeError as e:
        return f"Erreur de type : {e}"

def convertir_et_additionner(chaine1, chaine2):
    valeur1 = conversion_en_entier(chaine1)
    valeur2 = conversion_en_entier(chaine2)

    # Vérification si les conversions se sont bien passées
    if isinstance(valeur1, int) and isinstance(valeur2, int):
        return additionner(valeur1, valeur2)
    else:
        # Retourne les messages d'erreur si la conversion a échoué
        return valeur1 if isinstance(valeur1, str) else valeur2

# Test
print(convertir_et_additionner("3", "4"))
print(convertir_et_additionner(3.8, "4"))
# print(convertir_et_additionner("a", "4"))
```

Vous pouvez avoir plusieurs blocs `except` pour gérer différents types d'exceptions. Cela permet de réagir de manière appropriée à différentes erreurs qui peuvent survenir.

```
try:
    # Tentative d'exécution de code
    ...
except TypeError:
    # Gestion de TypeError
    ...
except ValueError:
    # Gestion de ValueError
    ...
```

In [ ]:
```
def operation_complexe(a, b, c):
    try:
        resultat = (a + b) / c
        return resultat
    except TypeError as e:
        return f"Erreur de type : {e}"
    except ZeroDivisionError as e:
        return f"Erreur de division par zéro: {e}"
    except Exception as e:
        return f"Autre erreur : {e}"

# Exemple d'utilisation

print(operation_complexe(1, 3, 2))
print(operation_complexe(1, 2, 0))
print(operation_complexe(1, "deux", 3))
```

Dans cet exemple, la fonction `operation_complexe` essaie d'effectuer une opération qui peut lever plusieurs types d'exceptions :

- `TypeError` si `a`, `b`, ou `c` n'est pas un nombre.
- `ZeroDivisionError` si `c` est zéro.
- `Exception` comme un filet de sécurité pour attraper d'autres types d'exceptions inattendues.

L'utilisation de plusieurs blocs `except` permet de donner des réponses spécifiques à chaque type d'erreur, améliorant ainsi la lisibilité et la maintenance du code.

3.2.2 Finally

Le bloc `finally` en Python est utilisé pour définir des actions qui doivent être exécutées après les blocs `try` et `except`, peu importe si une exception a été levée ou non.

Ce bloc est souvent utilisé pour des opérations de nettoyage qui doivent s'exécuter dans tous les cas, comme la fermeture de fichiers ou la libération de ressources externes.

In [ ]:
```
def calculer_division(a, b):
    try:
        resultat = a / b
        return resultat
    except ZeroDivisionError:
        return "Une division par zéro a été tentée."
    finally:
        print("Opération terminée (réussie ou non).")
        return a + b

# Exemple d'utilisation
print(calculer_division(10, 2))  # Pas d'erreur, affiche le résultat et le message de finally
print(calculer_division(10, 0))  # Attrape ZeroDivisionError, affiche le message d'erreur et le message de finally
```

In [ ]:
```
def exemple_finally():
    try:
        fichier = open("../data/Orgueil_et_Prejuges.txt", "r")
        contenu = fichier.read()
        return contenu
    except FileNotFoundError as e:
        return f"Erreur : {e}"
    finally:
        fichier.close()
        print("Fichier fermé.")

# Exemple d'utilisation
print(exemple_finally())
```

3.2.3 Else

Utilisez `else` lorsque vous avez besoin d'exécuter du code qui ne doit s'exécuter que si le bloc `try` s'est exécuté sans erreur, mais avant d'exécuter le code dans le bloc `finally`.

In [ ]:
```
def exemple_else(chaine):
    try:
        num = int(chaine)
    except ValueError:
        return "Ce n'est pas un nombre valide."
    else:
        return f"Conversion réussie : {num}"

