Cours de Sciences de Données avec Python

Fatoumata Diallo

Table des matières

1 Introduction à Python pour la Data Science

Python est l'un des langages les plus utilisés en **science des données** pour sa simplicité et l'écosystème de bibliothèques : `NumPy` (calcul numérique), `Pandas` (tableaux), `Matplotlib/Seaborn` (visualisation) et `scikit-learn` (apprentissage automatique).

Un premier programme très simple consiste à afficher un message à l'écran :

```
print("Bienvenue dans le cours de Python pour la Data Science !")
```

—

2 Variables et opérations de base

2.1 Variables et types

Une **variable** est un espace mémoire qui contient une valeur. En Python, il n'est pas nécessaire de déclarer le type d'une variable : il est détecté automatiquement (typage dynamique).

Les types fondamentaux sont :

— `int` : nombres entiers,
— `float` : nombres réels,
— `str` : chaînes de caractères,
— `bool` : valeurs logiques `True`/`False`.

```
x = 10            # entier (int)
y = 3.14          # flottant (float)
nom = "Fatou"     # chaîne (str)
valide = True     # booléen (bool)

print(type(x), type(y), type(nom), type(valide))
```

—

2.2 Opérations arithmétiques

Python peut être utilisé comme une calculatrice. Les opérations de base sont :

— `+` : addition,
— `-` : soustraction,
— `*` : multiplication,
— `/` : division réelle,
— `//` : division entière,
— `%` : modulo (reste),
— `**` : puissance.

```
a = 7
b = 2

print("Addition :", a + b)
print("Soustraction :", a - b)
print("Multiplication :", a * b)
print("Division :", a / b)          # 3.5
print("Division entière :", a // b) # 3
```

```
print("Modulo :", a % b)            # 1
print("Puissance :", a ** b)        # 49
```

—

2.3 Opérateurs logiques et booléens

Les opérateurs de comparaison permettent de tester des conditions :
— `==` : égalité,
— `!=` : différence,
— `>` : strictement supérieur,
— `<` : strictement inférieur,
— `>=`, `<=` : supérieur ou égal / inférieur ou égal.
Ils se combinent avec des opérateurs logiques :
— `and` : ET logique,
— `or` : OU logique,
— `not` : négation.

```
x = 5
y = 10

print(x > y)         # False
print(x < y)         # True
print(x == 5)        # True
print(x != y)        # True

print(x > 0 and y > 0)   # True
print(x > 0 or y < 0)    # True
print(not (x > y))       # True
```

—

2.4 Exercice pratique

Écrire un programme qui calcule l'aire d'un rectangle.

```
longueur = 5
largeur = 3
aire = longueur * largeur
print("Aire =", aire)
```

—

3 Conditions et boucles

3.1 Structures conditionnelles

Les structures conditionnelles permettent d'exécuter du code en fonction d'un test logique. La syntaxe est :
— `if` : si la condition est vraie,
— `elif` : sinon si une autre condition est vraie,

— `else` : sinon (aucune condition remplie).

```
x = 2

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

—

3.2 Boucle for

La boucle `for` permet de répéter un bloc d'instructions pour chaque élément d'une séquence ou d'un intervalle numérique.

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

—

3.3 Boucle while

La boucle `while` exécute un bloc tant qu'une condition reste vraie.

```
x = 0
while x < 5:
    print("x =", x)
    x += 1
```

—

3.4 Exemple pratique

Programme qui indique si un nombre est pair ou impair.

```
n = int(input("Entrez un nombre : "))
if n % 2 == 0:
    print(n, "est pair")
else:
    print(n, "est impair")
```

—

3.5 Exercice complémentaire

Calculer la somme des 10 premiers entiers positifs.

```
somme = 0
for i in range(1, 11):
    somme += i
print("Somme des 10 premiers entiers =", somme)
```

4 Fonctions en Python

4.1 Définition et intérêt

Une **fonction** est un bloc de code réutilisable qui effectue une tâche précise. Elle permet :

— de réduire la duplication du code,
— de structurer et organiser les programmes,
— de rendre le code plus lisible et maintenable.

En Python, une fonction se définit avec le mot-clé `def`. La syntaxe générale est :

```
def nom_de_fonction(parametres):
    # bloc d'instructions
    return resultat
```

—

4.2 Exemple simple : fonction sans paramètre

```
def saluer():
    print("Bonjour, bienvenue en Python !")

saluer()
```

—

4.3 Fonction avec paramètres et retour de valeur

```
def carre(x):
    return x**2

print(carre(4))   # 16
print(carre(10))  # 100
```

—

4.4 Paramètres par défaut

Il est possible de donner une valeur par défaut aux paramètres.

