Scientific Programming

Lecture Notes

Licence L2 — 2025–2026

Yaë Ulrich Gaba

"Talk is cheap. Show me the code."
— Linus Torvalds

March 25, 2026

Contents

Preface

Programming has become an indispensable tool for every scientist. Whether you study mathematics, physics, chemistry, or biology, being able to write a program allows you to automate calculations, visualize data, simulate phenomena, and test hypotheses.

These notes are designed for undergraduate students with **no prior programming experience**. We start from scratch and progress step by step, illustrating each concept with examples from the sciences. The goal is to make you autonomous in solving scientific problems with Python (and an introduction to R).

Prerequisites. None in programming. Basic high-school mathematics (functions, equations) is helpful but not required.

Philosophy. Each chapter follows the same pattern:

1. Concept explanation with a scientific analogy.
2. Annotated code with its output.
3. Common errors ("Common Pitfalls").
4. Scientific mini-project.
5. Graded exercises.

References.

- Downey — *Think Python*, O'Reilly (freely available online).
- Langtangen — *A Primer on Scientific Programming with Python*, Springer.
- VanderPlas — *Python Data Science Handbook*, O'Reilly (free online).

Chapter 0 — Installation and Setup

Before you start programming, you need to set up your working environment. This chapter guides you step by step.

Installing Python

Intuition

Python is a free **programming language** used by millions of scientists worldwide. Think of it as a supercharged calculator that understands instructions written in English.

Recommended method: Anaconda

Anaconda is a Python distribution that includes all the scientific libraries we will need.

1. Go to https://www.anaconda.com/download
2. Download the version for your system (Windows, macOS, or Linux).
3. Run the installer and follow the default instructions.
4. Verify the installation by opening a terminal and typing:

Terminal

```
python --version
```

Output

```
Python 3.12.4
```

Lightweight alternative: Python + pip

If you prefer a minimal installation:

1. Download Python from https://www.python.org/downloads/
2. Check "Add Python to PATH" during installation.
3. Install the required libraries:

Terminal

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

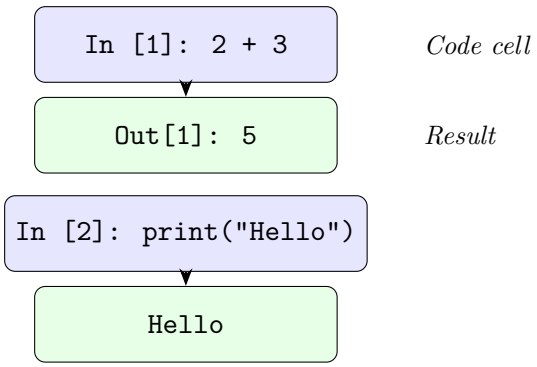
Jupyter Notebook

Jupyter Notebook is an interactive environment where you write code, see results, and add notes — like a digital lab notebook.

Terminal — Launch Jupyter

```
jupyter notebook
```

Your browser opens automatically. Click **New** → **Python 3** to create a new notebook.



Best Practice

Use **Shift + Enter** to execute a cell and move to the next one. Use **Ctrl + Enter** to execute without changing cells.

Text Editors

To write Python scripts (`.py` files), you can use:

- **VS Code** (recommended): free, with Python extension.
- **Spyder**: included in Anaconda, designed for scientists.
- **PyCharm**: free Community edition.

Verifying the Installation

Python — First test

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

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

# Quick test: plot sin(x)
x = np.linspace(0, 2 * np.pi, 100)
plt.plot(x, np.sin(x))
plt.title("sin(x) - Your first plot!")
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.grid(True)
plt.savefig('test_installation.pdf')
plt.show()
print("Plot saved!")
```

Output

```
Installation successful!
NumPy version: 1.26.4
Plot saved!
```

If everything works, you are ready to start Chapter 1!

Installing R (for Chapter 9)

1. Download R from https://cran.r-project.org/
2. Install **RStudio Desktop** from https://posit.co/download/rstudio-desktop/
3. Verify by opening RStudio and typing:

R Console

```
R.version.string
```

Output

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

Common Pitfalls

- **"Python is not recognized"**: you forgot to add Python to PATH. Reinstall with the option checked.

- **"ModuleNotFoundError"**: the library is not installed. Type `pip install module_name`.
- **Permission denied**: on macOS/Linux, try `pip install --user module_name`.

Chapter 1

Introduction to Programming

1.1 Why Program?

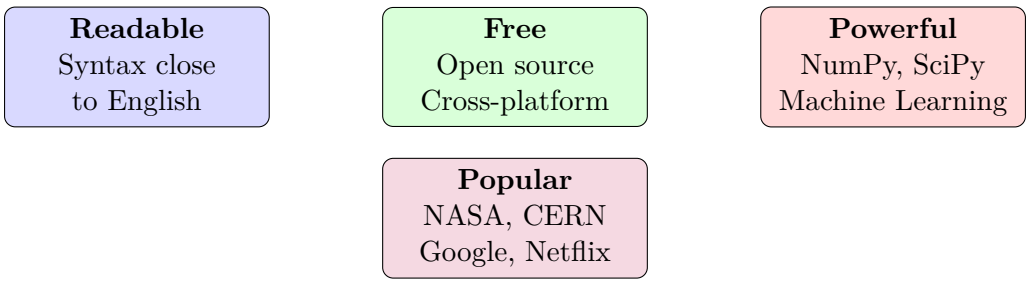
Intuition

A computer program is a **recipe** for the computer. Just as a recipe describes step by step how to prepare a dish, a program describes step by step how to solve a problem.

In science, programming allows you to:

- **Automate** repetitive calculations (compute 10,000 integrals).
- **Visualize** data (plot population growth).
- **Simulate** phenomena (pendulum motion, heat diffusion).
- **Analyze** large datasets (genomics, astronomy).

1.2 Why Python?



1.3 Your First Program

Python

```
print("Hello, future scientist!")
```

Output

```
Hello, future scientist!
```

Congratulations! You just wrote your first program. The `print()` function displays text on screen.

1.4 Python as a Calculator

Python can perform all basic mathematical operations:

Python — Basic calculations

```
# Addition and subtraction
print(3 + 5)
print(10 - 4)

# Multiplication and division
print(6 * 7)
print(15 / 4)      # Real division
print(15 // 4)     # Integer division
print(15 % 4)      # Remainder (modulo)

# Exponentiation
print(2 ** 10)     # 2 to the power 10
```

Output

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

Arithmetic Operators

Operator	Meaning	Example
`+`	Addition	`3 + 5` = 8
`-`	Subtraction	`10 - 4` = 6
`*`	Multiplication	`6 * 7` = 42
`/`	Real division	`15 / 4` = 3.75
`//`	Integer division	`15 // 4` = 3
`%`	Modulo (remainder)	`15 % 4` = 3
`**`	Exponentiation	`2 ** 10` = 1024

1.5 Comments

Comments are notes for humans. Python ignores them completely.

Python

```python
# This is a comment - Python ignores it
print(2 + 3)   # You can comment at end of line

# Speed of light in m/s
c = 299792458
print(f"Speed of light: {c} m/s")
```

Output

```
5
Speed of light: 299792458 m/s
```

Best Practice

Comment your code to explain **why** you are doing something, not **what** you are doing. A good comment says "Speed of light in m/s", not "Assign 299792458 to c".

1.6 Scientific Example: Kinetic Energy

Let us compute the kinetic energy $E_k = \frac{1}{2}mv^2$ of an object:

Python — Kinetic energy

```python
# Data
mass = 75.0         # kg (a person)
velocity = 10.0     # m/s (fast running)

# Compute kinetic energy
Ek = 0.5 * mass * velocity ** 2

print(f"Mass     : {mass} kg")
print(f"Velocity : {velocity} m/s")
print(f"Energy   : {Ek} J")
```

Output

```
Mass     : 75.0 kg
Velocity : 10.0 m/s
Energy   : 3750.0 J
```

1.7 Errors: Don't Be Afraid!

Common Pitfalls

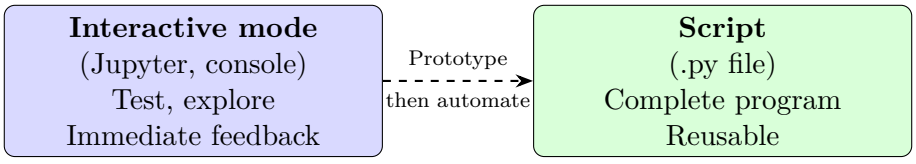
Errors are **normal** and part of learning. Python helps by indicating the error type and line number.

Python — Common errors

```
# Syntax error: missing parenthesis
# print("Hello"
# SyntaxError: '(' was never closed

# Name error: undefined variable
# print(x)
# NameError: name 'x' is not defined

# Type error: impossible operation
# "abc" + 5
# TypeError: can only concatenate str to str

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

Golden rule: always read the error message **from bottom to top**. The last line tells you what went wrong.

1.8 Interactive Mode vs Scripts



1.9 Mini-Project: Unit Conversions

Scientific Mini-Project

Write a program that converts temperatures between Celsius and Fahrenheit.
Formulas: $F = \frac{9}{5}C + 32$ and $C = \frac{5}{9}(F - 32)$

Python — Converter

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

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

# Conversion table
print("\n--- Conversion Table ---")
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}")
```

Output

```
100°C = 212.0°F
72°F = 22.2°C

--- Conversion Table ---
    °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 Exercises

Exercise 1.1 ($\star$ — Guided)**.** Compute the area of a circle with radius $r = 5$ using $A = \pi r^2$. Use `3.14159` for π.

Exercise 1.2 ($\star$ — Guided)**.** Compute the distance between points $(x_1, y_1) = (1, 2)$ and $(x_2, y_2) = (4, 6)$ using $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Hint: $\sqrt{x} = x^{0.5}$.

Exercise 1.3 ($\star\star$ — Independent)**.** Newton's law of gravitation gives the force between two masses: $F = G\frac{m_1 m_2}{r^2}$, with $G = 6.674 \times 10^{-11}$ N·m^2/kg^2. Compute the force between

Earth ($m_1 = 5.972 \times 10^{24}$ kg) and Moon ($m_2 = 7.342 \times 10^{22}$ kg), separated by $r = 3.844 \times 10^8$ m.

Exercise 1.4 ($\star\star\star$ — Scientific Challenge)**.** Compute the de Broglie wavelength $\lambda = h/(mv)$ for an electron ($m = 9.109 \times 10^{-31}$ kg) moving at $v = 10^6$ m/s, with $h = 6.626 \times 10^{-34}$ J·s. Compare with the size of an atom ($\sim 10^{-10}$ m).

Chapter Summary

- `print()` displays text and values.
- Python knows: `+`, `-`, `*`, `/`, `//`, `\%`, `**`.
- Comments start with `##`.
- Errors are normal — read the message from bottom to top.
- A `.py` script is a reusable program.

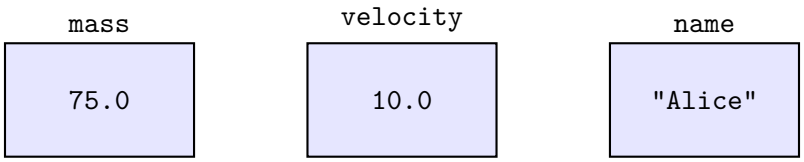
Chapter 2

Variables, Types, and Basic Operations

2.1 Variables: Labeled Boxes

Intuition

A **variable** is like a labeled box in which you store a value. The label (the name) lets you retrieve the content at any time.



