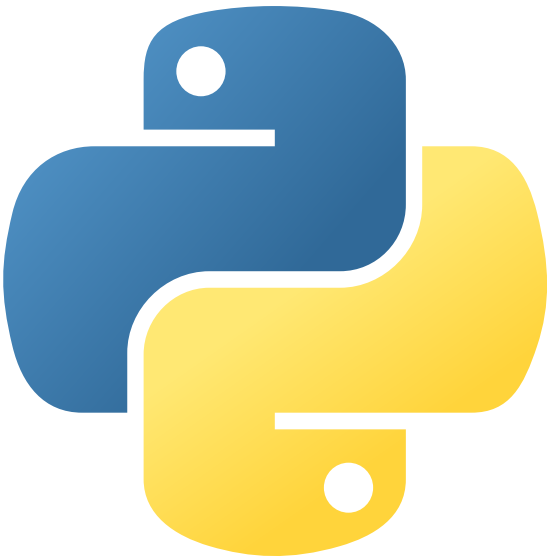


Introduction à Python

Emmanuel Evilafo

2024

Sommaire

12. **Pandas (Les Bases et Analyse de Données)**
    - Introduction à Pandas : Series et DataFrame
    - Chargement, manipulation et exploration des données avec Pandas
    - Exercice : Analyse de données avec Pandas
13. **Matplotlib (Graphiques de Base)**
    - Introduction à Matplotlib et création de graphiques simples
    - Personnalisation des graphiques (titres, labels, légendes)
    - Types de graphiques : courbes, histogrammes, barres, nuages de points
    - Exercice : Créer et personnaliser des graphiques avec Matplotlib

Introduction

Bienvenue dans ce cours d'**Initiation à Python**, conçu pour vous fournir une base solide dans l'un des langages de programmation les plus populaires et polyvalents au monde. Que vous soyez un **débutant complet** ou que vous ayez déjà des notions de programmation, ce cours vous guidera à travers les concepts fondamentaux de Python et vous aidera à acquérir les compétences nécessaires pour résoudre des problèmes concrets et mener à bien des projets de développement.

Objectifs du Cours

Ce cours a pour but de vous initier aux concepts clés de Python tout en vous donnant des outils pour **manipuler des données** et **créer des visualisations** claires. À la fin de ce cours, vous serez capable de :

- **Comprendre et écrire du code Python** pour des applications simples,
- Manipuler et analyser des **ensembles de données** avec les bibliothèques **NumPy** et **Pandas**,
- Créer des **visualisations de données** avec **Matplotlib** pour mieux comprendre et communiquer vos analyses.

Méthode d'Apprentissage

Chaque module de ce cours est structuré autour de :

- **Notions théoriques** : explication des concepts et de la syntaxe Python,
- **Exemples pratiques** : illustrations concrètes de l'utilisation des concepts,
- **Exercices** : pour vous permettre de pratiquer et d'ancrer les connaissances.

À la fin de chaque module, vous aurez l'opportunité de mettre en pratique ce que vous avez appris grâce à des exercices. Pour vous accompagner dans cette progression, n'hésitez pas à consulter le compte **GitHub** dédié à ce cours où vous trouverez les récapitulatifs et exercices correspondants.

Nous espérons que ce cours vous aidera à démarrer votre parcours de **développeur Python** et à explorer les nombreuses possibilités offertes par ce langage passionnant. Bonne formation !

Module 1 : Introduction à Python et Premiers Pas

Durée : 2 heures

Objectifs :

- Comprendre les bases de Python et ses domaines d'application.
- Installer Python et Anaconda pour un environnement de développement complet.
- Écrire et exécuter un premier programme Python.

1.1 Présentation de Python (30 minutes)

Qu'est-ce que Python ?

Python est un langage de programmation de haut niveau créé par **Guido van Rossum** en 1991. Reconnu pour sa simplicité, sa lisibilité et sa polyvalence, Python est utilisé dans plusieurs domaines, notamment :

- **Développement web** : avec des frameworks comme Django et Flask.
- **Data science** : manipulation et analyse de données avec des bibliothèques telles que NumPy, Pandas, et Matplotlib.
- **Automatisation des tâches** : scripts pour automatiser des processus répétitifs.
- **Intelligence artificielle et apprentissage automatique** : grâce à des outils comme TensorFlow et PyTorch.

Pourquoi apprendre Python ?

- **Facilité d'apprentissage** : Syntaxe claire et intuitive, idéale pour les débutants.
- **Polyvalence** : Utilisé dans une grande variété de projets.
- **Large écosystème** : Nombreuses bibliothèques pour accélérer le développement.
- **Communauté active** : Documentation et support facilement disponibles.

1.2 Installation de Python et Anaconda (40 minutes)

Installation de Python

1. Rendez-vous sur le site officiel python.org/downloads.
2. Téléchargez et installez la version appropriée pour votre système d'exploitation.
3. Pendant l'installation, cochez l'option **"Add Python to PATH"** pour permettre à votre système de reconnaître Python dans le terminal.

Installation d'Anaconda

Anaconda est une distribution Python qui comprend plusieurs bibliothèques essentielles ainsi que des outils comme **Jupyter Notebook** pour faciliter l'écriture de code interactif.

1. Téléchargez Anaconda depuis anaconda.com.
2. Suivez les instructions d'installation pour votre système d'exploitation.
3. Une fois installé, ouvrez **Anaconda Navigator** pour gérer vos environnements et accéder à des applications telles que Jupyter Notebook et Spyder.

Vérification de l'installation

Après l'installation d'Anaconda, ouvrez un terminal (ou Anaconda Prompt) et tapez la commande suivante pour vérifier que tout est en place :

```
bash
```

```
conda --version
```

Cette commande doit retourner la version de Conda, l'outil de gestion d'environnement d'Anaconda.

Lancement de Jupyter Notebook avec Anaconda

1. Ouvrez **Anaconda Navigator**.
2. Dans le tableau de bord, cliquez sur **Jupyter Notebook** pour lancer l'application.
3. Cela ouvrira une interface web vous permettant de créer et exécuter des notebooks Python.

1.3 Écriture et exécution de votre premier programme (50 minutes)

Le classique "Hello, World!"

Pour commencer, écrivons un programme simple. Dans Jupyter Notebook ou un éditeur de code (comme VSCode ou Spyder), tapez le code suivant :

```
print("Hello, World!")
```

Lorsque vous exécutez ce programme, il affichera :

```
Hello, World!
```

- **Explication** :
  - La fonction `print()` affiche du texte à l'écran.
  - `"Hello, World!"` est une chaîne de caractères (texte délimité par des guillemets).

Variables et types de données

Python permet de stocker des informations dans des **variables**. Les types de données les plus courants incluent :

- **Entiers (int)** : des nombres entiers, par exemple : 42.
- **Flottants (float)** : des nombres décimaux, par exemple : 3.14.
- **Chaînes de caractères (str)** : du texte, par exemple : "Bonjour".
- **Booléens (bool)** : valeurs logiques, `True` ou `False`.

Exemple :

```
nom = "Alice"
age = 25
est_majeur = True
print(nom, "a", age, "ans. Est-elle majeure ?", est_majeur)
```

Interagir avec l'utilisateur

Vous pouvez demander des informations à l'utilisateur à l'aide de la fonction `input()` :

```
nom = input("Entrez votre nom : ")
print("Bonjour", nom, "!")
```

Cela permet à l'utilisateur d'entrer son nom, et le programme lui répond de manière personnalisée.

Manipulations basiques

Exemple d'opérations sur des nombres :

```
a = 10
b = 3
print("Addition :", a + b)
print("Soustraction :", a - b)
print("Multiplication :", a * b)
print("Division :", a / b)
print("Division entière :", a // b)
```

Ce programme exécute des opérations arithmétiques simples.

1.4 Exercice pratique et Questions/Réponses (20 minutes)

Exercice : Créer un programme interactif

Écrivez un programme qui demande le nom et l'âge de l'utilisateur, puis affiche un message personnalisé :

```
nom = input("Quel est votre nom ? ")
age = int(input("Quel est votre âge ? "))
print("Bonjour", nom, "! Vous avez", age, "ans.")
```

Cet exercice permet de consolider les bases : interaction avec l'utilisateur, utilisation de variables et manipulation de types de données.

Module 2 : Variables, Opérations et Fonctions

Durée : 2 heures

Objectifs :

- Comprendre et manipuler les variables et les types de données en Python.
- Utiliser les opérateurs mathématiques et logiques.
- Créer et utiliser des fonctions pour structurer le code.

2.1 Variables et Types de Données (30 minutes)

Définition d'une variable

Une variable est une boîte qui stocke des informations. En Python, il suffit d'attribuer une valeur à un nom de variable pour la créer :

```
nom = "Alice"
age = 30
```

Ici, `nom` est une variable de type chaîne de caractères (**str**) et `age` est une variable de type entier (**int**).

Types de données de base en Python

Python prend en charge plusieurs types de données natifs, notamment :

- **Entiers (int)** : pour les nombres entiers.
- **Flottants (float)** : pour les nombres à virgule flottante.
- **Chaînes de caractères (str)** : pour du texte.
- **Booléens (bool)** : `True` ou `False`, pour exprimer la logique.

Exemples :

```
temperature = 36.6  # Type float
prenom = "Jean"     # Type str
est_actif = True    # Type bool
```

Typecasting (conversion de types)

Il est parfois nécessaire de convertir un type de données en un autre. Par exemple, convertir une chaîne en nombre :

```
age = int(input("Entrez votre âge : "))  # Conversion en entier
taille = float(input("Entrez votre taille : "))  # Conversion en flottant
```

Python offre des fonctions intégrées telles que `int()`, `float()`, `str()` pour convertir entre les types de données.

2.2 Opérations et opérateurs en Python (30 minutes)

Opérateurs arithmétiques

Les opérateurs de base permettent de faire des calculs sur des nombres :

Opérateur	Description	Exemple
`+`	Addition	`3 + 2 = 5`
`-`	Soustraction	`3 - 2 = 1`
`*`	Multiplication	`3 * 2 = 6`
`/`	Division	`3 / 2 = 1.5`
`//`	Division entière	`3 // 2 = 1`
`%`	Modulo (reste de division)	`3 % 2 = 1`
`**`	Exponentiation (puissance)	`3 ** 2 = 9`

Exemple d'utilisation :

```
a = 5
b = 2
print("Addition :", a + b)
print("Division entière :", a // b)
print("Puissance :", a ** b)
```

Opérateurs de comparaison

Ces opérateurs permettent de comparer des valeurs et retournent des booléens (`True` ou `False`) :

Opérateur	Description	Exemple
`==`	Égal à	`3 == 3` -> `True`
`!=`	Différent de	`3 != 2` -> `True`
`>`	Supérieur à	`3 > 2` -> `True`
`<`	Inférieur à	`2 < 3` -> `True`
`>=`	Supérieur ou égal à	`3 >= 3` -> `True`
`<=`	Inférieur ou égal à	`2 <= 3` -> `True`

Opérateurs logiques

Les opérateurs logiques sont utilisés pour combiner des conditions :

Opérateur	Description	Exemple
`and`	Les deux doivent être vraies	`(a > 1) and (b < 5)`
`or`	L'une ou l'autre doit être vraie	`(a > 1) or (b > 5)`
`not`	Inverse la condition	`not(a > 1)`

Exemple d'utilisation des opérateurs logiques :

```
a = 5
b = 2
print((a > 3) and (b < 5))  # True
```

2.3 Fonctions en Python (60 minutes)

Qu'est-ce qu'une fonction ?

Une fonction est un bloc de code qui effectue une tâche spécifique et peut être réutilisé. Python propose de nombreuses **fonctions intégrées** (comme `print()`), mais vous pouvez également **définir vos propres fonctions**.

Définition d'une fonction

Une fonction est définie avec le mot-clé `def`, suivi du nom de la fonction et de parenthèses. Exemple :

```
def dire_bonjour():
    print("Bonjour, tout le monde!")
```

Pour **appeler** cette fonction, il suffit d'écrire :

```
dire_bonjour()  # Affichera : Bonjour, tout le monde!
```

Fonctions avec paramètres

Les fonctions peuvent prendre des **paramètres** qui sont des données fournies lors de l'appel de la fonction :

```
def saluer(nom):
    print("Bonjour,", nom)
```

Appel de la fonction avec un argument :

```
saluer("Alice")  # Affichera : Bonjour, Alice
```

Fonctions avec retour de valeur

Les fonctions peuvent également renvoyer des valeurs à l'aide du mot-clé `return` :

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

Appel de la fonction :

```
resultat = additionner(3, 4)
print(resultat)  # Affichera : 7
```

Portée des variables (variables locales et globales)

Les variables définies à l'intérieur d'une fonction sont locales à cette fonction et ne peuvent pas être utilisées à l'extérieur. À l'inverse, les variables globales peuvent être accessibles dans toute la durée du programme.

Exemple :

```
x = 10  # Variable globale

def ma_fonction():
    x = 5  # Variable locale
    print("Dans la fonction :", x)

ma_fonction()  # Affiche : 5
print("En dehors de la fonction :", x)  # Affiche : 10
```

2.4 Exercice pratique et Questions/Réponses (30 minutes)

Exercice : Créer une calculatrice simple

Demandez à l'utilisateur d'entrer deux nombres et une opération (addition, soustraction, multiplication, division) et affichez le résultat.

```
def calculatrice(a, b, operation):
    if operation == '+':
        return a + b
    elif operation == '-':
```

```
        return a - b
    elif operation == '*':
        return a * b
    elif operation == '/':
        return a / b
    else:
        return "Opération non valide"

# Utilisation
a = float(input("Entrez le premier nombre : "))
b = float(input("Entrez le deuxième nombre : "))
operation = input("Entrez l'opération (+, -, *, /) : ")
resultat = calculatrice(a, b, operation)
print("Le résultat est :", resultat)
```

Cet exercice vous permettra de manipuler des fonctions, des variables et des opérateurs.

Module 3 : Structures de Contrôle

Durée : 2 heures

Objectifs :

- Comprendre et utiliser les structures de contrôle en Python.
- Maîtriser les conditions avec `if`, `elif`, et `else`.
- Utiliser les boucles `for` et `while` pour répéter des instructions.
- Comprendre les boucles imbriquées et les interruptions avec `break` et `continue`.

3.1 Les Conditions : `if`, `elif`, `else` (40 minutes)

Les **structures conditionnelles** permettent de contrôler le flux d'exécution du programme en fonction de conditions logiques.

Structure de base

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

```
age = 18

if age >= 18:
    print("Vous êtes majeur.")
```

Si la condition est **fausse**, le bloc `else` est exécuté :

```
age = 16

if age >= 18:
    print("Vous êtes majeur.")
else:
    print("Vous êtes mineur.")
```

Utilisation de `elif` pour plusieurs conditions

La clause `elif` (else if) permet de tester plusieurs conditions dans une structure :

```
note = 15

if note >= 16:
    print("Très bien.")
elif note >= 12:
    print("Bien.")
```

```
elif note >= 10:
    print("Passable.")
else:
    print("Insuffisant.")
```

Conditions imbriquées

Vous pouvez imbriquer des conditions pour des cas plus complexes :

```
age = 20
nationalite = "française"

if age >= 18:
    if nationalite == "française":
        print("Vous pouvez voter en France.")
    else:
        print("Vous ne pouvez pas voter en France.")
else:
    print("Vous n'êtes pas encore majeur.")
```

3.2 Les Boucles : `for` et `while` (50 minutes)

Les boucles permettent d'exécuter un bloc de code plusieurs fois. Python propose deux types de boucles principales : **for** et **while**.