```
def presentation(nom, age=18):
    print("Je m'appelle", nom, "et j'ai", age, "ans.")

presentation("Fatou", 25)
presentation("Julie")   # âge par défaut = 18
```

—

4.5 Arguments nommés et positionnels

Les arguments peuvent être passés :

— par position,

— par nom.

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

print(addition(5, 3))        # positionnels
print(addition(a=10, b=7))   # nommés
print(addition(b=4, a=6))    # ordre inversé possible
```

—

4.6 Fonctions anonymes (lambda)

Les fonctions `lambda` sont des fonctions courtes, écrites en une seule ligne. Elles sont souvent utilisées comme argument dans d'autres fonctions.

```python
f = lambda x: x**2
print(f(5))   # 25

produit = lambda x, y: x * y
print(produit(4, 3))   # 12
```

—

4.7 Exemple pratique

Écrire une fonction qui calcule la moyenne d'une liste de nombres.

```python
def moyenne(liste):
    return sum(liste) / len(liste)

notes = [12, 15, 9, 18, 14]
print("Moyenne :", moyenne(notes))
```

—

4.8 Exercice

1. Écrire une fonction `factorielle(n)` qui calcule $n! = 1 \times 2 \times ... \times n$. 2. Écrire une fonction `est_pair(n)` qui retourne `True` si n est pair, `False` sinon.

```python
def factorielle(n):
    res = 1
    for i in range(1, n+1):
        res *= i
    return res

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

print(factorielle(5))   # 120
print(est_pair(7))      # False
print(est_pair(10))     # True
```

5 Structures de données en Python

5.1 Les listes

Une **liste** est une structure de données ordonnée et modifiable (mutable). Elle peut contenir des éléments de différents types.

```
# Création de listes
nombres = [1, 2, 3, 4, 5]
melange = [10, "Python", 3.14, True]

# Indexation
print(nombres[0])      # premier élément
print(nombres[-1])     # dernier élément

# Slicing (tranches)
print(nombres[1:4])    # éléments d'indices 1 à 3
print(nombres[:3])     # du début jusqu'à l'indice 2
print(nombres[::2])    # un élément sur deux
```

Opérations principales sur les listes :

```
villes = ["Paris", "Berlin", "Londres"]

villes.append("Rome")        # ajout en fin
villes.insert(1, "Madrid")   # insertion à un indice
villes.remove("Paris")       # suppression d'un élément
dernier = villes.pop()       # enlève et retourne le dernier élément

print(len(villes))           # taille de la liste
print("Berlin" in villes)    # test d'appartenance

villes.sort()                # tri alphabétique
villes.reverse()             # inversion de l'ordre
```

—

5.2 Les tuples

Un **tuple** est similaire à une liste mais **immutable** : ses éléments ne peuvent pas être modifiés après création.

```
coordonnees = (10, 20)
print(coordonnees[0])    # accès par indice

# tuple d'un seul élément
t = (5,)
```

Les tuples sont utiles pour représenter des données fixes (par exemple, des coordonnées géographiques).

—

5.3 Les dictionnaires

Un **dictionnaire** est une collection non ordonnée d'associations `clé:valeur`. Les clés doivent être uniques et immuables (str, int, tuple).

```
etudiant = {
    "nom": "Fatou",
    "age": 25,
    "note": 16
}

print(etudiant["nom"])
etudiant["note"] = 18
etudiant["ville"] = "Paris"

print(etudiant.keys())
print(etudiant.values())
print(etudiant.items())
```

On peut parcourir un dictionnaire avec une boucle :

```
for cle, valeur in etudiant.items():
    print(cle, ":", valeur)
```

—

5.4 Les ensembles (sets)

Un **set** est une collection non ordonnée et sans doublons. Il est très utile pour effectuer des opérations ensemblistes.

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

print(A | B)    # union
print(A & B)    # intersection
print(A - B)    # différence
print(A ^ B)    # différence symétrique
```

—

5.5 Compréhensions de listes et de dictionnaires

Les **compréhensions** permettent de créer des listes, ensembles ou dictionnaires en une seule ligne.

```
carres = [x**2 for x in range(6)]
print(carres)  # [0, 1, 4, 9, 16, 25]

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

dico = {x: x**2 for x in range(4)}
print(dico)    # {0: 0, 1: 1, 2: 4, 3: 9}
```

—

5.6 Exercice pratique

1. Créer une liste de nombres de 1 à 20. 2. Extraire les carrés des nombres pairs dans une nouvelle liste. 3. Enregistrer le résultat dans un dictionnaire où la clé est le nombre et la valeur son carré.

```
nombres = list(range(1, 21))
carres_pairs = {x: x**2 for x in nombres if x % 2 == 0}
print(carres_pairs)
```

6 Opérations sur les structures de données

6.1 Opérations sur les listes

Les listes offrent un grand nombre de méthodes intégrées permettant de manipuler facilement des données.

- `append(x)` : ajoute un élément en fin de liste.
- `insert(i, x)` : insère un élément à une position donnée.
- `extend(L)` : étend une liste avec une autre liste.
- `remove(x)` : supprime la première occurrence d'un élément.
- `pop(i)` : supprime et retourne l'élément à la position `i` (ou le dernier si vide).
- `count(x)` : compte le nombre d'occurrences d'un élément.
- `index(x)` : retourne l'indice de la première occurrence.
- `sort()` : trie la liste.
- `reverse()` : inverse l'ordre des éléments.

```
villes = ["Paris", "Berlin", "Londres", "Madrid"]

villes.append("Rome")
villes.insert(1, "Bruxelles")
villes.remove("Paris")
dernier = villes.pop()

print("Liste finale :", villes)
print("Dernier élément retiré :", dernier)
```

—

6.2 Opérations sur les tuples

Les tuples étant immuables, les opérations sont limitées :

- accès par indice,
- découpage (slicing),
- concaténation avec `+`,
- répétition avec `*`.