Python — Creating variables

```
# Create variables
mass = 75.0           # decimal number (float)
age = 20              # integer (int)
name = "Alice"        # string (str)
is_student = True     # boolean (bool)

print(f"Name: {name}, Age: {age}")
print(f"Mass: {mass} kg")
print(f"Student: {is_student}")
```

Output

```
Name: Alice, Age: 20
Mass: 75.0 kg
Student: True
```

Warning

In Python, you do not "declare" a variable's type. Python figures it out automatically. This is called **dynamic typing**.

2.2 Data Types

Definition 2.1 (Fundamental types)**.** Python has four fundamental types:

- `int` — integer: `42`, `-7`, `0`
- `float` — decimal: `3.14`, `-0.5`, `6.022e23`
- `str` — string: `"Hello"`, `'xyz'`
- `bool` — boolean: `True`, `False`

Python — Checking types

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

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

Output

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

2.3 Scientific Notation

Python — Scientific notation

```
# Physical constants in scientific notation
c = 2.998e8         # speed of light (m/s)
h = 6.626e-34       # Planck's constant (J·s)
Na = 6.022e23       # Avogadro's number (mol⁻¹)
G = 6.674e-11       # gravitational constant

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

Output

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

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

2.4 Type Conversion

Python — Conversions

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

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

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

# Number -> str
message = "The answer is " + str(42)
print(message)
```

Output

```
42.0 <class 'float'>
3 <class 'int'>
25 3.14
The answer is 42
```

2.5 String Operations

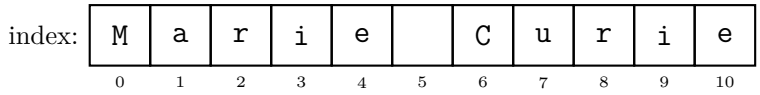
Python — Strings

```python
first = "Marie"
last = "Curie"

# Concatenation
full_name = first + " " + last
print(full_name)

# Repetition
print("-" * 30)

# Length
```

```
print(f"Length: {len(full_name)}")

# Index access (starts at 0!)
print(f"First letter: {full_name[0]}")
print(f"Last letter: {full_name[-1]}")

# Useful methods
print(full_name.upper())
print(full_name.lower())
print(full_name.replace("Curie", "Sklodowska-Curie"))
```

Output

```
Marie Curie
------------------------------
Length: 11
First letter: M
Last letter: e
MARIE CURIE
marie curie
Marie Sklodowska-Curie
```



2.6 f-strings (Formatted Strings)

Python — f-strings

```
element = "Hydrogen"
atomic_mass = 1.008
number = 1

# f-string: insert variables into text
print(f"Element {element} (Z={number}) has mass {atomic_mass} u")

# Number formatting
pi = 3.141592653589793
print(f"pi = {pi:.2f}")       # 2 decimals
print(f"pi = {pi:.6f}")       # 6 decimals
print(f"pi = {pi:.2e}")       # scientific notation

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

Output

```
Element Hydrogen (Z=1) has mass 1.008 u
pi = 3.14
pi = 3.141593
pi = 3.14e+00
H         1.008 u
C        12.011 u
O        15.999 u
```

2.7 Comparison Operators and Booleans

Python — Comparisons

```python
x = 10
print(x > 5)        # True
print(x == 10)       # True (equality)
print(x != 7)        # True (not equal)
print(x >= 15)       # False

# Logical operators
a = True
b = False
print(a and b)       # False
print(a or b)        # True
print(not a)         # False
```

Output

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

Comparison Operators

Operator	Meaning	Example
`==`	Equal to	`5 == 5` → `True`
`!=`	Not equal to	`5 != 3` → `True`
`<`, `>`	Less than, greater than	`3 < 5` → `True`
`<=`, `>=`	Less or equal, greater or equal	`5 >= 5` → `True`
`and`	Logical AND	`True and False` → `False`
`or`	Logical OR	`True or False` → `True`
`not`	Logical NOT	`not True` → `False`

2.8 User Input

Python — input

```python
# input() always returns a string!
# name = input("Your name: ")
# age = int(input("Your age: "))
# print(f"Hello {name}, you are {age} years old.")

# Simulation (for the course)
name = "Alice"
age = 20
print(f"Hello {name}, you are {age} years old.")
```

Output

```
Hello Alice, you are 20 years old.
```

2.9 Common Pitfalls

Common Pitfalls

Common errors

```python
# 1. Forgetting quotes for strings
# name = Alice        # NameError: name 'Alice' is not defined
name = "Alice"        # Correct

# 2. Confusing = (assignment) and == (comparison)
x = 5                 # assigns 5 to x
# x == 5              # compares x to 5, returns True

# 3. Integer division surprise
print(7 / 2)          # 3.5 (Python 3, ok)
print(7 // 2)         # 3 (integer division!)

# 4. Concatenating number and string
# print("I am " + 20 + " years old")  # TypeError
print("I am " + str(20) + " years old")  # OK
print(f"I am {20} years old")            # Even better
```

2.10 Mini-Project: Atom Identity Card

Scientific Mini-Project

Create a program that displays an atom's identity card.

Python

```python
# Identity card: Carbon
element = "Carbon"
symbol = "C"
atomic_number = 6
atomic_mass = 12.011  # u
electronegativity = 2.55  # Pauling

print("=" * 35)
print(f"  IDENTITY CARD: {element}")
print("=" * 35)
print(f"  Symbol             : {symbol}")
print(f"  Atomic number      : {atomic_number}")
print(f"  Atomic mass        : {atomic_mass} u")
print(f"  Electronegativity  : {electronegativity}")
print(f"  Number of protons  : {atomic_number}")
print(f"  Number of electrons: {atomic_number}")
print("=" * 35)
```

Output

```
==================================
  IDENTITY CARD: Carbon
==================================
  Symbol            : C
  Atomic number     : 6
  Atomic mass       : 12.011 u
  Electronegativity : 2.55
  Number of protons : 6
  Number of electrons: 6
==================================
```

2.11 Exercises

Exercise 2.1 ($\star$ — Guided)**.** Create variables for the Boltzmann constant ($k_B = 1.381 \times 10^{-23}$ J/K) and temperature $T = 300$ K. Compute thermal energy $E = k_B T$ and display the result.

Exercise 2.2 ($\star$ — Guided)**.** Create a variable `radius = 6371` (Earth's radius in km). Compute and display the circumference ($C = 2\pi r$) and surface area ($S = 4\pi r^2$) of Earth.

Exercise 2.3 ($\star\star$ — Independent)**.** Write a program that asks the user for a distance in kilometers and converts it to miles (1 km = 0.621 miles), meters, and centimeters.

Exercise 2.4 ($\star\star$ — Independent)**.** Create a program that computes BMI (Body Mass Index): BMI $= \frac{\text{mass (kg)}}{\text{height (m)}^2}$. Display the result with 1 decimal place.

Exercise 2.5 ($\star\star\star$ — Scientific Challenge)**.** The photon energy formula is $E = hf = hc/\lambda$. Compute the energy of a red photon ($\lambda = 700$ nm) and a blue photon ($\lambda = 450$ nm). Express results in joules and electron-volts (1 eV $= 1.602 \times 10^{-19}$ J).

Chapter Summary

- A variable stores a value: `x = 42`.
- Types: `int`, `float`, `str`, `bool`.
- `type(x)` gives the type, `int()`/`float()`/`str()` convert.
- Scientific notation: `6.022e23` $= 6.022 \times 10^{23}$.
- f-strings: `f"pi = \{pi:.2f\}"` for formatting.
- Comparison operators: `==`, `!=`, `<`, `>`, `and`, `or`.

Chapter 3

Control Structures — Conditions and Loops

Intuition

Imagine a cooking recipe: "If the batter is too runny, add flour; otherwise, bake for 30 minutes." A program works in exactly the same way: it makes *decisions* and *repeats* steps according to conditions. Control structures are the heart of every scientific algorithm.

3.1 Conditions: `if`, `elif`, `else`

In science, we often classify according to thresholds. Consider the *phase transitions* of water:

Python

```python
temperature = -5  # in degrees Celsius

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

print(f"At {temperature} C, water is {etat}.")
```

Output

```
At -5 C, water is solide.
```

Remark 3.1. Python evaluates conditions *in order*. As soon as a condition is true, the corresponding block is executed and the remaining ones are skipped.

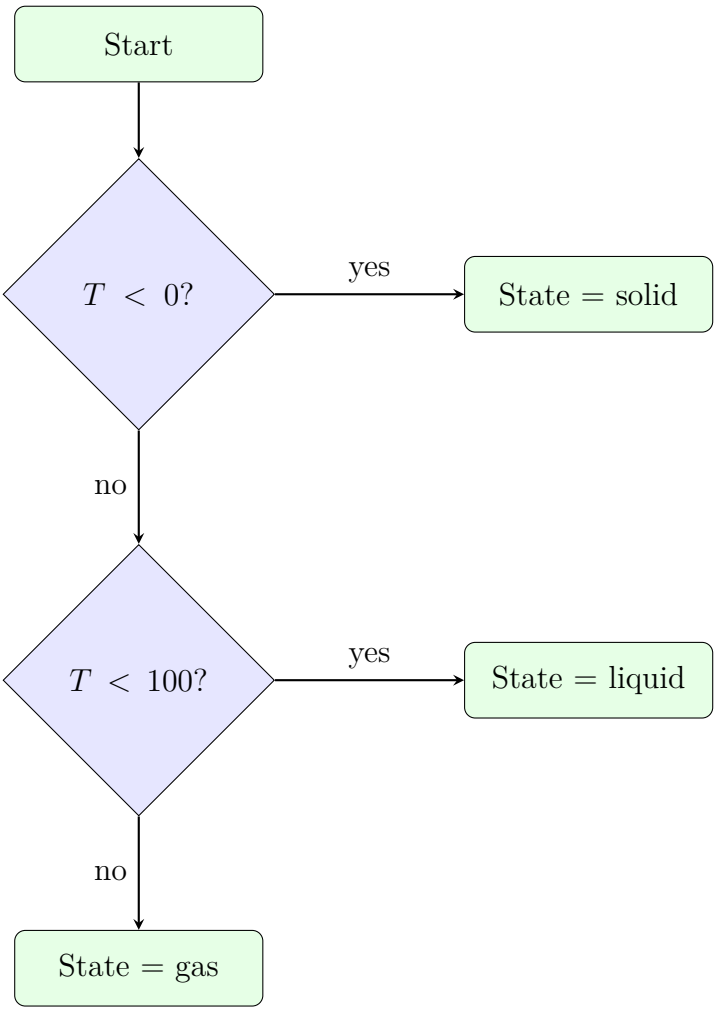


Figure 3.1: Flowchart of an `if/elif/else` structure for phase transitions.

3.1.1 Comparison operators and logical operators

Definition 3.2 (Comparison operators)**.** The operators `==`, `!=`, `<`, `>`, `<=`, `>=` return a Boolean (`True` or `False`). Conditions can be combined with `and`, `or`, `not`.

Python

```
pH = 3.2

if pH < 7 and pH >= 0:
    print("Acidic solution")
elif pH == 7:
    print("Neutral solution")
elif pH > 7 and pH <= 14:
    print("Basic solution")
else:
    print("Invalid pH value")
```

Output

```
Acidic solution
```

3.2 The `while` loop

The `while` loop repeats a block *as long as a condition is true*. In science, it is often used for *convergence* algorithms.

Example 3.3 (Convergence of a sequence)**.** The sequence $u_{n+1} = \frac{1}{2}\left(u_n + \frac{a}{u_n}\right)$ converges to $\sqrt{a}$ (Heron's method).

Python

