Programmation Scientifique

Notes de Cours

Licence L2 — 2025–2026

Yaë Ulrich Gaba

« Parler est bon, montrer est mieux, mais rien ne vaut faire. »
— Linus Torvalds

25 mars 2026

Table des matières

Préface

La programmation est devenue un outil indispensable pour tout scientifique. Que vous étudiiez les mathématiques, la physique, la chimie ou la biologie, savoir écrire un programme vous permet d'automatiser des calculs, de visualiser des données, de simuler des phénomènes et de tester des hypothèses.

Ce cours est conçu pour des étudiants de Licence **sans aucune expérience préalable en programmation**. Nous partons de zéro et progressons pas à pas, en illustrant chaque concept par des exemples issus des sciences. L'objectif est de vous rendre autonome pour résoudre des problèmes scientifiques avec Python (et une introduction à R).

Prérequis. Aucun en programmation. Des notions de mathématiques de lycée (fonctions, équations) sont utiles mais pas indispensables.

Philosophie. Chaque chapitre suit le même schéma :

1. Explication du concept avec une analogie scientifique.
2. Code commenté avec sa sortie.
3. Erreurs courantes (« Ce qui peut mal tourner »).
4. Mini-projet scientifique.
5. Exercices gradués.

Références.

- Downey — *Think Python*, O'Reilly (disponible gratuitement en ligne).
- Langtangen — *A Primer on Scientific Programming with Python*, Springer.
- VanderPlas — *Python Data Science Handbook*, O'Reilly (gratuit en ligne).

Chapitre 0 — Installation et Configuration

Avant de commencer à programmer, il faut préparer votre environnement de travail. Ce chapitre vous guide pas à pas.

Installer Python

Intuition

Python est un **langage de programmation** gratuit, utilisé par des millions de scientifiques dans le monde. C'est comme une calculatrice surpuissante qui comprend vos instructions écrites en anglais.

Méthode recommandée : Anaconda

Anaconda est une distribution Python qui inclut toutes les bibliothèques scientifiques dont nous aurons besoin.

1. Rendez-vous sur `https://www.anaconda.com/download`
2. Téléchargez la version pour votre système (Windows, macOS ou Linux).
3. Lancez l'installateur et suivez les instructions par défaut.
4. Vérifiez l'installation en ouvrant un terminal et en tapant :

Terminal

```
python --version
```

Sortie

```
Python 3.12.4
```

Alternative légère : Python seul + pip

Si vous préférez une installation minimale :

1. Téléchargez Python depuis `https://www.python.org/downloads/`

2. Cochez « Add Python to PATH » lors de l'installation.

3. Installez les bibliothèques nécessaires :

Terminal

```
pip install numpy matplotlib pandas scipy seaborn scikit-learn jupyter
```

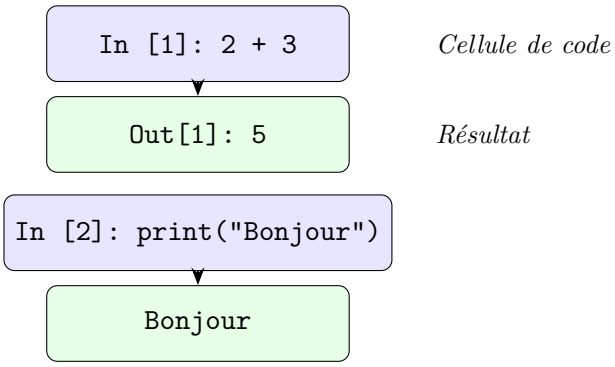
Jupyter Notebook

Jupyter Notebook est un environnement interactif où vous écrivez du code, voyez les résultats et ajoutez des notes — comme un cahier de laboratoire numérique.

Terminal — Lancer Jupyter

```
jupyter notebook
```

Votre navigateur s'ouvre automatiquement. Cliquez sur **New** → **Python 3** pour créer un nouveau notebook.



Bonne pratique

Utilisez **Shift + Entrée** pour exécuter une cellule et passer à la suivante. Utilisez **Ctrl + Entrée** pour exécuter sans changer de cellule.

Éditeurs de texte

Pour écrire des scripts Python (fichiers `.py`), vous pouvez utiliser :

- **VS Code** (recommandé) : gratuit, avec extension Python.
- **Spyder** : inclus dans Anaconda, conçu pour les scientifiques.
- **PyCharm** : version Community gratuite.

Vérification de l'installation

Python — Premier test

```python
import numpy as np
import matplotlib.pyplot as plt

print("Installation réussie !")
print(f"NumPy version : {np.__version__}")

# Petit test : tracer sin(x)
x = np.linspace(0, 2 * np.pi, 100)
plt.plot(x, np.sin(x))
plt.title("sin(x) - Votre premier graphique !")
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.grid(True)
plt.savefig('test_installation.pdf')
plt.show()
print("Graphique sauvegardé !")
```

Sortie

```
Installation réussie !
NumPy version : 1.26.4
Graphique sauvegardé !
```

Si tout fonctionne, vous êtes prêt(e) à commencer le chapitre 1 !

Installer R (pour le chapitre 9)

1. Téléchargez R depuis https://cran.r-project.org/
2. Installez **RStudio Desktop** depuis https://posit.co/download/rstudio-desktop/
3. Vérifiez en ouvrant RStudio et en tapant :

Console R

```
R.version.string
```

Sortie

```
[1] "R version 4.4.1 (2024-06-14)"
```

Ce qui peut mal tourner

- « **Python n'est pas reconnu** » : vous avez oublié d'ajouter Python au PATH. Réinstallez en cochant l'option.

- « **ModuleNotFoundError** » : la bibliothèque n'est pas installée. Tapez `pip install nom_du_module`.
- **Permission refusée** : sur macOS/Linux, essayez `pip install --user nom_du_module`.

Chapitre 1

Introduction à la Programmation

1.1 Pourquoi programmer ?

Intuition

Un programme informatique est une **recette de cuisine** pour l'ordinateur. Tout comme une recette décrit étape par étape comment préparer un plat, un programme décrit étape par étape comment résoudre un problème.

En sciences, programmer permet de :

- **Automatiser** des calculs répétitifs (calculer 10 000 intégrales).
- **Visualiser** des données (tracer l'évolution d'une population).
- **Simuler** des phénomènes (mouvement d'un pendule, diffusion de chaleur).
- **Analyser** de grandes quantités de données (génomique, astronomie).

1.2 Pourquoi Python ?

Lisible
Syntaxe proche de l'anglais

Gratuit
Open source
Multiplateforme

Puissant
NumPy, SciPy
Machine Learning

Populaire
NASA, CERN
Google, Netflix

1.3 Votre premier programme

Python

```
print("Bonjour, futur(e) scientifique !")
```

Sortie

```
Bonjour, futur(e) scientifique !
```

Félicitations ! Vous venez d'écrire votre premier programme. La fonction `print()` affiche du texte à l'écran.

1.4 Python comme calculatrice

Python peut effectuer toutes les opérations mathématiques de base :

Python — Calculs de base

```python
# Addition et soustraction
print(3 + 5)
print(10 - 4)

# Multiplication et division
print(6 * 7)
print(15 / 4)      # Division réelle
print(15 // 4)     # Division entière
print(15 % 4)      # Reste (modulo)

# Puissance
print(2 ** 10)      # 2 puissance 10
```

Sortie

```
8
6
42
3.75
3
3
1024
```

Opérateurs arithmétiques

Opérateur	Signification	Exemple
`+`	Addition	`3 + 5` = 8
`-`	Soustraction	`10 - 4` = 6
`*`	Multiplication	`6 * 7` = 42
`/`	Division réelle	`15 / 4` = 3.75
`//`	Division entière	`15 // 4` = 3
`\%`	Modulo (reste)	`15 \% 4` = 3
`**`	Puissance	`2 ** 10` = 1024

1.5 Commentaires

Les **commentaires** sont des notes pour les humains. Python les ignore complètement.

Python

```python
# Ceci est un commentaire - Python l'ignore
print(2 + 3)   # On peut commenter en fin de ligne

# Vitesse de la lumière en m/s
c = 299792458
print(f"Vitesse de la lumière : {c} m/s")
```

Sortie

```
5
Vitesse de la lumière : 299792458 m/s
```

Bonne pratique

Commentez votre code pour expliquer **pourquoi** vous faites quelque chose, pas **ce que** vous faites. Un bon commentaire dit « Vitesse de la lumière en m/s », pas « On affecte 299792458 à c ».

1.6 Exemple scientifique : Énergie cinétique

Calculons l'énergie cinétique $E_c = \frac{1}{2}mv^2$ d'un objet :

Python — Énergie cinétique

```python
# Données
masse = 75.0        # kg (une personne)
vitesse = 10.0      # m/s (course rapide)

# Calcul de l'énergie cinétique
Ec = 0.5 * masse * vitesse ** 2

print(f"Masse    : {masse} kg")
print(f"Vitesse  : {vitesse} m/s")
print(f"Énergie  : {Ec} J")
```

Sortie

```
Masse    : 75.0 kg
Vitesse  : 10.0 m/s
Énergie  : 3750.0 J
```

1.7 Les erreurs : ne pas avoir peur !

Ce qui peut mal tourner

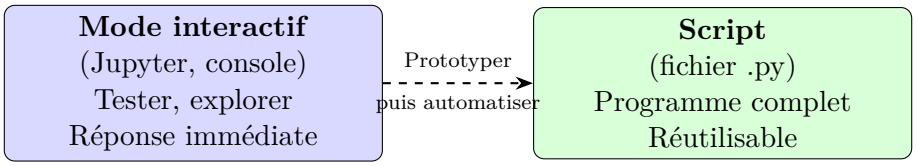
Les erreurs sont **normales** et font partie de l'apprentissage. Python vous aide en indiquant le type d'erreur et la ligne concernée.

Python — Erreurs courantes

```
# Erreur de syntaxe : parenthèse oubliée
# print("Bonjour"
# SyntaxError: '(' was never closed

# Erreur de nom : variable non définie
# print(x)
# NameError: name 'x' is not defined

# Erreur de type : opération impossible
# "abc" + 5
# TypeError: can only concatenate str to str

# Division par zéro
# 10 / 0
# ZeroDivisionError: division by zero
```

Règle d'or : lisez toujours le message d'erreur **de bas en haut**. La dernière ligne vous dit ce qui ne va pas.

1.8 Le mode interactif vs les scripts



1.9 Mini-projet : Conversions d'unités

Mini-projet scientifique

Écrivez un programme qui convertit des températures entre Celsius et Fahrenheit.
Formules : $F = \frac{9}{5}C + 32$ et $C = \frac{5}{9}(F - 32)$

Python — Convertisseur

```python
# Conversion Celsius -> Fahrenheit
celsius = 100
fahrenheit = (9/5) * celsius + 32
print(f"{celsius}°C = {fahrenheit}°F")

# Conversion Fahrenheit -> Celsius
fahrenheit = 72
celsius = (5/9) * (fahrenheit - 32)
print(f"{fahrenheit}°F = {celsius:.1f}°C")

# Table de conversion
print("\n--- Table de conversion ---")
print(f"{'°C':>6}  {'°F':>6}")
for c in range(0, 101, 10):
    f = (9/5) * c + 32
    print(f"{c:>6}  {f:>6.1f}")
```

Sortie

```
100°C = 212.0°F
72°F = 22.2°C

--- Table de conversion ---
    °C      °F
     0    32.0
    10    50.0
    20    68.0
    30    86.0
    40   104.0
    50   122.0
    60   140.0
    70   158.0
    80   176.0
    90   194.0
   100   212.0
```

1.10 Exercices

Exercice 1.1 ($\star$ — Guidé)**.** Calculez l'aire d'un cercle de rayon $r = 5$ en utilisant la formule $A = \pi r^2$. Utilisez `3.14159` pour π.

Exercice 1.2 ($\star$ — Guidé)**.** Calculez la distance entre deux points $(x_1, y_1) = (1, 2)$ et $(x_2, y_2) = (4, 6)$ avec la formule $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Indice : $\sqrt{x} = x^{0.5}$.

Exercice 1.3 ($\star\star$ — Indépendant)**.** La loi de la gravitation universelle donne la force entre deux masses : $F = G\frac{m_1 m_2}{r^2}$, avec $G = 6.674 \times 10^{-11}$ N·m^2/kg^2. Calculez la force

entre la Terre ($m_1 = 5.972 \times 10^{24}$ kg) et la Lune ($m_2 = 7.342 \times 10^{22}$ kg), distantes de $r = 3.844 \times 10^8$ m.

Exercice 1.4 ($\star\star\star$ — Défi scientifique)**.** Calculez la longueur d'onde de De Broglie $\lambda = h/(mv)$ pour un électron ($m = 9.109 \times 10^{-31}$ kg) se déplaçant à $v = 10^6$ m/s, avec $h = 6.626 \times 10^{-34}$ J·s. Comparez avec la taille d'un atome ($\sim 10^{-10}$ m).

Résumé du chapitre

- `print()` affiche du texte et des valeurs.
- Python connaît les opérations : `+`, `-`, `*`, `/`, `//`, `%`, `**`.
- Les commentaires commencent par `##`.
- Les erreurs sont normales — lisez le message de bas en haut.
- Un script `.py` est un programme réutilisable.

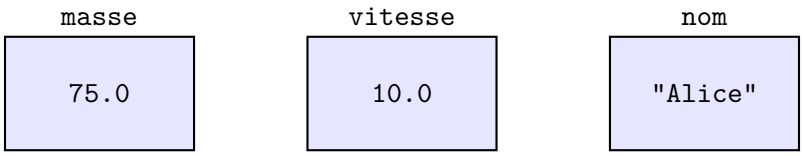
Chapitre 2

Variables, Types et Opérations de Base

2.1 Les variables : des boîtes avec des étiquettes

Intuition

Une **variable** est comme une boîte étiquetée dans laquelle on range une valeur. L'étiquette (le nom) permet de retrouver le contenu à tout moment.



Python — Créer des variables

```python
# Créer des variables
masse = 75.0          # nombre décimal (float)
age = 20              # nombre entier (int)
nom = "Alice"         # chaîne de caractères (str)
est_etudiant = True   # booléen (bool)

print(f"Nom : {nom}, Âge : {age}")
print(f"Masse : {masse} kg")
print(f"Étudiant : {est_etudiant}")
```

Sortie

```
Nom : Alice, Âge : 20
Masse : 75.0 kg
Étudiant : True
```

Attention

En Python, on ne « déclare » pas le type d'une variable. Python le déduit automatiquement. C'est le **typage dynamique**.