# Exemple d'utilisation
print(exemple_else("1233717237"))
print(exemple_else("abc"))
```

In [ ]:
```
def division_avec_verification(a, b):
    try:
        resultat = a / b
    except ZeroDivisionError:
        return "Une division par zéro a été tentée."
    else:
        if resultat > 50:
            return "Le résultat est étonnamment grand."
    finally:
        print("Opération de division terminée (réussie ou non).")
```

In [ ]:
```
# Exemple d'utilisation
print(division_avec_verification(1000, 10))  # Division réussie, vérification dans 'else', puis 'finally'
```

In [ ]:
```
print(division_avec_verification(10, 0))    # Attrape ZeroDivisionError, puis exécute 'finally'
```

3.3 Assertions

L'instruction `assert` est utilisée pour tester si une condition est vraie. Si la condition est fausse, `AssertionError` est levée.

Les assertions sont principalement utilisées comme un moyen de débogage, pour vérifier des conditions internes, tandis que les exceptions sont utilisées pour gérer des erreurs et des conditions exceptionnelles dans le flux normal du programme.

In [ ]:
```
def diviser_par_deux(n):
    assert n % 2 == 0, "Le nombre doit être pair."
    return n / 2
```

In [ ]:
```
# Exemple d'utilisation
print(diviser_par_deux(10))  # Fonctionne bien
```

In [ ]:
```
print(diviser_par_deux(3))  # Déclenche AssertionError
```

In [ ]:
```
def ma_fonction(x):
    assert isinstance(x, int), "La variable doit être un nombre entier"
    # ...

ma_fonction(10)  # OK
ma_fonction("toto")  # Assertion error
```

In [ ]:
```
def ma_fonction(x):
    assert x > 0, "La variable doit être positive"
    # ...

ma_fonction(5)  # OK
ma_fonction(-1)  # Assertion error
```

In [ ]:
```
def ma_fonction(x, y):
    assert x != y, "Les deux variables doivent être différentes"
    # ...

try:
    ma_fonction(10, 10)
except AssertionError:
    print("Les deux variables sont identiques")
```

3.4 Raise

L'instruction `raise` en Python est un outil crucial pour la gestion des exceptions, permettant aux développeurs de déclencher explicitement des exceptions lorsqu'une condition spécifique se produit dans le programme.

Les exceptions levées avec `raise` peuvent être capturées et gérées en amont, permettant aux développeurs de réagir de manière appropriée aux différents types de problèmes qui peuvent survenir, améliorant ainsi la robustesse et la fiabilité du code.

Dans l'exemple suivant, si `set_age` est appelée avec un âge négatif, une `ValueError` est levée, indiquant que l'âge ne peut pas être négatif.

In [ ]:
```
def set_age(age):
    if age < 0:
        raise ValueError("L'âge ne peut pas être négatif.")
    print(f"Âge défini à {age} ans.")

try:
    set_age(-10)
except ValueError as e:
    print(e)
```

Ici, `diviser` lève une `ZeroDivisionError` si le diviseur (`b`) est zéro. Cela empêche l'exécution d'une division par zéro, ce qui entraînerait une erreur en Python.

In [ ]:
```
def diviser(a, b):
    if b == 0:
        raise ZeroDivisionError("Le diviseur ne peut pas être zéro.")
    return a / b

try:
    resultat = diviser(10, 0)
except ZeroDivisionError as e:
    print(e)
```

In [ ]:
```
def ma_fonction(x):
    if x < 0:
        raise ValueError("La variable doit être positive")
    # ...

ma_fonction(10)  # OK
ma_fonction(-1)  # ValueError: La variable doit être positive
```

In [ ]:
```
def ma_fonction(x):
    try:
      y = 1 / x
    except ZeroDivisionError:
      raise ValueError("La variable ne peut pas être égale à 0")
    # ...