```
a = 2.0
u = 1.0        # initial value
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 in {iterations} iterations")
```

Output

```
sqrt(2.0) = 1.4142135623730951
Convergence in 4 iterations
```

Warning

A `while` loop whose condition *never* becomes false runs forever! Always add a safeguard:

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

3.3 The `for` loop and `range()`

The `for` loop iterates over an *iterable* (list, string, `range`, etc.).

Python

```python
# Compute the sum of squares from 1 to 10
somme = 0
for i in range(1, 11):
    somme += i**2

print(f"Sum of squares from 1 to 10 = {somme}")
# Verification: n(n+1)(2n+1)/6
print(f"Formula: {10 * 11 * 21 // 6}")
```

Output

```
Sum of squares from 1 to 10 = 385
Formula: 385
```

Definition 3.4 (`range(start, stop, step)`). `range(a, b, s)` generates integers from `a` to `b-1` with a step of `s`. Note: the upper bound `b` is *excluded*.

Python

```python
# Display multiples of 3 between 0 and 15
for n in range(0, 16, 3):
    print(n, end=" ")
```

Output

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

3.4 Nested loops

Python

```python
# Multiplication table (excerpt)
for i in range(1, 4):
    for j in range(1, 4):
        print(f"{i} x {j} = {i*j:2d}", end="   ")
    print()  # newline
```

Output

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

3.5 `break` and `continue`

Python

```python
# Find the first prime number > 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       # no need to test other divisors
    if est_premier:
        print(f"First prime > 50: {n}")
        break           # exit the while loop
    n += 1
```

Output

```
First prime > 50: 53
```

Python

```python
# Skip negative values in measurements
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   # skip to the next iteration
    somme += m
    compteur += 1

print(f"Mean (positive values) = {somme/compteur:.2f}")
```

Output

```
Mean (positive values) = 4.43
```

3.6 Common errors

Common Pitfalls

1. **Indentation error**: Python uses indentation to delimit blocks. Mixing spaces and tabs causes an `IndentationError`. Rule: always use **4 spaces**.
2. **Infinite loop**: forgetting to increment the control variable in a `while` leads to a loop that never stops.

```
# ERROR: i never changes!
i = 0
while i < 10:
    print(i)    # prints 0 indefinitely
```

3. **Off-by-one error**: `range(1, 10)` produces integers from 1 to **9**, not 10! To include 10, write `range(1, 11)`.
4. **Using `=` instead of `==`**: `=` is assignment, `==` is comparison. Writing `if x = 5` causes a `SyntaxError`.

3.7 Mini-project: Radioactive Decay

Scientific Mini-Project

The law of radioactive decay gives the number of remaining atoms:

$$N(t) = N_0 \times 0.5^{t/t_{1/2}}$$

where N_0 is the initial number of atoms and $t_{1/2}$ is the half-life.
Objective: simulate the decay of carbon-14 ($t_{1/2} = 5730$ years) and display the number of remaining atoms every 1000 years.

Python

```
N0 = 1_000_000        # initial number of atoms
demi_vie = 5730       # half-life of C-14 in years
seuil = N0 * 0.01     # stop when less than 1% remains

t = 0
N = N0
print(f"{'Time (yrs)':>12} {'N remaining':>12} {'% remaining':>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"\nAfter {t} years, less than 1% of the atoms remain.")
```

Output

```
 Time (yrs)   N remaining  % remaining
------------------------------------
           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%

After 38000 years, less than 1% of the atoms remain.
```

3.8 Exercises

Exercise 3.1 ($\star$ — BMI Classification)**.** Write a program that asks for a person's weight (kg) and height (m), computes the BMI (BMI = weight/height^2) and displays the category: underweight (< 18.5), normal (18.5–25), overweight (25–30), obese (≥ 30).

Exercise 3.2 ($\star$ — Countdown)**.** Use a `while` loop to display a countdown from 10 to 0, then display "Liftoff!".

Exercise 3.3 ($\star\star$ — Fibonacci Sequence)**.** Compute the first 20 terms of the Fibonacci sequence ($F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$) with a `for` loop. Verify that the ratio F_n/F_{n-1} converges to the golden ratio $\varphi \approx 1.618$.

Exercise 3.4 ($\star\star$ — Sieve of Eratosthenes)**.** Use nested loops to find all prime numbers less than 100.

Exercise 3.5 ($\star\star\star$ — Random Walk Simulation)**.** Simulate a 1D random walk: at each step, the particle moves by $+1$ or -1 with equal probability. After $N = 1000$ steps, display the final position. Repeat 100 times and compute the mean distance $\langle|x|\rangle$. Compare with the theoretical prediction $\sqrt{N} \approx 31.6$. *Hint:* use `import random` and `random.choice([-1, 1])`.

Key Functions

Chapter Summary: Control Structures

- **Conditions**: `if condition:` / `elif condition:` / `else:`
- `while` **loop**: repeats as long as the condition is true
- `for` **loop**: iterates over an iterable (`range()`, list, etc.)
- `range(a, b, s)`: integers from `a` to `b-1`, step `s`
- `break`: immediately exits the loop
- `continue`: skips to the next iteration
- **Indentation**: 4 spaces, always consistent
- **Safeguard**: always set a maximum number of iterations for `while`

Chapter 4

Functions and Scope

Intuition

A function is like a *machine* in a laboratory: you feed it *inputs* (reagents), it performs a *process* (chemical reaction) and produces an *output* (product). Rather than rebuilding the machine for each experiment, you reuse it. This is the **DRY** principle: "*Don't Repeat Yourself*".

4.1 Why use functions?

- **Reusability**: write a formula once and call it everywhere.
- **Readability**: a program broken into functions is easier to understand.
- **Debugging**: each function can be tested independently.
- **Modularity**: each function has a single responsibility.

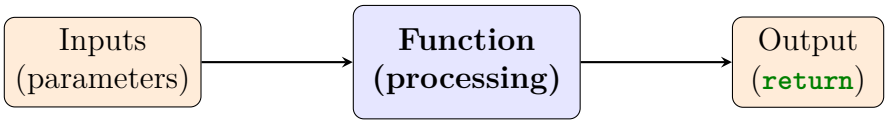


Figure 4.1: A function viewed as a machine: inputs → processing → output.

4.2 Defining and calling a function

Definition 4.1 (Function syntax)

```
def function_name(parameter1, parameter2):
    """Docstring: description of the function."""
    # function body
    return result
```

Python

```python
def energie_cinetique(masse, vitesse):
    """Compute the kinetic energy E = 0.5 * m * v^2."""
    return 0.5 * masse * vitesse**2

# Call the function
E = energie_cinetique(2.0, 3.0)
print(f"E = {E} J")
```

Output

```
E = 9.0 J
```

4.3 Docstrings

A *docstring* is a documentation string placed right after `def`. It is accessible via `help(function_name)`.

Python

```python
def force_gravitationnelle(m1, m2, r):
    """
    Compute the gravitational force between two masses.

    Parameters
    ----------
    m1 : float -- mass 1 (kg)
    m2 : float -- mass 2 (kg)
    r  : float -- distance between centers (m)

    Returns
    -------
    float -- force in Newtons
    """
    G = 6.674e-11  # gravitational constant
    return G * m1 * m2 / r**2

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

Output

```
Earth-Moon force: 1.98e+20 N
```

4.4. DEFAULT ARGUMENTS AND KEYWORD ARGUMENTS

4.4 Default arguments and keyword arguments

Python

```python
def chute_libre(t, g=9.81, v0=0):
    """Position of an object in free fall: y = v0*t + 0.5*g*t^2."""
    return v0 * t + 0.5 * g * t**2

# Using default values
print(f"Earth: y = {chute_libre(2.0):.2f} m")

# On the Moon (g = 1.62 m/s^2)
print(f"Moon : y = {chute_libre(2.0, g=1.62):.2f} m")

# With initial velocity
print(f"With v0=5: y = {chute_libre(2.0, v0=5):.2f} m")
```

Output

```
Earth: y = 19.62 m
Moon : y = 3.24 m
With v0=5: y = 29.62 m
```

Best Practice

Use keyword arguments to improve readability: `chute\_libre(t=2.0, g=1.62)` is clearer than `chute\_libre(2.0, 1.62)`.

4.5 Functions with multiple return values

Python

```python
import math

def coordonnees_polaires(x, y):
    """Convert Cartesian coordinates to polar coordinates."""
    r = math.sqrt(x**2 + y**2)
    theta = math.atan2(y, x)
    return r, theta    # returns a tuple

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

Output

```
r = 5.00, theta = 53.1 deg
```

4.6 Variable scope: local vs global

Definition 4.2 (Scope)**.** A variable defined *inside* a function is **local**: it only exists during the execution of the function. A variable defined outside is **global**.

Python

```python
x = 10          # global variable

def ma_fonction():
    x = 5       # local variable (different!)
    print(f"Inside the function: x = {x}")

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

Output

```
Inside the function: x = 5
Outside: x = 10
```

Warning

Avoid using `global` to modify global variables from within a function. *Always* prefer passing values as parameters and using `return`.

4.7 Lambda functions

`lambda` functions are short anonymous functions, useful for simple operations.

Python

```python
# Lambda function to convert Celsius to 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")
```

Output

```
 -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 Common errors

Common Pitfalls

1. **Forgetting `return`**: without `return`, a function returns `None` by default. The computation is performed but the result is lost!

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

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

2. **Mutable default argument**: using a list as a default value is dangerous, because it is shared across all calls!

```
# ERROR: the list is shared
def ajouter(element, liste=[]):
    liste.append(element)
    return liste

print(ajouter(1))  # [1]
print(ajouter(2))  # [1, 2] instead of [2]!

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

3. **Confusing `print` and `return`**: `print` displays on screen but does not return a value usable in an expression.

4.9 Mini-project: Physics Formula Calculator

Scientific Mini-Project

Create a module containing the following functions:

- `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)`: $V = RI$ (provide two quantities, compute the third)

Each function must have a complete docstring. Test them with known values.