2.2 Les types de données

Définition 2.1 (Types fondamentaux)**.** Python possède quatre types fondamentaux :

- `int` — entier : `42`, `-7`, `0`
- `float` — décimal : `3.14`, `-0.5`, `6.022e23`
- `str` — chaîne : `"Bonjour"`, `'xyz'`
- `bool` — booléen : `True`, `False`

Python — Vérifier le type

```
x = 42
y = 3.14
z = "physique"
b = True

print(type(x))   # <class 'int'>
print(type(y))   # <class 'float'>
print(type(z))   # <class 'str'>
print(type(b))   # <class 'bool'>
```

Sortie

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

2.3 Notation scientifique

Python — Notation scientifique

```
# Constantes physiques en notation scientifique
c = 2.998e8         # vitesse de la lumière (m/s)
h = 6.626e-34       # constante de Planck (J·s)
Na = 6.022e23       # nombre d'Avogadro (mol⁻¹)
G = 6.674e-11       # constante gravitationnelle

print(f"c  = {c}")
print(f"h  = {h}")
print(f"Na = {Na}")
print(f"G  = {G}")
```

Sortie

```
c  = 299800000.0
h  = 6.626e-34
```

```
Na = 6.022e+23
G  = 6.674e-11
```

2.4 Conversion de types

Python — Conversions

```python
# int -> float
x = float(42)
print(x, type(x))        # 42.0 <class 'float'>

# float -> int (tronque !)
y = int(3.99)
print(y, type(y))        # 3 <class 'int'>

# str -> int ou float
temperature = int("25")
pi_approx = float("3.14")
print(temperature, pi_approx)

# Nombre -> str
message = "La réponse est " + str(42)
print(message)
```

Sortie

```
42.0 <class 'float'>
3 <class 'int'>
25 3.14
La réponse est 42
```

2.5 Opérations sur les chaînes

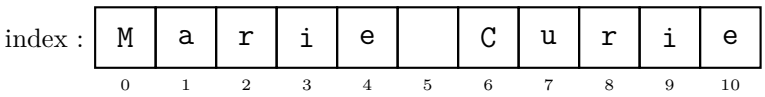
Python — Chaînes de caractères

```python
prenom = "Marie"
nom = "Curie"

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

# Répétition
print("-" * 30)

# Longueur
```

```python
print(f"Longueur : {len(nom_complet)}")

# Accès par index (commence à 0 !)
print(f"Première lettre : {nom_complet[0]}")
print(f"Dernière lettre : {nom_complet[-1]}")

# Méthodes utiles
print(nom_complet.upper())
print(nom_complet.lower())
print(nom_complet.replace("Curie", "Skłodowska-Curie"))
```

Sortie

```
Marie Curie
------------------------------
Longueur : 11
Première lettre : M
Dernière lettre : e
MARIE CURIE
marie curie
Marie Skłodowska-Curie
```



2.6 Les f-strings (chaînes formatées)

Python — f-strings

```python
element = "Hydrogène"
masse_atomique = 1.008
numero = 1

# f-string : insérer des variables dans du texte
print(f"L'élément {element} (Z={numero}) a une masse de {masse_atomique}
↪  u")

# Formatage des nombres
pi = 3.141592653589793
print(f"  = {pi:.2f}")      # 2 décimales
print(f"  = {pi:.6f}")      # 6 décimales
print(f"  = {pi:.2e}")      # notation scientifique

# Alignement
for element, masse in [("H", 1.008), ("C", 12.011), ("O", 15.999)]:
    print(f"{element:<5} {masse:>8.3f} u")
```

Sortie

```
L'élément Hydrogène (Z=1) a une masse de 1.008 u
  = 3.14
  = 3.141593
  = 3.14e+00
H         1.008 u
C        12.011 u
O        15.999 u
```

2.7 Opérateurs de comparaison et booléens

Python — Comparaisons

```python
x = 10
print(x > 5)        # True
print(x == 10)      # True (égalité)
print(x != 7)       # True (différent)
print(x >= 15)      # False

# Opérateurs logiques
a = True
b = False
print(a and b)      # False
print(a or b)       # True
print(not a)        # False
```

Sortie

```
True
True
True
False
False
True
False
```

Opérateurs de comparaison

Opérateur	Signification	Exemple
`==`	Égal à	`5 == 5` → `True`
`!=`	Différent de	`5 != 3` → `True`
`<`, `>`	Inférieur, supérieur	`3 < 5` → `True`
`<=`, `>=`	Inf. ou égal, sup. ou égal	`5 >= 5` → `True`
`and`	ET logique	`True and False` → `False`
`or`	OU logique	`True or False` → `True`
`not`	NON logique	`not True` → `False`

2.8 L'entrée utilisateur

Python — input

```
# input() retourne toujours une chaîne !
# nom = input("Votre nom : ")
# age = int(input("Votre âge : "))
# print(f"Bonjour {nom}, vous avez {age} ans.")

# Simulation (pour le cours)
nom = "Alice"
age = 20
print(f"Bonjour {nom}, vous avez {age} ans.")
```

Sortie

```
Bonjour Alice, vous avez 20 ans.
```

2.9 Ce qui peut mal tourner

Ce qui peut mal tourner

Erreurs fréquentes

```
# 1. Oublier les guillemets pour les chaînes
# nom = Alice        # NameError: name 'Alice' is not defined
nom = "Alice"        # Correct

# 2. Confondre = (affectation) et == (comparaison)
x = 5                # affecte 5 à x
# x == 5             # compare x à 5, retourne True

# 3. Diviser un entier et attendre un flottant
print(7 / 2)         # 3.5 (en Python 3, ok)
print(7 // 2)        # 3 (division entière !)

# 4. Concaténer un nombre et une chaîne
# print("J'ai " + 20 + " ans")  # TypeError
print("J'ai " + str(20) + " ans")  # OK
print(f"J'ai {20} ans")            # Encore mieux
```

2.10 Mini-projet : Fiche d'identité d'un atome

Mini-projet scientifique

Créez un programme qui affiche la fiche d'identité d'un atome.

Python

```
# Fiche d'identité : Carbone
element = "Carbone"
symbole = "C"
numero_atomique = 6
masse_atomique = 12.011  # u
electronegativite = 2.55  # Pauling

print("=" * 35)
print(f"  FICHE D'IDENTITÉ : {element}")
print("=" * 35)
print(f"  Symbole           : {symbole}")
print(f"  Numéro atomique   : {numero_atomique}")
print(f"  Masse atomique    : {masse_atomique} u")
print(f"  Électronégativité : {electronegativite}")
print(f"  Nombre de protons : {numero_atomique}")
print(f"  Nombre d'électrons: {numero_atomique}")
print("=" * 35)
```

Sortie

```
==================================
  FICHE D'IDENTITÉ : Carbone
==================================
  Symbole           : C
  Numéro atomique   : 6
  Masse atomique    : 12.011 u
  Électronégativité : 2.55
  Nombre de protons : 6
  Nombre d'électrons: 6
==================================
```

2.11 Exercices

Exercice 2.1 (⋆ — Guidé)**.** Créez des variables pour stocker la constante de Boltzmann ($k_B = 1.381 \times 10^{-23}$ J/K) et une température $T = 300$ K. Calculez l'énergie thermique $E = k_B T$ et affichez le résultat.

Exercice 2.2 (⋆ — Guidé)**.** Créez une variable `rayon = 6371` (rayon de la Terre en km). Calculez et affichez la circonférence ($C = 2\pi r$) et la surface ($S = 4\pi r^2$) de la Terre.

Exercice 2.3 (⋆⋆ — Indépendant)**.** Écrivez un programme qui demande à l'utilisateur une distance en kilomètres et la convertit en miles (1 km = 0.621 miles), en mètres et en centimètres.

Exercice 2.4 (⋆⋆ — Indépendant)**.** Créez un programme qui calcule l'IMC (Indice de Masse Corporelle) : IMC $= \frac{\text{masse (kg)}}{\text{taille (m)}^2}$. Affichez le résultat avec 1 décimale.

Exercice 2.5 (⋆⋆⋆ — Défi scientifique)**.** La formule de l'énergie d'un photon est $E = hf = hc/\lambda$. Calculez l'énergie d'un photon de lumière rouge ($\lambda = 700$ nm) et de lumière bleue ($\lambda = 450$ nm). Exprimez les résultats en joules et en électron-volts (1 eV $= 1.602 \times 10^{-19}$ J).

Résumé du chapitre

- Une variable stocke une valeur : `x = 42`.
- Types : `int`, `float`, `str`, `bool`.
- `type(x)` donne le type, `int()`/`float()`/`str()` convertissent.
- Notation scientifique : `6.022e23` $= 6.022 \times 10^{23}$.
- f-strings : `f"pi = \{pi:.2f\}"` pour le formatage.
- Opérateurs de comparaison : `==`, `!=`, `<`, `>`, `and`, `or`.

Chapitre 3

Structures de Contrôle — Conditions et Boucles

Intuition

Imaginez une recette de cuisine : « Si la pâte est trop liquide, ajoutez de la farine ; sinon, enfournez pendant 30 minutes. » Un programme fonctionne exactement de la même manière : il prend des *décisions* et *répète* des étapes selon des conditions. Les structures de contrôle sont le cœur de tout algorithme scientifique.

3.1 Les conditions : `if`, `elif`, `else`

En science, on classifie souvent selon des seuils. Considérons les *transitions de phase* de l'eau :

Python

```
temperature = -5  # en degrés Celsius

if temperature < 0:
    etat = "solide"
elif temperature < 100:
    etat = "liquide"
else:
    etat = "gazeux"

print(f"À {temperature}°C, l'eau est {etat}.")
```

Sortie

```
À -5°C, l'eau est solide.
```

Remarque 3.1. Python évalue les conditions *dans l'ordre*. Dès qu'une condition est vraie, le bloc correspondant est exécuté et les suivants sont ignorés.

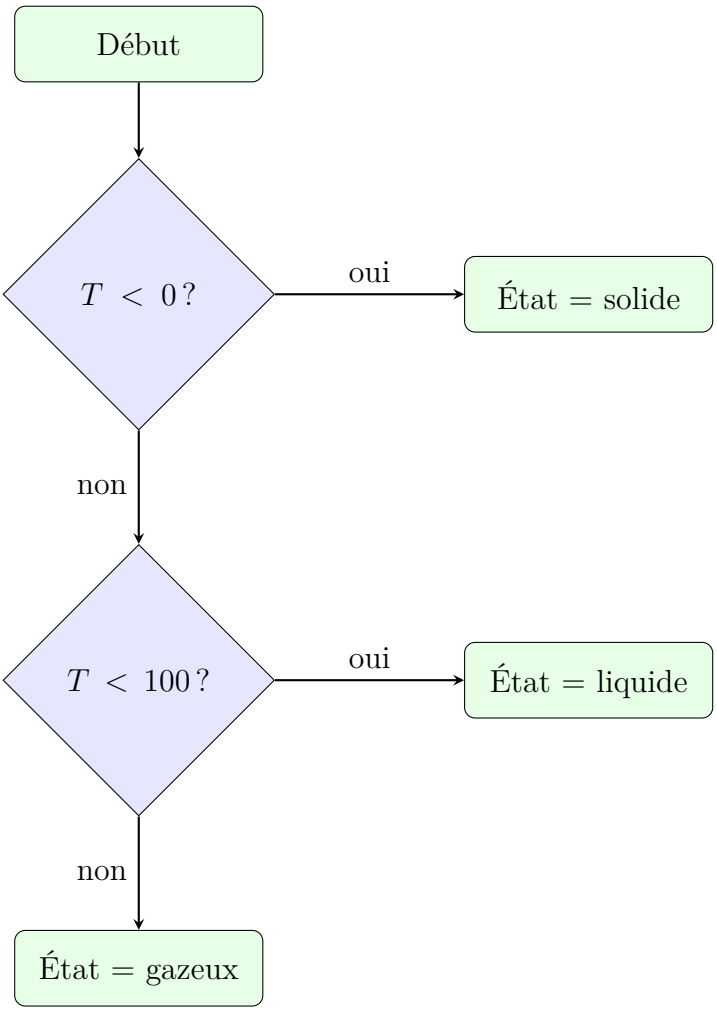


FIG. 3.1 : Organigramme d'une structure `if/elif/else` pour les transitions de phase.

3.1.1 Opérateurs de comparaison et opérateurs logiques

Définition 3.2 (Opérateurs de comparaison)**.** Les opérateurs `==`, `!=`, `<`, `>`, `<=`, `>=` renvoient un booléen (`True` ou `False`). On peut combiner des conditions avec `and`, `or`, `not`.

Python

```
pH = 3.2

if pH < 7 and pH >= 0:
    print("Solution acide")
elif pH == 7:
    print("Solution neutre")
elif pH > 7 and pH <= 14:
    print("Solution basique")
else:
    print("Valeur de pH invalide")
```

Sortie

```
Solution acide
```

3.2 La boucle `while`

La boucle `while` répète un bloc *tant qu'une condition est vraie*. En sciences, on l'utilise souvent pour des algorithmes de *convergence*.

Exemple 3.3 (Convergence d'une suite)**.** La suite $u_{n+1} = \frac{1}{2}\left(u_n + \frac{a}{u_n}\right)$ converge vers $\sqrt{a}$ (méthode de Héron).

Python

```
a = 2.0
u = 1.0        # valeur initiale
tolerance = 1e-10
iterations = 0

while abs(u**2 - a) > tolerance:
    u = 0.5 * (u + a / u)
    iterations += 1

print(f"sqrt({a})  {u}")
print(f"Convergence en {iterations} itérations")
```

Sortie

```
sqrt(2.0)  1.4142135623730951
Convergence en 4 itérations
```

Attention

Une boucle `while` dont la condition ne devient *jamais* fausse tourne à l'infini ! Ajoutez toujours un garde-fou :

```
max_iter = 10000
while condition and iterations < max_iter:
    ...
```

3.3 La boucle `for` et `range()`

La boucle `for` parcourt un *itérable* (liste, chaîne, `range`, etc.).

Python

```python
# Calculer la somme des carrés de 1 à 10
somme = 0
for i in range(1, 11):
    somme += i**2

print(f"Somme des carrés de 1 à 10 = {somme}")
# Vérification : n(n+1)(2n+1)/6
print(f"Formule : {10 * 11 * 21 // 6}")
```

Sortie

```
Somme des carrés de 1 à 10 = 385
Formule : 385
```

Définition 3.4 (`range(start, stop, step)`). `range(a, b, s)` génère les entiers de `a` à `b-1` avec un pas de `s`. Attention : la borne supérieure `b` est *exclue*.

Python

```python
# Afficher les multiples de 3 entre 0 et 15
for n in range(0, 16, 3):
    print(n, end=" ")
```

Sortie

```
0 3 6 9 12 15
```

3.4 Boucles imbriquées

Python

```python
# Table de multiplication (extrait)
for i in range(1, 4):
    for j in range(1, 4):
        print(f"{i} × {j} = {i*j:2d}", end="   ")
    print()  # retour à la ligne
```

Sortie

```
1 × 1 =  1   1 × 2 =  2   1 × 3 =  3
2 × 1 =  2   2 × 2 =  4   2 × 3 =  6
3 × 1 =  3   3 × 2 =  6   3 × 3 =  9
```

3.5 `break` et `continue`

Python

```python
# Trouver le premier nombre premier > 50
n = 51
while True:
    est_premier = True
    for d in range(2, int(n**0.5) + 1):
        if n % d == 0:
            est_premier = False
            break       # inutile de tester d'autres diviseurs
    if est_premier:
        print(f"Premier nombre premier > 50 : {n}")
        break           # sortir de la boucle while
    n += 1
```

Sortie

```
Premier nombre premier > 50 : 53
```

Python

```python
# Sauter les valeurs négatives dans des mesures
mesures = [2.3, -1.0, 4.5, -0.3, 3.8, 7.1]
somme = 0
compteur = 0
for m in mesures:
    if m < 0:
        continue   # passer à l'itération suivante
    somme += m
    compteur += 1

print(f"Moyenne (valeurs positives) = {somme/compteur:.2f}")
```

Sortie

```
Moyenne (valeurs positives) = 4.43
```

3.6 Erreurs courantes

Ce qui peut mal tourner

1. **Erreur d'indentation** : Python utilise l'indentation pour délimiter les blocs. Mélanger espaces et tabulations provoque une `IndentationError`. Règle : utilisez toujours **4 espaces**.

2. **Boucle infinie** : oublier d'incrémenter la variable de contrôle dans un `while`

mène à une boucle qui ne s'arrête jamais.

```
# ERREUR : i ne change jamais !
i = 0
while i < 10:
    print(i)    # affiche 0 indéfiniment
```

3. **Erreur de décalage (*off-by-one*)** : `range(1, 10)` produit les entiers de 1 à **9**, pas 10 ! Pour inclure 10, écrivez `range(1, 11)`.
4. **Utiliser `=` au lieu de `==`** : `=` est l'affectation, `==` est la comparaison. Écrire `if x = 5` provoque une `SyntaxError`.

3.7 Mini-projet : Désintégration radioactive

Mini-projet scientifique

La loi de désintégration radioactive donne le nombre d'atomes restants :

$$N(t) = N_0 \times 0{,}5^{\,t/t_{1/2}}$$

où N_0 est le nombre initial d'atomes et $t_{1/2}$ la demi-vie.
Objectif : simuler la désintégration du carbone-14 ($t_{1/2} = 5730$ ans) et afficher le nombre d'atomes restants chaque 1000 ans.

Python

```
N0 = 1_000_000        # nombre initial d'atomes
demi_vie = 5730       # demi-vie du C-14 en années
seuil = N0 * 0.01     # arrêter quand il reste < 1%

t = 0
N = N0
print(f"{'Temps (ans)':>12} {'N restants':>12} {'% restant':>10}")
print("-" * 36)

while N > seuil:
    pourcentage = N / N0 * 100
    print(f"{t:>12} {N:>12.0f} {pourcentage:>9.1f}%")
    t += 1000
    N = N0 * 0.5 ** (t / demi_vie)

print(f"\nAprès {t} ans, il reste moins de 1% des atomes.")
```

Sortie

```
 Temps (ans)   N restants  % restant
-------------------------------------
           0      1000000     100.0%
        1000       886076      88.6%
        2000       785128      78.5%
        3000       695685      69.6%
        4000       616441      61.6%
        5000       546274      54.6%
         ...
       37000        12055       1.2%

Après 38000 ans, il reste moins de 1% des atomes.
```

3.8 Exercices

Exercice 3.1 ($\star$ — Classification de l'IMC)**.** Écrivez un programme qui demande le poids (kg) et la taille (m) d'une personne, calcule l'IMC (IMC = poids/taille^2) et affiche la catégorie : sous-poids ($< 18{,}5$), normal (18,5–25), surpoids (25–30), obésité (≥ 30).

Exercice 3.2 ($\star$ — Compte à rebours)**.** Utilisez une boucle `while` pour afficher un compte à rebours de 10 à 0, puis affichez « Décollage ! ».