ma_fonction(10)  # OK
ma_fonction(0)  # ValueError: La variable ne peut pas être égale à 0
```

En résumé : Exceptions Intégrées

Voici quelques-unes des exceptions intégrées courantes en programmation Python ainsi que les erreurs qui les provoquent.

Exception	Cause de l'Erreur
AssertionError	Levée lorsque l'instruction assert échoue.
AttributeError	Levée lorsque l'assignation ou la référence à un attribut échoue.
EOFError	Levée lorsque input() atteint la fin du fichier.
FloatingPointError	Levée lorsqu'une opération sur les nombres à virgule flottante échoue.
GeneratorExit	Levée lorsque la méthode close() d'un générateur est appelée.
ImportError	Levée lorsque le module importé n'est pas trouvé.
IndexError	Levée lorsque l'index d'une séquence est hors de portée.
KeyError	Levée lorsque une clé n'est pas trouvée dans un dictionnaire.
KeyboardInterrupt	Levée lorsque l'utilisateur appuie sur la touche d'interruption (Ctrl+C ou Suppr).
MemoryError	Levée lorsqu'une opération manque de mémoire.
NameError	Levée lorsqu'une variable n'est pas trouvée dans la portée locale ou globale.
NotImplementedError	Levée par des méthodes abstraites.
OSError	Levée lorsqu'une opération système provoque une erreur liée au système.
OverflowError	Levée lorsque le résultat d'une opération arithmétique est trop grand pour être représenté.
ReferenceError	Levée lorsqu'une référence faible est utilisée pour accéder à un référent collecté par le ramasse-miettes.
RuntimeError	Levée lorsqu'une erreur ne rentre dans aucune autre catégorie.
StopIteration	Levée par next() pour indiquer qu'il n'y a plus d'élément à renvoyer par l'itérateur.
SyntaxError	Levée par l'analyseur lorsque une erreur de syntaxe est rencontrée.
IndentationError	Levée lorsqu'il y a une indentation incorrecte.
TabError	Levée lorsque l'indentation est composée de tabulations et d'espaces inconsistants.
SystemError	Levée lorsque l'interpréteur détecte une erreur interne.
SystemExit	Levée par la fonction sys.exit().
TypeError	Levée lorsqu'une fonction ou une opération est appliquée à un objet de type incorrect.
UnboundLocalError	Levée lorsqu'une référence est faite à une variable locale dans une fonction ou une méthode, mais aucune valeur n'a été liée à cette variable.
UnicodeDecodeError	Levée lorsqu'une erreur liée à Unicode se produit pendant le décodage.
ValueError	Levée lorsqu'une fonction reçoit un argument de type correct mais de valeur incorrecte.
ZeroDivisionError	Levée lorsque le deuxième opérande de l'opération de division ou de modulo est zéro.

Une liste exhaustive des exceptions intégrées en Python peut être trouvée dans la documentation de Python.

Nous pouvons voir toutes les exceptions intégrées en utilisant la fonction locale intégrée comme suit :

In [ ]:
```
print(dir(locals()['__builtins__']))
```

4. Lecture et écriture de fichiers

4.1 Gestion simple des fichiers

1. Fonction open()

- La fonction open() est utilisée pour ouvrir un fichier en Python.
- Syntaxe : open(nom_de_fichier, mode) .
- Le nom_de_fichier spécifie le chemin d'accès, tandis que le mode détermine le type d'opération (lecture, écriture, etc.).

2. Modes d'Ouverture de Fichier Mode | Description ---- | ---------- r | Ouvre un fichier en lecture. (par défaut) w | Ouvre un fichier en écriture. Crée un nouveau fichier s'il n'existe pas ou tronque le fichier s'il existe. x | Ouvre un fichier en création exclusive. Si le fichier existe déjà, l'opération échoue. a | Ouvre un fichier pour ajouter à la fin du fichier sans le tronquer. Crée un nouveau fichier s'il n'existe pas. t | Ouvre en mode texte. (par défaut) b | Ouvre en mode binaire.

3. Bonnes Pratiques

- Utiliser le gestionnaire de contexte with pour une meilleure gestion des exceptions et une fermeture automatique du fichier.

In [ ]:
```
chemin_fichier = "../data/fichier_lu.txt"