```
t1 = (1, 2, 3)
t2 = (4, 5)
```

```
3
4 print(t1 + t2)    # concaténation
5 print(t1 * 2)     # répétition
6 print(t1[1:])     # slicing
```

—

6.3 Opérations sur les dictionnaires

Les dictionnaires possèdent des méthodes puissantes pour gérer les données clé/valeur.

— `get(cle, defaut)` : retourne la valeur associée à une clé (ou une valeur par défaut).
— `update(dict)` : met à jour avec un autre dictionnaire.
— `pop(cle)` : supprime et retourne une valeur.
— `popitem()` : supprime et retourne la dernière paire clé/valeur.
— `clear()` : vide le dictionnaire.

```
1  etudiant = {"nom": "Fatou", "age": 25, "note": 16}
2
3  print(etudiant.get("ville", "Inconnue"))
4  etudiant.update({"ville": "Paris", "age": 26})
5  print(etudiant)
6
7  note = etudiant.pop("note")
8  print("Note supprimée :", note)
9
10 print("Clés :", etudiant.keys())
11 print("Valeurs :", etudiant.values())
```

—

6.4 Opérations sur les ensembles

Les ensembles permettent les opérations classiques de la théorie des ensembles.

```
1  A = {1, 2, 3, 4}
2  B = {3, 4, 5, 6}
3
4  print("Union :", A | B)
5  print("Intersection :", A & B)
6  print("Différence :", A - B)
7  print("Différence symétrique :", A ^ B)
8
9  A.add(7)          # ajout
10 B.remove(6)       # suppression
11 print(A, B)
```

—

6.5 Conversions entre structures

Python permet de convertir facilement entre différents types de structures.

```python
liste = [1, 2, 2, 3]
ensemble = set(liste)        # suppression des doublons
print(ensemble)              # {1, 2, 3}

dico = dict([(1, "un"), (2, "deux")])
print(dico)                  # {1: 'un', 2: 'deux'}

liste2 = list(dico.keys())
print(liste2)                # [1, 2]
```

—

6.6 Exercice pratique

1. Créer une liste de prénoms avec des doublons. 2. Supprimer les doublons grâce à un ensemble. 3. Construire un dictionnaire associant chaque prénom à sa longueur.

```python
prenoms = ["Julie", "Marc", "Julie", "Sophie", "Marc"]
unique = set(prenoms)

dico = {nom: len(nom) for nom in unique}
print(dico)
```

7 Manipulation avancée des chaînes de caractères

7.1 Définition

Une **chaîne de caractères** (type `str`) est une séquence immuable de symboles (lettres, chiffres, espaces, ponctuation). En Python, les chaînes sont entourées de guillemets simples ou doubles.

```python
texte1 = "Bonjour"
texte2 = 'Python'
```

—

7.2 Indexation et slicing

Comme les listes, les chaînes sont indexées à partir de 0 et supportent le découpage (`slicing`).

```python
mot = "Python"

print(mot[0])      # P
print(mot[-1])     # n
print(mot[1:4])    # yth
print(mot[:3])     # Pyt
print(mot[::-1])   # nohytP (inversé)
```

—

7.3 Concaténation et répétition

```python
prenom = "Fatou"
nom = "Diallo"

nom_complet = prenom + " " + nom
print(nom_complet)

rire = "ha" * 3
print(rire)   # hahaha
```

—

7.4 Méthodes utiles des chaînes

— `lower()` : met en minuscules,
— `upper()` : met en majuscules,
— `capitalize()` : met la première lettre en majuscule,
— `strip()` : supprime les espaces au début et à la fin,
— `replace(a, b)` : remplace `a` par `b`,
— `split(sep)` : découpe selon un séparateur,
— `join(liste)` : concatène les éléments d'une liste avec un séparateur.

```python
phrase = "  Python est Génial  "

print(phrase.lower())
print(phrase.upper())
print(phrase.strip())
print(phrase.replace("Génial", "puissant"))

mots = phrase.split()
print(mots)
print("-".join(mots))
```

—

7.5 Itération sur une chaîne

```python
for lettre in "Python":
    print(lettre)
```

—

7.6 Test d'appartenance

```python
texte = "Machine Learning"
print("Learn" in texte)     # True
print("Deep" not in texte)  # True
```

—

7.7 Formatage des chaînes

Python propose plusieurs manières d'insérer des valeurs dans une chaîne.

```
nom = "Fatou"
age = 25

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

# f-strings (recommandé depuis Python 3.6)
print(f"Je m'appelle {nom} et j'ai {age} ans")
```

—

7.8 Exercice pratique

1. Demander à l'utilisateur son prénom. 2. Afficher un message personnalisé en majuscules. 3. Indiquer le nombre de lettres dans le prénom.

```
prenom = input("Entrez votre prénom : ")

print("Bonjour", prenom.upper())
print("Votre prénom contient", len(prenom), "lettres")
```

8 Entrées et sorties (I/O)

8.1 Lecture et écriture dans la console

En Python, la fonction `print()` permet d'afficher un message, tandis que la fonction `input()` permet de saisir des données depuis le clavier.

```
# Affichage simple
print("Bienvenue dans le cours de Python !")