Exercice 3.3 ($\star\star$ — Suite de Fibonacci)**.** Calculez les 20 premiers termes de la suite de Fibonacci ($F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$) avec une boucle `for`. Vérifiez que le rapport F_n/F_{n-1} converge vers le nombre d'or $\varphi \approx 1{,}618$.

Exercice 3.4 ($\star\star$ — Crible d'Ératosthène)**.** Utilisez des boucles imbriquées pour trouver tous les nombres premiers inférieurs à 100.

Exercice 3.5 ($\star\star\star$ — Simulation de marche aléatoire)**.** Simulez une marche aléatoire 1D : à chaque pas, la particule se déplace de $+1$ ou -1 avec probabilité égale. Après $N = 1000$ pas, affichez la position finale. Répétez 100 fois et calculez la distance moyenne $\langle |x| \rangle$. Comparez avec la prédiction théorique $\sqrt{N} \approx 31{,}6$. *Indice :* utilisez `import random` et `random.choice([-1, 1])`.

Fonctions clés

Résumé du chapitre : Structures de contrôle

- **Conditions** : `if condition:` / `elif condition:` / `else:`
- **Boucle** `while` : répète tant que la condition est vraie
- **Boucle** `for` : parcourt un itérable (`range()`, liste, etc.)
- `range(a, b, s)` : entiers de `a` à `b-1`, pas de `s`
- `break` : sort immédiatement de la boucle
- `continue` : passe à l'itération suivante

- **Indentation** : 4 espaces, toujours cohérente
- **Garde-fou** : prévoir un nombre maximal d'itérations pour `while`

Chapitre 4

Fonctions et Portée

Intuition

Une fonction est comme une *machine* dans un laboratoire : on y introduit des *entrées* (réactifs), elle effectue un *traitement* (réaction chimique) et produit une *sortie* (produit). Plutôt que de reconstruire la machine à chaque expérience, on la réutilise. C'est le principe **DRY** : « *Don't Repeat Yourself* ».

4.1 Pourquoi utiliser des fonctions ?

- **Réutilisabilité** : écrire une formule une seule fois et l'appeler partout.
- **Lisibilité** : un programme découpé en fonctions est plus facile à comprendre.
- **Débogage** : on peut tester chaque fonction indépendamment.
- **Modularité** : chaque fonction a une responsabilité unique.

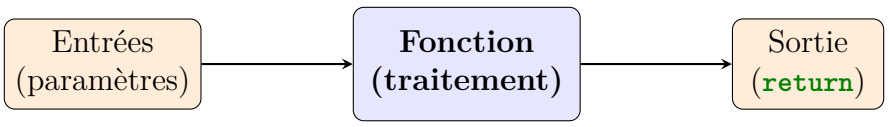


FIG. 4.1 : Une fonction vue comme une machine : entrées → traitement → sortie.

4.2 Définir et appeler une fonction

Définition 4.1 (Syntaxe d'une fonction)

```
def nom_fonction(parametre1, parametre2):
    """Docstring : description de la fonction."""
    # corps de la fonction
    return resultat
```

Python

```python
def energie_cinetique(masse, vitesse):
    """Calcule l'énergie cinétique E = 0.5 * m * v²."""
    return 0.5 * masse * vitesse**2

# Appel de la fonction
E = energie_cinetique(2.0, 3.0)
print(f"E = {E} J")
```

Sortie

```
E = 9.0 J
```

4.3 Docstrings

Une *docstring* est une chaîne de documentation placée juste après `def`. Elle est accessible via `help(nom_fonction)`.

Python

```python
def force_gravitationnelle(m1, m2, r):
    """
    Calcule la force gravitationnelle entre deux masses.

    Paramètres
    ----------
    m1 : float - masse 1 (kg)
    m2 : float - masse 2 (kg)
    r  : float - distance entre les centres (m)

    Retour
    ------
    float - force en Newtons
    """
    G = 6.674e-11  # constante gravitationnelle
    return G * m1 * m2 / r**2

# Force Terre-Lune
F = force_gravitationnelle(5.972e24, 7.342e22, 3.844e8)
print(f"Force Terre-Lune : {F:.2e} N")
```

Sortie

```
Force Terre-Lune : 1.98e+20 N
```

4.4 Arguments par défaut et arguments nommés

Python

```python
def chute_libre(t, g=9.81, v0=0):
    """Position d'un objet en chute libre : y = v0*t + 0.5*g*t².""""
    return v0 * t + 0.5 * g * t**2

# Utilisation avec valeurs par défaut
print(f"Terre : y = {chute_libre(2.0):.2f} m")

# Sur la Lune (g = 1.62 m/s²)
print(f"Lune  : y = {chute_libre(2.0, g=1.62):.2f} m")

# Avec vitesse initiale
print(f"Avec v0=5 : y = {chute_libre(2.0, v0=5):.2f} m")
```

Sortie

```
Terre : y = 19.62 m
Lune  : y = 3.24 m
Avec v0=5 : y = 29.62 m
```

Bonne pratique

Utilisez des arguments nommés pour améliorer la lisibilité : `chute\_libre(t=2.0, g=1.62)` est plus clair que `chute\_libre(2.0, 1.62)`.

4.5 Fonctions avec plusieurs valeurs de retour

Python

```python
import math

def coordonnees_polaires(x, y):
    """Convertit des coordonnées cartésiennes en polaires."""
    r = math.sqrt(x**2 + y**2)
    theta = math.atan2(y, x)
    return r, theta    # retourne un tuple

r, angle = coordonnees_polaires(3, 4)
print(f"r = {r:.2f},   = {math.degrees(angle):.1f}°")
```

Sortie

```
r = 5.00,   = 53.1°
```

4.6 Portée des variables : locale vs globale

Définition 4.2 (Portée (*scope*))**.** Une variable définie à l'intérieur d'une fonction est **locale** : elle n'existe que pendant l'exécution de la fonction. Une variable définie en dehors est **globale**.

Python

```python
x = 10          # variable globale

def ma_fonction():
    x = 5       # variable locale (différente !)
    print(f"Dans la fonction : x = {x}")

ma_fonction()
print(f"En dehors : x = {x}")
```

Sortie

```
Dans la fonction : x = 5
En dehors : x = 10
```

Attention

Évitez d'utiliser `global` pour modifier des variables globales depuis une fonction. Préférez *toujours* passer les valeurs en paramètres et utiliser `return`.

4.7 Fonctions lambda

Les fonctions `lambda` sont des fonctions anonymes courtes, utiles pour des opérations simples.

Python

```python
# Fonction lambda pour convertir Celsius en Kelvin
celsius_to_kelvin = lambda T: T + 273.15

temperatures_C = [-40, 0, 20, 37, 100]
temperatures_K = [celsius_to_kelvin(T) for T in temperatures_C]

for c, k in zip(temperatures_C, temperatures_K):
    print(f"{c:>4}°C = {k:>6.2f} K")
```

Sortie

```
 -40°C = 233.15 K
   0°C = 273.15 K
  20°C = 293.15 K
  37°C = 310.15 K
```

```
100°C = 373.15 K
```

4.8 Erreurs courantes

Ce qui peut mal tourner

1. **Oublier `return`** : sans `return`, une fonction renvoie `None` par défaut. Le calcul est effectué mais le résultat est perdu !

```
def aire_cercle(r):
    a = 3.14159 * r**2
    # Oubli de return a !

resultat = aire_cercle(5)
print(resultat)  # Affiche None
```

2. **Argument mutable par défaut** : utiliser une liste comme valeur par défaut est dangereux, car elle est partagée entre tous les appels !

```
# ERREUR : la liste est partagée
def ajouter(element, liste=[]):
    liste.append(element)
    return liste

print(ajouter(1))  # [1]
print(ajouter(2))  # [1, 2] au lieu de [2] !

# CORRECT : utiliser None
def ajouter(element, liste=None):
    if liste is None:
        liste = []
    liste.append(element)
    return liste
```

3. **Confondre `print` et `return`** : `print` affiche à l'écran mais ne renvoie pas de valeur utilisable dans une expression.

4.9 Mini-projet : Calculateur de formules physiques

Mini-projet scientifique

Créez un module contenant les fonctions suivantes :

- `energie\_cinetique(m, v)` : $E_c = \frac{1}{2}mv^2$
- `force\_gravitationnelle(m1, m2, r)` : $F = G\frac{m_1 m_2}{r^2}$

- `periode\_pendule(L, g=9.81)` : $T = 2\pi\sqrt{L/g}$
- `loi\_ohm(V=None, R=None, I=None)` : $V = RI$ (donner deux grandeurs, calculer la troisième)

Chaque fonction doit avoir une docstring complète. Testez-les avec des valeurs connues.

Python

```python
import math

def periode_pendule(L, g=9.81):
    """Période d'un pendule simple : T = 2√(L/g)."""
    return 2 * math.pi * math.sqrt(L / g)

def loi_ohm(V=None, R=None, I=None):
    """Loi d'Ohm : V = R × I. Fournir deux grandeurs."""
    if V is None:
        return R * I
    elif R is None:
        return V / I
    elif I is None:
        return V / R

# Tests
print(f"Pendule L=1m : T = {periode_pendule(1.0):.3f} s")
print(f"Loi d'Ohm : V = {loi_ohm(R=100, I=0.5):.1f} V")
print(f"Loi d'Ohm : I = {loi_ohm(V=12, R=100):.3f} A")
```

Sortie

```
Pendule L=1m : T = 2.006 s
Loi d'Ohm : V = 50.0 V
Loi d'Ohm : I = 0.120 A
```

4.10 Exercices

Exercice 4.1 (⋆ — Conversion de températures)**.** Écrivez deux fonctions `celsius\_to\_fahrenheit(T)` et `fahrenheit\_to\_celsius(T)`. Testez avec l'eau bouillante (100 °C = 212 °F).

Exercice 4.2 (⋆ — Factorielle)**.** Écrivez une fonction `factorielle(n)` qui calcule $n!$ avec une boucle. Vérifiez que `factorielle(10) == 3628800`.

Exercice 4.3 (⋆⋆ — Fonction récursive)**.** Réécrivez la factorielle de manière *récursive* : une fonction qui s'appelle elle-même. Quel est le cas de base ?

Exercice 4.4 (⋆⋆ — Statistiques descriptives)**.** Écrivez une fonction `statistiques(donnees)` qui reçoit une liste de nombres et renvoie la moyenne, la médiane et l'écart-type (sans utiliser de bibliothèque).

Exercice 4.5 ($\star\star\star$ — Méthode de Newton)**.** Implémentez la méthode de Newton pour trouver les zéros d'une fonction : $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. Votre fonction prendra en paramètres `f`, `df` (dérivée), `x0` (estimation initiale) et `tol` (tolérance). Testez avec $f(x) = x^2 - 2$ pour retrouver $\sqrt{2}$.

Fonctions clés

Résumé du chapitre : Fonctions et portée

- **Définition** : `def nom(params): ... return resultat`
- **Docstring** : chaîne de documentation entre triple guillemets
- **Arguments par défaut** : `def f(x, n=10):` — n vaut 10 si non précisé
- **Arguments nommés** : `f(x=3, n=20)` pour plus de clarté
- **Valeurs multiples** : `return a, b` renvoie un tuple
- **Portée locale** : les variables dans une fonction sont isolées
- **Lambda** : `lambda x: x**2` pour des fonctions courtes
- **Principe DRY** : ne jamais copier-coller du code, écrire une fonction

Chapitre 5

Structures de Données

Intuition

Un scientifique ne travaille jamais avec une seule donnée : il manipule des séries de mesures, des tableaux de résultats, des catalogues d'espèces. Python offre plusieurs *structures de données* pour organiser ces informations — chacune avec ses forces, comme les différents contenants d'un laboratoire (éprouvettes, boîtes de Pétri, classeurs).

5.1 Les listes

Définition 5.1 (Liste)**.** Une **liste** est une collection *ordonnée* et *modifiable* d'éléments. On la crée avec des crochets : `[e1, e2, ...]`.

5.1.1 Création et indexation

Python

```python
# Mesures de température (°C) sur une semaine
temperatures = [18.2, 19.5, 17.8, 21.0, 22.3, 20.1, 19.7]

print(f"Premier jour  : {temperatures[0]}°C")
print(f"Dernier jour  : {temperatures[-1]}°C")
print(f"Nombre de mesures : {len(temperatures)}")
```

Sortie

```
Premier jour  : 18.2°C
Dernier jour  : 19.7°C
Nombre de mesures : 7
```

5.1.2 Slicing (découpage)

Python

```python
# Jours ouvrés (lundi à vendredi)
jours_ouvres = temperatures[0:5]
print(f"Jours ouvrés : {jours_ouvres}")

# Un élément sur deux
print(f"Un sur deux : {temperatures[::2]}")

# Ordre inverse
print(f"Inversé : {temperatures[::-1]}")
```

Sortie

```
Jours ouvrés : [18.2, 19.5, 17.8, 21.0, 22.3]
Un sur deux : [18.2, 17.8, 22.3, 19.7]
Inversé : [19.7, 20.1, 22.3, 21.0, 17.8, 19.5, 18.2]
```

5.1.3 Méthodes utiles

Python

```python
masses = [12.5, 8.3, 15.7, 10.2]

masses.append(9.8)        # ajouter à la fin
print(f"Après append : {masses}")

dernier = masses.pop()    # retirer le dernier
print(f"Retiré : {dernier}, liste : {masses}")

masses.sort()             # trier en place
print(f"Trié : {masses}")

masses.reverse()          # inverser en place
print(f"Inversé : {masses}")
```

Sortie

```
Après append : [12.5, 8.3, 15.7, 10.2, 9.8]
Retiré : 9.8, liste : [12.5, 8.3, 15.7, 10.2]
Trié : [8.3, 10.2, 12.5, 15.7]
Inversé : [15.7, 12.5, 10.2, 8.3]
```

5.2 Compréhensions de listes

Les compréhensions offrent une syntaxe compacte pour créer des listes.

Python

```python
# Carrés des entiers de 1 à 10
carres = [x**2 for x in range(1, 11)]
print(f"Carrés : {carres}")

# Filtrer les températures > 20°C
temperatures = [18.2, 19.5, 17.8, 21.0, 22.3, 20.1, 19.7]
chaudes = [t for t in temperatures if t > 20]
print(f"Jours chauds : {chaudes}")

# Convertir en Fahrenheit
fahrenheit = [t * 9/5 + 32 for t in temperatures]
print(f"Fahrenheit : {[f'{f:.1f}' for f in fahrenheit]}")
```

Sortie

```
Carrés : [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
Jours chauds : [21.0, 22.3, 20.1]
Fahrenheit : ['64.8', '67.1', '64.0', '69.8', '72.1', '68.2', '67.5']
```

5.3 Les tuples

Définition 5.2 (Tuple)**.** Un **tuple** est une collection *ordonnée* mais *immuable* (non modifiable). On le crée avec des parenthèses : `(e1, e2, ...)`.

Python

```python
# Constantes physiques (ne doivent pas changer)
constantes = ("vitesse lumière", 299_792_458, "m/s")
print(f"{constantes[0]} = {constantes[1]} {constantes[2]}")

# Déballage (unpacking)
nom, valeur, unite = constantes
print(f"{nom} : {valeur} {unite}")

# Tentative de modification → erreur
# constantes[1] = 300_000_000  # TypeError !
```

Sortie

```
vitesse lumière = 299792458 m/s
vitesse lumière : 299792458 m/s
```

Remarque 5.3. Utilisez les tuples pour des données qui ne doivent pas être modifiées : coordonnées (x, y), constantes physiques, clés de dictionnaire.

5.4 Les dictionnaires

Définition 5.4 (Dictionnaire)**.** Un **dictionnaire** associe des *clés* à des *valeurs*. On le crée avec des accolades : `\{cle: valeur, ...\}`.