# Lire un fichier
with open(chemin_fichier, 'r', encoding='utf-8') as fichier:
    contenu = fichier.read()
    print(contenu)
```

In [ ]:
```
# Écrire dans un fichier
chemin_ecriture = '../data/fichier_ecris.txt'

with open(chemin_ecriture, 'w') as fichier:
    fichier.write("Bonjour!Comment allez-vous ?")
```

4. Lire un fichier ligne par ligne

- Utilisation de for line in file: pour lire un fichier ligne par ligne.
- Cela permet de traiter de grands fichiers sans surcharger la mémoire.
- Utiliser read() , readline() , readlines() **
  - read() lit tout le contenu du fichier.
  - readline() lit une ligne à la fois.
  - readlines() lit toutes les lignes et les retourne sous forme de liste.

In [ ]:
```
chemin_fichier = "../data/fichier_lu.txt"

# Lire un fichier ligne par ligne
with open(chemin_fichier, 'r', encoding='utf-8') as omelette:
    for ligne in omelette:
        print("****")
        print(ligne.strip())
```

Notez que ce qui précède est une méthode courante pour la lecture "paresseuse" de gros fichiers en Python, surtout lors de la lecture de gros fichiers sur un système avec une mémoire limitée.

In [ ]:
```
for line in open('vraiment_gros_fichier.dat'):
    process_data(line)
```

Alternativement, vous pouvez simplement utiliser yield:

In [ ]:
```
def lire_par_morceaux(objet_fichier, taille_morceau=1024):
    """Fonction paresseuse (générateur) pour lire un fichier morceau par morceau.
    Taille de morceau par défaut : 1 ko."""
    while True:
        donnees = objet_fichier.read(taille_morceau)
        if not donnees:
            break
        yield donnees

with open('vraiment_gros_fichier.dat') as f:
    for morceau in lire_par_morceaux(f):
        process_data(morceau)
```

5. Ecrire un fichier ligne par ligne

- writelines() prend une liste de chaînes de caractères et les écrit dans un fichier.

In [ ]:
```
# Écrire dans un fichier
chemin_ecriture = '../data/rapport.txt'

with open(chemin_ecriture, 'w',  encoding='utf-8') as fichier:
    fichier.write("Rapport du Jour\n")
    fichier.write("====================\n")
    fichier.write("Le temps était ensoleillé et le parc était paisible.\n")
```

In [ ]:
```
# Écrire une liste de lignes
lignes = ["Première ligne\n", "Deuxième ligne\n", "Troisième ligne\n"]

with open(chemin_ecriture, 'a', encoding='utf-8') as fichier:
    fichier.writelines(lignes)
```

En résumé : Méthodes de Fichier Python

Voici la liste complète des méthodes en mode texte avec une brève description :

Méthode	Description
close()	Ferme un fichier ouvert. Elle n'a aucun effet si le fichier est déjà fermé.
detach()	Sépare le tampon binaire sous-jacent du TextIOBase et le renvoie.
fileno()	Renvoie un numéro entier (descripteur de fichier) du fichier.
flush()	Vide le tampon d'écriture du flux de fichier.
isatty()	Renvoie True si le flux de fichier est interactif.
read(n)	Lit au plus n caractères du fichier. Lit jusqu'à la fin du fichier s'il est négatif ou None.
readable()	Renvoie True si le flux de fichier peut être lu.
readline(n=-1)	Lit et renvoie une ligne du fichier. Lit au plus n octets si spécifié.
readlines(n=-1)	Lit et renvoie une liste de lignes du fichier. Lit au plus n octets si spécifié.
seek(offset, from=SEEK_SET)	Modifie la position du fichier à offset octets, par rapport à from (début, courant, fin).
seekable()	Renvoie True si le flux de fichier prend en charge l'accès aléatoire.
tell()	Renvoie l'emplacement actuel du fichier.
truncate(size=None)	Redimensionne le flux de fichier à size octets. Si size n'est pas spécifié, redimensionne à l'emplacement actuel.
writable()	Renvoie True si le flux de fichier peut être écrit.
write(s)	Écrit la chaîne s dans le fichier et renvoie le nombre de caractères écrits.
writelines(lines)	Écrit une liste de lignes dans le fichier.

4.2 Travailler avec différents types de fichiers

- Utilisation du module `csv` pour lire et écrire des fichiers CSV.
- Utilisation du module `json` pour manipuler des données JSON.

In [ ]:
```
import csv