# Lecture d'une entrée utilisateur
nom = input("Entrez votre nom : ")
print("Bonjour", nom)
```

Par défaut, la fonction `input()` retourne une chaîne de caractères. Il faut donc convertir en entier ou flottant si nécessaire.

```
age = int(input("Entrez votre âge : "))
taille = float(input("Entrez votre taille en mètres : "))

print(f"Vous avez {age} ans et mesurez {taille} m")
```

—

8.2 Lecture et écriture dans les fichiers

La fonction `open()` permet d'ouvrir un fichier. On doit préciser le nom du fichier et le mode :

— `"r"` : lecture,
— `"w"` : écriture (écrase si le fichier existe),
— `"a"` : ajout en fin de fichier,
— `"b"` : mode binaire (optionnel).

```python
# Écriture dans un fichier
with open("exemple.txt", "w", encoding="utf-8") as f:
    f.write("Bonjour Python\n")
    f.write("Une deuxième ligne\n")

# Lecture du fichier
with open("exemple.txt", "r", encoding="utf-8") as f:
    contenu = f.read()
    print(contenu)
```

—

8.3 Lecture ligne par ligne

```python
with open("exemple.txt", "r", encoding="utf-8") as f:
    for ligne in f:
        print("Ligne :", ligne.strip())
```

—

8.4 Écriture en mode ajout

```python
with open("exemple.txt", "a", encoding="utf-8") as f:
    f.write("Ajout d'une nouvelle ligne\n")
```

—

8.5 Vérification de l'existence d'un fichier

Le module `os` permet de gérer les fichiers et dossiers.

```python
import os

if os.path.exists("exemple.txt"):
    print("Le fichier existe")
else:
    print("Fichier introuvable")
```

—

8.6 Exercice pratique

1. Demander à l'utilisateur son prénom et son âge. 2. Écrire ces informations dans un fichier texte. 3. Lire le fichier et afficher le contenu.

```python
nom = input("Entrez votre prénom : ")
age = input("Entrez votre âge : ")

with open("utilisateur.txt", "w", encoding="utf-8") as f:
    f.write(f"Nom : {nom}\n")
    f.write(f"Âge : {age}\n")

with open("utilisateur.txt", "r", encoding="utf-8") as f:
    print(f.read())
```

9 Modules et packages en Python

9.1 Définition

Un **module** est un fichier Python contenant des fonctions, classes ou variables réutilisables. Un **package** est un ensemble organisé de modules regroupés dans un dossier, souvent avec un fichier `__init__.py`.

Python propose une large bibliothèque standard, et il est possible d'installer des bibliothèques externes avec `pip`.

—

9.2 Importer un module

```python
import math

print(math.sqrt(16))      # racine carrée
print(math.pi)            # constante pi
```

On peut aussi importer uniquement certaines fonctions :

```python
from math import sin, cos

print(sin(0))
print(cos(0))
```

Ou encore donner un alias pour simplifier l'écriture :

```python
import numpy as np

A = np.array([1, 2, 3])
print(A * 2)
```

—

9.3 Modules utiles de la bibliothèque standard

— `math` : fonctions mathématiques,
— `random` : génération aléatoire,
— `statistics` : calculs statistiques,
— `os` : gestion des fichiers et dossiers,

— `glob` : recherche de fichiers avec motifs,
— `datetime` : gestion des dates et heures.

```
import random, statistics

liste = [1, 2, 3, 4, 5]

print(random.choice(liste))       # élément aléatoire
print(random.sample(liste, 3))    # échantillon sans remise
print(statistics.mean(liste))     # moyenne
```

—

9.4 Création d'un module personnalisé

Il est possible d'écrire ses propres modules. Par exemple, créer un fichier `utilitaires.py` :

```
# fichier utilitaires.py
def salutation():
    return "Bonjour depuis le module utilitaires"
```

Puis l'utiliser dans un autre script :

```
import utilitaires

print(utilitaires.salutation())
```

—

9.5 Installation de packages externes

Les bibliothèques externes s'installent avec `pip`. Exemple pour installer NumPy :

```
pip install numpy
```

Puis utilisation :

```
import numpy as np

A = np.arange(0, 10, 2)
print(A)
```

—

9.6 Exercice pratique

1. Importer le module `math` et calculer la valeur de $\sin(\pi/4)$. 2. Générer un nombre entier aléatoire entre 1 et 100 avec `random`. 3. Créer un module `calculs.py` contenant une fonction `cube(n)` et l'utiliser dans un script.

```
import math, random

print("sin(pi/4) =", math.sin(math.pi/4))
print("Nombre aléatoire :", random.randint(1, 100))

```

```python
# fichier calculs.py
def cube(n):
    return n**3

# fichier principal
import calculs
print(calculs.cube(3))
```

10 NumPy : Calcul scientifique avec Python

10.1 Introduction à NumPy

`NumPy` (Numerical Python) est une bibliothèque fondamentale pour le calcul scientifique. Elle fournit :

- le type `ndarray`, tableau multidimensionnel efficace,
- des fonctions rapides pour le calcul vectorisé,
- des outils d'algèbre linéaire, statistiques et manipulation de données.
-

10.2 Création de tableaux