5.4.1 Le tableau périodique comme dictionnaire

Python

```
elements = {
    "H":  {"nom": "Hydrogène",  "Z": 1,  "masse": 1.008},
    "He": {"nom": "Hélium",     "Z": 2,  "masse": 4.003},
    "C":  {"nom": "Carbone",    "Z": 6,  "masse": 12.011},
    "N":  {"nom": "Azote",      "Z": 7,  "masse": 14.007},
    "O":  {"nom": "Oxygène",    "Z": 8,  "masse": 15.999},
    "Fe": {"nom": "Fer",        "Z": 26, "masse": 55.845},
}

# Accès à un élément
carbone = elements["C"]
print(f"Carbone : Z={carbone['Z']}, masse={carbone['masse']} u")

# Masse molaire de l'eau H2O
masse_eau = 2 * elements["H"]["masse"] + elements["O"]["masse"]
print(f"Masse molaire H O = {masse_eau:.3f} g/mol")
```

Sortie

```
Carbone : Z=6, masse=12.011 u
Masse molaire H O = 18.015 g/mol
```

5.4.2 Itération sur un dictionnaire

Python

```
# Afficher tous les éléments
for symbole, info in elements.items():
    print(f"{symbole:>2} : {info['nom']:<12} Z={info['Z']:>2}")
```

Sortie

```
 H : Hydrogène    Z= 1
He : Hélium       Z= 2
 C : Carbone      Z= 6
 N : Azote        Z= 7
 O : Oxygène      Z= 8
Fe : Fer          Z=26
```

5.5 Les ensembles

Python

```python
# Éléments détectés dans deux échantillons
echantillon_A = {"H", "C", "O", "N", "S"}
echantillon_B = {"H", "C", "O", "Fe", "Ca"}

print(f"Communs    : {echantillon_A & echantillon_B}")
print(f"Tous       : {echantillon_A | echantillon_B}")
print(f"Unique à A : {echantillon_A - echantillon_B}")
```

Sortie

```
Communs    : {'O', 'H', 'C'}
Tous       : {'S', 'O', 'Fe', 'H', 'Ca', 'N', 'C'}
Unique à A : {'S', 'N'}
```

Liste	Tuple	Dictionnaire	Ensemble
Ordonnée	Ordonné	Clé → Valeur	Non ordonné
Modifiable	Immuable	Modifiable	Pas de doublons
`[1, 2, 3]`	`(1, 2, 3)`	`\{"a": 1\}`	`\{1, 2, 3\}`

FIG. 5.1 : Comparaison des structures de données en Python.

5.6 Erreurs courantes

Ce qui peut mal tourner

1. **Index hors limites** (`IndexError`) : une liste de 5 éléments a des indices de 0 à 4. Accéder à l'indice 5 provoque une erreur.

```python
mesures = [1.2, 3.4, 5.6]
print(mesures[3])  # IndexError !
# Correct : mesures[2] ou mesures[-1]
```

2. **Modifier une liste pendant l'itération** : supprimer des éléments d'une liste en la parcourant produit des résultats imprévisibles.

```python
# ERREUR : saute des éléments
donnees = [1, -2, 3, -4, 5]
for d in donnees:
    if d < 0:
        donnees.remove(d)  # Dangereux !
```

```python
# CORRECT : créer une nouvelle liste
donnees = [d for d in donnees if d >= 0]
```

3. **Clé inexistante** (`KeyError`) : accéder à une clé absente d'un dictionnaire. Utilisez `dict.get(cle, defaut)` pour éviter l'erreur.
4. **Confondre liste et tuple** : tenter de modifier un tuple avec `t[0] = 5` provoque un `TypeError`.

5.7 Mini-projet : Gestion des notes étudiantes

Mini-projet scientifique

Créez un système de gestion de notes utilisant un dictionnaire :

- Clés : noms des étudiants
- Valeurs : listes de notes
- Fonctions : ajouter un étudiant, ajouter une note, calculer la moyenne, trouver le major de promotion

Python

```python
def creer_promotion():
    """Crée un dictionnaire vide de promotion."""
    return {}

def ajouter_etudiant(promo, nom):
    """Ajoute un étudiant avec une liste de notes vide."""
    promo[nom] = []

def ajouter_note(promo, nom, note):
    """Ajoute une note à un étudiant."""
    promo[nom].append(note)

def moyenne(notes):
    """Calcule la moyenne d'une liste de notes."""
    return sum(notes) / len(notes) if notes else 0

def afficher_resultats(promo):
    """Affiche les résultats de la promotion."""
    print(f"{'Étudiant':<15} {'Notes':<25} {'Moyenne':>8}")
    print("-" * 50)
    for nom, notes in promo.items():
        notes_str = ", ".join(f"{n:.1f}" for n in notes)
        moy = moyenne(notes)
        print(f"{nom:<15} {notes_str:<25} {moy:>7.2f}")
```

```python
# Utilisation
promo = creer_promotion()
for nom in ["Alice", "Bob", "Claire"]:
    ajouter_etudiant(promo, nom)

ajouter_note(promo, "Alice", 15.5)
ajouter_note(promo, "Alice", 17.0)
ajouter_note(promo, "Alice", 14.0)
ajouter_note(promo, "Bob", 12.0)
ajouter_note(promo, "Bob", 11.5)
ajouter_note(promo, "Bob", 13.0)
ajouter_note(promo, "Claire", 18.0)
ajouter_note(promo, "Claire", 16.5)
ajouter_note(promo, "Claire", 19.0)

afficher_resultats(promo)

major = max(promo, key=lambda nom: moyenne(promo[nom]))
print(f"\nMajor de promotion : {major} ({moyenne(promo[major]):.2f}/20)")
```

Sortie

```
Étudiant        Notes                       Moyenne
--------------------------------------------------
Alice           15.5, 17.0, 14.0              15.50
Bob             12.0, 11.5, 13.0              12.17
Claire          18.0, 16.5, 19.0              17.83

Major de promotion : Claire (17.83/20)
```

5.8 Exercices

Exercice 5.1 ($\star$ — Manipulation de listes)**.** Créez une liste de 10 mesures de pH. Calculez la moyenne, le minimum et le maximum *sans utiliser* les fonctions `sum`, `min`, `max`.

Exercice 5.2 ($\star$ — Filtrage)**.** À partir d'une liste de températures, créez avec une compréhension de liste une nouvelle liste ne contenant que les valeurs entre 15 °C et 25 °C.

Exercice 5.3 ($\star\star$ — Compteur de fréquences)**.** Écrivez une fonction qui reçoit une chaîne de caractères et renvoie un dictionnaire comptant la fréquence de chaque lettre. Testez avec une séquence ADN (« ATCGGATCCA »).

Exercice 5.4 ($\star\star$ — Matrice comme liste de listes)**.** Représentez une matrice 3×3 comme une liste de listes. Écrivez une fonction qui calcule la transposée. Vérifiez avec la matrice identité.

Exercice 5.5 ($\star\star\star$ — Analyse de séquence protéique)**.** Créez un dictionnaire associant chaque acide aminé à sa masse molaire. Écrivez une fonction qui, étant donnée

une séquence d'acides aminés (lettres), calcule la masse totale de la protéine. Testez avec « MKVLWA ».

Fonctions clés

Résumé du chapitre : Structures de données

- **Liste** `[...]` : ordonnée, modifiable, méthodes `append`, `pop`, `sort`
- **Compréhension** : `[expr for x in iterable if cond]`
- **Tuple** `(...)` : ordonné, immuable, idéal pour les constantes
- **Dictionnaire** `\{cle: val\}` : accès par clé, méthodes `.keys()`, `.values()`, `.items()`
- **Ensemble** `\{...\}` : pas de doublons, opérations $\cap$, $\cup$, $\setminus$
- **Indexation** : commence à 0, indices négatifs depuis la fin
- **Slicing** : `liste[debut:fin:pas]`

Chapitre 6

NumPy — Tableaux et Calcul Numérique

Intuition

Imaginez devoir mesurer la température en 1 000 points d'une plaque chauffante. Avec une liste Python, chaque opération nécessite une boucle. NumPy permet de traiter *tous les points en une seule instruction*, comme un instrument qui mesure toute la plaque d'un coup. C'est la différence entre compter des grains de sable un par un et peser le tas entier.

6.1 Pourquoi NumPy ?

Python

```python
import time

# Comparaison : somme des carrés de 1 à 1 000 000
n = 1_000_000

# Avec une liste Python
debut = time.time()
resultat_liste = sum([i**2 for i in range(n)])
temps_liste = time.time() - debut

# Avec NumPy
import numpy as np
debut = time.time()
resultat_numpy = np.sum(np.arange(n)**2)
temps_numpy = time.time() - debut

print(f"Liste Python : {temps_liste:.4f} s")
print(f"NumPy        : {temps_numpy:.4f} s")
print(f"Accélération : ×{temps_liste/temps_numpy:.0f}")
```

Sortie

```
Liste Python : 0.2851 s
NumPy        : 0.0032 s
Accélération : ×89
```

6.2 Création de tableaux

Python

```python
import numpy as np

# À partir d'une liste
a = np.array([1.0, 2.0, 3.0, 4.0])
print(f"Array     : {a}")

# Tableaux spéciaux
zeros = np.zeros(5)
print(f"Zéros     : {zeros}")

uns = np.ones(4)
print(f"Uns       : {uns}")

# Séquence régulière
x = np.arange(0, 10, 2)
print(f"Arange    : {x}")

# N points équidistants
t = np.linspace(0, 1, 5)
print(f"Linspace  : {t}")
```

Sortie

```
Array     : [1. 2. 3. 4.]
Zéros     : [0. 0. 0. 0. 0.]
Uns       : [1. 1. 1. 1.]
Arange    : [0 2 4 6 8]
Linspace  : [0.   0.25 0.5  0.75 1.  ]
```

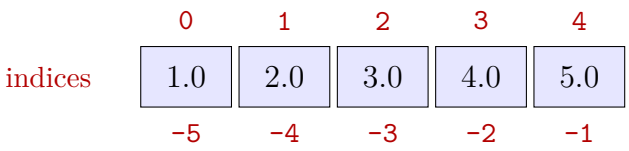


FIG. 6.1 : Indexation d'un tableau NumPy : indices positifs (haut) et négatifs (bas).

6.3 Opérations élément par élément

Python

```python
import numpy as np

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

print(f"a + b   = {a + b}")
print(f"a * b   = {a * b}")
print(f"a ** 2  = {a ** 2}")
print(f"a + 100 = {a + 100}")    # broadcasting
print(f"a > 2   = {a > 2}")       # comparaison
```

Sortie

```
a + b   = [11 22 33 44]
a * b   = [ 10  40  90 160]
a ** 2  = [ 1  4  9 16]
a + 100 = [101 102 103 104]
a > 2   = [False False  True  True]
```

Définition 6.1 (Broadcasting)**.** Le **broadcasting** permet à NumPy d'appliquer des opérations entre tableaux de tailles différentes. Un scalaire est automatiquement « étendu » à la taille du tableau.

6.4 Indexation et slicing

Python

```python
import numpy as np

x = np.array([10, 20, 30, 40, 50, 60, 70])

print(f"Élément 3     : {x[3]}")
print(f"Slice [2:5]   : {x[2:5]}")

# Indexation booléenne (fancy indexing)
masque = x > 35
print(f"Masque        : {masque}")
print(f"Valeurs > 35  : {x[masque]}")

# Modification conditionnelle
x[x < 30] = 0
print(f"Après modif   : {x}")
```

Sortie

```
Élément 3     : 40
Slice [2:5]   : [30 40 50]
Masque        : [False False False  True  True  True  True]
Valeurs > 35  : [40 50 60 70]
Après modif   : [ 0  0 30 40 50 60 70]
```

6.5 Fonctions mathématiques

Python

```python
import numpy as np

x = np.linspace(0, 2 * np.pi, 5)

print(f"x       = {np.round(x, 2)}")
print(f"sin(x)  = {np.round(np.sin(x), 4)}")
print(f"cos(x)  = {np.round(np.cos(x), 4)}")
print(f"exp(x)  = {np.round(np.exp(x), 2)}")

# Application : signal amorti
t = np.linspace(0, 5, 6)
signal = np.exp(-0.5 * t) * np.sin(2 * np.pi * t)
print(f"\nt       = {np.round(t, 1)}")
print(f"signal  = {np.round(signal, 4)}")
```

Sortie

```
x       = [0.   1.57 3.14 4.71 6.28]
sin(x)  = [ 0.    1.    0.   -1.   -0.  ]
cos(x)  = [ 1.    0.   -1.   -0.    1.  ]
exp(x)  = [  1.    4.81  23.14 111.32 535.49]

t       = [0. 1. 2. 3. 4. 5.]
signal  = [ 0.   -0.    0.   -0.    0.   -0.  ]
```

6.6 Statistiques descriptives

Python

```python
import numpy as np

# Mesures expérimentales (concentration en mg/L)
mesures = np.array([12.3, 11.8, 12.5, 11.9, 12.1, 12.4, 12.0, 11.7])
```

```
print(f"Moyenne      : {np.mean(mesures):.2f} mg/L")
print(f"Écart-type   : {np.std(mesures):.2f} mg/L")
print(f"Minimum      : {np.min(mesures):.2f} mg/L")
print(f"Maximum      : {np.max(mesures):.2f} mg/L")
print(f"Médiane      : {np.median(mesures):.2f} mg/L")
```

Sortie

```
Moyenne      : 12.09 mg/L
Écart-type   : 0.27 mg/L
Minimum      : 11.70 mg/L
Maximum      : 12.50 mg/L
Médiane      : 12.05 mg/L
```

6.7 Algèbre linéaire élémentaire

Python

```
import numpy as np

# Matrices 2×2
A = np.array([[1, 2],
              [3, 4]])

B = np.array([[5, 6],
              [7, 8]])

# Produit matriciel
C = A @ B    # équivalent à np.dot(A, B)
print("A @ B =")
print(C)

# Déterminant et inverse
det_A = np.linalg.det(A)
inv_A = np.linalg.inv(A)
print(f"\ndet(A) = {det_A:.1f}")
print(f"A ⁻¹ =\n{inv_A}")
```

Sortie

```
A @ B =
[[19 22]
 [43 50]]

det(A) = -2.0
A ⁻¹ =
[[-2.   1. ]
```

```
 [ 1.5 -0.5]]
```

6.8 Nombres aléatoires

Python

```python
import numpy as np

rng = np.random.default_rng(seed=42)

# Uniforme entre 0 et 1
u = rng.uniform(0, 1, size=5)
print(f"Uniforme  : {np.round(u, 3)}")

# Distribution normale (moyenne=0, écart-type=1)
n = rng.normal(0, 1, size=5)
print(f"Normale   : {np.round(n, 3)}")

# Entiers aléatoires (lancer de dé)
des = rng.integers(1, 7, size=10)
print(f"Lancers   : {des}")
```

Sortie

```
Uniforme  : [0.773 0.438 0.859 0.697 0.094]
Normale   : [ 0.314  0.459 -0.537  0.268 -0.232]
Lancers   : [5 2 3 6 1 4 2 5 3 6]
```

6.9 Erreurs courantes

Ce qui peut mal tourner

1. **Incompatibilité de dimensions (`shape mismatch`)** : opérer sur des tableaux de tailles incompatibles provoque une `ValueError`.

   ```python
   a = np.array([1, 2, 3])
   b = np.array([1, 2])
   # a + b → ValueError: operands could not be broadcast
   ```

2. **Oublier les opérations élément par élément** : l'opérateur `*` entre deux tableaux fait une multiplication élément par élément, *pas* un produit matriciel. Utilisez `@` ou `np.dot` pour le produit matriciel.

3. **Confusion `np.arange` / `np.linspace`** : `np.arange(0, 1, 0.1)` spécifie un *pas*, `np.linspace(0, 1, 10)` spécifie un *nombre de points*.

4. **Modifier une vue au lieu d'une copie** : le slicing NumPy crée une *vue* (pas une copie). Modifier la vue modifie l'original !

```
a = np.array([1, 2, 3, 4])
b = a[1:3]      # vue, pas copie
b[0] = 99       # a devient [1, 99, 3, 4] !
b = a[1:3].copy()  # copie indépendante
```

6.10 Mini-projet : Estimation de π par Monte Carlo

Mini-projet scientifique

La méthode de Monte Carlo estime π en tirant des points aléatoires dans un carré $[-1, 1]^2$ et en comptant ceux qui tombent dans le cercle unité ($x^2 + y^2 \leq 1$). Le rapport donne :

$$\pi \approx 4 \times \frac{\text{points dans le cercle}}{\text{points totaux}}$$

Python

```python
import numpy as np

rng = np.random.default_rng(seed=42)
N = 1_000_000

# Tirer N points aléatoires dans [-1, 1] × [-1, 1]
x = rng.uniform(-1, 1, size=N)
y = rng.uniform(-1, 1, size=N)

# Compter les points dans le cercle unité
dans_cercle = x**2 + y**2 <= 1
nb_dans_cercle = np.sum(dans_cercle)

# Estimation de pi
pi_estime = 4 * nb_dans_cercle / N
erreur = abs(pi_estime - np.pi)

print(f"Points totaux     : {N:,}")
print(f"Points dans cercle: {nb_dans_cercle:,}")
print(f"  estimé          : {pi_estime:.6f}")
print(f"  exact           : {np.pi:.6f}")
print(f"Erreur            : {erreur:.6f}")
```

Sortie

```
Points totaux     : 1,000,000
Points dans cercle: 785,436
  estimé          : 3.141744
  exact           : 3.141593
Erreur            : 0.000151
```

6.11 Exercices

Exercice 6.1 ($\star$ — Création de tableaux)**.** Créez les tableaux suivants : (a) les entiers de 0 à 99, (b) 50 valeurs équidistantes entre $-\pi$ et π, (c) une matrice 3×3 de zéros avec des 1 sur la diagonale (matrice identité).