Python

```python
import math

def periode_pendule(L, g=9.81):
    """Period of a simple pendulum: T = 2*pi*sqrt(L/g)."""
    return 2 * math.pi * math.sqrt(L / g)

def loi_ohm(V=None, R=None, I=None):
    """Ohm's law: V = R * I. Provide two quantities."""
    if V is None:
        return R * I
    elif R is None:
        return V / I
    elif I is None:
        return V / R

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

Output

```
Pendulum L=1m: T = 2.006 s
Ohm's law: V = 50.0 V
Ohm's law: I = 0.120 A
```

4.10 Exercises

Exercise 4.1 ($\star$ — Temperature Conversion)**.** Write two functions `celsius\_to\_fahrenheit(T)` and `fahrenheit\_to\_celsius(T)`. Test with boiling water (100 °C = 212 °F).

Exercise 4.2 ($\star$ — Factorial)**.** Write a function `factorielle(n)` that computes $n!$ using a loop. Verify that `factorielle(10) == 3628800`.

Exercise 4.3 ($\star\star$ — Recursive Function)**.** Rewrite the factorial in a *recursive* manner: a function that calls itself. What is the base case?

Exercise 4.4 ($\star\star$ — Descriptive Statistics)**.** Write a function `statistiques(donnees)` that receives a list of numbers and returns the mean, the median, and the standard deviation (without using any library).

Exercise 4.5 ($\star\star\star$ — Newton's Method)**.** Implement Newton's method for finding the zeros of a function: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. Your function should take as parameters `f`, `df` (derivative), `x0` (initial estimate), and `tol` (tolerance). Test with $f(x) = x^2 - 2$ to recover $\sqrt{2}$.

Key Functions

Chapter Summary: Functions and Scope

- **Definition**: `def name(params): ... return result`
- **Docstring**: documentation string in triple quotes
- **Default arguments**: `def f(x, n=10):` — n defaults to 10 if not specified
- **Keyword arguments**: `f(x=3, n=20)` for clarity
- **Multiple values**: `return a, b` returns a tuple
- **Local scope**: variables inside a function are isolated
- **Lambda**: `lambda x: x**2` for short functions
- **DRY principle**: never copy-paste code, write a function

Chapter 5

Data Structures

Intuition

A scientist never works with a single data point: they handle series of measurements, tables of results, catalogs of species. Python offers several *data structures* to organize this information — each with its strengths, like the different containers in a laboratory (test tubes, Petri dishes, binders).

5.1 Lists

Definition 5.1 (List)**.** A **list** is an *ordered* and *mutable* collection of elements. It is created with square brackets: `[e1, e2, ...]`.

5.1.1 Creation and indexing

Python

```
# Temperature measurements (deg C) over one week
temperatures = [18.2, 19.5, 17.8, 21.0, 22.3, 20.1, 19.7]

print(f"First day    : {temperatures[0]} C")
print(f"Last day     : {temperatures[-1]} C")
print(f"Number of measurements: {len(temperatures)}")
```

Output

```
First day    : 18.2 C
Last day     : 19.7 C
Number of measurements: 7
```

5.1.2 Slicing

Python

```python
# Weekdays (Monday to Friday)
jours_ouvres = temperatures[0:5]
print(f"Weekdays: {jours_ouvres}")

# Every other element
print(f"Every other: {temperatures[::2]}")

# Reverse order
print(f"Reversed: {temperatures[::-1]}")
```

Output

```
Weekdays: [18.2, 19.5, 17.8, 21.0, 22.3]
Every other: [18.2, 17.8, 22.3, 19.7]
Reversed: [19.7, 20.1, 22.3, 21.0, 17.8, 19.5, 18.2]
```

5.1.3 Useful methods

Python

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

masses.append(9.8)        # add to the end
print(f"After append: {masses}")

dernier = masses.pop()    # remove the last element
print(f"Removed: {dernier}, list: {masses}")

masses.sort()             # sort in place
print(f"Sorted: {masses}")

masses.reverse()          # reverse in place
print(f"Reversed: {masses}")
```

Output

```
After append: [12.5, 8.3, 15.7, 10.2, 9.8]
Removed: 9.8, list: [12.5, 8.3, 15.7, 10.2]
Sorted: [8.3, 10.2, 12.5, 15.7]
Reversed: [15.7, 12.5, 10.2, 8.3]
```

5.2 List comprehensions

List comprehensions offer a compact syntax for creating lists.

Python

```python
# Squares of integers from 1 to 10
carres = [x**2 for x in range(1, 11)]
print(f"Squares: {carres}")

# Filter temperatures > 20 deg 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"Hot days: {chaudes}")

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

Output

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

5.3 Tuples

Definition 5.2 (Tuple)**.** A **tuple** is an *ordered* but *immutable* (non-modifiable) collection. It is created with parentheses: `(e1, e2, ...)`.

Python

```python
# Physical constants (must not change)
constantes = ("speed of light", 299_792_458, "m/s")
print(f"{constantes[0]} = {constantes[1]} {constantes[2]}")

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

# Attempting to modify -> error
# constantes[1] = 300_000_000  # TypeError!
```

Output

```
speed of light = 299792458 m/s
speed of light: 299792458 m/s
```

Remark 5.3. Use tuples for data that should not be modified: coordinates (x, y), physical constants, dictionary keys.

5.4 Dictionaries

Definition 5.4 (Dictionary)**.** A **dictionary** maps *keys* to *values*. It is created with curly braces: `\{key: value, ...\}`.

5.4.1 The periodic table as a dictionary

Python

```python
elements = {
    "H":  {"name": "Hydrogen",  "Z": 1,  "mass": 1.008},
    "He": {"name": "Helium",    "Z": 2,  "mass": 4.003},
    "C":  {"name": "Carbon",    "Z": 6,  "mass": 12.011},
    "N":  {"name": "Nitrogen",  "Z": 7,  "mass": 14.007},
    "O":  {"name": "Oxygen",    "Z": 8,  "mass": 15.999},
    "Fe": {"name": "Iron",      "Z": 26, "mass": 55.845},
}

# Access an element
carbone = elements["C"]
print(f"Carbon: Z={carbone['Z']}, mass={carbone['mass']} u")

# Molar mass of water H2O
masse_eau = 2 * elements["H"]["mass"] + elements["O"]["mass"]
print(f"Molar mass H2O = {masse_eau:.3f} g/mol")
```

Output

```
Carbon: Z=6, mass=12.011 u
Molar mass H2O = 18.015 g/mol
```

5.4.2 Iterating over a dictionary

Python

```python
# Display all elements
for symbole, info in elements.items():
    print(f"{symbole:>2} : {info['name']:<12} Z={info['Z']:>2}")
```

Output

```
 H : Hydrogen     Z= 1
He : Helium       Z= 2
 C : Carbon       Z= 6
 N : Nitrogen     Z= 7
 O : Oxygen       Z= 8
Fe : Iron         Z=26
```

5.5 Sets

Python

```
# Elements detected in two samples
echantillon_A = {"H", "C", "O", "N", "S"}
echantillon_B = {"H", "C", "O", "Fe", "Ca"}

print(f"Common      : {echantillon_A & echantillon_B}")
print(f"All         : {echantillon_A | echantillon_B}")
print(f"Only in A   : {echantillon_A - echantillon_B}")
```

Output

```
Common      : {'O', 'H', 'C'}
All         : {'S', 'O', 'Fe', 'H', 'Ca', 'N', 'C'}
Only in A   : {'S', 'N'}
```

List	Tuple	Dictionary	Set
Ordered	Ordered	Key → Value	Unordered
Mutable	Immutable	Mutable	No duplicates
[1, 2, 3]	(1, 2, 3)	\{"a": 1\}	\{1, 2, 3\}

Figure 5.1: Comparison of data structures in Python.

5.6 Common errors

Common Pitfalls

1. **Index out of bounds** (`IndexError`): a list with 5 elements has indices from 0 to 4. Accessing index 5 causes an error.

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

2. **Modifying a list during iteration**: removing elements from a list while iterating over it produces unpredictable results.

```
# ERROR: skips elements
donnees = [1, -2, 3, -4, 5]
for d in donnees:
    if d < 0:
        donnees.remove(d)  # Dangerous!
```

```python
# CORRECT: create a new list
donnees = [d for d in donnees if d >= 0]
```

3. **Missing key** (`KeyError`): accessing a key that does not exist in a dictionary. Use `dict.get(key, default)` to avoid the error.
4. **Confusing list and tuple**: attempting to modify a tuple with `t[0] = 5` causes a `TypeError`.

5.7 Mini-project: Student Grade Management

Scientific Mini-Project

Create a grade management system using a dictionary:

- Keys: student names
- Values: lists of grades
- Functions: add a student, add a grade, compute the average, find the top student

Python

```python
def creer_promotion():
    """Create an empty class dictionary."""
    return {}

def ajouter_etudiant(promo, nom):
    """Add a student with an empty grade list."""
    promo[nom] = []

def ajouter_note(promo, nom, note):
    """Add a grade for a student."""
    promo[nom].append(note)

def moyenne(notes):
    """Compute the average of a list of grades."""
    return sum(notes) / len(notes) if notes else 0

def afficher_resultats(promo):
    """Display the results of the class."""
    print(f"{'Student':<15} {'Grades':<25} {'Average':>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
# Usage
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"\nTop student: {major} ({moyenne(promo[major]):.2f}/20)")
```

Output

```
Student         Grades                        Average
--------------------------------------------------
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

Top student: Claire (17.83/20)
```

5.8 Exercises

Exercise 5.1 ($\star$ — List Manipulation)**.** Create a list of 10 pH measurements. Compute the mean, minimum, and maximum *without using* the `sum`, `min`, `max` functions.

Exercise 5.2 ($\star$ — Filtering)**.** From a list of temperatures, create a new list using a list comprehension containing only values between 15 °C and 25 °C.

Exercise 5.3 ($\star\star$ — Frequency Counter)**.** Write a function that receives a string and returns a dictionary counting the frequency of each letter. Test with a DNA sequence ("ATCGGATCCA").

Exercise 5.4 ($\star\star$ — Matrix as a List of Lists)**.** Represent a 3×3 matrix as a list of lists. Write a function that computes the transpose. Verify with the identity matrix.

Exercise 5.5 ($\star\star\star$ — Protein Sequence Analysis)**.** Create a dictionary mapping each amino acid to its molar mass. Write a function that, given a sequence of amino acids (letters), computes the total mass of the protein. Test with "MKVLWA".

Key Functions

Chapter Summary: Data Structures

- **List** `[...]`: ordered, mutable, methods `append`, `pop`, `sort`
- **Comprehension**: `[expr for x in iterable if cond]`
- **Tuple** `(...)`: ordered, immutable, ideal for constants
- **Dictionary** `\{key: val\}`: access by key, methods `.keys()`, `.values()`, `.items()`
- **Set** `\{...\}`: no duplicates, operations $\cap$, $\cup$, $\setminus$
- **Indexing**: starts at 0, negative indices from the end
- **Slicing**: `list[start:stop:step]`

Chapter 6

NumPy — Arrays and Numerical Computing

Intuition

Imagine having to measure the temperature at 1,000 points on a hot plate. With a Python list, each operation requires a loop. NumPy lets you process *all points in a single instruction*, like an instrument that measures the entire plate at once. It is the difference between counting grains of sand one by one and weighing the whole pile.

6.1 Why NumPy?

Python

```python
import time

# Comparison: sum of squares from 1 to 1,000,000
n = 1_000_000

# With a Python list
debut = time.time()
resultat_liste = sum([i**2 for i in range(n)])
temps_liste = time.time() - debut

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

print(f"Python list: {temps_liste:.4f} s")
print(f"NumPy      : {temps_numpy:.4f} s")
print(f"Speedup    : x{temps_liste/temps_numpy:.0f}")
```

Output

```
Python list: 0.2851 s
NumPy      : 0.0032 s
Speedup    : x89
```

6.2 Creating arrays

Python

```python
import numpy as np

# From a list
a = np.array([1.0, 2.0, 3.0, 4.0])
print(f"Array     : {a}")

# Special arrays
zeros = np.zeros(5)
print(f"Zeros     : {zeros}")

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

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

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

Output

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

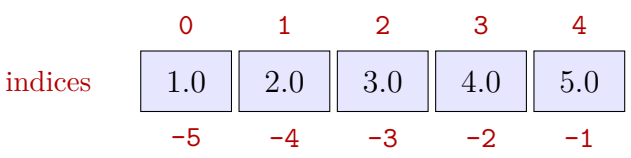


Figure 6.1: Indexing a NumPy array: positive indices (top) and negative indices (bottom).