La boucle `for`

La boucle `for` est utilisée pour parcourir une séquence (liste, chaîne de caractères, etc.).

```
# Parcourir une liste
fruits = ["pomme", "banane", "cerise"]

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

Utilisation de `range()` dans une boucle `for`

La fonction `range()` génère une séquence de nombres, très utile pour contrôler le nombre d'itérations dans une boucle :

```
for i in range(5):
    print("Itération", i)
```

Cela affichera les valeurs de `i` de 0 à 4.

Vous pouvez aussi spécifier un **début**, une **fin** et un **pas** dans `range()` :

```
for i in range(2, 10, 2):  # De 2 à 10, avec un pas de 2
    print(i)
```

La boucle `while`

La boucle `while` continue tant qu'une condition donnée est vraie :

```
compteur = 0

while compteur < 5:
    print("Compteur :", compteur)
    compteur += 1  # Augmente le compteur de 1
```

La boucle `while` est utile lorsque vous ne connaissez pas à l'avance le nombre d'itérations nécessaires.

Boucles infinies et comment les éviter

Faites attention aux boucles infinies ! Si la condition d'arrêt d'une boucle `while` n'est jamais remplie, la boucle continuera indéfiniment. Par exemple :

```
while True:
    print("Cette boucle ne s'arrête jamais")
```

Cela crée une boucle infinie ; pour l'arrêter, utilisez `Ctrl+C` dans le terminal.

3.3 Contrôler le flux des boucles : `break` et `continue` (30 minutes)

Utilisation de `break`

La commande `break` interrompt immédiatement une boucle, même si la condition n'est pas remplie ou les itérations ne sont pas terminées :

```
for i in range(10):
    if i == 5:
        break  # La boucle se termine dès que i vaut 5
    print(i)
```

Utilisation de `continue`

La commande `continue` saute une itération, mais continue la boucle pour les itérations suivantes :

```
for i in range(5):
    if i == 3:
        continue  # Saute l'affichage de 3
    print(i)
```

3.4 Boucles imbriquées (30 minutes)

Les boucles peuvent être imbriquées, c'est-à-dire qu'une boucle peut être à l'intérieur d'une autre boucle. Par exemple, pour parcourir une matrice (tableau à deux dimensions) :

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

for ligne in matrice:
    for element in ligne:
        print(element, end=" ")
    print()  # Saut de ligne après chaque ligne de la matrice
```

Cet exemple affiche chaque élément de la matrice sur une ligne séparée.

Exemple pratique : Imprimer un triangle d'étoiles

```
n = 5

for i in range(1, n + 1):
    for j in range(i):
        print("*", end="")
    print()  # Saut de ligne à la fin de chaque ligne
```

Résultat pour `n = 5` :

```
markdown

*
**
***
****
*****
```

3.5 Exercice pratique et Questions/Réponses (30 minutes)

Exercice : Jeu de devinettes

Créez un jeu où l'ordinateur génère un nombre aléatoire entre 1 et 20, et l'utilisateur doit deviner ce nombre. Après chaque tentative, le programme doit dire si le nombre est plus petit ou plus grand. Si l'utilisateur devine correctement, le jeu se termine.

```
import random

nombre_mystere = random.randint(1, 20)
essai = 0
devine = False

while not devine:
    nombre = int(input("Devinez le nombre (entre 1 et 20) : "))
    essai += 1
    if nombre == nombre_mystere:
        print(f"Bravo ! Vous avez trouvé en {essai} essai(s).")
        devine = True
    elif nombre < nombre_mystere:
        print("Trop petit.")
    else:
        print("Trop grand.")
```

Module 4 : Structures de Données (Listes, Tuples, Vecteurs, Matrices)

Durée : 2 heures

Objectifs :

- Comprendre et utiliser les principales structures de données en Python.
- Maîtriser la manipulation des listes et tuples.
- Introduction aux vecteurs et matrices avec la bibliothèque NumPy.
- Appliquer des méthodes et des fonctions sur les structures de données.

4.1 Les Listes en Python (40 minutes)

Les **listes** sont l'une des structures de données les plus utilisées en Python. Elles permettent de stocker une collection ordonnée d'éléments, qui peuvent être modifiés après leur création.

Création et manipulation de listes

Une liste est créée en plaçant des éléments entre des crochets `[ ]`, séparés par des virgules :

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

Accéder aux éléments d'une liste

Les éléments d'une liste sont accessibles via leur **index** (position dans la liste, commençant à 0) :

```
print(fruits[0])  # Affiche "pomme"
print(fruits[2])  # Affiche "cerise"
```

Modifier les éléments d'une liste

Les listes sont **mutables**, ce qui signifie que vous pouvez changer leurs éléments après leur création :

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

Méthodes utiles pour les listes

Python propose plusieurs méthodes pour manipuler des listes :

Méthode	Description	Exemple
`append(x)`	Ajoute un élément à la fin de la liste	`fruits.append("mangue")`
`remove(x)`	Supprime le premier élément égal à `x`	`fruits.remove("orange")`
`pop(i)`	Supprime et retourne l'élément à l'index `i`	`fruits.pop(1)`
`sort()`	Trie la liste en place	`fruits.sort()`
`reverse()`	Inverse l'ordre des éléments	`fruits.reverse()`
`len()`	Retourne la longueur de la liste	`len(fruits)`

Parcourir une liste avec une boucle

Il est facile de parcourir une liste avec une boucle `for` :

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

Vous pouvez également utiliser la fonction `enumerate()` pour obtenir à la fois les indices et les éléments :

```
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

4.2 Les Tuples en Python (20 minutes)

Les **tuples** sont similaires aux listes, mais ils sont **immuables**, c'est-à-dire que leurs éléments ne peuvent pas être modifiés une fois créés.

Création de tuples

Un tuple est défini en plaçant les éléments entre des parenthèses `()` :

```
couleurs = ("rouge", "vert", "bleu")
```

Accéder aux éléments d'un tuple

Les éléments d'un tuple sont accessibles de la même manière que pour les listes :

```
print(couleurs[1])  # Affiche "vert"
```

Immuabilité des tuples

Contrairement aux listes, il est impossible de modifier les éléments d’un tuple :

```
# couleurs[1] = "jaune"  # Provoquerait une erreur
```

Cependant, vous pouvez toujours accéder aux éléments et les parcourir avec une boucle `for` :

```
for couleur in couleurs:
    print(couleur)
```

Utilisations courantes des tuples

Les tuples sont souvent utilisés pour :

- Stocker des collections d'éléments qui ne doivent pas être modifiés.
- Retourner plusieurs valeurs à partir d'une fonction.

4.3 Introduction aux Vecteurs et Matrices avec NumPy (60 minutes)

Les **vecteurs** et **matrices** sont des structures de données essentielles pour la manipulation de données numériques, notamment en data science et en calcul scientifique. **NumPy** est une bibliothèque Python qui permet de travailler facilement avec des vecteurs et des matrices.

Installation de NumPy

Si NumPy n'est pas installé, utilisez la commande suivante dans votre terminal pour l'installer :

```
bash

pip install numpy
```

Création d'un vecteur (tableau unidimensionnel)

En NumPy, les vecteurs sont appelés **tableaux** (**ndarray**). Voici comment créer un tableau unidimensionnel :

```
import numpy as np

vecteur = np.array([1, 2, 3, 4, 5])
print(vecteur)
```

Création d'une matrice (tableau multidimensionnel)

Une matrice est un tableau à deux dimensions :

```
matrice = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(matrice)
```

Accéder aux éléments d'un tableau NumPy

Les éléments d'un tableau NumPy sont accessibles de la même manière que pour les listes, en utilisant des indices :

```
print(vecteur[2])  # Affiche 3
print(matrice[1, 2])  # Affiche 6 (ligne 2, colonne 3)
```

Opérations sur les tableaux NumPy

NumPy facilite les opérations mathématiques sur les tableaux :

```
# Opérations élémentaires
vecteur2 = vecteur + 2
print(vecteur2)  # Affiche [3, 4, 5, 6, 7]

# Addition de deux vecteurs
vecteur3 = vecteur + np.array([5, 4, 3, 2, 1])
print(vecteur3)  # Affiche [6, 6, 6, 6, 6]

# Multiplication d'une matrice par un scalaire
matrice2 = matrice * 2
print(matrice2)  # Chaque élément de la matrice est multiplié par 2
```

Méthodes et opérations courantes avec NumPy

Voici quelques fonctions courantes pour manipuler des tableaux NumPy :

Fonction	Description	Exemple
`np.zeros()`	Crée un tableau rempli de zéros	`np.zeros((2, 3))`
`np.ones()`	Crée un tableau rempli de 1	`np.ones((3, 3))`
`np.reshape()`	Change la forme d'un tableau	`vecteur.reshape((5, 1))`
`np.sum()`	Calcule la somme des éléments	`np.sum(matrice)`
`np.mean()`	Calcule la moyenne des éléments	`np.mean(vecteur)`
`np.dot()`	Produit matriciel (dot product)	`np.dot(matrice, matrice)`

4.4 Exercice pratique et Questions/Réponses (30 minutes)

Exercice : Manipuler des listes et des tableaux NumPy

1. **Manipuler une liste** : Créez une liste de vos cinq plats préférés. Modifiez un élément de la liste, puis ajoutez un nouveau plat à la fin. Enfin, triez la liste par ordre alphabétique.

2. **Manipuler un tableau NumPy** : Créez une matrice NumPy de taille 3x3. Multipliez chaque élément par 3.

Exemple :

```
# Liste des plats
plats = ["Pizza", "Burger", "Salade", "Sushi", "Tacos"]
plats[2] = "Pâtes"
plats.append("Ramen")
plats.sort()
print(plats)

# Matrice NumPy
import numpy as np
matrice = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
matrice = matrice * 3
print("Matrice multipliée par 3 :", matrice)
```

Module 5 : Dictionnaires et Compréhension de Liste

Durée : 2 heures

Objectifs :

- Comprendre et utiliser les **dictionnaires** en Python.
- Manipuler des **paires clé-valeur**.
- Introduire la **compréhension de liste** pour une manipulation efficace des données.
- Appliquer des méthodes et des fonctions pratiques aux dictionnaires et aux compréhensions de liste.

5.1 Les Dictionnaires en Python (60 minutes)

Les **dictionnaires** sont des structures de données qui associent des **clés** à des **valeurs**. Contrairement aux listes, qui sont indexées par des nombres, les dictionnaires sont indexés par des clés qui peuvent être de n'importe quel type immuable (chaînes, nombres, etc.).

Création d'un dictionnaire

Un dictionnaire se crée en plaçant les paires clé-valeur entre accolades `{}` et en les séparant par des deux-points `:`.

```
mon_dictionnaire = {
    "nom": "Jean",
    "age": 30,
    "profession": "ingénieur"
}
```

Accéder aux éléments d'un dictionnaire

Pour accéder aux valeurs d'un dictionnaire, il suffit de spécifier la clé correspondante entre crochets `[]` :

```
print(mon_dictionnaire["nom"])  # Affiche "Jean"
```

Ajouter ou modifier des éléments

Les dictionnaires sont **mutables**, donc on peut facilement ajouter ou modifier des paires clé-valeur :

```
mon_dictionnaire["ville"] = "Paris"  # Ajout d'une nouvelle paire
```

```
mon_dictionnaire["age"] = 31  # Modification de la valeur existante
```

Supprimer des éléments

Pour supprimer une paire clé-valeur, on peut utiliser la méthode `pop()` ou la commande `del` :

```
mon_dictionnaire.pop("profession")  # Supprime la clé "profession"
del mon_dictionnaire["ville"]  # Supprime la clé "ville"
```

Méthodes courantes pour manipuler les dictionnaires

Voici quelques méthodes et fonctions utiles pour travailler avec les dictionnaires :

Méthode	Description	Exemple
`keys()`	Retourne une vue contenant toutes les clés	`mon_dictionnaire.keys()`
`values()`	Retourne une vue contenant toutes les valeurs	`mon_dictionnaire.values()`
`items()`	Retourne une vue contenant les paires clé-valeur	`mon_dictionnaire.items()`
`get()`	Retourne la valeur associée à une clé donnée	`mon_dictionnaire.get("nom")`
`update()`	Met à jour le dictionnaire avec d'autres paires	`mon_dictionnaire.update({"age": 32})`

Parcourir un dictionnaire

Vous pouvez parcourir les clés, les valeurs ou les deux en même temps avec une boucle `for` :

```
# Parcourir les clés
for cle in mon_dictionnaire.keys():
    print(cle)

# Parcourir les valeurs
for valeur in mon_dictionnaire.values():
    print(valeur)

# Parcourir les paires clé-valeur
for cle, valeur in mon_dictionnaire.items():
    print(f"{cle} : {valeur}")
```

5.2 La Compréhension de Liste (40 minutes)

La **compréhension de liste** (list comprehension) est une méthode concise pour créer des listes à partir d'itérables tout en appliquant une condition ou une transformation à chaque élément.

Création d'une liste avec compréhension

La syntaxe de la compréhension de liste est :

```
nouvelle_liste = [expression for élément in iterable]
```

Exemple simple : créer une liste contenant les carrés des nombres de 1 à 5 :

```
carrés = [x ** 2 for x in range(1, 6)]
print(carrés)  # Affiche [1, 4, 9, 16, 25]
```

Ajout de conditions

Vous pouvez aussi ajouter une condition à une compréhension de liste :

```
pairs = [x for x in range(10) if x % 2 == 0]
print(pairs)  # Affiche [0, 2, 4, 6, 8]
```

Exemples avancés

1. **Doublez les éléments d'une liste :**

```
liste = [1, 2, 3, 4, 5]
doubles = [x * 2 for x in liste]
print(doubles)  # Affiche [2, 4, 6, 8, 10]
```

2. **Convertir les éléments d'une liste en majuscules :**

```
mots = ["python", "est", "puissant"]
mots_majuscules = [mot.upper() for mot in mots]
print(mots_majuscules)  # Affiche ['PYTHON', 'EST', 'PUISSANT']
```

3. **Filtrer les éléments pairs d'une liste :**

```
nombres = [1, 2, 3, 4, 5, 6]
nombres_pairs = [n for n in nombres if n % 2 == 0]
print(nombres_pairs)  # Affiche [2, 4, 6]
```

4. **Créer une liste à partir d'un dictionnaire :**

```
eleves = {"Alice": 15, "Bob": 18, "Clara": 12}
admis = [nom for nom, age in eleves.items() if age >= 16]
print(admis)  # Affiche ['Bob']
```

5.3 Exercice pratique et Questions/Réponses (20 minutes)

Exercice : Manipuler des dictionnaires et utiliser la compréhension de liste

1. **Exercice sur les dictionnaires :** Créez un dictionnaire contenant le nom, l'âge et la profession de trois personnes. Ajoutez un nouvel attribut (comme la ville) à chaque personne, puis modifiez l'un des âges. Parcourez ensuite le dictionnaire pour afficher les informations de chaque personne.
2. **Exercice sur la compréhension de liste :** Créez une liste contenant les nombres de 1 à 20, puis générez une nouvelle liste avec les carrés des nombres pairs uniquement, en utilisant la compréhension de liste.

Exemple :

```
# Dictionnaire
personnes = {
    "Alice": {"age": 25, "profession": "docteur"},
    "Bob": {"age": 30, "profession": "ingénieur"},
    "Charlie": {"age": 22, "profession": "designer"}
}

# Ajout de la ville
for nom in personnes:
    personnes[nom]["ville"] = "Paris"

# Modification de l'âge de Bob
personnes["Bob"]["age"] = 31

# Affichage des informations
for nom, infos in personnes.items():
    print(f"{nom} : {infos}")

# Compréhension de liste pour les carrés des nombres pairs
nombres = [x for x in range(1, 21)]
carres_pairs = [x ** 2 for x in nombres if x % 2 == 0]
print(carres_pairs)  # Affiche les carrés des nombres pairs
```

Module 6 : Fonctions de Base et Modules de Base

Durée : 2 heures

Objectifs :

- Comprendre la définition et l'utilisation des **fonctions** en Python.
- Savoir comment passer des arguments et obtenir des résultats à partir d'une fonction.
- Découvrir les **modules** standards et comment les importer dans un programme Python.
- Maîtriser quelques modules de base essentiels pour les projets Python.

6.1 Introduction aux Fonctions en Python (60 minutes)

Une **fonction** est un bloc de code réutilisable conçu pour accomplir une tâche spécifique. Elle permet de structurer un programme en morceaux modulaires et d'éviter la redondance de code.

Définition d'une fonction

Les fonctions en Python sont définies à l'aide du mot-clé `def`, suivi du nom de la fonction et d'une liste de paramètres entre parenthèses. Le corps de la fonction est ensuite indéxé par une indentation.