Exercice 6.2 ($\star$ — Conversion de températures)**.** Créez un tableau de températures en Celsius de -40 à 100 (pas de 10). Convertissez-le en Fahrenheit ($F = 1{,}8 \times C + 32$) *sans boucle*.

Exercice 6.3 ($\star\star$ — Analyse de données expérimentales)**.** Générez 1000 mesures simulées d'une distribution normale ($\mu = 50$, $\sigma = 5$). Calculez la moyenne, l'écart-type, et comptez combien de mesures sont à plus de 2σ de la moyenne. Comparez avec la prédiction théorique ($\approx 5\%$).

Exercice 6.4 ($\star\star$ — Produit scalaire)**.** Calculez l'angle entre les vecteurs $\vec{u} = (1, 2, 3)$ et $\vec{v} = (4, 5, 6)$ en utilisant la formule $\cos\theta = \frac{\vec{u}\cdot\vec{v}}{|\vec{u}||\vec{v}|}$.

Exercice 6.5 ($\star\star\star$ — Résolution de système linéaire)**.** Résolvez le système d'équations linéaires suivant avec `np.linalg.solve` :

$$\begin{cases} 2x + 3y - z = 1 \\ 4x + y + 2z = 2 \\ -x + 2y + 3z = 3 \end{cases}$$

Vérifiez votre résultat en calculant $A\vec{x}$ et en comparant avec $\vec{b}$.

Fonctions clés

Résumé du chapitre : NumPy

- **Création** : `np.array`, `np.zeros`, `np.ones`, `np.linspace`, `np.arange`
- **Opérations** : `+`, `-`, `*`, `/`, `**` s'appliquent élément par élément
- **Broadcasting** : un scalaire est étendu automatiquement
- **Maths** : `np.sin`, `np.cos`, `np.exp`, `np.log`, `np.sqrt`
- **Stats** : `np.mean`, `np.std`, `np.min`, `np.max`, `np.median`
- **Algèbre** : `A @ B` (produit matriciel), `np.linalg.det`, `np.linalg.inv`, `np.linalg.solve`

- **Aléatoire** : `rng = np.random.default\_rng(seed)`
- **Indexation booléenne** : `x[x > 0]` filtre les éléments positifs

Chapitre 7

Matplotlib — Visualisation Scientifique

Intuition

« Une image vaut mille mots », et en science, un graphique vaut mille tableaux de données. Matplotlib est la bibliothèque de référence en Python pour créer des figures de qualité publication. Pensez-y comme le « papier millimétré numérique » du scientifique moderne.

7.1 Premier graphique : `plt.plot()`

Python

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi, 200)
y_sin = np.sin(x)
y_cos = np.cos(x)

plt.plot(x, y_sin, label="sin(x)")
plt.plot(x, y_cos, label="cos(x)", linestyle="--")
plt.xlabel("x (radians)")
plt.ylabel("y")
plt.title("Fonctions trigonométriques")
plt.legend()
plt.grid(True)
plt.show()
```

Sortie

```
[Figure affichée : deux courbes sin(x) et cos(x) sur [0, 2 ],
avec légende, grille et titres des axes.]
```

Remarque 7.1. La convention est d'importer `matplotlib.pyplot as plt`. Toutes les fonc-

tions de traçage sont alors accessibles via `plt.nom\_fonction()`.

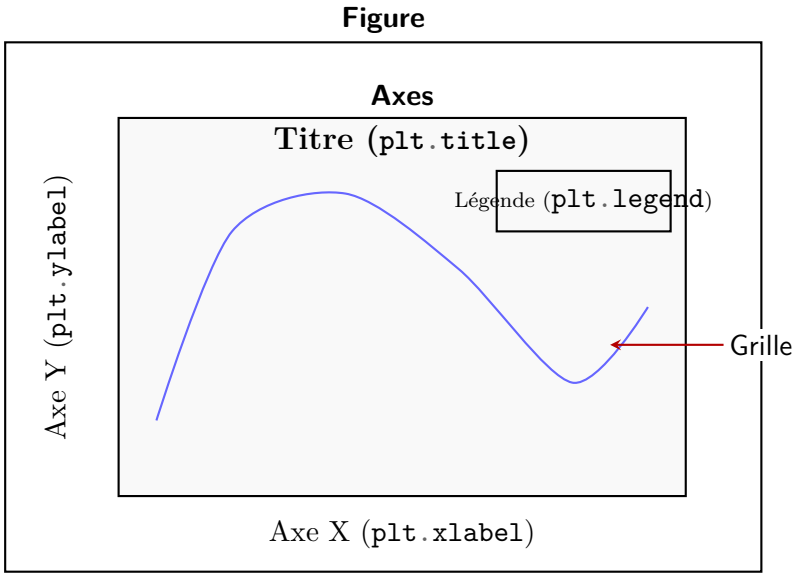


FIG. 7.1 : Anatomie d'une figure Matplotlib : figure, axes, titre, étiquettes et légende.

7.2 Nuages de points : `plt.scatter()`

Python

```python
import numpy as np
import matplotlib.pyplot as plt

# Simuler des données expérimentales
rng = np.random.default_rng(42)
n = 50
concentration = rng.uniform(0, 10, n)
absorbance = 0.42 * concentration + rng.normal(0, 0.3, n)

plt.scatter(concentration, absorbance, color="blue", alpha=0.6,
            label="Mesures")

# Droite de régression
coeffs = np.polyfit(concentration, absorbance, 1)
x_fit = np.linspace(0, 10, 100)
plt.plot(x_fit, np.polyval(coeffs, x_fit), "r-",
         label=f"Régression : y = {coeffs[0]:.2f}x + {coeffs[1]:.2f}")

plt.xlabel("Concentration (mol/L)")
plt.ylabel("Absorbance")
plt.title("Loi de Beer-Lambert")
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```

Sortie

[Figure affichée : nuage de points bleus avec droite de régression rouge, titre « Loi de Beer-Lambert », axes étiquetés.]

7.3 Histogrammes : `plt.hist()`

Python

```python
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(42)

# Distribution des masses d'un échantillon
masses = rng.normal(loc=75, scale=8, size=1000)

plt.hist(masses, bins=30, color="green", alpha=0.7,
         edgecolor="black", density=True)
plt.xlabel("Masse (kg)")
plt.ylabel("Densité de probabilité")
plt.title("Distribution des masses (n = 1000)")
plt.axvline(np.mean(masses), color="red", linestyle="--",
            label=f"Moyenne = {np.mean(masses):.1f} kg")
plt.legend()
plt.show()
```

Sortie

[Figure affichée : histogramme vert avec 30 barres, ligne rouge en pointillés marquant la moyenne.]

7.4 Diagrammes en barres : `plt.bar()`

Python

```python
import matplotlib.pyplot as plt

elements = ["H", "C", "N", "O", "S"]
abondance = [63.0, 9.5, 1.4, 25.5, 0.6]  # % dans le corps humain

plt.bar(elements, abondance, color=["red", "gray", "blue",
        "cyan", "yellow"], edgecolor="black")
plt.xlabel("Élément chimique")
plt.ylabel("Abondance (%)")
plt.title("Composition élémentaire du corps humain")
plt.show()
```

Sortie

```
[Figure affichée : diagramme en barres colorées montrant
l'abondance des éléments dans le corps humain.]
```

7.5 Sous-graphiques : `plt.subplots()`

Python

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 4 * np.pi, 300)

fig, axes = plt.subplots(2, 2, figsize=(10, 8))

# sin(x)
axes[0, 0].plot(x, np.sin(x), color="blue")
axes[0, 0].set_title("sin(x)")
axes[0, 0].grid(True)

# cos(x)
axes[0, 1].plot(x, np.cos(x), color="red")
axes[0, 1].set_title("cos(x)")
axes[0, 1].grid(True)

# exp(-x/5) * sin(x)  - signal amorti
axes[1, 0].plot(x, np.exp(-x / 5) * np.sin(x), color="green")
axes[1, 0].set_title("Signal amorti")
axes[1, 0].grid(True)

# x² et x³
axes[1, 1].plot(x, x**2, label="$x^2$")
axes[1, 1].plot(x, x**3 / 10, label="$x^3/10$")
axes[1, 1].set_title("Polynômes")
axes[1, 1].legend()
axes[1, 1].grid(True)

plt.tight_layout()
plt.show()
```

Sortie

```
[Figure affichée : grille 2×2 de sous-graphiques avec
sin(x), cos(x), signal amorti et polynômes.]
```

Bonne pratique

Utilisez toujours `plt.tight\_layout()` ou `fig.tight\_layout()` pour éviter que les étiquettes se chevauchent entre sous-graphiques.

7.6 Personnalisation avancée

Python

```python
import numpy as np
import matplotlib.pyplot as plt

t = np.linspace(0, 10, 500)
y1 = np.sin(2 * np.pi * 0.5 * t)
y2 = np.sin(2 * np.pi * 1.0 * t)

plt.figure(figsize=(10, 5))
plt.plot(t, y1, "b-", linewidth=2, label="f = 0.5 Hz")
plt.plot(t, y2, "r--", linewidth=1.5, label="f = 1.0 Hz")

plt.xlabel("Temps (s)", fontsize=14)
plt.ylabel("Amplitude", fontsize=14)
plt.title("Signaux sinusoïdaux", fontsize=16, fontweight="bold")
plt.legend(fontsize=12, loc="upper right")
plt.xlim(0, 10)
plt.ylim(-1.5, 1.5)
plt.grid(True, alpha=0.3)

# Annotation
plt.annotate("Maximum", xy=(0.5, 1), xytext=(2, 1.3),
             arrowprops=dict(arrowstyle="->"), fontsize=11)

plt.show()
```

Sortie

```
[Figure affichée : deux signaux sinusoïdaux avec styles différents,
annotation fléchée pointant vers le maximum.]
```

7.7 Sauvegarder une figure : `plt.savefig()`

Python

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 200)
```

```python
y = np.exp(-x**2 / 2) / np.sqrt(2 * np.pi)

plt.plot(x, y, "k-", linewidth=2)
plt.fill_between(x, y, alpha=0.3)
plt.xlabel("x")
plt.ylabel("f(x)")
plt.title("Distribution normale standard")

# Sauvegarder en haute résolution
plt.savefig("gaussienne.png", dpi=300, bbox_inches="tight")
plt.savefig("gaussienne.pdf", bbox_inches="tight")
print("Figure sauvegardée en PNG et PDF.")
```

Sortie

```
Figure sauvegardée en PNG et PDF.
```

Attention

Appelez `plt.savefig()` **avant** `plt.show()`, sinon la figure sera vide lors de la sauvegarde. L'option `bbox\_inches="tight"` évite de couper les étiquettes.

7.8 Figures scientifiques : trajectoire d'un projectile

Python

```python
import numpy as np
import matplotlib.pyplot as plt

g = 9.81
angles = [30, 45, 60, 75]  # degrés
v0 = 20  # m/s

plt.figure(figsize=(10, 6))

for theta_deg in angles:
    theta = np.radians(theta_deg)
    t_vol = 2 * v0 * np.sin(theta) / g
    t = np.linspace(0, t_vol, 200)
    x = v0 * np.cos(theta) * t
    y = v0 * np.sin(theta) * t - 0.5 * g * t**2
    plt.plot(x, y, linewidth=2, label=f" = {theta_deg}°")

plt.xlabel("Distance horizontale (m)", fontsize=13)
plt.ylabel("Hauteur (m)", fontsize=13)
plt.title("Trajectoire d'un projectile (v  = 20 m/s)", fontsize=14)
plt.legend(fontsize=11)
plt.grid(True, alpha=0.3)
```

```
plt.ylim(bottom=0)
plt.show()
```

Sortie

```
[Figure affichée : quatre trajectoires paraboliques pour différents
angles de lancement, montrant que 45° donne la portée maximale.]
```

7.9 Erreurs courantes

Ce qui peut mal tourner

1. **Oublier `plt.show()`** : dans un script (hors Jupyter), sans `plt.show()`, aucune fenêtre ne s'affiche. La figure est créée en mémoire mais reste invisible.
2. **Mauvaise taille de figure** : par défaut, les figures sont petites. Utilisez `plt.figure(figsize=(largeur, hauteur))` pour contrôler la taille en pouces.
3. **Sauvegarder après `plt.show()`** : `plt.show()` vide la figure en cours. Tout appel à `plt.savefig()` après donnera un fichier vide.
4. **Confondre `plt.plot` et l'interface orientée objet** : pour des sous-graphiques, utilisez `ax.plot()` et non `plt.plot()` qui agit sur le dernier axe actif.
5. **Oublier les étiquettes** : un graphique sans titre ni étiquettes d'axes est inutilisable en contexte scientifique. Ajoutez *toujours* `xlabel`, `ylabel` et `title`.

7.10 Mini-projet : Visualisation du mouvement d'un projectile

Mini-projet scientifique

Créez un programme complet qui :

1. Demande la vitesse initiale v_0 et l'angle de lancement θ.
2. Calcule la trajectoire, la portée et la hauteur maximale.
3. Produit une figure avec :
   - La trajectoire parabolique.
   - Un point marqué au sommet (hauteur max).
   - Une annotation indiquant la portée.
   - Un encadré avec les paramètres (v_0, θ, portée, $h_{\max}$).
4. Sauvegarde la figure en PDF.

Python

```
import numpy as np
import matplotlib.pyplot as plt

v0 = 25.0        # m/s
theta_deg = 55   # degrés
g = 9.81

theta = np.radians(theta_deg)
t_vol = 2 * v0 * np.sin(theta) / g
portee = v0**2 * np.sin(2 * theta) / g
h_max = (v0 * np.sin(theta))**2 / (2 * g)
t_max = v0 * np.sin(theta) / g

t = np.linspace(0, t_vol, 300)
x = v0 * np.cos(theta) * t
y = v0 * np.sin(theta) * t - 0.5 * g * t**2

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(x, y, "b-", linewidth=2.5)
ax.plot(v0 * np.cos(theta) * t_max, h_max, "ro", markersize=10)
ax.annotate(f"Sommet\n({h_max:.1f} m)",
            xy=(v0 * np.cos(theta) * t_max, h_max),
            xytext=(portee * 0.6, h_max * 0.95),
            arrowprops=dict(arrowstyle="->"), fontsize=11)

info = (f"v  = {v0} m/s\n"
        f"  = {theta_deg}°\n"
        f"Portée = {portee:.1f} m\n"
        f"h_max = {h_max:.1f} m")
ax.text(0.02, 0.95, info, transform=ax.transAxes,
        fontsize=11, verticalalignment="top",
        bbox=dict(boxstyle="round", facecolor="wheat", alpha=0.8))
```

```python
ax.set_xlabel("Distance (m)", fontsize=13)
ax.set_ylabel("Hauteur (m)", fontsize=13)
ax.set_title("Trajectoire d'un projectile", fontsize=15)
ax.set_ylim(bottom=0)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig("projectile.pdf", bbox_inches="tight")
plt.show()
print(f"Portée : {portee:.1f} m, Hauteur max : {h_max:.1f} m")
```

Sortie

```
Portée : 59.2 m, Hauteur max : 21.4 m
```

7.11 Exercices

Exercice 7.1 ($\star$ — Courbe exponentielle)**.** Tracez la fonction $f(x) = e^{-x/3}\cos(2\pi x)$ sur $[0, 10]$. Ajoutez un titre, des étiquettes d'axes et une grille.

Exercice 7.2 ($\star$ — Histogramme de lancers de dés)**.** Simulez 10 000 lancers de deux dés. Tracez l'histogramme de la somme obtenue. Quelle somme est la plus fréquente ?

Exercice 7.3 ($\star\star$ — Diagramme de phases)**.** Tracez un diagramme de phases simplifié de l'eau : pression (y) en fonction de la température (x). Utilisez `plt.fill\_between` pour colorier les zones solide, liquide et gazeuse.

Exercice 7.4 ($\star\star$ — Sous-graphiques multiples)**.** Créez une figure 2×2 montrant : (a) $\sin(x)$, (b) $\cos(x)$, (c) $\tan(x)$ (limité à $[-5, 5]$), (d) e^x et $\ln(x)$. Chaque sous-graphique doit avoir son titre et sa grille.

Exercice 7.5 ($\star\star\star$ — Carte de chaleur)**.** Créez une matrice 20×20 représentant la température d'une plaque chauffante (bords froids à 0 °C, centre chaud à 100 °C). Affichez-la avec `plt.imshow()` et ajoutez une barre de couleur. *Indice* : initialisez une matrice de zéros et ajoutez une gaussienne centrée.