```python
import numpy as np

# À partir d'une liste Python
A = np.array([1, 2, 3, 4])
print(A)

# Tableaux multidimensionnels
B = np.array([[1, 2, 3], [4, 5, 6]])
print(B)

# Tableaux spéciaux
zeros = np.zeros((3, 3))
ones = np.ones((2, 4))
full = np.full((2, 2), 7)
identite = np.eye(3)

print(zeros)
print(identite)
```

-

10.3 Génération de séquences numériques

```python
# Arange : intervalle
x = np.arange(0, 10, 2)   # [0 2 4 6 8]

# Linspace : valeurs régulièrement espacées
y = np.linspace(0, 1, 5)  # [0.   0.25 0.5  0.75 1. ]
```

10.4 Dimensions et formes

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

print(A.shape)   # dimensions (2, 3)
print(A.ndim)    # nombre d'axes (2)
print(A.size)    # nombre total d'éléments (6)
```

10.5 Indexation et slicing

```python
A = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])

print(A[0, 1])      # élément ligne 0, colonne 1
print(A[:, 1])      # toutes les lignes, colonne 1
print(A[1, :])      # ligne 1 complète
print(A[0:2, 0:2]) # sous-matrice
```

10.6 Broadcasting

Le **broadcasting** permet d'effectuer des opérations entre tableaux de formes différentes, tant que les dimensions sont compatibles.

```python
A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.array([10, 20, 30])

print(A + B)
# [[11 22 33]
#  [14 25 36]]
```

10.7 Opérations mathématiques

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

print(np.sqrt(A))
print(np.exp(A))
print(np.log(A))
print(np.sin(A))
```

10.8 Statistiques de base

```
A = np.random.randint(0, 10, (3, 4))

print("Moyenne :", A.mean())
print("Variance :", np.var(A))
print("Écart-type :", np.std(A))
print("Somme par colonne :", A.sum(axis=0))
print("Somme par ligne :", A.sum(axis=1))
```

—

10.9 Algèbre linéaire

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

print("Produit matriciel :")
print(A.dot(B))

# Déterminant et inverse
print("Determinant :", np.linalg.det(A))
print("Inverse :\n", np.linalg.inv(A))

# Valeurs propres et vecteurs propres
valeurs, vecteurs = np.linalg.eig(A)
print("Valeurs propres :", valeurs)
print("Vecteurs propres :\n", vecteurs)
```

—

10.10 Gestion des valeurs manquantes (NaN)

```
A = np.array([1, 2, np.nan, 4])

print("Moyenne classique :", np.mean(A))      # NaN
print("Moyenne corrigée :", np.nanmean(A))    # ignore les NaN
print("Nombre de NaN :", np.isnan(A).sum())
```

—

10.11 Exercice pratique

1. Créer une matrice aléatoire 5×5 d'entiers entre 0 et 20. 2. Remplacer tous les éléments < 10 par 0. 3. Calculer la moyenne de chaque ligne.

```
np.random.seed(0)
M = np.random.randint(0, 20, (5, 5))

M[M < 10] = 0
print(M)
```

```
6
7 print("Moyenne par ligne :", M.mean(axis=1))
```

11 Visualisation des données avec Matplotlib

11.1 Introduction

La visualisation des données transforme des chiffres en graphiques pour révéler *tendances*, *relations* et *structures*. Avec **Matplotlib**, on produit des tracés 2D/3D, personnalise styles et annote les figures.

11.2 Tracé de courbes avec `plot`

`plot()` relie des points (x_i, y_i) pour visualiser des fonctions ou séries temporelles. Paramètres clés : `c` (couleur), `lw` (épaisseur), `ls` (style), `label` (légende).

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  x = np.linspace(0, 2, 10)
5  y = x**2
6
7  plt.figure(figsize=(5,5))
8  plt.plot(x, y, c="red", lw=3, ls="--", label="quadratique")
9  plt.plot(x, x**3, label="cubique")
10 plt.title("Courbes quadratique et cubique")
11 plt.xlabel("abscisses")
12 plt.ylabel("ordonnées")
13 plt.legend()
14 plt.show()
```

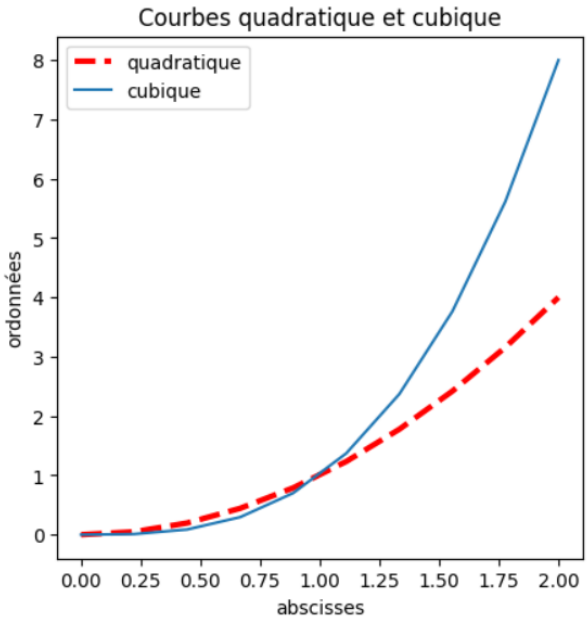

FIGURE 1 – Exemple de tracés avec `plot()`.

11.3 Nuage de points (`scatter`)

Représente la relation entre deux variables continues. Permet de détecter corrélations, clusters et valeurs aberrantes.