```
def saluer():
    print("Bonjour tout le monde!")
```

Appel d'une fonction

Après avoir défini une fonction, vous pouvez l'appeler simplement en utilisant son nom suivi de parenthèses.

```
saluer()  # Affiche "Bonjour tout le monde!"
```

Fonctions avec paramètres

Les fonctions peuvent également prendre des **paramètres** pour rendre leur comportement plus flexible.

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

```
saluer_nom("Alice")  # Affiche "Bonjour Alice!"
```

Retourner une valeur

Une fonction peut renvoyer une valeur avec l'instruction `return` :

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


resultat = additionner(3, 5)
print(resultat)  # Affiche 8
```

Arguments par défaut

Vous pouvez définir des **valeurs par défaut** pour les paramètres :

```
def saluer_nom(nom="inconnu"):
    print(f"Bonjour {nom}!")


saluer_nom()  # Affiche "Bonjour inconnu!"
saluer_nom("Marie")  # Affiche "Bonjour Marie!"
```

Passage d'arguments : positionnels et nommés

Python permet de passer des arguments de deux manières : **positionnels** (dans l'ordre où ils apparaissent) ou **nommés** (en spécifiant explicitement la valeur pour chaque paramètre).

```
def presentation(nom, age):
    print(f"Je m'appelle {nom} et j'ai {age} ans.")

# Arguments positionnels
presentation("Jean", 30)

# Arguments nommés
presentation(age=25, nom="Luc")
```

Fonctions Lambda (Fonctions Anonymes)

Les **fonctions lambda** sont des fonctions anonymes définies en une seule ligne. Elles sont utiles pour des opérations simples.

```
addition = lambda x, y: x + y
print(addition(2, 3))  # Affiche 5
```

6.2 Modules en Python (40 minutes)

Un **module** en Python est un fichier qui contient des fonctions, classes ou variables que vous pouvez utiliser dans d'autres programmes. Les modules permettent d'organiser le code et de le rendre plus modulaire et réutilisable.

Importer un module

Pour utiliser un module, il faut l'importer dans votre script avec l'instruction `import` :

```
import math

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

Importer des fonctions spécifiques d'un module

Vous pouvez également importer une fonction spécifique à partir d'un module :

```
from math import sqrt

print(sqrt(25))  # Affiche 5.0
```

Modules intégrés dans Python

Python propose de nombreux **modules standards** qui sont directement disponibles après installation. Voici quelques-uns des plus courants :

Module	Description	Exemple d'utilisation
`math`	Fonctions mathématiques avancées	`math.sqrt(9)`
`random`	Génération de nombres aléatoires	`random.randint(1, 10)`
`datetime`	Manipulation de dates et heures	`datetime.now()`
`os`	Interactions avec le système d'exploitation	`os.listdir()`
`sys`	Accès aux fonctions liées à l'interpréteur Python	`sys.argv`

Créer et importer son propre module

Vous pouvez également créer vos propres modules en sauvegardant des fonctions dans un fichier `.py` et en les important dans d'autres scripts.

Par exemple, supposons que vous ayez un fichier appelé `utilitaires.py` :

```
# Contenu de utilitaires.py
def bonjour():
    print("Salut du module utilitaires!")
```

Dans un autre script, vous pouvez l'importer ainsi :

```
import utilitaires

utilitaires.bonjour()  # Affiche "Salut du module utilitaires!"
```

6.3 Exercice pratique et Questions/Réponses (20 minutes)

Exercice : Créer des fonctions et importer des modules

1. **Exercice sur les fonctions :** Créez une fonction qui prend deux paramètres et renvoie leur produit. Ajoutez un argument par défaut pour l'un des paramètres. Ensuite, appelez cette fonction avec différents arguments et affichez les résultats.
2. **Exercice sur les modules :** Utilisez le module `random` pour générer cinq nombres aléatoires entre 1 et 50. Ensuite, utilisez le module `datetime` pour afficher la date et l'heure actuelles.

Exemple :

```
# Fonction avec paramètre par défaut
def multiplier(a, b=2):
    return a * b

print(multiplier(5))  # Affiche 10
print(multiplier(3, 4))  # Affiche 12

# Utilisation de modules
import random
import datetime

# Génération de nombres aléatoires
for _ in range(5):
    print(random.randint(1, 50))

# Affichage de la date et heure actuelles
print(datetime.datetime.now())
```

Module 7 : Numpy (Les Bases)

Durée : 2 heures

Objectifs :

- Comprendre les bases de la bibliothèque **NumPy**.
- Manipuler des tableaux multidimensionnels (**ndarray**).
- Réaliser des opérations arithmétiques avec des tableaux NumPy.
- Saisir les avantages de NumPy pour les calculs numériques rapides et efficaces en Python.

7.1 Introduction à NumPy (30 minutes)

NumPy est l'une des bibliothèques les plus utilisées pour le calcul numérique en Python. Elle est particulièrement performante pour les manipulations de **tableaux** et les **opérations mathématiques** sur de grandes quantités de données. NumPy offre la possibilité de travailler avec des **tableaux multidimensionnels** appelés **ndarray**.

Installation de NumPy

Si NumPy n'est pas déjà installé dans votre environnement, vous pouvez l'installer avec la commande suivante :

```
bash

pip install numpy
```

Importer NumPy

La convention standard pour importer NumPy est d'utiliser l'alias `np` :

```
import numpy as np
```

7.2 Les Tableaux NumPy (60 minutes)

Un **tableau NumPy** est une structure de données similaire à une liste en Python, mais beaucoup plus performante pour le traitement d'éléments numériques.

Création de tableaux

Vous pouvez créer un tableau NumPy à partir d'une liste Python, ou bien utiliser des fonctions intégrées de NumPy pour générer des tableaux.

Créer un tableau à partir d'une liste :

```
import numpy as np

mon_tableau = np.array([1, 2, 3, 4, 5])
print(mon_tableau)
# Affiche : [1 2 3 4 5]
```

Créer un tableau multidimensionnel :

Un tableau NumPy peut également être **multidimensionnel** :

```
tableau_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(tableau_2d)
# Affiche :
# [[1 2 3]
#  [4 5 6]]
```

Fonctions utiles pour créer des tableaux :

- **`np.zeros(shape)`** : crée un tableau rempli de zéros.

```
zeros = np.zeros((3, 3))
print(zeros)
# Affiche :
# [[0. 0. 0.]
#  [0. 0. 0.]
#  [0. 0. 0.]]
```

- **`np.ones(shape)`** : crée un tableau rempli de uns.

```
ones = np.ones((2, 4))
print(ones)
# Affiche :
# [[1. 1. 1. 1.]
#  [1. 1. 1. 1.]]
```

- **`np.arange(start, stop, step)`** : génère un tableau contenant une séquence d'éléments à intervalles réguliers.

```
sequence = np.arange(0, 10, 2)
print(sequence)
# Affiche : [0 2 4 6 8]
```

- **`np.linspace(start, stop, num)`** : génère un tableau contenant un certain nombre d'éléments régulièrement espacés entre deux valeurs.

```
lin_space = np.linspace(0, 1, 5)
print(lin_space)
# Affiche : [0.   0.25 0.5  0.75 1.  ]
```

7.3 Propriétés des Tableaux NumPy (30 minutes)

Les tableaux NumPy possèdent plusieurs **attributs** utiles pour en comprendre la structure et les manipuler efficacement.

La dimension d'un tableau :

L'attribut `ndim` permet de connaître le nombre de dimensions d'un tableau.

```
tableau = np.array([[1, 2, 3], [4, 5, 6]])
print(tableau.ndim)  # Affiche 2
```

La forme d'un tableau :

L'attribut `shape` donne la forme du tableau (le nombre de lignes et de colonnes).

```
print(tableau.shape)  # Affiche (2, 3)
```

Le type des éléments :

L'attribut `dtype` permet de vérifier le type des éléments du tableau.

```
print(tableau.dtype)  # Affiche int64 ou int32, selon l'architecture
```

Changer la forme d'un tableau :

La fonction `reshape()` permet de redimensionner un tableau sans en changer les données.

```
tableau_reshaped = tableau.reshape(3, 2)
print(tableau_reshaped)
# Affiche :
# [[1 2]
#  [3 4]
#  [5 6]]
```

Opérations mathématiques sur les tableaux :

Les tableaux NumPy permettent de réaliser des **opérations élémentaires** directement sur tous les éléments.

```
tableau = np.array([1, 2, 3, 4])
print(tableau * 2)
# Affiche : [2 4 6 8]

# Addition de deux tableaux
tableau2 = np.array([10, 20, 30, 40])
print(tableau + tableau2)
# Affiche : [11 22 33 44]
```

7.4 Exercice pratique et Questions/Réponses (30 minutes)

Exercice : Manipuler des tableaux NumPy

1. **Créer des tableaux :** Créez un tableau contenant les nombres de 1 à 9. Redimensionnez-le ensuite en un tableau 3x3.
2. **Opérations sur les tableaux :** Multipliez un tableau par une constante, puis additionnez deux tableaux de même forme.
3. **Opérations avancées :** Utilisez `np.arange()` pour générer une séquence de nombres de 0 à 20, puis extraire uniquement les éléments pairs de cette séquence.

Exemple :

```
import numpy as np

# Créer un tableau et redimensionner
tableau = np.arange(1, 10)
tableau_reshaped = tableau.reshape(3, 3)
print(tableau_reshaped)

# Opérations sur les tableaux
tableau2 = np.array([1, 2, 3])
print(tableau2 * 3)  # Multiplie par 3

# Générer une séquence et extraire les éléments pairs
sequence = np.arange(0, 21)
pairs = sequence[sequence % 2 == 0]
print(pairs)
```

Module 8 : Numpy (Slicing et Indexing)

Durée : 2 heures

Objectifs :

- Comprendre et maîtriser le **slicing** (découpage) dans les tableaux NumPy.
- Manipuler les **indices** pour accéder et modifier des éléments dans un tableau.
- Apprendre à utiliser les techniques avancées de **slicing** et **indexing** pour travailler efficacement avec des tableaux multidimensionnels.

8.1 Introduction au Slicing et Indexing (20 minutes)

Le **slicing** et l'**indexing** sont des techniques utilisées pour extraire ou modifier des sous-parties d'un tableau NumPy. Elles permettent de sélectionner des **éléments spécifiques** dans un tableau de manière efficace, sans créer de copies inutiles.

Accéder aux éléments d'un tableau

Pour accéder à un élément spécifique dans un tableau NumPy, vous utilisez la syntaxe d'**indexation** avec des crochets `[]`. Les indices commencent à partir de **0** en Python.

Exemple pour un tableau unidimensionnel :

```
import numpy as np

tableau = np.array([10, 20, 30, 40, 50])
print(tableau[2])  # Affiche 30
```

Exemple pour un tableau multidimensionnel :

Pour un tableau à deux dimensions (2D), l'indexation se fait via une paire d'indices.

```
tableau_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(tableau_2d[1, 2])  # Affiche 6 (élément ligne 1, colonne 2)
```

8.2 Slicing dans les Tableaux NumPy (50 minutes)

Le **slicing** permet d'extraire des sous-ensembles d'un tableau NumPy à l'aide de la syntaxe `start:stop:step`.

Slicing dans un tableau 1D

Vous pouvez extraire une partie du tableau en spécifiant les **indices de début** et de **fin** :

```
tableau = np.array([10, 20, 30, 40, 50, 60])
sous_tableau = tableau[1:4]  # Prend les éléments aux indices 1, 2 et 3
print(sous_tableau)  # Affiche [20 30 40]
```

- **Indice de début :** inclusif
- **Indice de fin :** exclusif
- **Pas (step) :** permet de sauter des éléments

```
sous_tableau_step = tableau[0:6:2]  # Prend un élément sur deux
print(sous_tableau_step)  # Affiche [10 30 50]
```

Slicing dans un tableau 2D

Avec les tableaux multidimensionnels, vous pouvez effectuer un slicing sur plusieurs axes (lignes et colonnes) :

```
tableau_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Extraire les 2 premières lignes et les colonnes 1 à 3
sous_tableau_2d = tableau_2d[0:2, 1:3]
print(sous_tableau_2d)
# Affiche :
# [[2 3]
#  [5 6]]
```

Slicing avec un pas

Vous pouvez également utiliser un **pas** pour extraire des éléments à intervalles réguliers dans une matrice.

```
tableau_2d = np.array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13,
14, 15]])

# Prendre une ligne sur deux et une colonne sur deux
sous_tableau_step = tableau_2d[::2, ::2]
print(sous_tableau_step)
# Affiche :
# [[ 0  2]
#  [ 8 10]]
```

Slicing inversé

Vous pouvez inverser un tableau ou une partie de celui-ci en utilisant un **pas négatif** :

```
tableau_inverse = tableau[::-1]
print(tableau_inverse)  # Affiche [60 50 40 30 20 10]
```

8.3 Indexation Avancée dans NumPy (30 minutes)

En plus de l'indexation simple, NumPy permet d'utiliser des **listes d'indices**, des **tableaux booléens**, et des **conditions** pour sélectionner ou modifier des éléments.

Indexation par listes d'indices

Vous pouvez sélectionner des éléments spécifiques d'un tableau à l'aide d'une liste d'indices.

```
tableau = np.array([10, 20, 30, 40, 50])
indices = [0, 2, 4]
print(tableau[indices])  # Affiche [10 30 50]
```

Indexation booléenne

L'indexation booléenne permet de sélectionner des éléments selon une condition logique.

```
tableau = np.array([10, 20, 30, 40, 50])
condition = tableau > 30
print(tableau[condition])  # Affiche [40 50]
```

Vous pouvez également utiliser cette technique pour modifier certains éléments en fonction d'une condition.

```
tableau[tableau < 30] = 0
print(tableau)  # Affiche [ 0  0 30 40 50]
```

Indexation par condition multiple

Vous pouvez combiner plusieurs conditions pour filtrer un tableau.