Fonctions clés

Résumé du chapitre : Matplotlib

- **Courbes** : `plt.plot(x, y, style, label=...)`
- **Nuages** : `plt.scatter(x, y, color=..., alpha=...)`
- **Histogrammes** : `plt.hist(data, bins=..., density=True)`
- **Barres** : `plt.bar(categories, valeurs)`
- **Sous-graphiques** : `fig, axes = plt.subplots(nrows, ncols)`
- **Étiquettes** : `plt.xlabel`, `plt.ylabel`, `plt.title`

- **Légende** : `plt.legend()` (après avoir utilisé `label=...`)
- **Grille** : `plt.grid(True)`
- **Sauvegarde** : `plt.savefig("nom.pdf", dpi=300, bbox\_inches="tight")`
- **Règle d'or** : `savefig` *avant* `show`

Chapitre 8

Pandas — Analyse de Données

Intuition

Imaginez un tableur comme Excel, mais piloté entièrement par du code Python. C'est exactement ce que propose **Pandas** : une bibliothèque puissante pour charger, explorer, filtrer et analyser des données tabulaires. En sciences, la majorité des données expérimentales se présentent sous forme de tableaux — Pandas est l'outil idéal pour les manipuler.

8.1 Séries et DataFrames

Pandas repose sur deux structures fondamentales :

Définition 8.1 (Series et DataFrame)**.** Une **Series** est une colonne de données étiquetée (comme un vecteur avec un index). Un **DataFrame** est un tableau à deux dimensions composé de plusieurs Series partageant le même index.

DataFrame

index	nom	masse	charge
0	proton	1.673	+1
1	neutron	1.675	0
2	électron	0.0009	−1

Index — Colonnes (Series)

Python

```python
import pandas as pd

# Cr\'eer une Series
masses = pd.Series([1.673, 1.675, 0.0009],
                   index=["proton", "neutron", "electron"])
print(masses)
```

Sortie

```
proton      1.6730
neutron     1.6750
electron    0.0009
dtype: float64
```

8.2 Charger des données : `pd.read\_csv`

En pratique, on charge rarement des données à la main. Pandas lit de nombreux formats : CSV, Excel, JSON, SQL, etc. Nous utiliserons le jeu de données `tips` disponible dans Seaborn.

Python

```python
import seaborn as sns

tips = sns.load_dataset("tips")
print(type(tips))
print(tips.shape)
```

Sortie

```
<class 'pandas.core.frame.DataFrame'>
(244, 7)
```

Le DataFrame `tips` contient 244 lignes (observations) et 7 colonnes.

8.3 Exploration rapide

Python

```python
# Premi`eres lignes
print(tips.head(3))
```

Sortie

```
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
```

Python

```python
# R\'esum\'e statistique
print(tips.describe().round(2))
```

Sortie

```
       total_bill     tip    size
count      244.00  244.00  244.00
mean        19.79    3.00    2.57
std          8.90    1.38    0.95
min          3.07    1.00    1.00
25%         13.35    2.00    2.00
50%         17.80    2.90    2.00
75%         24.13    3.56    3.00
max         50.81   10.00    6.00
```

Python

```
tips.info()
```

Sortie

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   total_bill  244 non-null    float64
 1   tip         244 non-null    float64
 2   sex         244 non-null    category
 3   smoker      244 non-null    category
 4   day         244 non-null    category
 5   time        244 non-null    category
 6   size        244 non-null    int64
```

8.4 Sélection de données

Définition 8.2 (loc vs iloc)**.** `loc` sélectionne par **étiquettes** (noms de lignes/colonnes). `iloc` sélectionne par **positions entières** (indices numériques).

Python

```
# S\'electionner une colonne
print(tips["total_bill"].head(3))

# S\'electionner par position (ligne 0, colonne 1)
print(tips.iloc[0, 1])      # 1.01

# S\'electionner par \'etiquette
print(tips.loc[0, "tip"])    # 1.01

# Plusieurs colonnes
```

```python
subset = tips[["total_bill", "tip"]].head(3)
print(subset)
```

Sortie

```
0    16.99
1    10.34
2    21.01
Name: total_bill, dtype: float64
1.01
1.01
   total_bill   tip
0       16.99  1.01
1       10.34  1.66
2       21.01  3.50
```

8.5 Filtrage avec des masques booléens

Python

```python
# Additions sup\'erieures `a 40 €
gros_repas = tips[tips["total_bill"] > 40]
print(gros_repas[["total_bill", "tip", "size"]])
```

Sortie

```
     total_bill   tip  size
11        35.26  5.00     4
23        39.42  7.58     4
39        31.27  5.00     3
47        32.40  6.00     4
56        38.01  3.00     1
59        48.27  6.73     4
170       50.81  10.00    3
182       45.35  3.50     3
197       43.11  5.00     4
212       48.33  9.00     4
```

Python

```python
# Conditions multiples : d\^iners du dimanche avec pourboire > 5 €
filtre = tips[(tips["day"] == "Sun") & (tips["tip"] > 5)]
print(f"Nombre de r\'esultats : {len(filtre)}")
print(filtre[["total_bill", "tip", "day"]].head())
```

Sortie

```
Nombre de resultats : 11
     total_bill    tip  day
23        39.42   7.58  Sun
39        31.27   5.00  Sun
44        30.40   5.60  Sun
47        32.40   6.00  Sun
52        34.81   5.20  Sun
```

8.6 Agrégation avec GroupBy

L'opération `groupby` permet de regrouper les données selon une variable catégorielle, puis d'appliquer une fonction d'agrégation.

Python

```python
# Pourboire moyen par jour
par_jour = tips.groupby("day")["tip"].mean().round(2)
print(par_jour)
```

Sortie

```
day
Thur    2.77
Fri     2.73
Sat     2.99
Sun     3.26
Name: tip, dtype: float64
```

Python

```python
# Statistiques multiples
stats = tips.groupby("sex")[["total_bill", "tip"]].agg(["mean", "std"])
print(stats.round(2))
```

Sortie

```
       total_bill        tip
             mean   std mean  std
sex
Male        20.74  9.25 3.09 1.49
Female      18.06  8.01 2.83 1.16
```

8.7 Visualisation rapide avec `df.plot()`

Pandas intègre matplotlib pour créer des graphiques en une ligne.

Python

```python
import matplotlib.pyplot as plt

# Histogramme des additions
tips["total_bill"].plot(kind="hist", bins=20, edgecolor="black",
                        title="Distribution des additions")
plt.xlabel("Addition (€)")
plt.ylabel("Fr\'equence")
plt.show()

# Pourboire moyen par jour (diagramme en barres)
tips.groupby("day")["tip"].mean().plot(kind="bar", color="blue",
                                       title="Pourboire moyen par jour")
plt.ylabel("Pourboire moyen (€)")
plt.show()
```

Python

```python
# Nuage de points : addition vs pourboire
tips.plot(kind="scatter", x="total_bill", y="tip", alpha=0.6,
          title="Pourboire en fonction de l'addition")
plt.xlabel("Addition (€)")
plt.ylabel("Pourboire (€)")
plt.show()
```

Remarque 8.3. Pour des visualisations plus élaborées, on peut utiliser Seaborn ou Matplotlib directement. La méthode `df.plot()` est idéale pour une exploration rapide.

8.8 Créer de nouvelles colonnes

Python

```python
# Calculer le pourcentage de pourboire
tips["tip_pct"] = (tips["tip"] / tips["total_bill"] * 100).round(1)
print(tips[["total_bill", "tip", "tip_pct"]].head(4))
```

Sortie

```
   total_bill   tip  tip_pct
0       16.99  1.01      5.9
1       10.34  1.66     16.1
2       21.01  3.50     16.7
3       23.68  3.31     14.0
```

Ce qui peut mal tourner

Erreurs fréquentes avec Pandas

1. **SettingWithCopyWarning** — Quand on modifie une « tranche » d'un DataFrame, Pandas ne sait pas si on modifie l'original ou une copie. Solution : utilisez `.loc[]` ou `.copy()` explicitement.

```
# MAUVAIS : peut d\'eclencher un warning
subset = tips[tips["day"] == "Sun"]
subset["tip_pct"] = subset["tip"] / subset["total_bill"]

# BON : copie explicite
subset = tips[tips["day"] == "Sun"].copy()
subset["tip_pct"] = subset["tip"] / subset["total_bill"]
```

2. **Confondre `loc` et `iloc`** — `loc[1]` accède à la ligne dont l'**étiquette** est 1 ; `iloc[1]` accède à la **deuxième** ligne (position 1). Après un filtrage, les étiquettes ne sont plus consécutives !

3. **Oublier les parenthèses dans les conditions multiples** — Écrivez `(cond1) \& (cond2)`, pas `cond1 \& cond2`.

Mini-projet scientifique

Analyse du dataset Tips

1. Chargez le dataset `tips` avec Seaborn.
2. Calculez le pourboire moyen selon le moment du repas (`time`).
3. Déterminez quel jour a le plus grand nombre de clients (somme de `size`).
4. Créez une colonne `price\_per\_person` = `total\_bill / size`.
5. Tracez un histogramme de `price\_per\_person` et commentez la distribution.
6. Testez si les fumeurs laissent un pourboire différent (groupby + mean).

8.9 Exercices

Exercice 8.1 (⋆). Chargez le dataset `sns.load\_dataset("penguins")`. Affichez les 5 premières lignes, la forme du DataFrame et le résumé statistique. Combien y a-t-il de valeurs manquantes par colonne ? (Indice : `.isna().sum()`)

Exercice 8.2 (⋆). À partir du dataset `tips`, sélectionnez toutes les lignes où l'addition dépasse 30 € et le pourboire est inférieur à 3 €. Combien de lignes obtient-on ?

Exercice 8.3 (⋆⋆). Utilisez `groupby` pour calculer la moyenne et l'écart-type du pourboire selon le sexe *et* le moment du repas (groupby sur deux colonnes). Quelle catégorie laisse le pourboire moyen le plus élevé ?

Exercice 8.4 (⋆⋆). Créez une nouvelle colonne `tip\_category` qui vaut `"g\'en\'ereux"` si `tip\_pct > 20`, `"normal"` si entre 10 et 20, et `"faible"` sinon. Utilisez `pd.cut` ou `np.where`. Tracez un diagramme en barres du nombre de repas par catégorie.

Exercice 8.5 (⋆⋆⋆). Chargez le dataset `sns.load\_dataset("planets")`. Analysez la distribution des méthodes de découverte d'exoplanètes au fil des années : groupez par `method` et `year`, comptez les découvertes, et tracez l'évolution temporelle pour les 3 méthodes les plus fréquentes.

Fonctions clés

Chapitre 8 — Pandas : l'essentiel

Opération	Syntaxe
Charger un CSV	`pd.read\_csv("fichier.csv")`
Premières lignes	`df.head(n)`
Dimensions	`df.shape`
Résumé statistique	`df.describe()`
Sélection colonne	`df["col"]` ou `df.col`
Sélection par position	`df.iloc[ligne, colonne]`
Sélection par étiquette	`df.loc[ligne, "colonne"]`
Filtrage	`df[df["col"] > valeur]`
Agrégation	`df.groupby("col")["val"].mean()`
Nouvelle colonne	`df["new"] = expression`
Graphique rapide	`df.plot(kind="hist")`

Chapitre 9

Introduction à R pour la Statistique

Intuition

Si Python est un couteau suisse généraliste, **R** est un instrument de précision conçu spécifiquement pour la statistique. Créé par des statisticiens pour des statisticiens, R excelle dans l'analyse de données, les tests d'hypothèses et la visualisation. De nombreux chercheurs en biologie, écologie et sciences sociales l'utilisent quotidiennement.

9.1 Pourquoi R ?

Définition 9.1 (Le langage R)**.** **R** est un langage de programmation libre et gratuit, spécialisé dans le calcul statistique et la représentation graphique. Il dispose d'un écosystème de plus de 20 000 paquets sur le dépôt CRAN.

- Généraliste
- Machine learning
- Automatisation
- NumPy, Pandas, SciPy

- Statistique native
- Tests d'hypothèses
- Visualisation (ggplot2)
- Bioconductor, tidyverse

9.2 Variables et vecteurs

En R, l'opérateur d'affectation usuel est `<-` (flèche). La structure de base est le **vecteur**, créé avec la fonction `c()`.

R

```r
# Affectation
x <- 42
nom <- "physique"
```

```r
# Vecteur num\'erique
mesures <- c(12.3, 14.1, 11.8, 13.5, 12.9)
print(mesures)

# Op\'erations vectoris\'ees
mesures_cm <- mesures * 100
print(mesures_cm)
```

Sortie

```
[1] 12.3 14.1 11.8 13.5 12.9
[1] 1230 1410 1180 1350 1290
```

Attention

En R, l'indexation commence à **1** (et non 0 comme en Python). Ainsi, `mesures[1]` renvoie le *premier* élément, soit 12.3.

R

```r
# Indexation (commence `a 1 !)
print(mesures[1])    # Premier \'el\'ement
print(mesures[2:4])  # \'El\'ements 2 `a 4 (inclus !)

# S\'equences et r\'ep\'etitions
indices <- 1:10
print(indices)
repetitions <- rep(0, 5)
print(repetitions)
```

Sortie

```
[1] 12.3
[1] 14.1 11.8 13.5
 [1]  1  2  3  4  5  6  7  8  9 10
[1] 0 0 0 0 0
```

9.3 Data frames en R

R

```r
# Cr\'eer un data frame
experience <- data.frame(
  sujet = c("A", "B", "C", "D", "E"),
  dose  = c(10, 20, 30, 40, 50),
  effet = c(2.1, 4.5, 5.8, 7.2, 8.9)
```

```
)
print(experience)
str(experience)
```

Sortie

```
  sujet dose effet
1     A   10   2.1
2     B   20   4.5
3     C   30   5.8
4     D   40   7.2
5     E   50   8.9
'data.frame': 5 obs. of  3 variables:
 $ sujet: chr  "A" "B" "C" "D" ...
 $ dose : num  10 20 30 40 50
 $ effet: num  2.1 4.5 5.8 7.2 8.9
```

9.4 Fonctions statistiques de base

R intègre nativement toutes les fonctions statistiques essentielles.

R

```
notes <- c(12, 15, 8, 14, 16, 11, 13, 9, 17, 10)

cat("Moyenne     :", mean(notes), "\n")
cat("M\'ediane    :", median(notes), "\n")
cat("\'Ecart-type :", sd(notes), "\n")
cat("Variance    :", var(notes), "\n")
cat("Min / Max   :", min(notes), "/", max(notes), "\n")

# R\'esum\'e complet
summary(notes)
```

Sortie

```
Moyenne     : 12.5
Mediane     : 12.5
Ecart-type  : 3.027650
Variance    : 9.166667
Min / Max   : 8 / 17
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   8.00   10.25   12.50   12.50   14.75   17.00
```

9.5 Tests statistiques

Définition 9.2 (Test de Student)**.** Le **test t de Student** compare les moyennes de deux groupes pour déterminer si la différence observée est statistiquement significative (p-valeur < 0.05).

R

```r
# Deux groupes : traitement vs contr\^ole
traitement <- c(23.1, 25.4, 22.8, 26.1, 24.5, 27.0)
controle   <- c(19.2, 20.1, 18.5, 21.3, 19.8, 20.5)

resultat <- t.test(traitement, controle)
print(resultat)
```

Sortie

```
 Welch Two Sample t-test

data:  traitement and controle
t = 5.1893, df = 9.5182, p-value = 0.0004507
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 2.633574 6.566426
sample estimates:
mean of x mean of y
 24.81667  20.21667
```

R

```r
# Test de corr\'elation
x <- c(1, 2, 3, 4, 5, 6, 7, 8)
y <- c(2.1, 4.3, 5.8, 8.2, 9.7, 12.1, 13.8, 16.2)
cor.test(x, y)
```

Sortie

```
 Pearson's product-moment correlation

data:  x and y
t = 42.085, df = 6, p-value = 4.386e-08
95 percent confidence interval:
 0.9971346 0.9999227
sample estimates:
      cor
0.9984112
```

9.6 Régression linéaire avec `lm()`

R

```r
# R\'egression lin\'eaire : effet en fonction de la dose
modele <- lm(effet ~ dose, data = experience)
summary(modele)
```

Sortie

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.14000    0.34351   0.408  0.71062
dose         0.17200    0.01033  16.649  0.00048 ***
---
Residual standard error: 0.3742 on 3 degrees of freedom
Multiple R-squared:  0.9893, Adjusted R-squared:  0.9857
```

Le coefficient $R^2 = 0.989$ indique un excellent ajustement linéaire : l'effet augmente de 0.172 unités par unité de dose.