# Lire un fichier CSV
with open('../data/donnees.csv', mode ='r') as fichier_csv:
    lecteur_csv = csv.reader(fichier_csv)
    for ligne in lecteur_csv:
        print(ligne)
```

In [ ]:
```
# Dictionnaire
donnees = [{"nom": "Dupont", "age": 30},
           {"nom": "Martin", "age": 40, "ville": "Nice", "profession": "Médecin"}]
print(donnees)
```

In [ ]:
```
import json

# Écrire et lire des données JSON
with open('../data/donnees.json', 'w', encoding='utf-8') as fichier_json:
    json.dump(donnees, fichier_json)
```

In [ ]:
```
with open('../data/donnees.json', 'r', encoding='utf-8') as fichier_json:
    donnees_lues = json.load(fichier_json)
    print(donnees_lues)
```

5. Modules et Packages

Les modules et les packages Python sont des outils essentiels pour organiser et partager du code. Ils permettent de créer des programmes plus modulaires et plus réutilisables.

5.1. Modules

Qu'est-ce qu'un module ?

Un module en Python est un fichier contenant du code Python pouvant être importé et utilisé dans d'autres programmes. Il permet de regrouper des fonctions, des classes et des variables dans un fichier unique, ce qui facilite la réutilisation du code et la création de programmes plus modulaires.

Créer un module

Pour créer un module, il suffit de créer un **fichier .py** contenant le code que vous souhaitez partager.

Le nom du fichier doit correspondre au nom du module. Par exemple, un module nommé utils.py pourrait contenir les fonctions suivantes :

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

def soustraire(a, b):
  return a - b

def multiplier(a, b):
  return a * b
```

Importer un module local

Pour utiliser un module dans un autre programme, vous devez l'importer en utilisant l'instruction import . Par exemple, pour importer le module utils.py dans un programme, vous pouvez utiliser le code suivant :

```
import utils

# Appeler les fonctions du module
resultat = utils.additionner(1, 2)
print(resultat)  # 3

resultat = utils.soustraire(5, 3)
print(resultat)  # 2
```

In [ ]:
```
import fichier

fichier.addition(2, 4 )
```

Importer un module installé https://docs.python.org/3/py-modindex.html

Pour utiliser un module dans un autre programme, vous devez l'importer en utilisant l'instruction import . Par exemple, le code suivant importe le module math et utilise les fonctions log10 et floor :

In [ ]:
```
import csv
import math

# help(csv)
# help(math)

math.log10(34)
math.floor(977713.710937)
```

5.2. Packages

Un package Python est un ensemble de modules regroupés dans un dossier. Les packages permettent de mieux organiser le code et de le distribuer plus facilement. La structure d'un package est la suivante :

```
└─── package
   ├─── __init__.py
   ├─── module1.py
   └─── module2.py
```

Le fichier `__init__.py` est un fichier vide qui permet d'initialiser le package. Pour importer un module d'un package, vous devez utiliser le nom du package suivi du nom du module, séparés par un point. Par exemple, pour importer le module `module1` du package `package`, vous devez utiliser l'instruction suivante :

```
import package.module1
```

Pour installer un package Python, vous pouvez utiliser la commande `pip`. Par exemple, pour installer le package `requests`, vous pouvez utiliser la commande suivante :

```
pip install requests
```

In [ ]:
```
from monpackage import operations

print(operations.somme(2, 3))
print(operations.division(2, 3))
```

In [ ]:
```
from operations import division # plus nécessaire de préciser le module et importe que la fonction

resultat = division(4, 2)
print(resultat)
```

In [ ]:
```
from operations import division, somme # plusieurs fonctions d'un module

resultat = somme(4, 2)
print(resultat)
```