```
tableau = np.array([10, 20, 30, 40, 50])
resultat = tableau[(tableau > 20) & (tableau < 50)]
print(resultat)  # Affiche [30 40]
```

8.4 Exercice pratique et Questions/Réponses (20 minutes)

Exercice : Slicing et Indexation

1. **Manipulation de slicing** : Créez un tableau contenant les nombres de 0 à 99. Utilisez le slicing pour extraire les éléments entre les indices 20 et 40, avec un pas de 2.

2. **Slicing sur un tableau 2D** : Créez un tableau 2D de dimension 5x5 contenant les nombres de 1 à 25. Extrayez les éléments situés dans les 3 premières lignes et les colonnes 2 à 4.
3. **Indexation avancée** : Utilisez un tableau de nombres pour créer un sous-tableau contenant uniquement les nombres pairs supérieurs à 10 et inférieurs à 50.

Exemple :

```
import numpy as np

# Slicing sur un tableau 1D
tableau = np.arange(100)
sous_tableau = tableau[20:40:2]
print(sous_tableau)  # Affiche [20 22 24 26 28 30 32 34 36 38]

# Slicing sur un tableau 2D
tableau_2d = np.arange(1, 26).reshape(5, 5)
sous_tableau_2d = tableau_2d[0:3, 1:4]
print(sous_tableau_2d)
# Affiche :
# [[ 2  3  4]
#  [ 7  8  9]
#  [12 13 14]]

# Indexation booléenne
tableau = np.array([5, 12, 19, 32, 47, 58])
sous_tableau_pairs = tableau[(tableau % 2 == 0) & (tableau > 10) & (tableau
< 50)]
print(sous_tableau_pairs)  # Affiche [12 32]
```

Module 9 : Numpy (Mathématiques et Statistiques)

Durée : 2 heures

Objectifs :

- Apprendre à utiliser les fonctions mathématiques et statistiques offertes par **NumPy**.
- Calculer des statistiques descriptives sur des tableaux de données.
- Manipuler et appliquer des fonctions mathématiques pour des opérations élémentaires et avancées.

9.1 Introduction aux Fonctions Mathématiques dans NumPy (30 minutes)

NumPy fournit une large gamme de fonctions mathématiques qui permettent d'effectuer des opérations sur des tableaux. Ces fonctions sont optimisées pour la **vitesse** et l'**efficacité** dans la manipulation de données numériques.

Opérations élémentaires

Les opérations élémentaires dans NumPy peuvent être appliquées à un tableau entier ou à des éléments spécifiques.

Addition, soustraction, multiplication et division :

```
import numpy as np

tableau = np.array([10, 20, 30, 40])

# Addition
print(tableau + 5)  # Affiche [15 25 35 45]

# Multiplication
print(tableau * 2)  # Affiche [20 40 60 80]

# Division
print(tableau / 10)  # Affiche [1. 2. 3. 4.]
```

Fonctions mathématiques universelles (ufuncs)

NumPy offre des **ufuncs**, des fonctions universelles, pour appliquer des opérations mathématiques élémentaires et avancées sur des tableaux.

- **`np.sqrt(x)`** : Calcul de la racine carrée.
- **`np.exp(x)`** : Calcul de l'exponentielle.
- **`np.log(x)`** : Calcul du logarithme naturel.

```
tableau = np.array([1, 4, 9, 16])

racine_carree = np.sqrt(tableau)
print(racine_carree)  # Affiche [1. 2. 3. 4.]

logarithme = np.log(np.array([1, np.e, np.e**2]))
print(logarithme)  # Affiche [0. 1. 2.]
```

9.2 Opérations Mathématiques Avancées (40 minutes)

Les opérations mathématiques avancées sont essentielles pour effectuer des calculs scientifiques ou des analyses de données plus poussées.

Produit scalaire et produit matriciel

NumPy permet de réaliser des produits scalaires et des produits de matrices.

Produit scalaire :

Le produit scalaire de deux vecteurs se calcule en multipliant les éléments correspondants des deux vecteurs, puis en additionnant les résultats.

```
vecteur_1 = np.array([1, 2, 3])
vecteur_2 = np.array([4, 5, 6])

produit_scalaire = np.dot(vecteur_1, vecteur_2)
print(produit_scalaire)  # Affiche 32
```

Produit matriciel :

Le produit matriciel se calcule entre deux matrices et respecte les règles de l'algèbre linéaire.

```
matrice_1 = np.array([[1, 2], [3, 4]])
matrice_2 = np.array([[5, 6], [7, 8]])

produit_matriciel = np.dot(matrice_1, matrice_2)
print(produit_matriciel)
# Affiche :
# [[19 22]
#  [43 50]]
```

Fonctions trigonométriques

NumPy dispose de fonctions pour réaliser des opérations sur les angles :

- **`np.sin(x)`** : Calcul du sinus.
- **`np.cos(x)`** : Calcul du cosinus.
- **`np.tan(x)`** : Calcul de la tangente.

```
angles = np.array([0, np.pi/2, np.pi])

sinus = np.sin(angles)
print(sinus)  # Affiche [0. 1. 0.]
```

9.3 Statistiques de Base avec NumPy (40 minutes)

NumPy propose des fonctions puissantes pour calculer des **statistiques descriptives** sur des ensembles de données.

Moyenne, Médiane, Variance et Écart-type

- **`np.mean(x)`** : Moyenne des éléments d'un tableau.
- **`np.median(x)`** : Médiane des éléments d'un tableau.
- **`np.var(x)`** : Variance.
- **`np.std(x)`** : Écart-type.

```
tableau = np.array([1, 2, 3, 4, 5])

# Moyenne
moyenne = np.mean(tableau)
print(moyenne)  # Affiche 3.0

# Médiane
mediane = np.median(tableau)
print(mediane)  # Affiche 3.0

# Variance
variance = np.var(tableau)
print(variance)  # Affiche 2.0

# Écart-type
ecart_type = np.std(tableau)
print(ecart_type)  # Affiche 1.414
```

Somme et Produit des éléments

- **`np.sum(x)`** : Somme des éléments.
- **`np.prod(x)`** : Produit des éléments.

```
tableau = np.array([1, 2, 3, 4])

somme = np.sum(tableau)
print(somme)  # Affiche 10

produit = np.prod(tableau)
print(produit)  # Affiche 24
```

Min, Max et Écart entre Min et Max

- **`np.min(x)`** : Minimum des éléments.
- **`np.max(x)`** : Maximum des éléments.
- **`np.ptp(x)`** : Écart entre le minimum et le maximum (peak-to-peak).

```
tableau = np.array([1, 3, 7, 2, 9])

minimum = np.min(tableau)
print(minimum)  # Affiche 1

maximum = np.max(tableau)
print(maximum)  # Affiche 9

ecart = np.ptp(tableau)
print(ecart)  # Affiche 8
```

9.4 Exercice pratique et Questions/Réponses (30 minutes)

Exercice : Opérations Mathématiques et Statistiques

1. **Calcul de statistiques** : Créez un tableau contenant 10 valeurs aléatoires entre 1 et 100. Calculez la moyenne, la médiane, l'écart-type et la variance de ce tableau.
2. **Produit matriciel et produit scalaire** : Créez deux matrices 2x2 et calculez leur produit matriciel. Ensuite, créez deux vecteurs et calculez leur produit scalaire.
3. **Fonctions mathématiques avancées** : Créez un tableau de valeurs entre 0 et 2π et calculez le sinus, le cosinus et la tangente de chaque valeur.

Exemple :

```
import numpy as np

# 1. Statistiques descriptives
tableau = np.random.randint(1, 101, size=10)
print("Tableau :", tableau)

moyenne = np.mean(tableau)
mediane = np.median(tableau)
ecart_type = np.std(tableau)
variance = np.var(tableau)

print(f"Moyenne : {moyenne}, Médiane : {mediane}, Écart-type :
{ecart_type}, Variance : {variance}")

# 2. Produit matriciel
matrice_1 = np.array([[1, 2], [3, 4]])
matrice_2 = np.array([[5, 6], [7, 8]])

produit_matriciel = np.dot(matrice_1, matrice_2)
print("Produit Matriciel :", produit_matriciel)

# 3. Calcul trigonométrique
angles = np.linspace(0, 2 * np.pi, 5)
sinus = np.sin(angles)
```

```
cosinus = np.cos(angles)
tangente = np.tan(angles)

print("Sinus :", sinus)
print("Cosinus :", cosinus)
print("Tangente :", tangente)
```

Module 10 : Numpy (Broadcasting)

Durée : 2 heures

Objectifs :

- Comprendre le concept de **broadcasting** dans NumPy.
- Apprendre à manipuler des tableaux de différentes formes et dimensions.
- Utiliser le broadcasting pour effectuer des **opérations arithmétiques** sans avoir besoin de boucles explicites.
- Gérer les cas où le broadcasting n'est pas possible et comment contourner ces limitations.

10.1 Introduction au Broadcasting (30 minutes)

Le **broadcasting** est une fonctionnalité puissante de NumPy qui permet d'appliquer des opérations arithmétiques sur des tableaux de tailles différentes sans avoir besoin de les redimensionner manuellement. Il optimise les calculs en éliminant les boucles explicites et améliore les performances en travaillant directement sur des structures de données vectorielles.

Définition

Le broadcasting permet à NumPy d'étendre automatiquement la plus petite dimension d'un tableau pour qu'il corresponde à la taille de l'autre tableau, ce qui permet de réaliser des opérations sans copier inutilement les données.

Principe du Broadcasting

L'idée principale du broadcasting est que si les formes de deux tableaux sont compatibles, NumPy ajuste automatiquement la forme des tableaux pour permettre les opérations élémentaires (addition, multiplication, etc.).

```
import numpy as np

tableau_1 = np.array([1, 2, 3])
tableau_2 = np.array([[10], [20], [30]])

# Broadcasting entre un tableau 1D et un tableau 2D
resultat = tableau_1 + tableau_2
print(resultat)
# Affiche :
# [[11 12 13]
#  [21 22 23]
#  [31 32 33]]
```

Conditions pour le Broadcasting

Deux tableaux peuvent être "broadcastés" si :

1. Leurs formes sont **compatibles** à partir de la droite (lorsqu'on aligne les dimensions à partir de la droite).
2. L'une des dimensions est égale à **1**, ou bien les deux dimensions sont égales.

Exemple de formes compatibles :

- Un tableau de forme `(3, 1)` peut être broadcasté avec un tableau de forme `(1, 4)`.

10.2 Opérations avec Broadcasting (40 minutes)

Le **broadcasting** est couramment utilisé pour effectuer des opérations arithmétiques sur des tableaux de tailles différentes, comme l'addition, la soustraction, la multiplication, etc.

Addition et Soustraction avec Broadcasting

Lorsque vous additionnez ou soustrayez des tableaux de formes compatibles, NumPy ajuste la plus petite dimension pour permettre l'opération.

```
tableau_a = np.array([1, 2, 3])
tableau_b = np.array([[10], [20], [30]])

# Broadcasting d'addition
resultat = tableau_a + tableau_b
print(resultat)
# Affiche :
# [[11 12 13]
#  [21 22 23]
#  [31 32 33]]

# Broadcasting de soustraction
resultat = tableau_b - tableau_a
print(resultat)
# Affiche :
# [[ 9  8  7]
#  [19 18 17]
#  [29 28 27]]
```

Multiplication et Division avec Broadcasting

Le broadcasting s'applique également aux opérations de multiplication et de division. Vous pouvez multiplier ou diviser des tableaux de tailles compatibles sans devoir ajuster leurs formes manuellement.

```
tableau_a = np.array([1, 2, 3])
```

```
tableau_b = np.array([[2], [4], [6]])

# Broadcasting de multiplication
resultat = tableau_a * tableau_b
print(resultat)
# Affiche :
# [[ 2  4  6]
#  [ 4  8 12]
#  [ 6 12 18]]

# Broadcasting de division
resultat = tableau_b / tableau_a
print(resultat)
# Affiche :
# [[2.  1.  0.66666667]
#  [4.  2.  1.33333333]
#  [6.  3.  2.        ]]
```

Différences de formes : le Cas des Scalars

NumPy permet également de broadcast des **scalars** (valeurs uniques) sur des tableaux. Par exemple, si vous voulez ajouter une constante à chaque élément d'un tableau, NumPy applique automatiquement le broadcasting.

```
tableau = np.array([10, 20, 30])

# Broadcasting avec un scalaire
resultat = tableau + 5
print(resultat)
# Affiche [15 25 35]
```

Fonctions Universelles (ufuncs) avec Broadcasting

Les **ufuncs** de NumPy, comme `np.sin()`, `np.exp()`, etc., appliquent également le broadcasting de manière automatique sur des tableaux de formes différentes.

```
angles = np.array([0, np.pi/2, np.pi])

# Broadcasting de la fonction trigonométrique np.sin
sinus = np.sin(angles)
print(sinus)
# Affiche [0. 1. 0.]
```

10.3 Limitations et Erreurs de Broadcasting (20 minutes)

Bien que le broadcasting soit extrêmement utile, il présente certaines **limites**. Par exemple, deux tableaux ne peuvent pas être broadcastés si leurs formes ne sont pas compatibles selon les règles définies précédemment.

Cas d'erreur de Broadcasting

Si vous essayez de broadcast deux tableaux avec des dimensions incompatibles, NumPy renverra une erreur.

```
tableau_a = np.array([1, 2, 3])
tableau_b = np.array([4, 5])

# Tentative de broadcasting de formes incompatibles
# Cette opération génère une erreur
resultat = tableau_a + tableau_b
```

Dans cet exemple, les formes des tableaux `(3,)` et `(2,)` ne sont pas compatibles, donc une erreur sera levée.

Contournement des Limites de Broadcasting

Lorsque le broadcasting n'est pas possible directement, il est possible d'utiliser des méthodes comme **`np.reshape()`** ou **`np.expand_dims()`** pour ajuster manuellement les dimensions des tableaux.

```
tableau_a = np.array([1, 2, 3])
tableau_b = np.array([4, 5])

# Utilisation de reshape pour rendre les formes compatibles
tableau_a_reshaped = tableau_a.reshape(3, 1)
resultat = tableau_a_reshaped + tableau_b
print(resultat)
# Affiche :
# [[5 6]
#  [6 7]
#  [7 8]]
```

10.4 Exercice pratique et Questions/Réponses (30 minutes)

Exercice : Utilisation du Broadcasting

1. **Opérations élémentaires avec Broadcasting** : Créez un tableau 1D de 5 éléments et un tableau 2D de dimension (3, 1). Appliquez une addition, une soustraction, une multiplication et une division avec broadcasting entre les deux tableaux.
2. **Broadcasting avec un scalaire** : Créez un tableau 2D de dimension (4, 4). Ajoutez une constante de 10 à chaque élément du tableau en utilisant le broadcasting.
3. **Reshape et Broadcasting** : Créez deux tableaux de dimensions différentes qui ne sont pas broadcastables directement. Utilisez `reshape()` pour ajuster les formes et permettre le broadcasting.

Exemple :

```
import numpy as np

# 1. Opérations avec Broadcasting
tableau_1d = np.array([1, 2, 3, 4, 5])
```

```
tableau_2d = np.array([[10], [20], [30]])

# Addition
resultat_add = tableau_1d + tableau_2d
print(resultat_add)

# 2. Broadcasting avec un scalaire
tableau_2d = np.arange(16).reshape(4, 4)
resultat = tableau_2d + 10
print(resultat)

# 3. Broadcasting après reshape
tableau_a = np.array([1, 2, 3])
tableau_b = np.array([4, 5])

# Utilisation de reshape
tableau_a_reshaped = tableau_a.reshape(3, 1)
resultat = tableau_a_reshaped + tableau_b
print(resultat)
```

Module 11 : Numpy (Tableau ndarray)

Durée : 2 heures

Objectifs :

- Comprendre la structure et le fonctionnement des **tableaux multidimensionnels (ndarray)** dans NumPy.
- Apprendre à créer, manipuler et transformer des tableaux **ndarray**.
- Explorer les propriétés des tableaux **ndarray** : dimensions, formes, types de données, etc.
- Appliquer des opérations avancées sur les tableaux **ndarray**.

11.1 Introduction aux Tableaux Multidimensionnels (30 minutes)

Un **ndarray** (N-dimensional array) est l'objet de base utilisé dans NumPy pour stocker et manipuler des données de manière efficace. Contrairement aux listes Python classiques, les tableaux **ndarray** sont :

- **Homogènes** : tous les éléments du tableau doivent être du même type.
- **Multidimensionnels** : ils peuvent contenir des données de plusieurs dimensions (1D, 2D, 3D, etc.).

Caractéristiques des Tableaux ndarray

1. **Shape (forme)** : C'est le nombre d'éléments dans chaque dimension du tableau. La forme d'un tableau indique sa structure (exemple : un tableau 2D de taille `(3, 4)` possède 3 lignes et 4 colonnes).
2. **Dimensions (ndim)** : Le nombre total de dimensions du tableau.
3. **Dtype (type de données)** : Le type des éléments dans le tableau (par exemple, `int`, `float`, etc.).
4. **Size** : Le nombre total d'éléments dans le tableau.

Exemple de création d'un tableau **ndarray** :

```
import numpy as np

# Création d'un tableau 2D (3x3)
tableau = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print("Tableau :\n", tableau)
print("Dimensions :", tableau.ndim)
print("Forme :", tableau.shape)
print("Taille totale :", tableau.size)
print("Type des données :", tableau.dtype)
```

11.2 Création de Tableaux ndarray (40 minutes)

NumPy offre plusieurs méthodes pour créer des tableaux **ndarray** de différentes dimensions.

Tableaux à partir de listes

La manière la plus simple de créer un tableau **ndarray** est d'utiliser une liste ou une liste imbriquée (pour des tableaux multidimensionnels).