9.7 Graphiques de base en R

R

```r
# Nuage de points avec droite de r\'egression
plot(experience$dose, experience$effet,
     xlab = "Dose (mg)", ylab = "Effet",
     main = "Relation dose-effet", pch = 19, col = "blue")
abline(modele, col = "red", lwd = 2)

# Histogramme
hist(rnorm(1000), breaks = 30, col = "lightblue",
     main = "Distribution normale simul\'ee",
     xlab = "Valeur")

# Bo\^ite `a moustaches
boxplot(traitement, controle,
        names = c("Traitement", "Contr\^ole"),
        col = c("coral", "lightgreen"),
        main = "Comparaison des groupes")
```

9.8 Comparaison Python vs R

Python

Exemple 9.3 (Même analyse dans les deux langages).

```python
import numpy as np
from scipy import stats

data = [12, 15, 8, 14, 16, 11, 13, 9, 17, 10]
print(f"Moyenne : {np.mean(data):.2f}")
print(f"Ecart-type : {np.std(data, ddof=1):.2f}")

# Test t
groupe_a = [23.1, 25.4, 22.8, 26.1, 24.5, 27.0]
groupe_b = [19.2, 20.1, 18.5, 21.3, 19.8, 20.5]
t_stat, p_val = stats.ttest_ind(groupe_a, groupe_b)
print(f"p-value : {p_val:.6f}")
```

R

```r
data <- c(12, 15, 8, 14, 16, 11, 13, 9, 17, 10)
cat("Moyenne :", mean(data), "\n")
cat("Ecart-type :", sd(data), "\n")

# Test t
groupe_a <- c(23.1, 25.4, 22.8, 26.1, 24.5, 27.0)
groupe_b <- c(19.2, 20.1, 18.5, 21.3, 19.8, 20.5)
resultat <- t.test(groupe_a, groupe_b)
cat("p-value :", resultat$p.value, "\n")
```

Remarque 9.4. Notez que `sd()` en R calcule l'écart-type *corrigé* $(n-1)$ par défaut, comme `np.std(data, ddof=1)` en Python. La fonction `np.std()` sans `ddof=1` divise par n, ce qui donne un résultat différent.

Ce qui peut mal tourner

Erreurs fréquentes en R

1. **Indexation à partir de 1** — En R, `x[1]` est le *premier* élément. En Python, `x[1]` est le *deuxième*. Cette différence est source de nombreuses erreurs quand on passe d'un langage à l'autre.

2. **`<-` vs `=`** — En R, on privilégie `<-` pour l'affectation. Le signe `=` fonctionne aussi dans la plupart des cas, mais `<-` est la convention standard. Attention à ne pas écrire `x < -5` (comparaison !) au lieu de `x <- 5` (affectation).

3. **Facteurs inattendus** — R convertit parfois automatiquement les chaînes en `factor`. Utilisez `stringsAsFactors = FALSE` dans `data.frame()` si nécessaire.

4. **Oublier `na.rm = TRUE`** — Les fonctions comme `mean()` retournent `NA` si le vecteur contient des valeurs manquantes. Ajoutez `na.rm = TRUE`.

Mini-projet scientifique

Analyse statistique comparative en R Réalisez une étude complète en R :

1. Créez un data frame avec deux groupes expérimentaux (15 mesures chacun), simulés avec `rnorm(15, mean=..., sd=...)`.
2. Calculez les statistiques descriptives de chaque groupe (`mean`, `sd`, `median`).
3. Réalisez un test t de Student pour comparer les deux groupes.
4. Tracez des boîtes à moustaches côte à côte.
5. Ajoutez une troisième variable et effectuez une régression linéaire avec `lm()`.
6. Interprétez les résultats : la différence est-elle significative ? Le modèle linéaire est-il pertinent (R^2) ?

9.9 Exercices

Exercice 9.1 ($\star$)**.** Créez un vecteur contenant les températures moyennes mensuelles d'une ville (12 valeurs). Calculez la moyenne, la médiane et l'écart-type. Quel mois est le plus chaud ? Utilisez `which.max()`.

Exercice 9.2 ($\star$)**.** Créez un data frame avec les colonnes `element`, `symbole` et `masse\_atomique` pour 5 éléments chimiques. Affichez le résumé avec `summary()` et accédez à la colonne `masse\_atomique` avec `\$`.

Exercice 9.3 ($\star\star$)**.** Générez 100 valeurs aléatoires suivant une loi normale ($\mu = 50$, $\sigma = 10$) avec `rnorm()`. Tracez un histogramme et superposez la courbe théorique avec `curve(dnorm(x, 50, 10), add = TRUE)`.

Exercice 9.4 ($\star\star$)**.** Simulez une expérience : générez deux échantillons de taille 30, l'un de moyenne 100 et l'autre de moyenne 105 (même écart-type 15). Effectuez un test t. Répétez 1000 fois et comptez le pourcentage de fois où $p < 0.05$ (c'est la **puissance** du test).

Exercice 9.5 ($\star\star\star$)**.** Chargez le jeu de données intégré `iris` en R. Réalisez une analyse complète : statistiques descriptives par espèce, test ANOVA avec `aov()` pour comparer les longueurs de sépales entre espèces, et régression linéaire entre longueur et largeur de pétales. Produisez 3 graphiques pertinents.

Fonctions clés

Chapitre 9 — R : l'essentiel

Concept	Syntaxe R
Affectation	`x <- valeur`
Vecteur	`c(1, 2, 3)`
Séquence	`1:10` ou `seq(0, 1, by=0.1)`
Data frame	`data.frame(col1=..., col2=...)`
Moyenne	`mean(x)`
Écart-type	`sd(x)`
Résumé	`summary(x)`
Test t	`t.test(groupe1, groupe2)`
Corrélation	`cor.test(x, y)`
Régression	`lm(y ~ x, data=df)`
Graphique	`plot(x, y)`, `hist(x)`, `boxplot(x)`

Chapitre 10

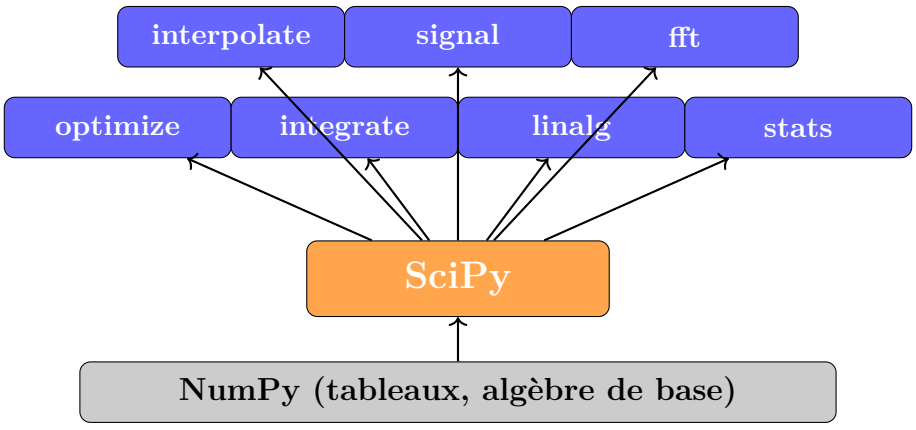
Calcul Scientifique avec SciPy

Intuition

NumPy fournit les briques de base (tableaux, algèbre linéaire élémentaire), mais le scientifique a souvent besoin d'outils plus spécialisés : ajuster une courbe à des données expérimentales, intégrer une fonction, résoudre un système d'équations ou effectuer des tests statistiques. C'est exactement le rôle de **SciPy**, une bibliothèque construite au-dessus de NumPy.

10.1 Vue d'ensemble de SciPy

Définition 10.1 (SciPy)**. SciPy** (Scientific Python) est une bibliothèque open source qui fournit des algorithmes numériques efficaces organisés en sous-modules thématiques.



10.2 `scipy.optimize` — Optimisation et ajustement

10.2.1 Ajuster une courbe expérimentale avec `curve\_fit`

En physique, on mesure souvent une décroissance exponentielle (radioactivité, décharge d'un condensateur). Ajustons un modèle $A(t) = A_0\, e^{-\lambda t}$ à des données bruitées.

Python

```python
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

# Mod`ele exponentiel
def decroissance(t, A0, lam):
    return A0 * np.exp(-lam * t)

# Donn\'ees simul\'ees (avec bruit)
np.random.seed(42)
t_data = np.linspace(0, 10, 20)
y_data = 5.0 * np.exp(-0.3 * t_data) + 0.2 * np.random.randn(20)

# Ajustement
popt, pcov = curve_fit(decroissance, t_data, y_data)
A0_fit, lam_fit = popt
print(f"A0 = {A0_fit:.3f}, lambda = {lam_fit:.3f}")
```

Sortie

```
A0 = 5.047, lambda = 0.308
```

Les valeurs retrouvées ($A_0 \approx 5.05$, $\lambda \approx 0.31$) sont proches des paramètres réels ($A_0 = 5$, $\lambda = 0.3$).

Python

```python
# Visualisation
t_fine = np.linspace(0, 10, 200)
plt.scatter(t_data, y_data, label="Donn\'ees", color="black", s=20)
plt.plot(t_fine, decroissance(t_fine, *popt), "r-", label="Ajustement")
plt.xlabel("Temps (s)")
plt.ylabel("Activit\'e (Bq)")
plt.title("Ajustement d'une d\'ecroissance exponentielle")
plt.legend()
plt.show()
```

10.2.2 Minimisation avec `minimize`

Python

```python
from scipy.optimize import minimize

# Trouver le minimum de f(x) = (x - 3)^2 + 2*sin(x)
def f(x):
    return (x - 3)**2 + 2 * np.sin(x)
```

```python
result = minimize(f, x0=0)  # Point de d\'epart x0 = 0
print(f"Minimum en x = {result.x[0]:.4f}")
print(f"Valeur f(x) = {result.fun:.4f}")
```

Sortie

```
Minimum en x = 3.3424
Valeur f(x) = -1.8770
```

10.3 `scipy.integrate` — Intégration numérique

La fonction `quad` calcule une intégrale définie numériquement.

Définition 10.2 (Intégration numérique)**.** L'intégration numérique (ou quadrature) approxime la valeur d'une intégrale définie $\int_a^b f(x)\,dx$ en évaluant f en un nombre fini de points.

Python

```python
from scipy.integrate import quad

# Calculer l'int\'egrale de sin(x) de 0 `a pi
resultat, erreur = quad(np.sin, 0, np.pi)
print(f"Integrale de sin(x) de 0 a pi = {resultat:.6f}")
print(f"Valeur exacte = {2.0:.6f}")

# Int\'egrale gaussienne : int de -inf `a +inf de exp(-x^2)
val, err = quad(lambda x: np.exp(-x**2), -np.inf, np.inf)
print(f"Integrale gaussienne = {val:.6f}")
print(f"sqrt(pi) = {np.sqrt(np.pi):.6f}")
```

Sortie

```
Integrale de sin(x) de 0 a pi = 2.000000
Valeur exacte = 2.000000
Integrale gaussienne = 1.772454
sqrt(pi) = 1.772454
```

10.4 `scipy.linalg` — Algèbre linéaire

10.4.1 Résoudre un système linéaire

Considérons le système :

$$\begin{cases} 2x + y = 5 \\ x + 3y = 7 \end{cases} \iff \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \end{pmatrix}$$

Python

```python
from scipy.linalg import solve, eig

A = np.array([[2, 1],
              [1, 3]])
b = np.array([5, 7])

x = solve(A, b)
print(f"Solution : x = {x[0]:.2f}, y = {x[1]:.2f}")
```

Sortie

```
Solution : x = 1.60, y = 1.80
```

10.4.2 Valeurs propres

Python

```python
# Valeurs propres et vecteurs propres
valeurs, vecteurs = eig(A)
print("Valeurs propres :", valeurs.real)
print("Vecteur propre 1 :", vecteurs[:, 0].real)
```

Sortie

```
Valeurs propres : [1.38196601 3.61803399]
Vecteur propre 1 : [-0.85065081  0.52573111]
```

10.5 `scipy.stats` — Statistiques

Python

```python
from scipy import stats

# G\'en\'erer des donn\'ees suivant une loi normale
np.random.seed(0)
echantillon = stats.norm.rvs(loc=170, scale=10, size=100)

# Statistiques descriptives
print(f"Moyenne : {np.mean(echantillon):.2f}")
print(f"Ecart-type : {np.std(echantillon, ddof=1):.2f}")

# Test de normalit\'e (Shapiro-Wilk)
stat, p = stats.shapiro(echantillon)
print(f"Test Shapiro-Wilk : p = {p:.4f}")
if p > 0.05:
```

```
    print("-> Distribution compatible avec la normalit\'e")
```

Sortie

```
Moyenne : 171.77
Ecart-type : 10.14
Test Shapiro-Wilk : p = 0.8453
-> Distribution compatible avec la normalite
```

Python

```
# Test t de Student (deux \'echantillons)
groupe_a = stats.norm.rvs(loc=170, scale=10, size=50, random_state=1)
groupe_b = stats.norm.rvs(loc=175, scale=10, size=50, random_state=2)

t_stat, p_val = stats.ttest_ind(groupe_a, groupe_b)
print(f"t = {t_stat:.3f}, p-value = {p_val:.4f}")
```

Sortie

```
t = -2.574, p-value = 0.0116
```

10.6 `scipy.interpolate` — Interpolation

L'interpolation permet d'estimer des valeurs entre des points de mesure.

Python

```
from scipy.interpolate import interp1d

# Points de mesure (temp\'erature en fonction du temps)
temps = np.array([0, 1, 2, 3, 4, 5])
temperature = np.array([20.0, 22.5, 28.3, 31.1, 29.0, 25.5])

# Interpolation lin\'eaire et cubique
f_lin = interp1d(temps, temperature, kind="linear")
f_cub = interp1d(temps, temperature, kind="cubic")

t_fin = np.linspace(0, 5, 100)
print(f"T `a t=2.5 (lin\'eaire) : {f_lin(2.5):.2f} °C")
print(f"T `a t=2.5 (cubique)  : {f_cub(2.5):.2f} °C")
```

Sortie

```
T a t=2.5 (lineaire) : 29.70 °C
T a t=2.5 (cubique)  : 30.22 °C
```

Python

```
plt.scatter(temps, temperature, color="black", zorder=5,
            label="Mesures")
plt.plot(t_fin, f_lin(t_fin), "--", label="Lin\'eaire")
plt.plot(t_fin, f_cub(t_fin), "-", label="Cubique")
plt.xlabel("Temps (h)")
plt.ylabel("Temp\'erature (°C)")
plt.legend()
plt.title("Interpolation de donn\'ees de temp\'erature")
plt.show()
```

Ce qui peut mal tourner

Erreurs fréquentes avec SciPy

1. **Confondre `numpy.linalg` et `scipy.linalg`** — Les deux existent, mais `scipy.linalg` est plus complet et plus optimisé. Préférez toujours `scipy.linalg` pour les calculs sérieux.
2. **Oublier le point de départ dans `minimize`** — La fonction `minimize` nécessite un `x0`. Un mauvais choix initial peut conduire à un minimum *local* au lieu du minimum global.
3. **Mauvais modèle dans `curve\_fit`** — Si le modèle mathématique ne correspond pas aux données, l'ajustement sera mauvais. Vérifiez toujours visuellement le résultat.
4. **Ignorer la matrice de covariance** — `curve\_fit` retourne également `pcov`, qui permet de calculer les incertitudes sur les paramètres ajustés : `np.sqrt(np.diag(pcov))`.

Mini-projet scientifique

Analyse et ajustement de données expérimentales Simulez une expérience de décharge d'un condensateur :

1. Générez des données bruitées suivant $V(t) = V_0\, e^{-t/\tau}$ avec $V_0 = 12\,\mathrm{V}$ et $\tau = 3\,\mathrm{s}$ (ajoutez du bruit gaussien).
2. Utilisez `curve\_fit` pour retrouver V_0 et τ.
3. Calculez les incertitudes à partir de la matrice de covariance.
4. Intégrez numériquement l'énergie dissipée : $E = \int_0^{20} \frac{V(t)^2}{R}\, dt$ avec $R = 1\,\mathrm{k\Omega}$.
5. Tracez les données, la courbe ajustée et les barres d'erreur.
6. Comparez le τ obtenu avec la valeur théorique.

10.7 Exercices

Exercice 10.1 ($\star$)**.** Calculez numériquement l'intégrale $\int_0^1 e^{-x^2}\,dx$ avec `quad`. Comparez avec la valeur approchée ≈ 0.7468.

Exercice 10.2 ($\star$)**.** Résolvez le système linéaire : $3x + 2y - z = 1$, $2x - y + 4z = -2$, $x + y + z = 0$. Vérifiez en calculant $A \cdot x$ avec NumPy.