In [ ]:
```
import random
# help(random)

nombre_aleatoire = random.randint(1, 90)
liste_nombre_aleatoire = random.sample(range(1, 100), 18)
print(nombre_aleatoire, '\n', liste_nombre_aleatoire)
```

In [ ]:
```
import random

# tirer au sort aléatoire un objet d'une liste
objects = ['table', "stylo", "livre"]

object_aleatoire = random.choice(objects)
print(f" l'objet tiré est : {object_aleatoire}")
```

In [ ]:
```
from monpackage import pierre_feuille_ciseaux

pierre_feuille_ciseaux.jeu()
```

6. Bonnes Pratiques de Programmation

6.1 Introduction

Les bonnes pratiques en programmation sont des standards ou des conventions qui facilitent la lecture, la compréhension et la maintenance du code par d'autres développeurs. Elles jouent un rôle crucial dans le travail d'équipe et la gestion de projets à long terme.

Pourquoi Suivre les Bonnes Pratiques ?

- **Lisibilité** : Code plus clair et facile à comprendre.
- **Maintenance** : Simplification des mises à jour et corrections.
- **Collaboration** : Uniformité du code au sein d'une équipe.

Exemple Illustratif Comparons deux morceaux de code pour voir l'impact des bonnes pratiques.

Code Sans Bonnes Pratiques

```
def f(x,y):return x+y
```

Code Avec Bonnes Pratiques

```
def add_numbers(x, y):
    """Add two numbers and return the result."""
    return x + y
```

6.2 La conformité PEP

La conformité à la PEP 8 concerne les règles de style de codage. Elle vise à rendre le code plus lisible et cohérent. Voici quelques points clés et exemples :

1. **Indentation :** Utiliser 4 espaces par niveau d'indentation.
   - Mauvais : `if x: x += 1`
   - Bon :
     ```
     if x:
         x += 1
     ```
2. **Longueur de Ligne :** Limiter les lignes à 79 caractères.
   - Mauvais : une seule ligne très longue.
   - Bon : diviser en plusieurs lignes avec des parenthèses, crochets, ou accolades.
3. **Espaces :** Utiliser des espaces autour des opérateurs et après les virgules.
   - Mauvais : `def fonction(x,y=0):`
   - Bon : `def fonction(x, y = 0):`
4. **Noms de Variables :** Utiliser des noms descriptifs et éviter les abréviations.
   - Mauvais : `n` , `df` , `x1`
   - Bon : `nombre` , `dataframe` , `variable_explicative`
5. **Commentaires :** Ils doivent être clairs et pertinents, pas redondants.
   - Mauvais : `x = x + 1  # Augmente x de 1`
   - Bon : `x = x + 1  # Compense pour le décalage`

Ces exemples illustrent comment l'adhésion à la PEP 8 peut améliorer la lisibilité et la cohérence du code. Pour une compréhension détaillée, il est conseillé de se référer directement à la PEP 8.

[ici] : (https://peps.python.org/pep-0008/)

6.3 Outils de contrôle qualité du code

Il est en effet important d'évaluer la qualité du code Python pour assurer sa lisibilité, sa maintenabilité et sa conformité aux recommandations de la PEP 8 (style de code) et de la PEP 257 (docstrings). Vous pouvez utiliser différents outils et sites pour vous aider dans cette tâche. Voici quelques options :

1. **pep8online.com** : Comme vous l'avez mentionné, c'est un site en ligne qui permet de vérifier la conformité du code à la PEP 8 en collant simplement le code dans la zone de texte et en cliquant sur le bouton "Check code". Il vous donnera des indications sur les problèmes potentiels de style.

2. **pylint** : C'est un outil en ligne de commande qui offre une analyse plus approfondie du code. Il génère des rapports avec des notes de conformité à la PEP 8 et d'autres recommandations de style. Vous pouvez l'installer via pip (`pip install pylint`) et l'exécuter sur un fichier ou un répertoire.

3. **flake8** : C'est un autre outil en ligne de commande qui combine plusieurs vérifications statiques, y compris la conformité à la PEP 8. Il peut être installé via pip (`pip install flake8`) et utilisé de manière similaire à pylint.