6.3 Element-wise operations

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}")       # comparison
```

Output

```
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]
```

Definition 6.1 (Broadcasting)**. Broadcasting** allows NumPy to apply operations between arrays of different sizes. A scalar is automatically "extended" to match the size of the array.

6.4 Indexing and slicing

Python

```python
import numpy as np

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

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

# Boolean indexing (fancy indexing)
masque = x > 35
print(f"Mask          : {masque}")
print(f"Values > 35   : {x[masque]}")

# Conditional modification
x[x < 30] = 0
print(f"After modif   : {x}")
```

Output

```
Element 3     : 40
Slice [2:5]   : [30 40 50]
Mask          : [False False False  True  True  True  True]
Values > 35   : [40 50 60 70]
After modif   : [ 0  0 30 40 50 60 70]
```

6.5 Mathematical functions

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: damped signal
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)}")
```

Output

```
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 Descriptive statistics

Python

```python
import numpy as np

# Experimental measurements (concentration in mg/L)
mesures = np.array([12.3, 11.8, 12.5, 11.9, 12.1, 12.4, 12.0, 11.7])

print(f"Mean        : {np.mean(mesures):.2f} mg/L")
```

```
print(f"Std dev      : {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"Median       : {np.median(mesures):.2f} mg/L")
```

Output

```
Mean         : 12.09 mg/L
Std dev      : 0.27 mg/L
Minimum      : 11.70 mg/L
Maximum      : 12.50 mg/L
Median       : 12.05 mg/L
```

6.7 Elementary linear algebra

Python

```
import numpy as np

# 2x2 matrices
A = np.array([[1, 2],
              [3, 4]])

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

# Matrix product
C = A @ B    # equivalent to np.dot(A, B)
print("A @ B =")
print(C)

# Determinant and 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}")
```

Output

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

det(A) = -2.0
A^(-1) =
[[-2.   1. ]
 [ 1.5 -0.5]]
```

6.8 Random numbers

Python

```python
import numpy as np

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

# Uniform between 0 and 1
u = rng.uniform(0, 1, size=5)
print(f"Uniform   : {np.round(u, 3)}")

# Normal distribution (mean=0, std=1)
n = rng.normal(0, 1, size=5)
print(f"Normal    : {np.round(n, 3)}")

# Random integers (dice roll)
des = rng.integers(1, 7, size=10)
print(f"Rolls     : {des}")
```

Output

```
Uniform   : [0.773 0.438 0.859 0.697 0.094]
Normal    : [ 0.314  0.459 -0.537  0.268 -0.232]
Rolls     : [5 2 3 6 1 4 2 5 3 6]
```

6.9 Common errors

Common Pitfalls

1. **Dimension mismatch (`shape mismatch`)**: operating on arrays of incompatible sizes causes a `ValueError`.

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

2. **Forgetting element-wise operations**: the `*` operator between two arrays performs element-wise multiplication, *not* matrix multiplication. Use `@` or `np.dot` for the matrix product.

3. **Confusing `np.arange` / `np.linspace`**: `np.arange(0, 1, 0.1)` specifies a *step*, `np.linspace(0, 1, 10)` specifies a *number of points*.

4. **Modifying a view instead of a copy**: NumPy slicing creates a *view* (not a copy). Modifying the view modifies the original!

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

```
b = a[1:3]      # view, not copy
b[0] = 99       # a becomes [1, 99, 3, 4]!
b = a[1:3].copy()  # independent copy
```

6.10 Mini-project: Estimating π with Monte Carlo

Scientific Mini-Project

The Monte Carlo method estimates π by drawing random points in a square $[-1, 1]^2$ and counting those that fall inside the unit circle ($x^2 + y^2 \leq 1$). The ratio gives:

$$\pi \approx 4 \times \frac{\text{points inside the circle}}{\text{total points}}$$

Python

```python
import numpy as np

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

# Draw N random points in [-1, 1] x [-1, 1]
x = rng.uniform(-1, 1, size=N)
y = rng.uniform(-1, 1, size=N)

# Count points inside the unit circle
dans_cercle = x**2 + y**2 <= 1
nb_dans_cercle = np.sum(dans_cercle)

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

print(f"Total points       : {N:,}")
print(f"Points in circle   : {nb_dans_cercle:,}")
print(f"Estimated pi       : {pi_estime:.6f}")
print(f"Exact pi           : {np.pi:.6f}")
print(f"Error              : {erreur:.6f}")
```

Output

```
Total points       : 1,000,000
Points in circle   : 785,436
Estimated pi       : 3.141744
Exact pi           : 3.141593
Error              : 0.000151
```

6.11 Exercises

Exercise 6.1 ($\star$ — Array Creation)**.** Create the following arrays: (a) integers from 0 to 99, (b) 50 equally spaced values between $-\pi$ and π, (c) a 3×3 matrix of zeros with 1s on the diagonal (identity matrix).

Exercise 6.2 ($\star$ — Temperature Conversion)**.** Create an array of temperatures in Celsius from -40 to 100 (step of 10). Convert it to Fahrenheit ($F = 1.8 \times C + 32$) *without a loop.*

Exercise 6.3 ($\star\star$ — Experimental Data Analysis)**.** Generate 1000 simulated measurements from a normal distribution ($\mu = 50$, $\sigma = 5$). Compute the mean, standard deviation, and count how many measurements are more than 2σ from the mean. Compare with the theoretical prediction ($\approx 5\%$).

Exercise 6.4 ($\star\star$ — Dot Product)**.** Compute the angle between vectors $\vec{u} = (1, 2, 3)$ and $\vec{v} = (4, 5, 6)$ using the formula $\cos\theta = \frac{\vec{u}\cdot\vec{v}}{|\vec{u}||\vec{v}|}$.

Exercise 6.5 ($\star\star\star$ — Solving a Linear System)**.** Solve the following system of linear equations using `np.linalg.solve`:

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

Verify your result by computing $A\vec{x}$ and comparing with $\vec{b}$.

Key Functions

Chapter Summary: NumPy

- **Creation**: `np.array`, `np.zeros`, `np.ones`, `np.linspace`, `np.arange`
- **Operations**: `+`, `-`, `*`, `/`, `**` are applied element-wise
- **Broadcasting**: a scalar is automatically extended
- **Math**: `np.sin`, `np.cos`, `np.exp`, `np.log`, `np.sqrt`
- **Stats**: `np.mean`, `np.std`, `np.min`, `np.max`, `np.median`
- **Algebra**: `A @ B` (matrix product), `np.linalg.det`, `np.linalg.inv`, `np.linalg.solve`
- **Random**: `rng = np.random.default\_rng(seed)`
- **Boolean indexing**: `x[x > 0]` filters positive elements

Chapter 7

Matplotlib — Scientific Visualization

Intuition

"A picture is worth a thousand words," and in science, a graph is worth a thousand data tables. Matplotlib is the reference library in Python for creating publication-quality figures. Think of it as the "digital graph paper" of the modern scientist.

7.1 First plot: `plt.plot()`

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("Trigonometric functions")
plt.legend()
plt.grid(True)
plt.show()
```

Output

```
[Figure displayed: two curves sin(x) and cos(x) on [0, 2pi],
with legend, grid, and axis labels.]
```

Remark 7.1. The convention is to import `matplotlib.pyplot as plt`. All plotting functions are then accessible via `plt.function\_name()`.

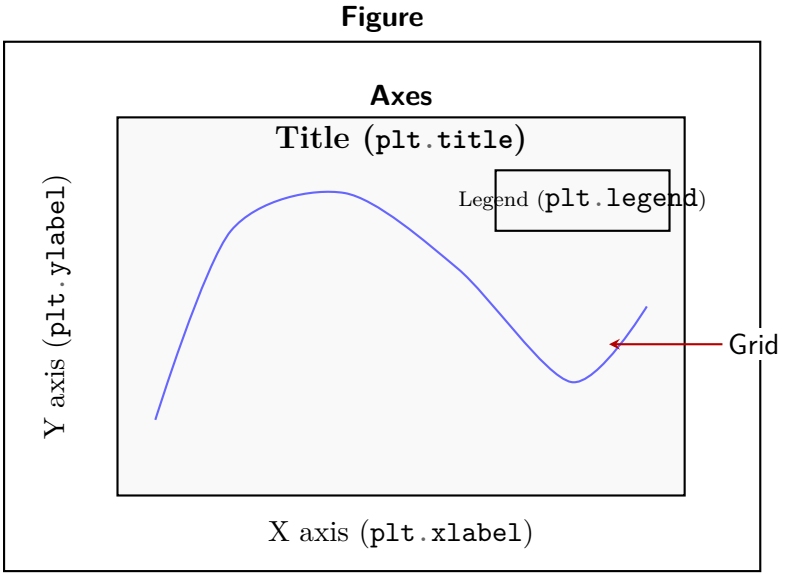


Figure 7.1: Anatomy of a Matplotlib figure: figure, axes, title, labels, and legend.

7.2 Scatter plots: `plt.scatter()`

Python

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

# Simulate experimental data
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="Measurements")

# Regression line
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"Regression: y = {coeffs[0]:.2f}x + {coeffs[1]:.2f}")

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

Output

```
[Figure displayed: blue scatter points with red regression line,
title "Beer-Lambert Law", labeled axes.]
```

7.3 Histograms: `plt.hist()`

Python

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

rng = np.random.default_rng(42)

# Mass distribution of a sample
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("Mass (kg)")
plt.ylabel("Probability density")
plt.title("Mass distribution (n = 1000)")
plt.axvline(np.mean(masses), color="red", linestyle="--",
            label=f"Mean = {np.mean(masses):.1f} kg")
plt.legend()
plt.show()
```

Output

```
[Figure displayed: green histogram with 30 bars,
dashed red line marking the mean.]
```

7.4 Bar charts: `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]  # % in the human body

plt.bar(elements, abondance, color=["red", "gray", "blue",
        "cyan", "yellow"], edgecolor="black")
plt.xlabel("Chemical element")
plt.ylabel("Abundance (%)")
plt.title("Elemental composition of the human body")
```

```
plt.show()
```

Output

```
[Figure displayed: colored bar chart showing
the abundance of elements in the human body.]
```

7.5 Subplots: `plt.subplots()`

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

# x^2 and x^3
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("Polynomials")
axes[1, 1].legend()
axes[1, 1].grid(True)

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

Output

```
[Figure displayed: 2x2 grid of subplots with
sin(x), cos(x), damped signal, and polynomials.]
```

Best Practice

Always use `plt.tight\_layout()` or `fig.tight\_layout()` to prevent labels from overlapping between subplots.

7.6 Advanced customization

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("Time (s)", fontsize=14)
plt.ylabel("Amplitude", fontsize=14)
plt.title("Sinusoidal signals", 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()
```

Output

```
[Figure displayed: two sinusoidal signals with different styles,
arrow annotation pointing to the maximum.]
```

7.7 Saving a figure: `plt.savefig()`

Python

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

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

```
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("Standard normal distribution")

# Save in high resolution
plt.savefig("gaussienne.png", dpi=300, bbox_inches="tight")
plt.savefig("gaussienne.pdf", bbox_inches="tight")
print("Figure saved in PNG and PDF.")
```

Output

```
Figure saved in PNG and PDF.
```

Warning

Call `plt.savefig()` **before** `plt.show()`, otherwise the figure will be empty when saved. The option `bbox\_inches="tight"` prevents labels from being clipped.