```
plt.scatter(x, y, c="blue", marker="o", alpha=0.8, label="points")
plt.title("Nuage de points")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```

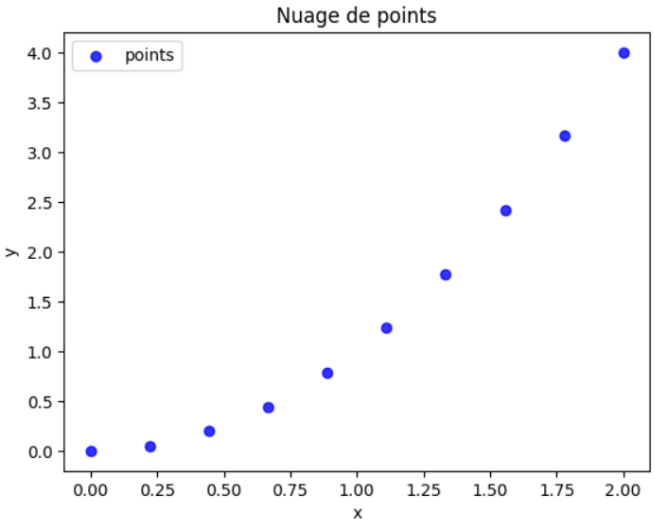

FIGURE 2 – Nuage de points avec `scatter()`.

11.4 Plusieurs graphiques dans une figure

`subplot(rows, cols, index)` découpe la figure en grille pour comparer plusieurs tracés.

```
plt.subplot(2, 2, 1)
plt.plot(x, y, c="red")

plt.subplot(2, 2, 2)
plt.plot(x, x**2, c="blue")

plt.subplot(2, 2, 3)
plt.plot(x, np.sin(x), c="green")

plt.subplot(2, 2, 4)
plt.plot(x, np.cos(x), c="orange")

plt.suptitle("Multiples sous-graphiques")
plt.tight_layout()
plt.show()
```

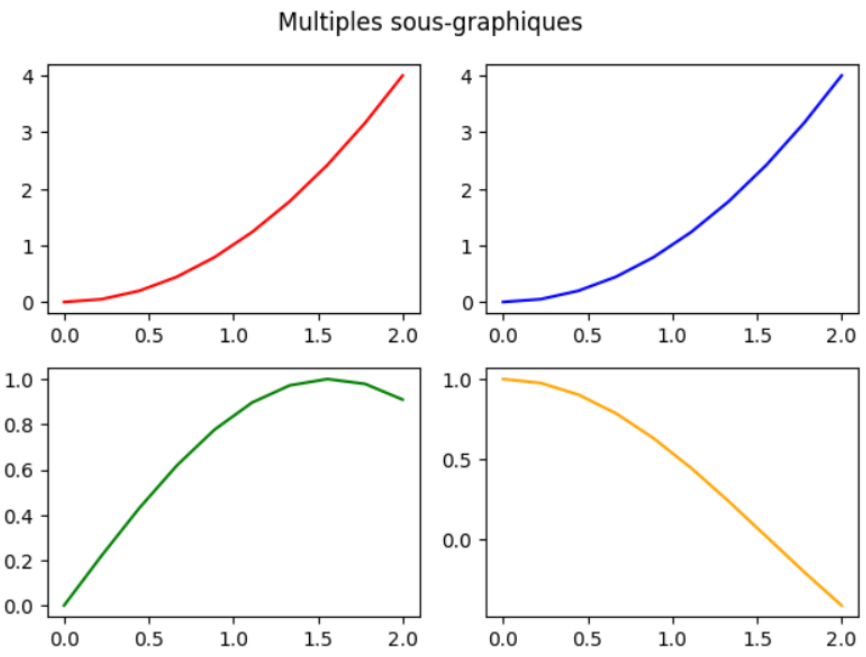

FIGURE 3 – Comparaison de tracés via `subplot()`.

11.5 Interface orientée objet

L'API orientée objet (`fig`, `ax`) donne un contrôle fin (axes, titres, ticks, annotations).

```
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title("Approche orientée objet")
ax.set_xlabel("x")
ax.set_ylabel("y")
plt.show()
```

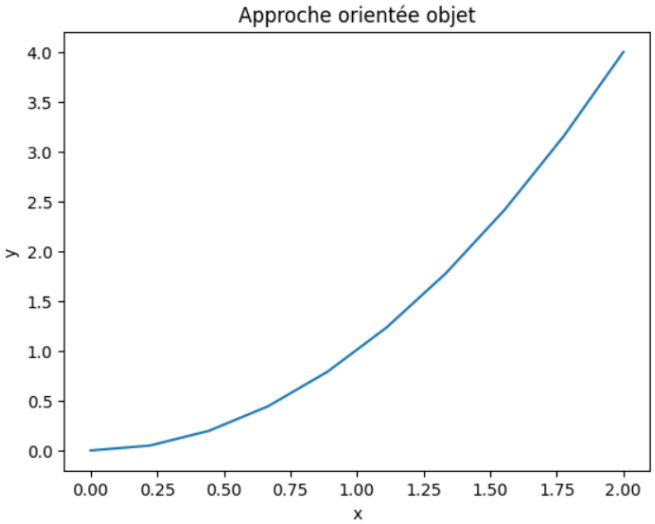

FIGURE 4 – Tracé avec l'API orientée objet.