4. **IDEs (Environnements de Développement Intégrés)** : De nombreux IDEs populaires comme PyCharm, VSCode, et Sublime Text incluent des extensions ou des fonctionnalités intégrées pour vérifier automatiquement la conformité à la PEP 8 et afficher des avertissements ou des erreurs directement dans l'éditeur pendant que vous écrivez du code.

5. **GitHub Actions ou Travis CI** : Si vous travaillez sur des projets collaboratifs, vous pouvez configurer des services d'intégration continue comme GitHub Actions ou Travis CI pour exécuter automatiquement des vérifications de conformité à la PEP 8 à chaque modification de code.

6. **Précommit Hooks** : Vous pouvez également configurer des hooks précommit avec des outils tels que `pre-commit` , qui exécutent automatiquement des vérifications de style chaque fois que vous vous apprêtez à valider votre code.

L'utilisation d'un ou plusieurs de ces outils et méthodes peut grandement contribuer à améliorer la qualité de votre code Python en garantissant sa conformité aux recommandations de la PEP 8 et de la PEP 257. Cela rendra votre code plus lisible, plus facile à maintenir et plus cohérent, ce qui est essentiel pour un développement logiciel efficace et collaboratif.

7.Certification avec DataCamp - bonus

Aller plus loin en programmation Python !

Êtes-vous fatigué de:

- Passer des heures à essayer de comprendre Python par vous-même?
- Manquer d'opportunités pour développer vos compétences?
- Ne pas avancer dans votre carrière?

Imaginez avoir accès à:

✔ Des cours de pointe

✔ Des exercices pratiques

✔ Un soutien d'experts

Pour maîtriser Python facilement avec DataCamp, la plateforme que j'utilise quotidiennement!

En vous inscrivant avec mon **lien affilié**, vous profiterez de:

50% de remise sur l'abonnement

Un gain de temps précieux

Pas d'efforts de recherche fastidieux

Les experts ont tout condensé pour vous dans des formations complètes et récentes.

1) Cliquez **ici** pour l'offre exclusive

2) Commencez à maîtriser Python aujourd'hui

3) Investissez dans vos compétences

4) Rejoignez la communauté DataCamp comme moi

5) Apprenez d'autres langages et technologies (SQL, Prompt Engineering, NLP...)!

5) Trouver un travail directement sur la plateforme !

8. Références

Dans cette section, nous fournissons une liste de références pour approfondir votre compréhension de certains sujets abordés dans ce notebook :

1. **Python's any and all functions**: Documentation officielle sur les fonctions any et all de Python, qui permettent de vérifier si au moins un élément d'une séquence est vrai ou si tous les éléments sont vrais, respectivement.
2. **Python's eval() function**: Documentation officielle sur la fonction eval() de Python, qui évalue une expression Python passée sous forme de chaîne de caractères.
3. **filter() in Python**: Documentation officielle sur la fonction filter() de Python, qui filtre les éléments d'une séquence en fonction d'une fonction de filtrage donnée.
4. **dir() in Python**: Documentation officielle sur la fonction dir() de Python, qui retourne la liste des noms dans l'espace de noms local ou global.
5. **isinstance and issubclass in Python**: Documentation officielle sur les fonctions isinstance() et issubclass() de Python, utilisées pour vérifier les types d'objets.
6. **len() in Python**: Documentation officielle sur la fonction len() de Python, qui retourne la longueur (le nombre d'éléments) d'un objet.
7. **iter() in Python**: Documentation officielle sur la fonction iter() de Python, qui retourne un itérateur pour un objet.
8. **Why should I use operator.itemgetter(x) instead of [x]?**: Une discussion sur Stack Overflow expliquant pourquoi il est préférable d'utiliser operator.itemgetter(x) plutôt que [x] pour accéder aux éléments d'une séquence.