7.8 Scientific figures: projectile trajectory

Python

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

g = 9.81
angles = [30, 45, 60, 75]  # degrees
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 = {theta_deg} deg")

plt.xlabel("Horizontal distance (m)", fontsize=13)
plt.ylabel("Height (m)", fontsize=13)
plt.title("Projectile trajectory (v0 = 20 m/s)", fontsize=14)
plt.legend(fontsize=11)
plt.grid(True, alpha=0.3)
```

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

Output

```
[Figure displayed: four parabolic trajectories for different
launch angles, showing that 45 deg gives the maximum range.]
```

7.9 Common errors

Common Pitfalls

1. **Forgetting `plt.show()`**: in a script (outside Jupyter), without `plt.show()`, no window appears. The figure is created in memory but remains invisible.
2. **Wrong figure size**: by default, figures are small. Use `plt.figure(figsize=(width, height))` to control the size in inches.
3. **Saving after `plt.show()`**: `plt.show()` clears the current figure. Any call to `plt.savefig()` afterwards will produce an empty file.
4. **Confusing `plt.plot` and the object-oriented interface**: for subplots, use `ax.plot()` rather than `plt.plot()` which acts on the last active axes.
5. **Forgetting labels**: a graph without a title or axis labels is unusable in a scientific context. *Always* add `xlabel`, `ylabel`, and `title`.

7.10 Mini-project: Projectile Motion Visualization

Scientific Mini-Project

Create a complete program that:

1. Asks for the initial velocity v_0 and the launch angle θ.
2. Computes the trajectory, range, and maximum height.
3. Produces a figure with:
   - The parabolic trajectory.
   - A marked point at the apex (maximum height).
   - An annotation indicating the range.
   - A text box with the parameters (v_0, θ, range, $h_{\max}$).
4. Saves the figure as a PDF.

Python

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

v0 = 25.0        # m/s
theta_deg = 55   # degrees
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"Apex\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"v0 = {v0} m/s\n"
        f"theta = {theta_deg} deg\n"
        f"Range = {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))

ax.set_xlabel("Distance (m)", fontsize=13)
ax.set_ylabel("Height (m)", fontsize=13)
ax.set_title("Projectile trajectory", 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"Range: {portee:.1f} m, Max height: {h_max:.1f} m")
```

Output

```
Range: 59.2 m, Max height: 21.4 m
```

7.11 Exercises

Exercise 7.1 ($\star$ — Exponential Curve)**.** Plot the function $f(x) = e^{-x/3}\cos(2\pi x)$ on $[0, 10]$. Add a title, axis labels, and a grid.

Exercise 7.2 ($\star$ — Dice Roll Histogram)**.** Simulate 10,000 rolls of two dice. Plot the histogram of the sum obtained. Which sum is the most frequent?

Exercise 7.3 ($\star\star$ — Phase Diagram)**.** Plot a simplified phase diagram of water: pressure (y) as a function of temperature (x). Use `plt.fill\_between` to color the solid, liquid, and gas regions.

Exercise 7.4 ($\star\star$ — Multiple Subplots)**.** Create a 2×2 figure showing: (a) $\sin(x)$, (b) $\cos(x)$, (c) $\tan(x)$ (limited to $[-5, 5]$), (d) e^x and $\ln(x)$. Each subplot must have its own title and grid.

Exercise 7.5 ($\star\star\star$ — Heatmap)**.** Create a 20×20 matrix representing the temperature of a hot plate (cold edges at 0 °C, hot center at 100 °C). Display it with `plt.imshow()` and add a color bar. *Hint:* initialize a matrix of zeros and add a centered Gaussian.

Key Functions

Chapter Summary: Matplotlib

- **Curves**: `plt.plot(x, y, style, label=...)`
- **Scatter**: `plt.scatter(x, y, color=..., alpha=...)`
- **Histograms**: `plt.hist(data, bins=..., density=True)`
- **Bars**: `plt.bar(categories, values)`
- **Subplots**: `fig, axes = plt.subplots(nrows, ncols)`
- **Labels**: `plt.xlabel`, `plt.ylabel`, `plt.title`
- **Legend**: `plt.legend()` (after using `label=...`)
- **Grid**: `plt.grid(True)`
- **Save**: `plt.savefig("name.pdf", dpi=300, bbox\_inches="tight")`
- **Golden rule**: `savefig` *before* `show`

Chapter 8

Pandas — Data Analysis

Intuition

Imagine a spreadsheet like Excel, but driven entirely by Python code. That is exactly what **Pandas** offers: a powerful library for loading, exploring, filtering, and analyzing tabular data. In science, most experimental data come in tabular form — Pandas is the ideal tool to manipulate them.

8.1 Series and DataFrames

Pandas relies on two fundamental structures:

Definition 8.1 (Series and DataFrame)**.** A **Series** is a labeled column of data (like a vector with an index). A **DataFrame** is a two-dimensional table composed of several Series sharing the same index.

DataFrame

index	name	mass	charge
0	proton	1.673	+1
1	neutron	1.675	0
2	electron	0.0009	−1

Index — Columns (Series)

Python

```python
import pandas as pd

# Create a Series
masses = pd.Series([1.673, 1.675, 0.0009],
                   index=["proton", "neutron", "electron"])
print(masses)
```

Output

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

8.2 Loading data: `pd.read\_csv`

In practice, we rarely enter data by hand. Pandas can read many formats: CSV, Excel, JSON, SQL, etc. We will use the `tips` dataset available in Seaborn.

Python

```python
import seaborn as sns

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

Output

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

The DataFrame `tips` contains 244 rows (observations) and 7 columns.

8.3 Quick exploration

Python

```python
# First rows
print(tips.head(3))
```

Output

```
   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
# Statistical summary
print(tips.describe().round(2))
```

Output

```
       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()
```

Output

```
<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 Selecting data

Definition 8.2 (loc vs iloc)**.** `loc` selects by **labels** (row/column names). `iloc` selects by **integer positions** (numeric indices).

Python

```
# Select a column
print(tips["total_bill"].head(3))

# Select by position (row 0, column 1)
print(tips.iloc[0, 1])      # 1.01

# Select by label
print(tips.loc[0, "tip"])    # 1.01

# Multiple columns
```

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

Output

```
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 Filtering with Boolean masks

Python

```
# Bills greater than 40
large_meals = tips[tips["total_bill"] > 40]
print(large_meals[["total_bill", "tip", "size"]])
```

Output

```
     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

```
# Multiple conditions: Sunday dinners with tip > 5
filtered = tips[(tips["day"] == "Sun") & (tips["tip"] > 5)]
print(f"Number of results: {len(filtered)}")
print(filtered[["total_bill", "tip", "day"]].head())
```

Output

```
Number of results: 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 Aggregation with GroupBy

The `groupby` operation groups data by a categorical variable, then applies an aggregation function.

Python

```python
# Average tip per day
per_day = tips.groupby("day")["tip"].mean().round(2)
print(per_day)
```

Output

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

Python

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

Output

```
       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 Quick visualization with `df.plot()`

Pandas integrates matplotlib to create plots in a single line.

Python

```python
import matplotlib.pyplot as plt

# Histogram of total bills
tips["total_bill"].plot(kind="hist", bins=20, edgecolor="black",
                        title="Distribution of total bills")
plt.xlabel("Total bill")
plt.ylabel("Frequency")
plt.show()

# Average tip per day (bar chart)
tips.groupby("day")["tip"].mean().plot(kind="bar", color="blue",
                                       title="Average tip per day")
plt.ylabel("Average tip")
plt.show()
```

Python

```python
# Scatter plot: total bill vs tip
tips.plot(kind="scatter", x="total_bill", y="tip", alpha=0.6,
          title="Tip as a function of total bill")
plt.xlabel("Total bill")
plt.ylabel("Tip")
plt.show()
```

Remark 8.3. For more elaborate visualizations, one can use Seaborn or Matplotlib directly. The `df.plot()` method is ideal for quick exploration.

8.8 Creating new columns

Python

```python
# Compute the tip percentage
tips["tip_pct"] = (tips["tip"] / tips["total_bill"] * 100).round(1)
print(tips[["total_bill", "tip", "tip_pct"]].head(4))
```

Output

```
   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
```

Common Pitfalls

Common errors with Pandas

1. **SettingWithCopyWarning** — When modifying a "slice" of a DataFrame, Pandas does not know whether you are modifying the original or a copy. Solution: use `.loc[]` or `.copy()` explicitly.

```
# BAD: may trigger a warning
subset = tips[tips["day"] == "Sun"]
subset["tip_pct"] = subset["tip"] / subset["total_bill"]

# GOOD: explicit copy
subset = tips[tips["day"] == "Sun"].copy()
subset["tip_pct"] = subset["tip"] / subset["total_bill"]
```

2. **Confusing `loc` and `iloc`** — `loc[1]` accesses the row whose **label** is 1; `iloc[1]` accesses the **second** row (position 1). After filtering, labels are no longer consecutive!

3. **Forgetting parentheses in multiple conditions** — Write `(cond1) \& (cond2)`, not `cond1 \& cond2`.

Scientific Mini-Project

Analyzing the Tips dataset

1. Load the `tips` dataset with Seaborn.
2. Compute the average tip by meal time (`time`).
3. Determine which day has the highest number of customers (sum of `size`).
4. Create a column `price\_per\_person` = `total\_bill / size`.
5. Plot a histogram of `price\_per\_person` and comment on the distribution.
6. Test whether smokers leave a different tip (groupby + mean).

8.9 Exercises

Exercise 8.1 ($\star$)**.** Load the dataset `sns.load\_dataset("penguins")`. Display the first 5 rows, the shape of the DataFrame, and the statistical summary. How many missing values are there per column? (Hint: `.isna().sum()`)

Exercise 8.2 ($\star$)**.** From the `tips` dataset, select all rows where the total bill exceeds 30 and the tip is less than 3. How many rows do you get?

Exercise 8.3 ($\star\star$)**.** Use `groupby` to compute the mean and standard deviation of the tip by sex *and* meal time (groupby on two columns). Which category leaves the highest average tip?

Exercise 8.4 (⋆⋆). Create a new column `tip\_category` that equals `"generous"` if `tip\_pct > 20`, `"normal"` if between 10 and 20, and `"low"` otherwise. Use `pd.cut` or `np.where`. Plot a bar chart of the number of meals per category.

Exercise 8.5 (⋆ ⋆ ⋆). Load the dataset `sns.load\_dataset("planets")`. Analyze the distribution of exoplanet discovery methods over the years: group by `method` and `year`, count the discoveries, and plot the temporal evolution for the 3 most frequent methods.

Key Functions

Chapter 8 — Pandas: the essentials

Operation	Syntax
Load a CSV	`pd.read\_csv("file.csv")`
First rows	`df.head(n)`
Dimensions	`df.shape`
Statistical summary	`df.describe()`
Column selection	`df["col"]` or `df.col`
Selection by position	`df.iloc[row, column]`
Selection by label	`df.loc[row, "column"]`
Filtering	`df[df["col"] > value]`
Aggregation	`df.groupby("col")["val"].mean()`
New column	`df["new"] = expression`
Quick plot	`df.plot(kind="hist")`

Chapter 9

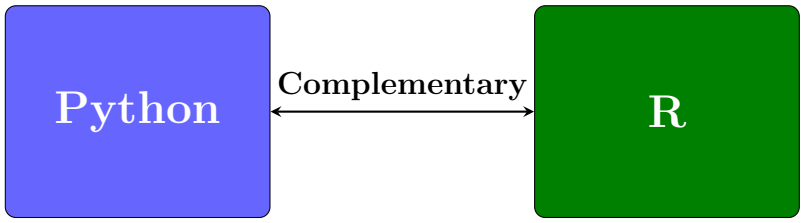
Introduction to R for Statistics

Intuition

If Python is a general-purpose Swiss army knife, **R** is a precision instrument designed specifically for statistics. Created by statisticians for statisticians, R excels at data analysis, hypothesis testing, and visualization. Many researchers in biology, ecology, and social sciences use it daily.

9.1 Why R?

Definition 9.1 (The R language)**.** **R** is a free and open-source programming language specialized in statistical computing and graphical representation. It has an ecosystem of more than 20,000 packages on the CRAN repository.