Sous-figures partageant l'axe x :

```
fig, ax = plt.subplots(2, 1, sharex=True, figsize=(6,5))
ax[0].plot(x, y, label="x^2"); ax[0].legend()
ax[1].plot(x, np.sin(x), label="sin(x)"); ax[1].legend()
fig.suptitle("Sous-figures avec abscisses partagées")
plt.tight_layout()
plt.show()
```

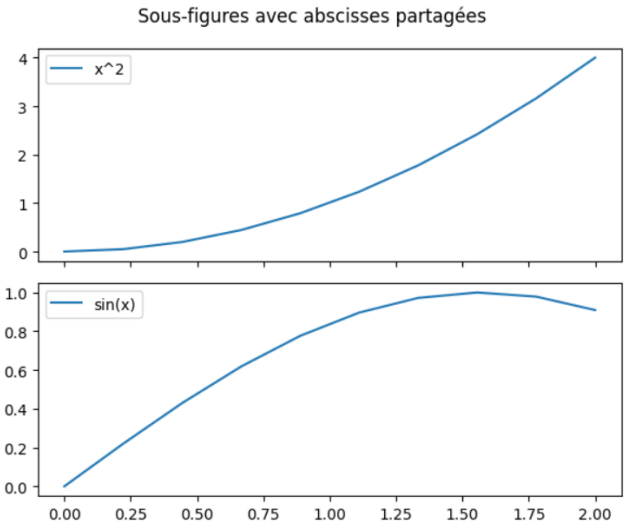

FIGURE 5 – Sous-figures (`subplots`) avec `sharex=True`.

11.6 Histogrammes

Un histogramme approxime la distribution d'une variable continue en comptant les observations par *bins*.

```
data = np.random.randn(1000)  # échantillon gaussien

plt.hist(data, bins=30, color="skyblue", edgecolor="black")
plt.title("Histogramme d'une loi normale")
plt.xlabel("Valeurs")
plt.ylabel("Fréquence")
plt.grid(alpha=.3)
plt.show()
```

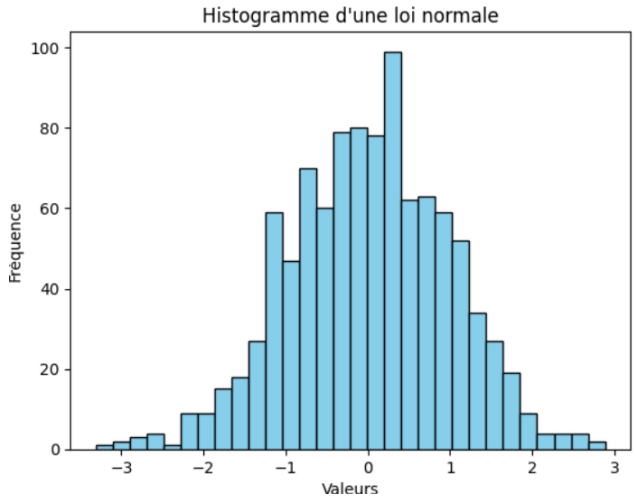

FIGURE 6 – Distribution univariée via `hist()`.

11.7 Histogramme 2D

hist2d() : estime la densité conjointe de deux variables (carte de chaleur des fréquences).

```
np.random.seed(0)
x = np.random.randn(1000)      # 1000 valeurs pour x
y = np.random.randn(1000)      # 1000 valeurs pour y

plt.hist2d(x, y, bins=30, cmap="Blues")
plt.colorbar(label="Comptes")
plt.title("Histogramme 2D")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

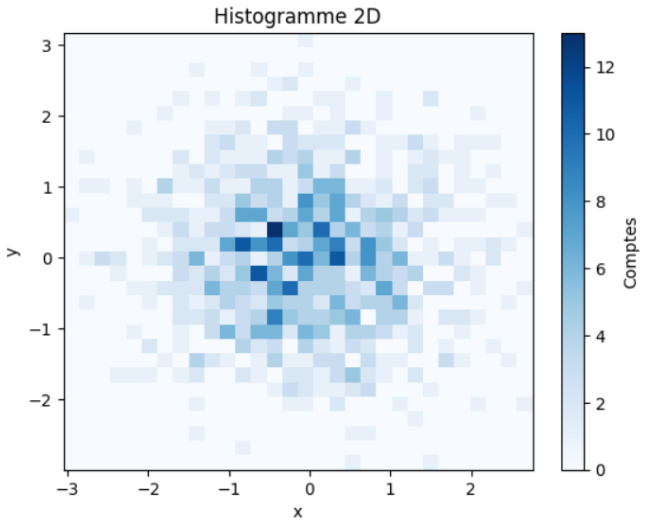

FIGURE 7 – Histogramme 2D représentant la distribution conjointe de deux variables aléatoires.

11.8 Courbes de niveau (`contour`)

Représente $z = f(x, y)$ par des lignes de même valeur (*isovaleurs*). Très utile pour visualiser des surfaces (p. ex. fonctions de coût).