```
# Tableau 1D à partir d'une liste
tableau_1d = np.array([1, 2, 3, 4])

# Tableau 2D à partir d'une liste imbriquée
tableau_2d = np.array([[1, 2], [3, 4], [5, 6]])
```

Méthodes de création rapide

1. **`np.zeros(shape)`** : Crée un tableau rempli de zéros.
2. **`np.ones(shape)`** : Crée un tableau rempli de 1.
3. **`np.full(shape, value)`** : Crée un tableau rempli d'une valeur donnée.
4. **`np.eye(n)`** : Crée une matrice identité de taille `n x n`.
5. **`np.random.rand(shape)`** : Crée un tableau avec des valeurs aléatoires uniformes entre 0 et 1.

Exemple :

```
# Tableau 3x3 de zéros
zeros = np.zeros((3, 3))

# Tableau 2x2 de 1
ones = np.ones((2, 2))

# Matrice identité 4x4
identity = np.eye(4)

print(zeros)
print(ones)
print(identity)
```

Création de Tableaux avec des Séquences Numériques

NumPy propose des fonctions pour créer des tableaux contenant des séquences de nombres de manière automatique.

- **`np.arange(start, stop, step)`** : Crée une séquence numérique avec un intervalle régulier.
- **`np.linspace(start, stop, num)`** : Génère une séquence de nombres espacés de manière égale entre deux valeurs données.

```
python

# Séquence de nombres de 0 à 9
sequence = np.arange(0, 10)

# Séquence de 5 nombres entre 0 et 1
espace = np.linspace(0, 1, 5)
```

```
print(sequence)  # Affiche [0 1 2 3 4 5 6 7 8 9]
print(espace)    # Affiche [0.  0.25 0.5  0.75 1. ]
```

11.3 Manipulation et Transformation des Tableaux ndarray (40 minutes)

Les tableaux NumPy sont flexibles et peuvent être transformés ou manipulés selon les besoins.

Redimensionnement des Tableaux

Les tableaux peuvent être redimensionnés avec **`np.reshape()`** pour ajuster leur forme.

```
tableau = np.arange(1, 13)  # Crée un tableau 1D de 12 éléments
reshaped_tableau = tableau.reshape(3, 4)  # Redimensionne en un tableau 2D
de 3x4

print(reshaped_tableau)
```

Transposition de Tableaux

La transposition est utile pour inverser les axes d'un tableau (par exemple, passer d'un tableau ligne à un tableau colonne).

- **`np.transpose()`** ou **`ndarray.T`** : Permet de transposer les dimensions d'un tableau.

```
# Création d'un tableau 2D
tableau = np.array([[1, 2], [3, 4], [5, 6]])

# Transposition
transpose = tableau.T

print(transpose)
```

Empilement de Tableaux

NumPy permet d'empiler des tableaux selon différentes dimensions.

- **`np.vstack()`** : Empile des tableaux verticalement.
- **`np.hstack()`** : Empile des tableaux horizontalement.

```
tableau_1 = np.array([1, 2, 3])
tableau_2 = np.array([4, 5, 6])

# Empilement vertical
vstacked = np.vstack((tableau_1, tableau_2))

# Empilement horizontal
hstacked = np.hstack((tableau_1, tableau_2))

print(vstacked)
print(hstacked)
```

Concaténation et Fractionnement

Les tableaux peuvent être **concaténés** ou **fractionnés** le long d'un axe donné.

- **`np.concatenate()`** : Concatène des tableaux le long d'un axe.
- **`np.split()`** : Fractionne un tableau en sous-tableaux.

```
# Concaténation
concatene = np.concatenate((tableau_1, tableau_2))

# Fractionnement
fractionne = np.split(concatene, 3)

print(concatene)
print(fractionne)
```

11.4 Exercice pratique et Questions/Réponses (30 minutes)

Exercice : Manipulation des Tableaux ndarray

1. **Création et Manipulation de Tableaux** :
   - Créez un tableau 2D de dimension `(4, 5)` rempli de valeurs aléatoires.
   - Redimensionnez-le en un tableau de forme `(5, 4)`.
   - Effectuez la transposition du tableau.
2. **Empilement et Concaténation** :
   - Créez deux tableaux 1D contenant chacun 6 éléments.
   - Empilez-les verticalement et horizontalement.
   - Concaténez les deux tableaux sur un seul axe.
3. **Fractionnement** :
   - Créez un tableau 1D contenant 12 éléments.
   - Fractionnez ce tableau en 4 sous-tableaux de taille égale.

Exemple :

```
import numpy as np

# 1. Création et manipulation
tableau = np.random.rand(4, 5)  # Tableau aléatoire
reshaped_tableau = tableau.reshape(5, 4)
transpose = tableau.T

print("Tableau original :\n", tableau)
print("Tableau redimensionné :\n", reshaped_tableau)
print("Tableau transposé :\n", transpose)

# 2. Empilement et concaténation
tableau_1 = np.array([1, 2, 3, 4, 5, 6])
tableau_2 = np.array([7, 8, 9, 10, 11, 12])

vstacked = np.vstack((tableau_1, tableau_2))
hstacked = np.hstack((tableau_1, tableau_2))
concatene = np.concatenate((tableau_1, tableau_2))

print("Empilement vertical :\n", vstacked)
```

```
print("Empilement horizontal :\n", hstacked)
print("Tableau concaténé :\n", concatene)

# 3. Fractionnement
tableau_3 = np.arange(12)
fractionne = np.split(tableau_3, 4)

print("Tableau fractionné :\n", fractionne)
```

Module 12 : Numpy (Slicing et Indexing)

Durée : 2 heures

Objectifs :

- Comprendre les concepts de **slicing** (découpage) et d'**indexing** (indexation) dans NumPy.
- Apprendre à accéder et modifier les éléments d'un tableau **ndarray**.
- Manipuler des sous-ensembles de tableaux à l'aide de techniques avancées d'indexation.
- Gérer des tableaux multidimensionnels avec des méthodes efficaces de slicing.

12.1 Introduction à l'Indexing dans NumPy (30 minutes)

L'**indexing** permet d'accéder à des éléments spécifiques dans un tableau **ndarray**. Il est similaire à l'indexation des listes Python, mais avec des possibilités supplémentaires pour les tableaux multidimensionnels.

Indexation de base pour les tableaux 1D

Les indices dans NumPy commencent à 0, tout comme dans les listes Python. On peut accéder à un élément spécifique avec son index.

```
import numpy as np

# Créer un tableau 1D
tableau = np.array([10, 20, 30, 40, 50])

# Accéder au troisième élément
print(tableau[2])  # Affiche 30
```

Indexation pour les tableaux multidimensionnels

Dans un tableau à plusieurs dimensions, on utilise une paire d'indices pour accéder à un élément. Par exemple, pour accéder à un élément dans un tableau 2D, on utilise `[ligne, colonne]`.

```
python

# Créer un tableau 2D
tableau_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Accéder à l'élément de la deuxième ligne, troisième colonne
print(tableau_2d[1, 2])  # Affiche 6
```

Indexation négative

NumPy permet d'utiliser des indices négatifs pour accéder aux éléments en partant de la fin du tableau.

```
# Dernier élément du tableau 1D
print(tableau[-1])  # Affiche 50

# Dernier élément du tableau 2D
print(tableau_2d[-1, -1])  # Affiche 9
```

12.2 Slicing (Découpage) dans NumPy (40 minutes)

Le **slicing** permet de sélectionner des sous-parties d'un tableau. Il utilise la syntaxe suivante : `[start:stop:step]`. Cela fonctionne pour les tableaux de toute dimension.

Slicing des tableaux 1D

Le slicing dans un tableau 1D est similaire à celui des listes Python.

```
# Créer un tableau 1D
tableau = np.array([10, 20, 30, 40, 50])

# Sélectionner les éléments du deuxième au quatrième
sous_tableau = tableau[1:4]

print(sous_tableau)  # Affiche [20 30 40]
```

Slicing des tableaux 2D

Dans les tableaux multidimensionnels, vous pouvez spécifier un slice pour chaque dimension séparée par une virgule.

```
# Créer un tableau 2D
tableau_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Sélectionner les deux premières lignes et les deux premières colonnes
sous_tableau_2d = tableau_2d[:2, :2]

print(sous_tableau_2d)  # Affiche [[1 2] [4 5]]
```

Slicing avec un step

Vous pouvez également spécifier un `step` (pas) pour sauter des éléments.

```
# Sélectionner les éléments avec un step de 2
sous_tableau_step = tableau[::2]

print(sous_tableau_step)  # Affiche [10 30 50]
```

Slicing des tableaux multidimensionnels avec step

Dans les tableaux 2D et plus, le step peut être appliqué à chaque dimension.

```
# Slicing avec un step sur les lignes et colonnes
sous_tableau_step_2d = tableau_2d[::2, ::2]

print(sous_tableau_step_2d)  # Affiche [[1 3] [7 9]]
```

12.3 Indexation Avancée (40 minutes)

NumPy propose des techniques avancées pour accéder aux éléments en utilisant des listes d'indices ou des conditions logiques.

Indexation par tableau d'indices

Il est possible d'utiliser des listes ou des tableaux d'indices pour sélectionner plusieurs éléments à la fois.

```
# Créer un tableau
tableau = np.array([10, 20, 30, 40, 50])

# Utiliser un tableau d'indices pour sélectionner des éléments spécifiques
indices = [0, 2, 4]
selection = tableau[indices]

print(selection)  # Affiche [10 30 50]
```

Indexation booléenne

Vous pouvez sélectionner des éléments selon une condition logique.

```
# Créer un tableau
tableau = np.array([10, 20, 30, 40, 50])

# Sélectionner les éléments supérieurs à 30
condition = tableau > 30
selection = tableau[condition]

print(selection)  # Affiche [40 50]
```

Indexation multidimensionnelle avec des listes d'indices

Pour les tableaux multidimensionnels, vous pouvez également utiliser des listes d'indices pour chaque dimension.

```
# Créer un tableau 2D
tableau_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Utiliser une liste d'indices pour sélectionner des éléments spécifiques
selection = tableau_2d[[0, 1, 2], [0, 1, 2]]

print(selection)  # Affiche [1 5 9]
```

12.4 Exercice pratique et Questions/Réponses (30 minutes)

Exercice : Indexation et Slicing

1. **Indexation de Base :**
   - Créez un tableau 1D contenant 10 éléments. Sélectionnez le cinquième élément, puis le dernier élément en utilisant un index négatif.
2. **Slicing :**
   - Créez un tableau 2D de dimension (4x4) et sélectionnez les deux premières lignes et colonnes.
3. **Slicing avec Step :**
   - Créez un tableau 1D contenant les nombres de 1 à 20, puis sélectionnez les éléments pairs.
4. **Indexation Avancée :**
   - Créez un tableau 2D de dimension (3x3) et utilisez une liste d'indices pour sélectionner les éléments situés sur la diagonale.

Exemple :

```
import numpy as np

# 1. Indexation de Base
tableau = np.arange(1, 11)
cinquieme_element = tableau[4]
dernier_element = tableau[-1]

print("Cinquième élément :", cinquieme_element)
print("Dernier élément :", dernier_element)

# 2. Slicing
tableau_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13,
14, 15, 16]])
sous_tableau_2d = tableau_2d[:2, :2]

print("Sous-tableau 2D :\n", sous_tableau_2d)

# 3. Slicing avec Step
tableau_1d = np.arange(1, 21)
elements_pairs = tableau_1d[1::2]

print("Éléments pairs :", elements_pairs)

# 4. Indexation Avancée
tableau_diag = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
diagonale = tableau_diag[[0, 1, 2], [0, 1, 2]]

print("Éléments sur la diagonale :", diagonale)
```

Module 13 : Numpy (Mathématiques et Statistiques)

Durée : 2 heures

Objectifs :

- Comprendre et appliquer les principales fonctions mathématiques de NumPy.
- Effectuer des opérations arithmétiques élémentaires et avancées sur des tableaux.
- Apprendre à utiliser les fonctions statistiques de NumPy pour analyser des données.
- Maîtriser le calcul de mesures centrales, de dispersion, et d'autres statistiques descriptives.

13.1 Opérations Arithmétiques de Base avec NumPy (30 minutes)

NumPy permet d'effectuer des opérations mathématiques élémentaires sur des tableaux de manière très efficace, sans nécessiter de boucles explicites.

Addition, Soustraction, Multiplication et Division

Les opérations arithmétiques se réalisent élément par élément entre les tableaux.

```
import numpy as np

# Créer deux tableaux 1D
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])

# Addition
somme = a + b

# Soustraction
difference = a - b

# Multiplication
produit = a * b

# Division
quotient = a / b

print("Somme :", somme)
print("Différence :", difference)
print("Produit :", produit)
print("Quotient :", quotient)
```

Puissance et Racine Carrée

NumPy offre des fonctions pour effectuer des opérations de puissance et de racine carrée sur des tableaux.

```
# Élévation au carré
carre = a ** 2

# Racine carrée
racine_carre = np.sqrt(a)

print("Élévation au carré :", carre)
print("Racine carrée :", racine_carre)
```

Exponentielle et Logarithme

Vous pouvez également calculer l'exponentielle et le logarithme des éléments d'un tableau.

```
# Exponentielle
exp = np.exp(a)

# Logarithme naturel
log = np.log(a)

print("Exponentielle :", exp)
print("Logarithme naturel :", log)
```

13.2 Fonctions Mathématiques Avancées (30 minutes)

NumPy propose plusieurs fonctions mathématiques avancées telles que les fonctions trigonométriques, hyperboliques, etc.

Fonctions Trigonométriques

Les fonctions trigonométriques comme **sin()**, **cos()**, et **tan()** sont disponibles dans NumPy.

```
# Angles en radians
angles = np.array([0, np.pi/2, np.pi])

# Calcul de sin, cos, tan
sinus = np.sin(angles)
cosinus = np.cos(angles)
tangente = np.tan(angles)

print("Sinus :", sinus)
print("Cosinus :", cosinus)
print("Tangente :", tangente)
```

Fonctions Hyperboliques

NumPy offre également les fonctions hyperboliques telles que **sinh()**, **cosh()**, et **tanh()**.

```
# Fonctions hyperboliques
sinh = np.sinh(angles)
cosh = np.cosh(angles)
tanh = np.tanh(angles)

print("Sinh :", sinh)
print("Cosh :", cosh)
print("Tanh :", tanh)
```

13.3 Statistiques de Base avec NumPy (40 minutes)

Les tableaux NumPy peuvent être analysés en calculant des mesures statistiques comme la moyenne, la médiane, ou l'écart-type.

Moyenne, Médiane et Somme

NumPy propose des fonctions pour calculer la **moyenne** et la **médiane** d'un tableau, ainsi que la somme des éléments.

```
# Créer un tableau
data = np.array([10, 20, 30, 40, 50])

# Moyenne
moyenne = np.mean(data)

# Médiane
mediane = np.median(data)

# Somme des éléments
somme = np.sum(data)

print("Moyenne :", moyenne)
print("Médiane :", mediane)
print("Somme :", somme)
```

Ecart-type et Variance

Ces mesures permettent de quantifier la dispersion des données autour de la moyenne.