- General-purpose
- Machine learning
- Automation
- NumPy, Pandas, SciPy

- Native statistics
- Hypothesis testing
- Visualization (ggplot2)
- Bioconductor, tidyverse

9.2 Variables and vectors

In R, the usual assignment operator is `<-` (arrow). The basic structure is the **vector**, created with the `c()` function.

R

```r
# Assignment
x <- 42
name <- "physics"
```

```r
# Numeric vector
measurements <- c(12.3, 14.1, 11.8, 13.5, 12.9)
print(measurements)

# Vectorized operations
measurements_cm <- measurements * 100
print(measurements_cm)
```

Output

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

Warning

In R, indexing starts at **1** (not 0 as in Python). Thus, `measurements[1]` returns the *first* element, i.e. 12.3.

R

```r
# Indexing (starts at 1!)
print(measurements[1])    # First element
print(measurements[2:4])  # Elements 2 to 4 (inclusive!)

# Sequences and repetitions
indices <- 1:10
print(indices)
repetitions <- rep(0, 5)
print(repetitions)
```

Output

```
[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 in R

R

```r
# Create a data frame
experiment <- data.frame(
  subject = c("A", "B", "C", "D", "E"),
  dose    = c(10, 20, 30, 40, 50),
  effect  = c(2.1, 4.5, 5.8, 7.2, 8.9)
)
```

```
print(experiment)
str(experiment)
```

Output

```
  subject dose effect
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:
 $ subject: chr  "A" "B" "C" "D" ...
 $ dose   : num  10 20 30 40 50
 $ effect : num  2.1 4.5 5.8 7.2 8.9
```

9.4 Basic statistical functions

R natively includes all essential statistical functions.

R

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

cat("Mean        :", mean(grades), "\n")
cat("Median      :", median(grades), "\n")
cat("Std. dev.   :", sd(grades), "\n")
cat("Variance    :", var(grades), "\n")
cat("Min / Max   :", min(grades), "/", max(grades), "\n")

# Full summary
summary(grades)
```

Output

```
Mean        : 12.5
Median      : 12.5
Std. dev.   : 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 Statistical tests

Definition 9.2 (Student's t-test)**.** The **Student's t-test** compares the means of two groups to determine whether the observed difference is statistically significant (p-value < 0.05).

R

```r
# Two groups: treatment vs control
treatment <- c(23.1, 25.4, 22.8, 26.1, 24.5, 27.0)
control   <- c(19.2, 20.1, 18.5, 21.3, 19.8, 20.5)

result <- t.test(treatment, control)
print(result)
```

Output

```
 Welch Two Sample t-test

data:  treatment and control
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
# Correlation test
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)
```

Output

```
 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 Linear regression with `lm()`

R

```r
# Linear regression: effect as a function of dose
model <- lm(effect ~ dose, data = experiment)
summary(model)
```

Output

```
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
```

The coefficient $R^2 = 0.989$ indicates an excellent linear fit: the effect increases by 0.172 units per unit of dose.

9.7 Basic plots in R

R

```r
# Scatter plot with regression line
plot(experiment$dose, experiment$effect,
     xlab = "Dose (mg)", ylab = "Effect",
     main = "Dose-effect relationship", pch = 19, col = "blue")
abline(model, col = "red", lwd = 2)

# Histogram
hist(rnorm(1000), breaks = 30, col = "lightblue",
     main = "Simulated normal distribution",
     xlab = "Value")

# Box plot
boxplot(treatment, control,
        names = c("Treatment", "Control"),
        col = c("coral", "lightgreen"),
        main = "Group comparison")
```

9.8 Python vs R comparison

Python

Example 9.3 (Same analysis in both languages).

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

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

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

R

```r
data <- c(12, 15, 8, 14, 16, 11, 13, 9, 17, 10)
cat("Mean:", mean(data), "\n")
cat("Std. dev.:", sd(data), "\n")

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

Remark 9.4. Note that `sd()` in R computes the *corrected* standard deviation ($n-1$) by default, like `np.std(data, ddof=1)` in Python. The function `np.std()` without `ddof=1` divides by n, which gives a different result.

Common Pitfalls

Common errors in R

1. **Indexing from 1** — In R, `x[1]` is the *first* element. In Python, `x[1]` is the *second*. This difference is the source of many errors when switching between languages.
2. **`<-` vs `=`** — In R, `<-` is preferred for assignment. The `=` sign also works in most cases, but `<-` is the standard convention. Be careful not to write `x < -5` (comparison!) instead of `x <- 5` (assignment).
3. **Unexpected factors** — R sometimes automatically converts strings to `factor`. Use `stringsAsFactors = FALSE` in `data.frame()` if needed.
4. **Forgetting `na.rm = TRUE`** — Functions like `mean()` return `NA` if the vector contains missing values. Add `na.rm = TRUE`.

Scientific Mini-Project

Comparative statistical analysis in R Carry out a complete study in R:

1. Create a data frame with two experimental groups (15 measurements each), simulated with `rnorm(15, mean=..., sd=...)`.
2. Compute the descriptive statistics for each group (`mean`, `sd`, `median`).
3. Perform a Student's t-test to compare the two groups.
4. Draw side-by-side box plots.
5. Add a third variable and perform a linear regression with `lm()`.
6. Interpret the results: is the difference significant? Is the linear model appropriate (R^2)?

9.9 Exercises

Exercise 9.1 ($\star$)**.** Create a vector containing the average monthly temperatures of a city (12 values). Compute the mean, median, and standard deviation. Which month is the hottest? Use `which.max()`.

Exercise 9.2 ($\star$)**.** Create a data frame with the columns `element`, `symbol`, and `atomic\_mass` for 5 chemical elements. Display the summary with `summary()` and access the `atomic\_mass` column with `\$`.

Exercise 9.3 ($\star\star$)**.** Generate 100 random values following a normal distribution ($\mu = 50$, $\sigma = 10$) with `rnorm()`. Plot a histogram and overlay the theoretical curve with `curve(dnorm(x, 50, 10), add = TRUE)`.

Exercise 9.4 ($\star\star$)**.** Simulate an experiment: generate two samples of size 30, one with mean 100 and the other with mean 105 (same standard deviation 15). Perform a t-test. Repeat 1000 times and count the percentage of times $p < 0.05$ (this is the **statistical power** of the test).

Exercise 9.5 ($\star\star\star$)**.** Load the built-in dataset `iris` in R. Carry out a complete analysis: descriptive statistics by species, ANOVA test with `aov()` to compare sepal lengths between species, and linear regression between petal length and petal width. Produce 3 relevant plots.

Key Functions

Chapter 9 — R: the essentials

Concept	R syntax
Assignment	`x <- value`
Vector	`c(1, 2, 3)`
Sequence	`1:10` or `seq(0, 1, by=0.1)`
Data frame	`data.frame(col1=..., col2=...)`
Mean	`mean(x)`
Standard deviation	`sd(x)`
Summary	`summary(x)`
t-test	`t.test(group1, group2)`
Correlation	`cor.test(x, y)`
Regression	`lm(y \~{} x, data=df)`
Plot	`plot(x, y)`, `hist(x)`, `boxplot(x)`

Chapter 10

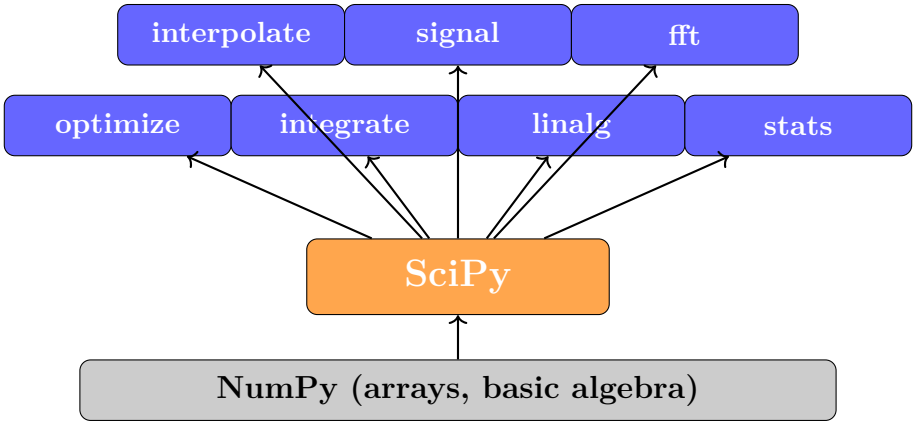
Scientific Computing with SciPy

Intuition

NumPy provides the building blocks (arrays, elementary linear algebra), but scientists often need more specialized tools: fitting a curve to experimental data, integrating a function, solving a system of equations, or performing statistical tests. This is exactly the role of **SciPy**, a library built on top of NumPy.

10.1 Overview of SciPy

Definition 10.1 (SciPy)**. SciPy** (Scientific Python) is an open-source library that provides efficient numerical algorithms organized into thematic submodules.



10.2 `scipy.optimize` — Optimization and fitting

10.2.1 Fitting an experimental curve with `curve\_fit`

In physics, one often measures an exponential decay (radioactivity, capacitor discharge). Let us fit a model $A(t) = A_0\, e^{-\lambda t}$ to noisy data.

Python

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

# Exponential model
def decay(t, A0, lam):
    return A0 * np.exp(-lam * t)

# Simulated data (with noise)
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)

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

Output

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

The recovered values ($A_0 \approx 5.05$, $\lambda \approx 0.31$) are close to the true parameters ($A_0 = 5$, $\lambda = 0.3$).

Python

```python
# Visualization
t_fine = np.linspace(0, 10, 200)
plt.scatter(t_data, y_data, label="Data", color="black", s=20)
plt.plot(t_fine, decay(t_fine, *popt), "r-", label="Fit")
plt.xlabel("Time (s)")
plt.ylabel("Activity (Bq)")
plt.title("Exponential decay fit")
plt.legend()
plt.show()
```

10.2.2 Minimization with `minimize`

Python

```python
from scipy.optimize import minimize

# Find the minimum of 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)  # Starting point x0 = 0
print(f"Minimum at x = {result.x[0]:.4f}")
print(f"Value f(x) = {result.fun:.4f}")
```

Output

```
Minimum at x = 3.3424
Value f(x) = -1.8770
```

10.3 `scipy.integrate` — Numerical integration

The function `quad` computes a definite integral numerically.

Definition 10.2 (Numerical integration)**.** Numerical integration (or quadrature) approximates the value of a definite integral $\int_a^b f(x)\,dx$ by evaluating f at a finite number of points.

Python

```python
from scipy.integrate import quad

# Compute the integral of sin(x) from 0 to pi
result, error = quad(np.sin, 0, np.pi)
print(f"Integral of sin(x) from 0 to pi = {result:.6f}")
print(f"Exact value = {2.0:.6f}")

# Gaussian integral: int from -inf to +inf of exp(-x^2)
val, err = quad(lambda x: np.exp(-x**2), -np.inf, np.inf)
print(f"Gaussian integral = {val:.6f}")
print(f"sqrt(pi) = {np.sqrt(np.pi):.6f}")
```

Output

```
Integral of sin(x) from 0 to pi = 2.000000
Exact value = 2.000000
Gaussian integral = 1.772454
sqrt(pi) = 1.772454
```

10.4 `scipy.linalg` — Linear algebra

10.4.1 Solving a linear system

Consider the system:

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

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}")
```

Output

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

10.4.2 Eigenvalues

Python

```python
# Eigenvalues and eigenvectors
eigenvalues, eigenvectors = eig(A)
print("Eigenvalues:", eigenvalues.real)
print("Eigenvector 1:", eigenvectors[:, 0].real)
```