9. **Python's Sorting HowTo wiki**: Guide officiel sur la manière de trier en Python, présentant différentes méthodes de tri pour différentes situations.
10. **Python's Generators wiki**: Page wiki officielle sur les générateurs en Python, expliquant leur utilisation et leur fonctionnement.
11. **Python's Decorators wiki**: Page wiki officielle sur les décorateurs en Python, fournissant des informations sur leur utilisation et leur création.
12. **Python's Decorator Library**: Bibliothèque officielle de décorateurs en Python, offrant une variété de décorateurs prêts à l'emploi pour différentes tâches.
13. **How to make a chain of function decorators?** : Une discussion sur Stack Overflow expliquant comment créer une chaîne de décorateurs de fonctions en Python.
14. **A Guide to Python's Magic Methods** : Guide détaillé sur les méthodes magiques en Python, qui permettent de personnaliser le comportement des objets Python.
15. **The Iterator Protocol: How "For Loops" Work in Python**: Article expliquant en détail le protocole de l'itérateur en Python et comment les boucles for fonctionnent en utilisant ce protocole.
16. **How to make an iterator in Python**: Article expliquant comment créer un itérateur personnalisé en Python en implémentant les méthodes spéciales **iter**() et **next**().
17. **Overusing lambda expressions in Python**: Article expliquant les dangers de la surutilisation des expressions lambda en Python et quand il est approprié de les utiliser.
18. **Python Errors and Built -in Exceptions**: Documentation officielle sur les erreurs et exceptions intégrées en Python, expliquant les différentes erreurs et exceptions qui peuvent se produire dans un programme Python.
19. **Python's Functions Are First-Class**: Article expliquant ce que signifie le fait que les fonctions sont des citoyens de première classe en Python, et comment cela influence la programmation en Python.
20. **The Zen of Python, Explained**: Article expliquant les principes philosophiques derrière le "Zen de Python", qui sont les principes directeurs de la conception du langage Python.
21. **How to use Python's min() and max() with nested lists** : Article expliquant comment utiliser les fonctions min() et max() de Python avec des listes imbriquées pour trouver les éléments minimaux et maximaux.
22. **for/else in Python**: Article expliquant l'utilisation de la clause else avec les boucles for en Python et dans quels cas elle est utile.
23. **Why does python use 'else' after for and while loops?** : Une discussion sur Stack Overflow expliquant pourquoi Python utilise la clause else après les boucles for et while, et comment elle fonctionne.
24. **Generator comprehensions in Python**: Documentation officielle sur les expressions de générateur en Python, qui permettent de créer des générateurs de manière concise et élégante.
25. **How to re-raise an exception in nested try/except blocks?**: Une discussion sur Stack Overflow expliquant comment ré-élever une exception dans des blocs try/except imbriqués en Python.
26. **Python File I/O**: Documentation officielle sur les opérations d'entrée/sortie de fichiers en Python, expliquant comment lire et écrire des fichiers en Python.
27. **Lazy Method for Reading Big File in Python?**: Une discussion sur Stack Overflow proposant des méthodes efficaces pour lire de gros fichiers de manière paresseuse en Python.
28. **How do I merge two dictionaries in a single expression (taking union of dictionaries)?** : Une discussion sur Stack Overflow expliquant comment fusionner deux dictionnaires en une seule expression en Python.
29. **Python Dictionary Tips**: Article fournissant des conseils utiles sur l'utilisation efficace des dictionnaires en Python.
30. **What are the differences between type() and isinstance()?**: Une discussion sur Stack Overflow expliquant les différences entre les fonctions type() et isinstance() en Python et quand les utiliser.
31. **Python 3 Tutorial by Aman Chadha**: Ce tutoriel Python 3 fournit une introduction complète au langage de programmation Python, couvrant les concepts de base, les structures de contrôle, les fonctions, les classes, les modules et bien plus encore. Il est rédigé de manière claire et accessible, ce qui en fait une ressource utile pour les débutants et les programmeurs intermédiaires qui souhaitent renforcer leurs connaissances en Python.

En consultant ces références, vous pourrez approfondir votre compréhension des concepts abordés dans ce notebook et continuer à développer vos compétences en Python.