```
# Ecart-type
ecart_type = np.std(data)

# Variance
variance = np.var(data)

print("Ecart-type :", ecart_type)
print("Variance :", variance)
```

Minimum, Maximum et Range

NumPy offre des fonctions pour trouver le **minimum**, le **maximum**, et la différence entre ces deux valeurs.

```
# Minimum
minimum = np.min(data)

# Maximum
maximum = np.max(data)

# Range (étendue)
range_val = maximum - minimum

print("Minimum :", minimum)
print("Maximum :", maximum)
print("Range :", range_val)
```

Percentiles

Les **percentiles** permettent de diviser les données en groupes selon une distribution.

```
# Calcul du 25e, 50e (médiane) et 75e percentiles
percentile_25 = np.percentile(data, 25)
percentile_50 = np.percentile(data, 50)  # Médiane
percentile_75 = np.percentile(data, 75)

print("25e percentile :", percentile_25)
print("50e percentile :", percentile_50)
print("75e percentile :", percentile_75)
```

13.4 Exercice pratique et Questions/Réponses (20 minutes)

Exercice : Mathématiques et Statistiques avec NumPy

1. **Opérations Arithmétiques** :
   - Créez deux tableaux 1D et effectuez une addition, une multiplication, et une division élément par élément.
2. **Calculs Mathématiques Avancés** :
   - Créez un tableau de valeurs d'angles en radians. Calculez les sinus, cosinus, et tangentes de ces angles.
3. **Statistiques Descriptives** :
   - Créez un tableau de données et calculez la moyenne, la médiane, l'écart-type et le 90e percentile.

Exemple :

```
import numpy as np

# 1. Opérations Arithmétiques
a = np.array([10, 20, 30, 40])
b = np.array([2, 3, 4, 5])

addition = a + b
```

```
multiplication = a * b
division = a / b

print("Addition :", addition)
print("Multiplication :", multiplication)
print("Division :", division)

# 2. Calculs Trigonométriques
angles = np.array([0, np.pi/4, np.pi/2, np.pi])
sinus = np.sin(angles)
cosinus = np.cos(angles)
tangente = np.tan(angles)

print("Sinus :", sinus)
print("Cosinus :", cosinus)
print("Tangente :", tangente)

# 3. Statistiques Descriptives
data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
moyenne = np.mean(data)
mediane = np.median(data)
ecart_type = np.std(data)
percentile_90 = np.percentile(data, 90)

print("Moyenne :", moyenne)
print("Médiane :", mediane)
print("Écart-type :", ecart_type)
print("90e percentile :", percentile_90)
```

Module 14 : Pandas (Les Bases et Analyse de Données)

Durée : 2 heures

Objectifs :

- Comprendre les fondamentaux de la bibliothèque **Pandas** pour la manipulation et l'analyse de données.
- Apprendre à utiliser les **DataFrames** et **Series**, les structures de données de base de Pandas.
- Charger, examiner, nettoyer et manipuler des ensembles de données.
- Appliquer des méthodes d'analyse de données pour extraire des informations utiles.

14.1 Introduction à Pandas (20 minutes)

Pandas est une bibliothèque Python essentielle pour la manipulation et l'analyse de données. Elle est largement utilisée dans des domaines tels que la data science, les statistiques et l'apprentissage automatique. Pandas offre des outils performants pour gérer des données tabulaires, notamment via ses structures **Series** et **DataFrame**.

Installation de Pandas

Si vous n'avez pas encore Pandas installé, vous pouvez l'installer avec la commande suivante :

```
bash

pip install pandas
```

Importation de Pandas

Commencez toujours par importer la bibliothèque :

```
import pandas as pd
```

14.2 Structure de Données Principales : Series et DataFrame (30 minutes)

Pandas repose sur deux structures principales : **Series** et **DataFrame**.

Les Series

Une **Series** est une structure de données unidimensionnelle, similaire à une liste ou un tableau.

```
import pandas as pd

# Créer une Series à partir d'une liste
serie = pd.Series([10, 20, 30, 40])

print(serie)
```

Les Series peuvent également être créées à partir de dictionnaires, où les clés représentent les indices.

```
# Créer une Series à partir d'un dictionnaire
data = {'a': 1, 'b': 2, 'c': 3}
serie = pd.Series(data)

print(serie)
```

Les DataFrames

Un **DataFrame** est une structure bidimensionnelle, similaire à une feuille de calcul ou à une table de base de données.

```
# Créer un DataFrame à partir d'un dictionnaire
data = {'Nom': ['Alice', 'Bob', 'Charlie'], 'Âge': [25, 30, 35], 'Ville':
['Paris', 'Lyon', 'Marseille']}
df = pd.DataFrame(data)

print(df)
```

Un DataFrame est constitué de lignes et de colonnes, chaque colonne pouvant être une **Series**.

14.3 Chargement de Données dans Pandas (20 minutes)

Pandas permet de charger des données depuis différentes sources, telles que des fichiers CSV, Excel, ou des bases de données SQL.

Charger un fichier CSV

Le format CSV est l'un des formats les plus courants pour les ensembles de données. Pandas facilite le chargement d'un fichier CSV avec **read_csv**.

```
# Charger un fichier CSV
df = pd.read_csv('data.csv')

# Afficher les 5 premières lignes
print(df.head())
```

Charger un fichier Excel

De même, vous pouvez charger des fichiers Excel avec **read_excel** :

```
# Charger un fichier Excel
df = pd.read_excel('data.xlsx')

# Afficher les 5 premières lignes
print(df.head())
```

Examiner les Données

Une fois les données chargées, vous pouvez utiliser plusieurs méthodes pour obtenir des informations sur votre DataFrame :

- **head()** : afficher les premières lignes.
- **tail()** : afficher les dernières lignes.
- **info()** : obtenir des informations générales sur le DataFrame.
- **describe()** : obtenir des statistiques descriptives.

```
print(df.info())
print(df.describe())
```

14.4 Manipulation des Données (40 minutes)

Pandas propose une vaste gamme de méthodes pour manipuler des DataFrames et les préparer à l'analyse.

Sélection des Colonnes

Vous pouvez accéder aux colonnes d'un DataFrame en utilisant des crochets ou des attributs.

```
# Sélectionner une colonne
colonne_age = df['Âge']

print(colonne_age)
```

Filtrage des Lignes

Vous pouvez filtrer les lignes selon certaines conditions.

```
# Filtrer les lignes où l'âge est supérieur à 30
df_age_30 = df[df['Âge'] > 30]

print(df_age_30)
```

Ajout et Suppression de Colonnes

Vous pouvez ajouter une nouvelle colonne ou supprimer une colonne existante.

```
# Ajouter une colonne "Salaire"
df['Salaire'] = [40000, 50000, 60000]

# Supprimer une colonne
df = df.drop(columns=['Salaire'])

print(df)
```

Gestion des Valeurs Manquantes

Les données réelles contiennent souvent des valeurs manquantes. Pandas propose plusieurs techniques pour les gérer, comme **fillna()** pour remplir les valeurs manquantes ou **dropna()** pour supprimer les lignes contenant des valeurs manquantes.

```
# Remplacer les valeurs manquantes par la moyenne
df['Âge'].fillna(df['Âge'].mean(), inplace=True)

# Supprimer les lignes contenant des valeurs manquantes
df.dropna(inplace=True)
```

14.5 Analyse de Données avec Pandas (30 minutes)

Pandas simplifie l'analyse de données grâce à ses méthodes statistiques et de regroupement.

GroupBy et Agrégation

Vous pouvez regrouper les données selon une ou plusieurs colonnes et appliquer des fonctions d'agrégation telles que **mean()**, **sum()**, ou **count()**.

```
# Calculer l'âge moyen par ville
age_moyen = df.groupby('Ville')['Âge'].mean()

print(age_moyen)
```

Tri des Données

Pandas permet de trier les DataFrames par les valeurs d'une colonne ou plusieurs colonnes.

```
# Trier les données par âge
df_trie = df.sort_values(by='Âge')

print(df_trie)
```

Appliquer des Fonctions à une Colonne

Pandas offre la possibilité d'appliquer des fonctions à une colonne entière grâce à **apply()**.

```
# Appliquer une fonction pour convertir l'âge en mois
df['Âge_en_mois'] = df['Âge'].apply(lambda x: x * 12)

print(df)
```

14.6 Exercice pratique et Questions/Réponses (30 minutes)

Exercice : Analyse de Données avec Pandas

1. **Charger et Explorer des Données** :
   - Chargez un fichier CSV dans un DataFrame. Utilisez les fonctions **head()**, **info()**, et **describe()** pour explorer les données.
2. **Filtrage et Sélection** :
   - Sélectionnez les lignes où l'âge est supérieur à 30 ans et affichez uniquement les colonnes **Nom** et **Ville**.
3. **Manipulation des Données** :
   - Ajoutez une colonne qui calcule le nombre de mois pour chaque âge, puis triez les données selon cette nouvelle colonne.
4. **GroupBy et Agrégation** :
   - Regroupez les données par ville et calculez l'âge moyen pour chaque ville.

Exemple :

```
import pandas as pd

# 1. Charger et explorer les données
df = pd.read_csv('data.csv')
print(df.head())
print(df.info())
print(df.describe())

# 2. Filtrage et sélection
df_filtree = df[df['Âge'] > 30][['Nom', 'Ville']]
print(df_filtree)

# 3. Manipulation des données
df['Âge_en_mois'] = df['Âge'] * 12
df_trie = df.sort_values(by='Âge_en_mois')
print(df_trie)

# 4. GroupBy et agrégation
age_moyen_ville = df.groupby('Ville')['Âge'].mean()
print(age_moyen_ville)
```

Module 13 : Matplotlib (Graphiques de Base)

Durée : 2 heures

Objectifs :

- Découvrir la bibliothèque **Matplotlib** pour créer des visualisations graphiques en Python.
- Apprendre à générer des graphiques simples (courbes, histogrammes, barres, nuages de points).
- Personnaliser les graphiques avec des titres, légendes, et annotations.
- Appliquer des techniques de visualisation pour explorer et présenter des données.

13.1 Introduction à Matplotlib (20 minutes)

Matplotlib est la bibliothèque de référence en Python pour la création de graphiques. Elle permet de générer des graphiques statiques, animés et interactifs.

Installation de Matplotlib

Si Matplotlib n'est pas installé, vous pouvez l'installer avec la commande suivante :

```
pip install matplotlib
```

Importation de Matplotlib

La convention standard pour importer Matplotlib est d'utiliser `pyplot`, un sous-module de Matplotlib.

```
import matplotlib.pyplot as plt
```

13.2 Création de Graphiques Simples (30 minutes)

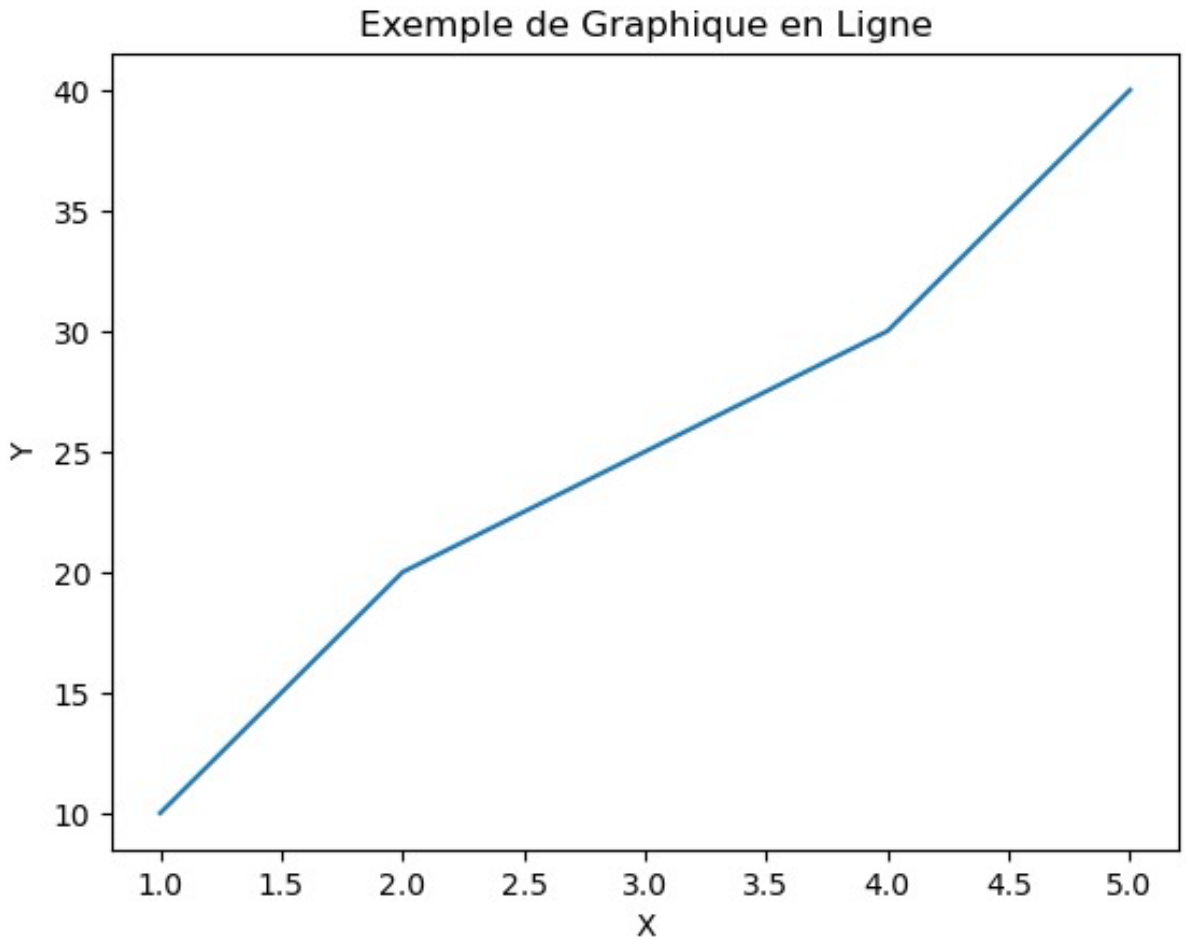
Matplotlib permet de créer des graphiques de base comme des **courbes**, des **histogrammes**, des **diagrammes en barres** et des **nuages de points**.

Graphique en Ligne (Courbe)

Un graphique en ligne est utilisé pour représenter des données continues.