Exercice 10.3 ($\star\star$)**.** Générez 200 points suivant une loi normale ($\mu = 50$, $\sigma = 8$). Effectuez un test de Shapiro-Wilk et un test de Kolmogorov-Smirnov (`stats.kstest`) pour vérifier la normalité. Interprétez les p-valeurs.

Exercice 10.4 ($\star\star$)**.** Des données expérimentales suivent la loi $y = a\sin(bx + c)$. Générez des données avec $a = 3$, $b = 2$, $c = 0.5$ plus du bruit. Utilisez `curve\_fit` pour retrouver les paramètres. Attention : donnez des valeurs initiales raisonnables avec le paramètre `p0`.

Exercice 10.5 ($\star\star\star$)**.** Modélisez la cinétique d'une réaction chimique du second ordre : $\frac{d[A]}{dt} = -k[A]^2$. La solution analytique est $[A](t) = \frac{[A]_0}{1+k[A]_0 t}$. Générez des données bruitées, ajustez le modèle pour trouver k et $[A]_0$, puis intégrez numériquement $\int_0^{10}[A](t)\,dt$ (quantité totale restante). Comparez intégration numérique et solution analytique.

Fonctions clés

Chapitre 10 — SciPy : l'essentiel

Sous-module	Fonctions clés
`scipy.optimize`	`minimize(f, x0)`, `curve\_fit(modele, x`
`scipy.integrate`	`quad(f, a, b)` — intégrale définie
`scipy.linalg`	`solve(A, b)`, `eig(A)`
`scipy.stats`	`ttest\_ind`, `shapiro`, distributions
`scipy.interpolate`	`interp1d(x, y, kind="cubic")`
Astuces	
Incertitudes fit	`np.sqrt(np.diag(pcov))`
Bornes infinies	`quad(f, -np.inf, np.inf)`
Valeurs initiales	`curve\_fit(f, x, y, p0=[a0, b0])`

Chapitre 11

Bonnes Pratiques — Tests, Documentation, Git

Intuition

Écrire du code qui fonctionne est une chose ; écrire du code **fiable**, **lisible** et **reproductible** en est une autre. En science, un résultat doit être vérifiable et reproductible. Ce chapitre présente les outils et méthodes qui font la différence entre un script jetable et un projet scientifique de qualité : style de code, documentation, tests et contrôle de version.

11.1 Style de code : PEP 8

Définition 11.1 (PEP 8)**. PEP 8** est le guide de style officiel de Python. Il définit des conventions de nommage, d'indentation et de mise en forme pour rendre le code lisible et cohérent.

Python

```python
# MAUVAIS STYLE
def f(x,y,z):
    A=x*y+z
    if(A>10):
        return A
    else:
        return A*2

# BON STYLE (PEP 8)
def calculer_score(longueur, largeur, bonus):
    """Calcule le score en fonction des dimensions et du bonus."""
    score = longueur * largeur + bonus
    if score > 10:
        return score
    else:
        return score * 2
```

Bonne pratique

Règles essentielles de PEP 8 :

- **Indentation** : 4 espaces (jamais de tabulations).
- **Longueur de ligne** : maximum 79 caractères.
- **Noms de variables** : `snake\_case` (ex. : `masse\_molaire`).
- **Noms de classes** : `CamelCase` (ex. : `ParticuleChargee`).
- **Constantes** : `MAJUSCULES` (ex. : `VITESSE\_LUMIERE`).
- **Espaces autour des opérateurs** : `x = 3 + y`, pas `x=3+y`.

Python

```python
# Conventions de nommage
CONSTANTE_BOLTZMANN = 1.380649e-23   # Constante (MAJUSCULES)
temperature_kelvin = 300.0           # Variable (snake_case)

class MoleculeGaz:                   # Classe (CamelCase)
    def __init__(self, masse, vitesse):
        self.masse = masse
        self.vitesse = vitesse

    def energie_cinetique(self):     # M\'ethode (snake_case)
        return 0.5 * self.masse * self.vitesse**2
```

11.2 Docstrings et documentation

Une **docstring** est une chaîne de documentation placée juste après la définition d'une fonction, d'une classe ou d'un module.

Python

```python
def energie_cinetique(masse, vitesse):
    """
    Calcule l'\'energie cin\'etique d'un objet.

    Parameters
    ----------
    masse : float
        Masse de l'objet en kilogrammes (kg).
    vitesse : float
        Vitesse de l'objet en m\`etres par seconde (m/s).

    Returns
    -------
```

```
    float
        \'Energie cin\'etique en joules (J).

    Examples
    --------
    >>> energie_cinetique(2.0, 3.0)
    9.0
    """
    return 0.5 * masse * vitesse**2
```

Python

```
# Acc\'eder `a la documentation
help(energie_cinetique)
```

Sortie

```
Help on function energie_cinetique in module __main__:

energie_cinetique(masse, vitesse)
    Calcule l'energie cinetique d'un objet.

    Parameters
    ----------
    masse : float
        Masse de l'objet en kilogrammes (kg).
    vitesse : float
        Vitesse de l'objet en metres par seconde (m/s).

    Returns
    -------
    float
        Energie cinetique en joules (J).
```

Remarque 11.2. Le style utilisé ci-dessus s'appelle le **style NumPy**. C'est le standard dans l'écosystème scientifique Python. D'autres styles existent (Google, Sphinx), mais le style NumPy est le plus répandu en sciences.

11.3 Tests avec `assert`

Tester son code, c'est comme vérifier une mesure expérimentale : on compare le résultat obtenu au résultat attendu.

Python

```
def celsius_vers_kelvin(temp_c):
```

```
    """Convertit une temp\'erature de Celsius en Kelvin."""
    return temp_c + 273.15

# Tests simples avec assert
assert celsius_vers_kelvin(0) == 273.15, "Erreur : 0°C != 273.15 K"
assert celsius_vers_kelvin(100) == 373.15, "Erreur : 100°C"
assert celsius_vers_kelvin(-273.15) == 0.0, "Erreur : z\'ero absolu"
print("Tous les tests sont pass\'es !")
```

Sortie

```
Tous les tests sont passes !
```

Python

```
import math

def aire_cercle(rayon):
    """Calcule l'aire d'un cercle."""
    if rayon < 0:
        raise ValueError("Le rayon doit \^etre positif.")
    return math.pi * rayon**2

# Tests
assert abs(aire_cercle(1) - math.pi) < 1e-10
assert abs(aire_cercle(0) - 0) < 1e-10
assert abs(aire_cercle(2) - 4 * math.pi) < 1e-10

# Test de l'exception
try:
    aire_cercle(-1)
    assert False, "Aurait d\^u lever une erreur"
except ValueError:
    pass  # C'est le comportement attendu

print("Tous les tests sont pass\'es !")
```

Sortie

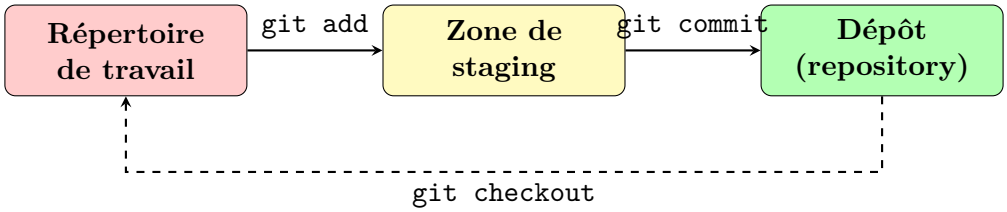
```
Tous les tests sont passes !
```

Attention

Pour les nombres à virgule flottante, ne testez *jamais* l'égalité exacte. Utilisez une tolérance : `abs(obtenu - attendu) < 1e-10` ou `math.isclose(obtenu, attendu)`.

11.4 Contrôle de version avec Git

Définition 11.3 (Git)**. Git** est un système de contrôle de version distribué. Il enregistre l'historique complet des modifications d'un projet, permettant de revenir en arrière, de collaborer et de travailler sur des branches parallèles.



Python

```
# Commandes Git essentielles (dans le terminal)
# Ces commandes sont ex\'ecut\'ees dans un shell, pas en Python

# 1. Initialiser un d\'ep\^ot
# $ git init

# 2. V\'erifier l'\'etat
# $ git status

# 3. Ajouter des fichiers
# $ git add analyse.py
# $ git add .               # Tout ajouter

# 4. Enregistrer (commit)
# $ git commit -m "Ajout de l'analyse initiale"

# 5. Consulter l'historique
# $ git log --oneline
```

Python

Exemple 11.4 (Session Git typique)**.**

```
# Commandes Git typiques dans le terminal :
#
# $ mkdir mon_projet && cd mon_projet
# $ git init
# Initialized empty Git repository in .../mon_projet/.git/
#
# $ echo "print('Bonjour')" > main.py
# $ git add main.py
# $ git commit -m "Premier commit : script principal"
# [master (root-commit) a1b2c3d] Premier commit : script principal
#
# $ echo "# Mon Projet" > README.md
# $ git add README.md
# $ git commit -m "Ajout du README"
#
```

```
# $ git log --oneline
# b4e5f6g Ajout du README
# a1b2c3d Premier commit : script principal
```

11.5 Environnements virtuels

Définition 11.5 (Environnement virtuel)**.** Un **environnement virtuel** est un répertoire isolé contenant une installation Python indépendante. Chaque projet peut ainsi avoir ses propres dépendances sans conflit avec d'autres projets.

Python

```
# Cr\'eer un environnement virtuel
# $ python -m venv mon_env

# L'activer
# Linux/Mac :   $ source mon_env/bin/activate
# Windows :     $ mon_env\Scripts\activate

# Installer des paquets
# (mon_env) $ pip install numpy scipy matplotlib

# Sauvegarder les d\'ependances
# (mon_env) $ pip freeze > requirements.txt

# Reproduire l'environnement ailleurs
# $ pip install -r requirements.txt
```

Bonne pratique

Créez **toujours** un environnement virtuel pour chaque projet. Ajoutez le dossier de l'environnement (`mon_env/`) au fichier `.gitignore` pour ne pas le versionner.

11.6 Structure d'un projet scientifique

Python

```
# Structure recommand\'ee pour un projet scientifique
#
# mon_projet/
# |-- README.md              # Description du projet
# |-- requirements.txt       # D\'ependances
# |-- .gitignore             # Fichiers `a ignorer
# |-- src/
# |   |-- __init__.py
# |   |-- analyse.py         # Fonctions d'analyse
```

```
# |   |-- visualisation.py   # Fonctions de trac\'e
# |-- tests/
# |   |-- test_analyse.py    # Tests unitaires
# |-- data/
# |   |-- mesures.csv        # Donn\'ees brutes
# |-- notebooks/
# |   |-- exploration.ipynb  # Carnets Jupyter
# |-- resultats/
#     |-- figures/            # Graphiques g\'en\'er\'es
```

Python

```
# Exemple de fichier .gitignore
#
# __pycache__/
# *.pyc
# mon_env/
# .ipynb_checkpoints/
# resultats/figures/*.png
```

Ce qui peut mal tourner

Erreurs fréquentes en bonnes pratiques

1. **Ne pas tester son code** — « Ça a l'air de marcher » n'est pas un test. Même un simple `assert` vaut mieux que rien. Les erreurs silencieuses sont les plus dangereuses en science.
2. **Ne pas versionner** — Sans Git, une modification malheureuse peut détruire des heures de travail. Faites des commits **réguliers** avec des messages **descriptifs**.
3. **Noms de variables obscurs** — `x1`, `tmp`, `data2` ne disent rien. Préférez `temperature\_kelvin`, `masse\_molaire`, `concentrations\_initiales`.
4. **Code non documenté** — Dans six mois, vous ne vous souviendrez plus de ce que fait votre fonction. Ajoutez une docstring à chaque fonction.
5. **Tout dans un seul fichier** — Découpez votre projet en modules logiques. Un fichier de 2000 lignes est difficile à maintenir.

Mini-projet scientifique

Organiser un mini-projet scientifique Créez un projet complet en suivant les bonnes pratiques :

1. Créez la structure de répertoires décrite ci-dessus.
2. Initialisez un dépôt Git et créez un `.gitignore`.

3. Dans `src/analyse.py`, écrivez trois fonctions documentées (par exemple : chargement de données, calcul de statistiques, ajustement).
4. Dans `tests/test_analyse.py`, écrivez au moins 5 tests avec `assert`.
5. Créez un environnement virtuel et un `requirements.txt`.
6. Faites au moins 3 commits Git avec des messages clairs.
7. Vérifiez votre historique avec `git log --oneline`.

11.7 Exercices

Exercice 11.1 (⋆)**.** Corrigez le code suivant pour respecter PEP 8. Renommez les variables, ajoutez des espaces et une docstring.

```
def f(L):
    s=0
    for i in L:
        if i>0: s+=i
    return s
```

Exercice 11.2 (⋆)**.** Écrivez une fonction `est\_premier(n)` qui teste si un entier est premier. Ajoutez une docstring au style NumPy et au moins 5 tests avec `assert` (incluant les cas limites : 0, 1, 2, nombres négatifs).

Exercice 11.3 (⋆⋆)**.** Créez un module `physique.py` contenant trois fonctions documentées : `force\_gravitationnelle(m1, m2, d)`, `energie\_potentielle(m, h)` et `periode\_pendule(L)`. Écrivez un fichier de tests séparé vérifiant chaque fonction avec des valeurs connues.

Exercice 11.4 (⋆⋆)**.** Initialisez un dépôt Git dans un nouveau répertoire. Créez un fichier `calculs.py`, faites un premier commit. Modifiez le fichier, faites un deuxième commit. Utilisez `git diff HEAD~1` pour voir les différences. Utilisez `git log --oneline` pour afficher l'historique.

Exercice 11.5 (⋆⋆⋆)**.** Reprenez l'un de vos projets des chapitres précédents (par exemple l'analyse de données Pandas ou l'ajustement SciPy). Réorganisez-le en projet structuré : séparez le code en modules, ajoutez des docstrings à toutes les fonctions, écrivez des tests, créez un environnement virtuel avec `requirements.txt`, et versionnez le tout avec Git (au moins 5 commits).

Fonctions clés

Chapitre	11	—	Bonnes	pratiques	:	l'essentiel

Pratique	Description
PEP 8	Style officiel Python : `snake\_case`, 4 espaces, 79 car.
Docstrings	Documentation intégrée au code (style NumPy)
`assert`	Vérification : `assert obtenu == attendu`
`math.isclose`	Comparaison flottante avec tolérance
`git init`	Initialiser un dépôt
`git add + commit`	Enregistrer des modifications
`git log`	Consulter l'historique
`python -m venv`	Créer un environnement virtuel
`pip freeze`	Sauvegarder les dépendances
`.gitignore`	Exclure des fichiers du suivi Git

Référence rapide Python

.1 Types et opérations

Type	Description
`int`	Entier
`float`	Nombre à virgule flottante
`str`	Chaîne de caractères
`bool`	Booléen (`True`/`False`)
`list`	Liste (modifiable)
`tuple`	Tuple (non modifiable)
`dict`	Dictionnaire (clé-valeur)
`set`	Ensemble (sans doublons)

.2 Fonctions intégrées

Fonction	Description
`print()`	Afficher
`len()`	Longueur
`type()`	Type d'un objet
`range()`	Séquence d'entiers
`input()`	Lire une entrée utilisateur
`int()`, `float()`, `str()`	Conversion de type
`abs()`, `round()`, `max()`, `min()`	Mathématiques de base
`sorted()`, `reversed()`	Tri et inversion
`enumerate()`, `zip()`	Itération avancée

.3 NumPy

Fonction	Description
`np.array()`	Créer un tableau
`np.zeros()`, `np.ones()`	Tableaux préremplis
`np.linspace()`, `np.arange()`	Séquences
`np.sin()`, `np.cos()`, `np.exp()`	Fonctions mathématiques
`np.mean()`, `np.std()`	Statistiques
`np.dot()`, `np.linalg.solve()`	Algèbre linéaire

.4 Référence rapide R

Fonction R	Description
`c()`	Créer un vecteur
`mean()`, `sd()`, `var()`	Statistiques
`t.test()`, `cor.test()`	Tests statistiques
`lm()`	Régression linéaire
`plot()`, `hist()`, `boxplot()`	Graphiques
`read.csv()`, `data.frame()`	Données

Bibliographie

[1] Downey, A.B. (2024). *Think Python*. 3e éd., O'Reilly.

[2] Langtangen, H.P. (2016). *A Primer on Scientific Programming with Python*. 5e éd., Springer.

[3] VanderPlas, J. (2016). *Python Data Science Handbook*. O'Reilly.

[4] Wickham, H. & Grolemund, G. (2023). *R for Data Science*. 2e éd., O'Reilly.