```
X = np.linspace(-5, 5, 80)
Y = np.linspace(-5, 5, 80)
X, Y = np.meshgrid(X, Y)
Z = np.sin(np.sqrt(X**2 + Y**2))

cs = plt.contour(X, Y, Z, levels=20, cmap="viridis")
plt.clabel(cs, inline=True, fontsize=8)
plt.colorbar(label="z")
plt.xlabel("X"); plt.ylabel("Y")
plt.title("Courbes de niveau")
plt.show()
```

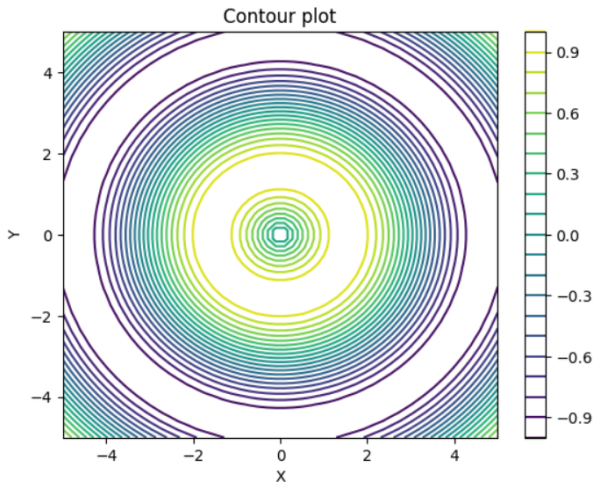

FIGURE 8 – Courbes de niveau pour $z = f(x, y)$.

11.9 Graphiques 3D

Les surfaces 3D aident à explorer des phénomènes dépendant de deux variables (topographie, surfaces de décision).

```
from mpl_toolkits.mplot3d import Axes3D  # optionnel en versions récentes

f = lambda X, Y: np.sin(X) + np.cos(X + Y)
X = np.linspace(0, 5, 100)
Y = np.linspace(0, 5, 100)
X, Y = np.meshgrid(X, Y)
Z = f(X, Y)

fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111, projection="3d")
ax.plot_surface(X, Y, Z, cmap="viridis", linewidth=0, antialiased=True)
ax.set_xlabel("X"); ax.set_ylabel("Y"); ax.set_zlabel("Z")
ax.set_title("Surface 3D")
```

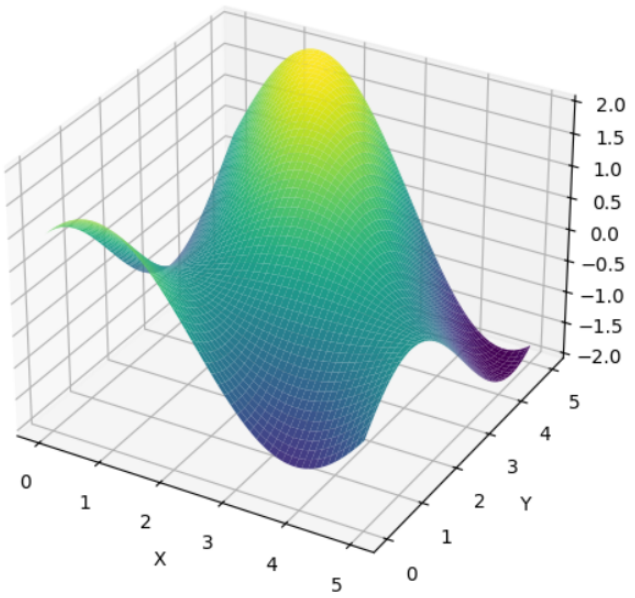

FIGURE 9 – Surface 3D d'une fonction $z = f(x, y)$.

11.10 Matrice de corrélation avec `imshow`

`imshow()` affiche une matrice comme une image. La matrice de corrélation révèle les relations linéaires entre variables (positives, négatives, nulles).

```
data = np.random.randn(200, 5)    # 5 variables
corr = np.corrcoef(data.T)

plt.imshow(corr, cmap="coolwarm", vmin=-1, vmax=1, interpolation="none")
plt.colorbar(label="Corrélation")
plt.title("Matrice de corrélation")
plt.xticks(range(5), [f"X{i}" for i in range(1,6)])
plt.yticks(range(5), [f"X{i}" for i in range(1,6)])
plt.show()
```

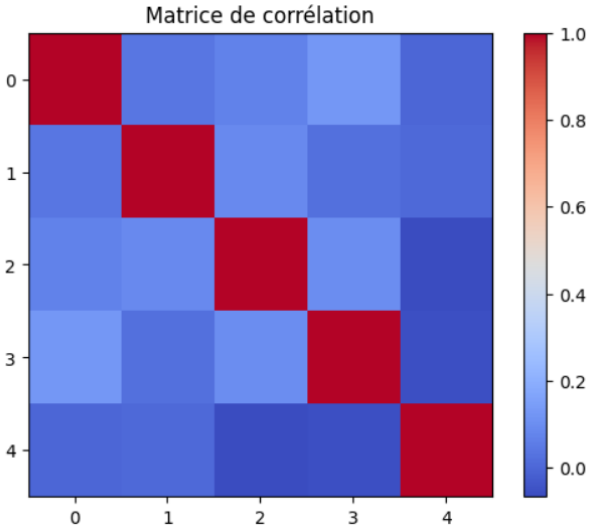

FIGURE 10 – Visualisation d'une matrice de corrélation avec `imshow()`.