```
# Données
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

# Créer un graphique en ligne
plt.plot(x, y)
```

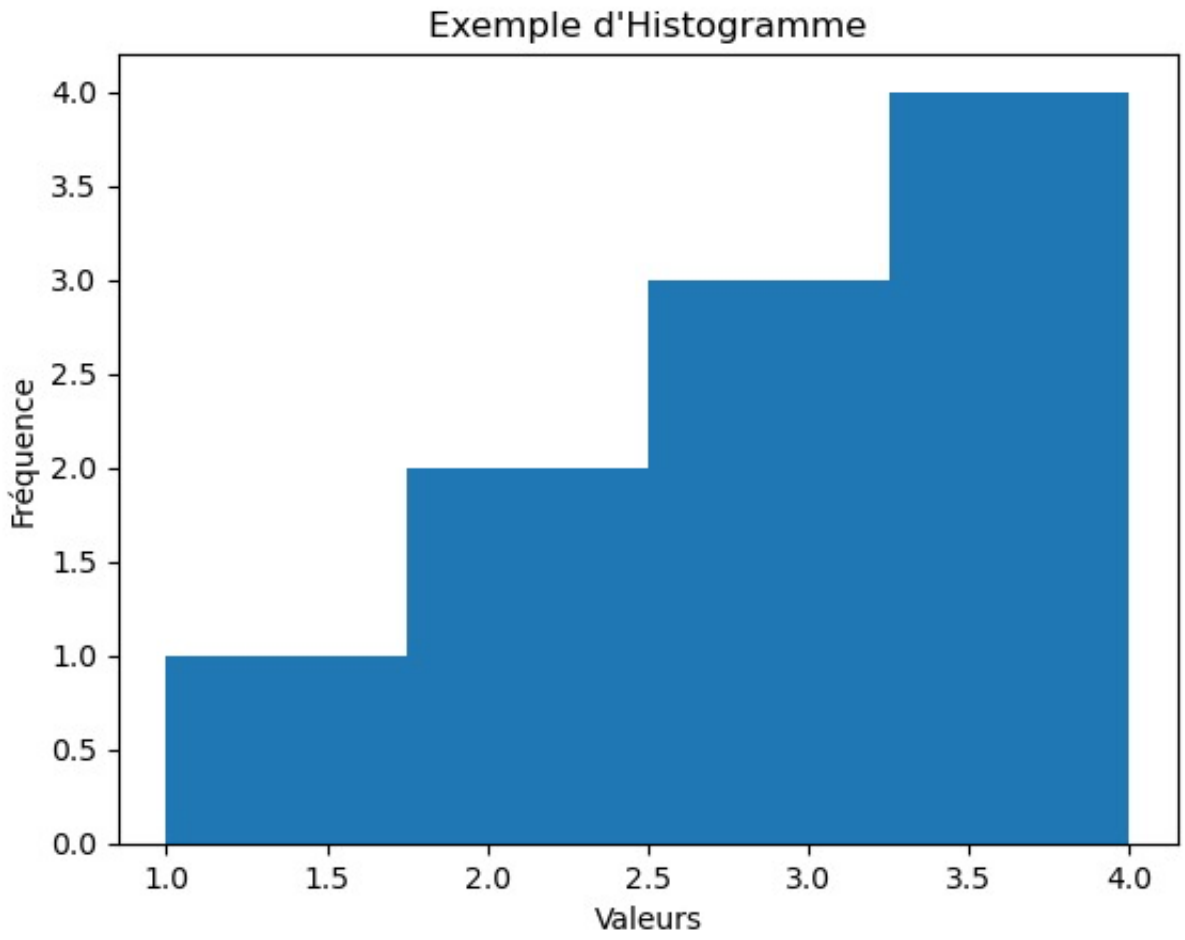
```
# Ajouter un titre et des labels
plt.title('Exemple de Graphique en Ligne')
plt.xlabel('X')
plt.ylabel('Y')

# Afficher le graphique
plt.show()
```



Histogramme

Un histogramme est utilisé pour visualiser la distribution d'un ensemble de données.

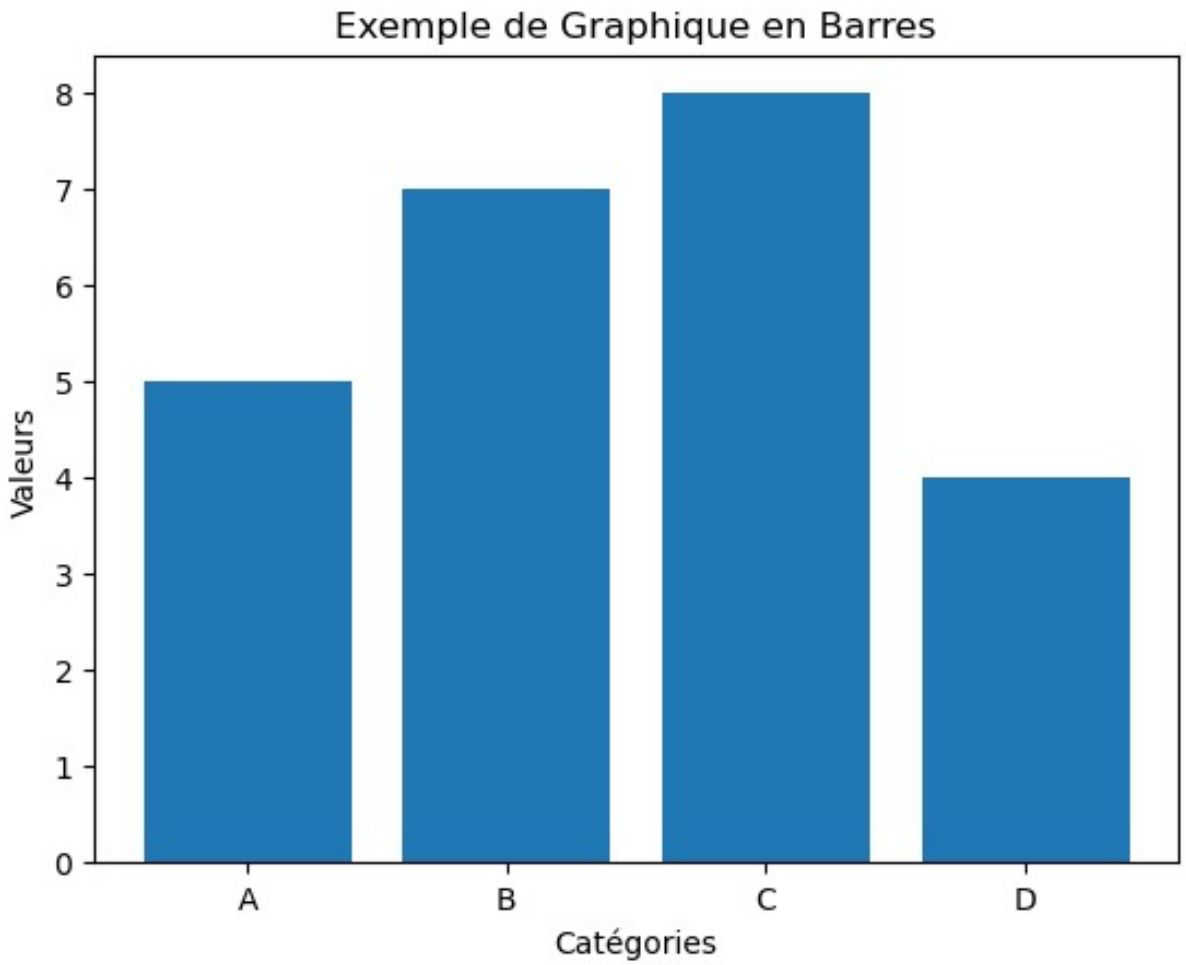
```
# Données
data = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]

# Créer un histogramme
plt.hist(data, bins=4)

# Ajouter un titre et des labels
plt.title('Exemple d\'Histogramme')
plt.xlabel('Valeurs')
plt.ylabel('Fréquence')

# Afficher le graphique
plt.show()
```



Graphique en Barres

Un diagramme en barres permet de comparer des valeurs discrètes.

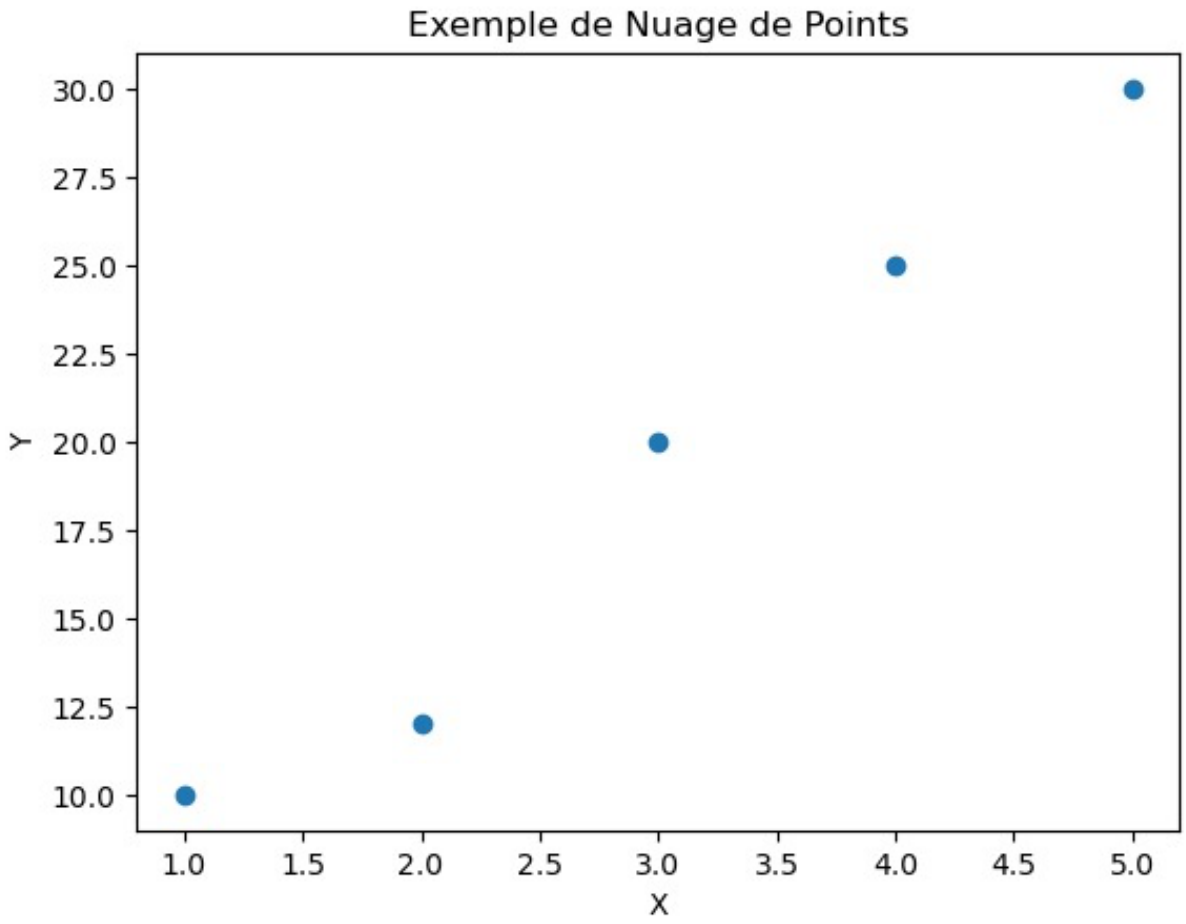
```
# Données
categories = ['A', 'B', 'C', 'D']
valeurs = [5, 7, 8, 4]

# Créer un graphique en barres
plt.bar(categories, valeurs)

# Ajouter un titre et des labels
plt.title('Exemple de Graphique en Barres')
plt.xlabel('Catégories')
plt.ylabel('Valeurs')

# Afficher le graphique
plt.show()
```



Nuage de Points (Scatter Plot)

Un nuage de points est utilisé pour visualiser la relation entre deux variables.

```
# Données
x = [1, 2, 3, 4, 5]
y = [10, 12, 20, 25, 30]

# Créer un nuage de points
plt.scatter(x, y)

# Ajouter un titre et des labels
plt.title('Exemple de Nuage de Points')
plt.xlabel('X')
plt.ylabel('Y')

# Afficher le graphique
plt.show()
```



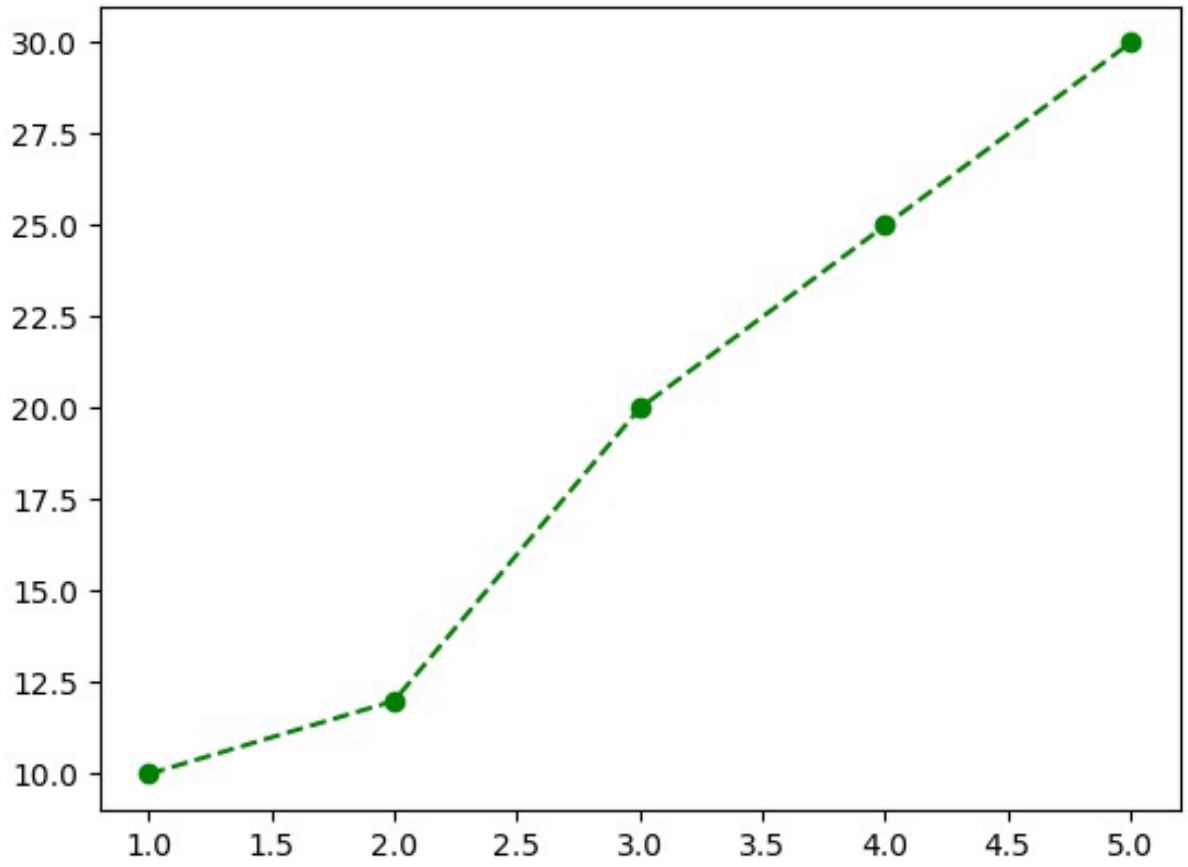
13.3 Personnalisation des Graphiques (40 minutes)

Matplotlib permet de **personnaliser** chaque aspect d'un graphique, depuis les couleurs des lignes jusqu'aux titres et annotations.

Personnalisation des Lignes et Marques

Vous pouvez personnaliser la couleur, le style de ligne et les marqueurs des points de données.

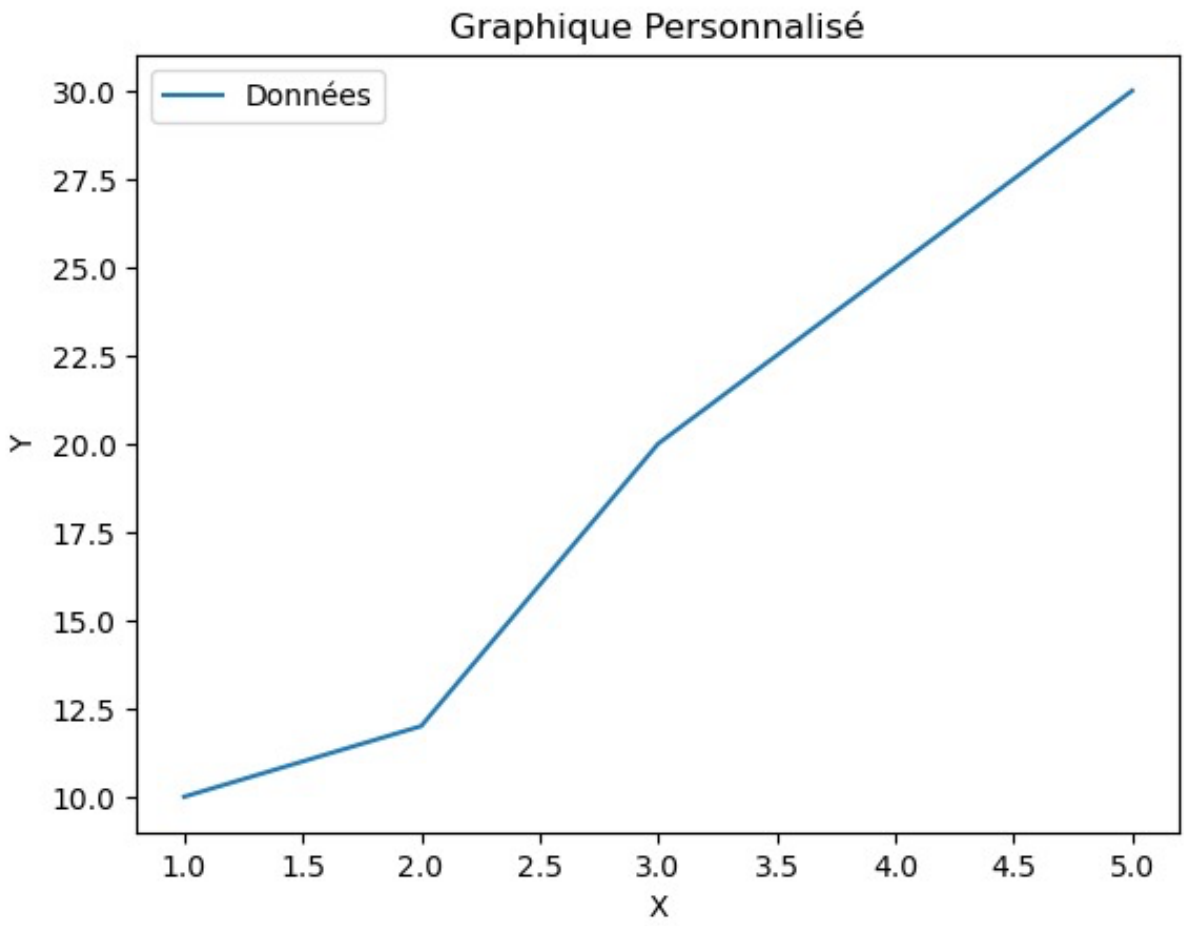
```
# Créer un graphique avec des options personnalisées
plt.plot(x, y, color='green', linestyle='--', marker='o')

# Afficher le graphique
plt.show()
```



Ajout de Titre, Labels et Légendes

Il est possible d'ajouter des **titres**, des **labels** pour les axes, et une **légende** pour mieux expliquer le graphique.

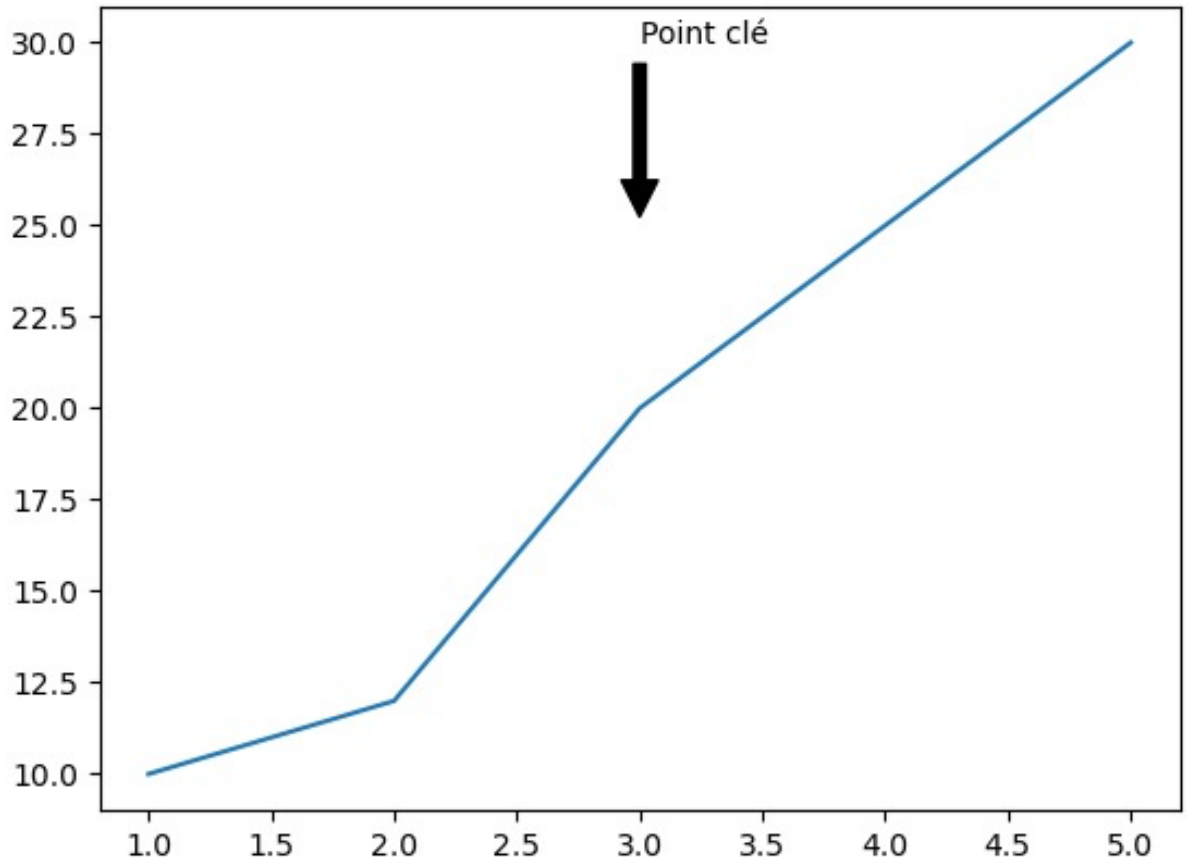
```
plt.plot(x, y, label='Données')

# Ajouter titre et légendes
plt.title('Graphique Personnalisé')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()

plt.show()
```



Ajout d'Annotations

Les annotations permettent de mettre en valeur certaines parties du graphique.

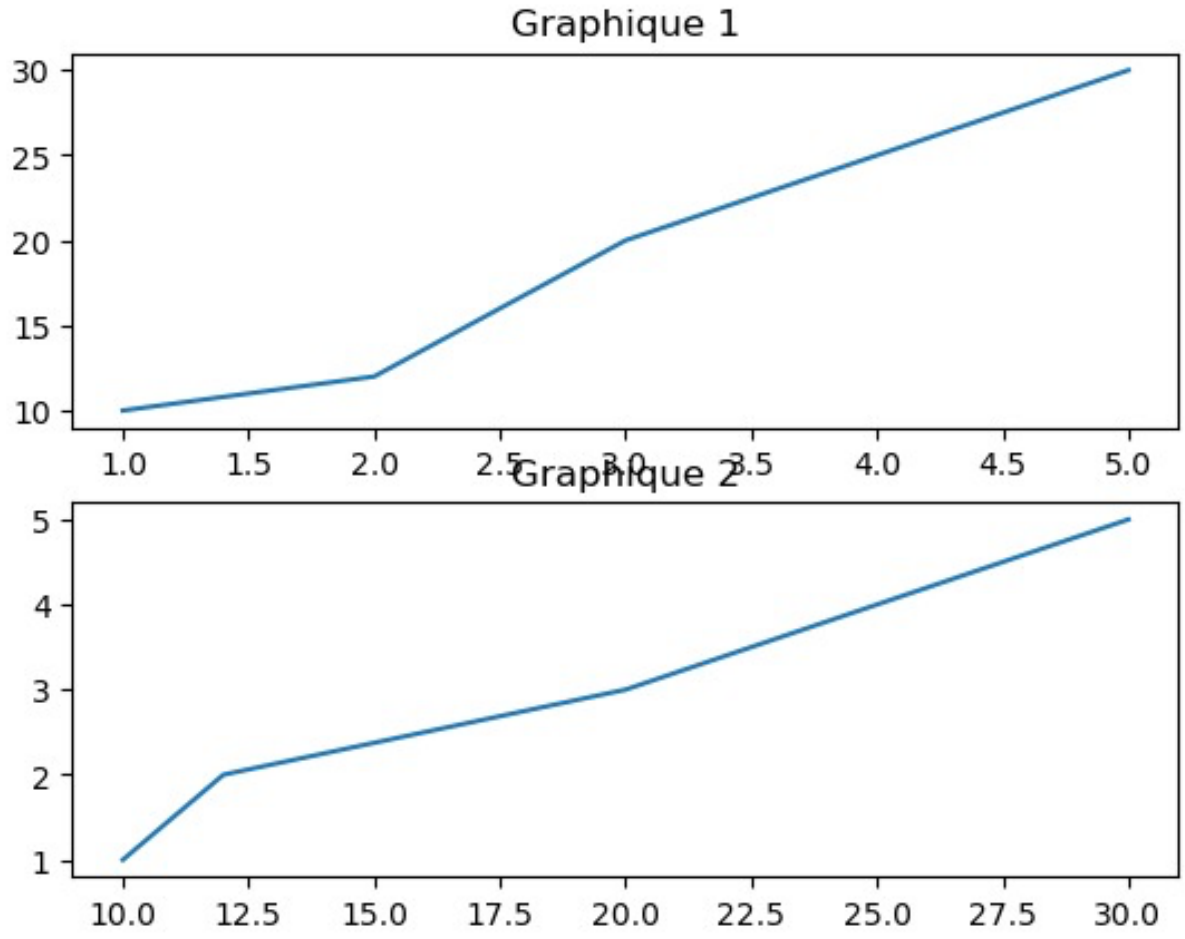
```
# Créer un graphique et ajouter une annotation
plt.plot(x, y)
plt.annotate('Point clé', xy=(3, 25), xytext=(3, 30),
arrowprops=dict(facecolor='black', shrink=0.05))

plt.show()
```



Sous-graphes (Subplots)

Vous pouvez afficher plusieurs graphiques dans une même figure avec **subplots()**.

```
# Créer deux sous-graphes
fig, axs = plt.subplots(2)

# Premier graphique
axs[0].plot(x, y)
axs[0].set_title('Graphique 1')

# Deuxième graphique
axs[1].plot(y, x)
axs[1].set_title('Graphique 2')

plt.show()
```



13.4 Sauvegarde et Affichage des Graphiques (10 minutes)

Une fois que vous avez généré un graphique, vous pouvez le **sauvegarder** dans différents formats d'image, tels que PNG ou PDF.

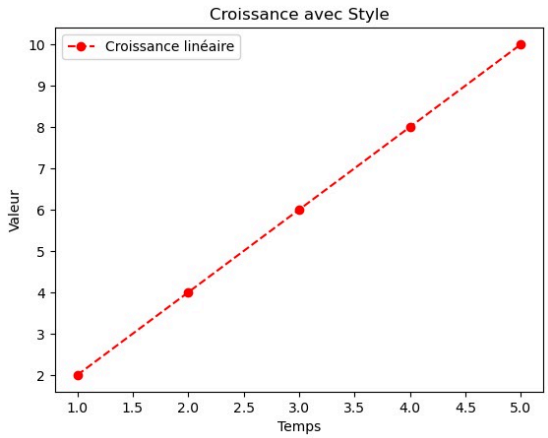
```
# Sauvegarder le graphique
plt.plot(x, y)
plt.savefig('graphique.png')

# Afficher le graphique
plt.show()
```

13.5 Exercice Pratique et Questions/Réponses (20 minutes)

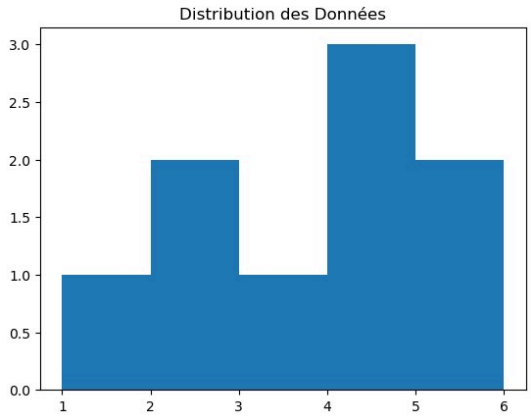
1. **Courbe et Personnalisation** : Créez un graphique en ligne avec des données de votre choix. Personnalisez les couleurs, les titres, et ajoutez une légende.
2. **Histogramme et Barres** : Créez un histogramme et un graphique en barres avec des données aléatoires. Ajoutez des labels pour chaque graphique.
3. **Nuage de Points et Annotations** : Créez un nuage de points représentant une relation entre deux variables. Ajoutez une annotation sur le point le plus important.

Exemple :

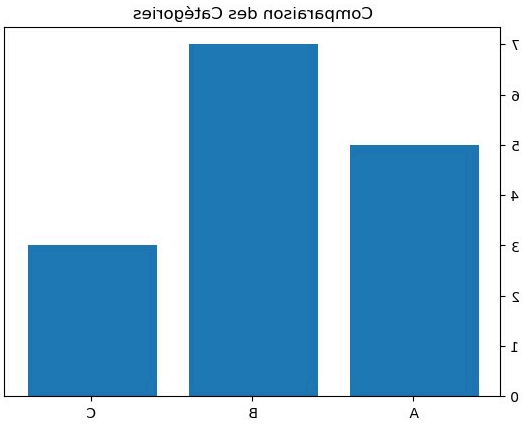
```
import matplotlib.pyplot as plt

# 1. Courbe personnalisée
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
plt.plot(x, y, color='red', linestyle='--', marker='o', label='Croissance
linéaire')
plt.title('Croissance avec Style')
plt.xlabel('Temps')
plt.ylabel('Valeur')
plt.legend()
plt.show()
```



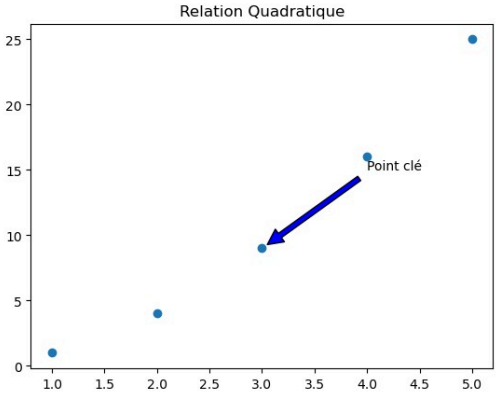

```
# 2. Histogramme et Barres
data = [1, 2, 2, 3, 4, 4, 4, 5, 6]
plt.hist(data, bins=5)
plt.title('Distribution des Données')
plt.show()
```




```
categories = ['A', 'B', 'C']
valeurs = [5, 7, 3]
plt.bar(categories, valeurs)
plt.title('Comparaison des Catégories')
plt.show()
```




```
# 3. Nuage de Points
x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]
plt.scatter(x, y)
plt.title('Relation Quadratique')
plt.annotate('Point clé', xy=(3, 9), xytext=(4, 15),
arrowprops=dict(facecolor='blue', shrink=0.05))
plt.show()
```



Prochaine Étape

Pour aller plus loin et approfondir vos compétences, nous vous invitons à consulter le compte **GitHub** dédié à ce cours. Vous y trouverez un récapitulatif complet des notions abordées ainsi que des **exercices pratiques** pour renforcer votre apprentissage. Les exercices vous permettront de mettre en pratique les concepts étudiés et de vous familiariser davantage avec Python, NumPy, Pandas, et Matplotlib.

N'hésitez pas à consulter régulièrement le dépôt GitHub pour des mises à jour et de nouveaux contenus !

Accédez aux fichiers exercices ici : https://github.com/Evilafo/Cours-python

Bon courage pour la suite et bonne pratique de Python !