Output

```
Eigenvalues: [1.38196601 3.61803399]
Eigenvector 1: [-0.85065081  0.52573111]
```

10.5 `scipy.stats` — Statistics

Python

```python
from scipy import stats

# Generate data following a normal distribution
np.random.seed(0)
sample = stats.norm.rvs(loc=170, scale=10, size=100)

# Descriptive statistics
print(f"Mean: {np.mean(sample):.2f}")
print(f"Std. dev.: {np.std(sample, ddof=1):.2f}")

# Normality test (Shapiro-Wilk)
stat, p = stats.shapiro(sample)
print(f"Shapiro-Wilk test: p = {p:.4f}")
if p > 0.05:
```

```
    print("-> Distribution consistent with normality")
```

Output

```
Mean: 171.77
Std. dev.: 10.14
Shapiro-Wilk test: p = 0.8453
-> Distribution consistent with normality
```

Python

```
# Student's t-test (two samples)
group_a = stats.norm.rvs(loc=170, scale=10, size=50, random_state=1)
group_b = stats.norm.rvs(loc=175, scale=10, size=50, random_state=2)

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

Output

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

10.6 `scipy.interpolate` — Interpolation

Interpolation allows estimating values between measurement points.

Python

```
from scipy.interpolate import interp1d

# Measurement points (temperature as a function of time)
time = np.array([0, 1, 2, 3, 4, 5])
temperature = np.array([20.0, 22.5, 28.3, 31.1, 29.0, 25.5])

# Linear and cubic interpolation
f_lin = interp1d(time, temperature, kind="linear")
f_cub = interp1d(time, temperature, kind="cubic")

t_fine = np.linspace(0, 5, 100)
print(f"T at t=2.5 (linear): {f_lin(2.5):.2f} deg C")
print(f"T at t=2.5 (cubic):  {f_cub(2.5):.2f} deg C")
```

Output

```
T at t=2.5 (linear): 29.70 deg C
T at t=2.5 (cubic):  30.22 deg C
```

Python

```
plt.scatter(time, temperature, color="black", zorder=5,
            label="Measurements")
plt.plot(t_fine, f_lin(t_fine), "--", label="Linear")
plt.plot(t_fine, f_cub(t_fine), "-", label="Cubic")
plt.xlabel("Time (h)")
plt.ylabel("Temperature (deg C)")
plt.legend()
plt.title("Temperature data interpolation")
plt.show()
```

Common Pitfalls

Common errors with SciPy

1. **Confusing `numpy.linalg` and `scipy.linalg`** — Both exist, but `scipy.linalg` is more complete and more optimized. Always prefer `scipy.linalg` for serious computations.
2. **Forgetting the starting point in `minimize`** — The function `minimize` requires an `x0`. A poor initial choice can lead to a *local* minimum instead of the global minimum.
3. **Wrong model in `curve\_fit`** — If the mathematical model does not match the data, the fit will be poor. Always check the result visually.
4. **Ignoring the covariance matrix** — `curve\_fit` also returns `pcov`, which allows computing the uncertainties on the fitted parameters: `np.sqrt(np.diag(pcov))`.

Scientific Mini-Project

Experimental data analysis and fitting Simulate a capacitor discharge experiment:

1. Generate noisy data following $V(t) = V_0\, e^{-t/\tau}$ with $V_0 = 12\,\mathrm{V}$ and $\tau = 3\,\mathrm{s}$ (add Gaussian noise).
2. Use `curve\_fit` to recover V_0 and τ.
3. Compute the uncertainties from the covariance matrix.
4. Numerically integrate the dissipated energy: $E = \int_0^{20} \frac{V(t)^2}{R}\, dt$ with $R = 1\,\mathrm{k\Omega}$.
5. Plot the data, the fitted curve, and the error bars.
6. Compare the obtained τ with the theoretical value.

10.7 Exercises

Exercise 10.1 ($\star$)**.** Numerically compute the integral $\displaystyle\int_0^1 e^{-x^2}\,dx$ with `quad`. Compare with the approximate value ≈ 0.7468.

Exercise 10.2 ($\star$)**.** Solve the linear system: $3x+2y-z=1$, $2x-y+4z=-2$, $x+y+z=0$. Verify by computing $A \cdot x$ with NumPy.

Exercise 10.3 ($\star\star$)**.** Generate 200 points following a normal distribution ($\mu = 50$, $\sigma = 8$). Perform a Shapiro-Wilk test and a Kolmogorov-Smirnov test (`stats.kstest`) to check for normality. Interpret the p-values.

Exercise 10.4 ($\star\star$)**.** Experimental data follow the law $y = a\sin(bx + c)$. Generate data with $a = 3$, $b = 2$, $c = 0.5$ plus noise. Use `curve\_fit` to recover the parameters. Note: provide reasonable initial values with the `p0` parameter.

Exercise 10.5 ($\star\star\star$)**.** Model the kinetics of a second-order chemical reaction: $\frac{d[A]}{dt} = -k[A]^2$. The analytical solution is $[A](t) = \frac{[A]_0}{1+k[A]_0 t}$. Generate noisy data, fit the model to find k and $[A]_0$, then numerically integrate $\int_0^{10}[A](t)\,dt$ (total remaining quantity). Compare numerical integration with the analytical solution.

Key Functions

Chapter 10 — SciPy: the essentials

Submodule	Key functions
`scipy.optimize`	`minimize(f, x0)`, `curve\_fit(model,`
`scipy.integrate`	`quad(f, a, b)` — definite integral
`scipy.linalg`	`solve(A, b)`, `eig(A)`
`scipy.stats`	`ttest\_ind`, `shapiro`, distributions
`scipy.interpolate`	`interp1d(x, y, kind="cubic")`

Tips	
Fit uncertainties	`np.sqrt(np.diag(pcov))`
Infinite bounds	`quad(f, -np.inf, np.inf)`
Initial values	`curve\_fit(f, x, y, p0=[a0, b0])`

CHAPTER 10. SCIENTIFIC COMPUTING WITH SCIPY

Chapter 11

Best Practices — Testing, Documentation, Git

Intuition

Writing code that works is one thing; writing code that is **reliable**, **readable**, and **reproducible** is another. In science, a result must be verifiable and reproducible. This chapter presents the tools and methods that make the difference between a throwaway script and a quality scientific project: code style, documentation, testing, and version control.

11.1 Code style: PEP 8

Definition 11.1 (PEP 8)**. PEP 8** is the official Python style guide. It defines naming conventions, indentation rules, and formatting guidelines to make code readable and consistent.

Python

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

# GOOD STYLE (PEP 8)
def compute_score(length, width, bonus):
    """Compute the score based on dimensions and bonus."""
    score = length * width + bonus
    if score > 10:
        return score
    else:
        return score * 2
```

Best Practice

Essential PEP 8 rules:

- **Indentation**: 4 spaces (never tabs).
- **Line length**: maximum 79 characters.
- **Variable names**: `snake_case` (e.g., `molar_mass`).
- **Class names**: `CamelCase` (e.g., `ChargedParticle`).
- **Constants**: `UPPER_CASE` (e.g., `SPEED_OF_LIGHT`).
- **Spaces around operators**: `x = 3 + y`, not `x=3+y`.

Python

```python
# Naming conventions
BOLTZMANN_CONSTANT = 1.380649e-23    # Constant (UPPER_CASE)
temperature_kelvin = 300.0           # Variable (snake_case)

class GasMolecule:                   # Class (CamelCase)
    def __init__(self, mass, velocity):
        self.mass = mass
        self.velocity = velocity

    def kinetic_energy(self):        # Method (snake_case)
        return 0.5 * self.mass * self.velocity**2
```

11.2 Docstrings and documentation

A **docstring** is a documentation string placed right after the definition of a function, class, or module.

Python

```python
def kinetic_energy(mass, velocity):
    """
    Compute the kinetic energy of an object.

    Parameters
    ----------
    mass : float
        Mass of the object in kilograms (kg).
    velocity : float
        Velocity of the object in meters per second (m/s).

    Returns
    -------
```

```
    float
        Kinetic energy in joules (J).

    Examples
    --------
    >>> kinetic_energy(2.0, 3.0)
    9.0
    """
    return 0.5 * mass * velocity**2
```

Python

```
# Access the documentation
help(kinetic_energy)
```

Output

```
Help on function kinetic_energy in module __main__:

kinetic_energy(mass, velocity)
    Compute the kinetic energy of an object.

    Parameters
    ----------
    mass : float
        Mass of the object in kilograms (kg).
    velocity : float
        Velocity of the object in meters per second (m/s).

    Returns
    -------
    float
        Kinetic energy in joules (J).
```

Remark 11.2. The style used above is called the **NumPy style**. It is the standard in the scientific Python ecosystem. Other styles exist (Google, Sphinx), but the NumPy style is the most widespread in science.

11.3 Testing with `assert`

Testing your code is like verifying an experimental measurement: you compare the obtained result with the expected result.

Python

```python
def celsius_to_kelvin(temp_c):
    """Convert a temperature from Celsius to Kelvin."""
    return temp_c + 273.15

# Simple tests with assert
assert celsius_to_kelvin(0) == 273.15, "Error: 0 deg C != 273.15 K"
assert celsius_to_kelvin(100) == 373.15, "Error: 100 deg C"
assert celsius_to_kelvin(-273.15) == 0.0, "Error: absolute zero"
print("All tests passed!")
```

Output

```
All tests passed!
```

Python

```python
import math

def circle_area(radius):
    """Compute the area of a circle."""
    if radius < 0:
        raise ValueError("Radius must be positive.")
    return math.pi * radius**2

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

# Test the exception
try:
    circle_area(-1)
    assert False, "Should have raised an error"
except ValueError:
    pass  # This is the expected behavior

print("All tests passed!")
```

Output

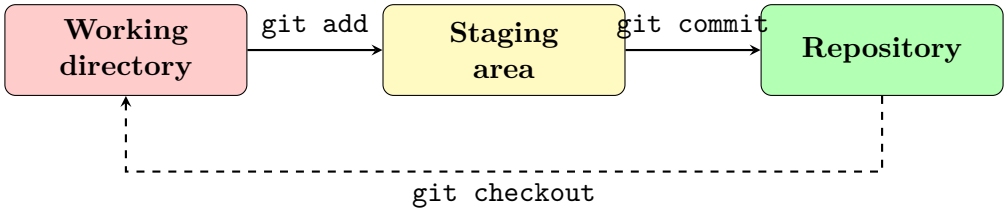
```
All tests passed!
```

Warning

For floating-point numbers, *never* test for exact equality. Use a tolerance: `abs(obtained - expected) < 1e-10` or `math.isclose(obtained, expected)`.

11.4 Version control with Git

Definition 11.3 (Git)**. Git** is a distributed version control system. It records the complete history of changes to a project, allowing you to go back in time, collaborate, and work on parallel branches.



Python

```
# Essential Git commands (in the terminal)
# These commands are run in a shell, not in Python

# 1. Initialize a repository
# $ git init

# 2. Check status
# $ git status

# 3. Add files
# $ git add analysis.py
# $ git add .               # Add everything

# 4. Commit
# $ git commit -m "Add initial analysis"

# 5. View history
# $ git log --oneline
```

Python

Example 11.4 (Typical Git session)**.**

```
# Example: a typical Git session in the terminal:
#
# $ mkdir my_project && cd my_project
# $ git init
# Initialized empty Git repository in .../my_project/.git/
#
# $ echo "print('Hello')" > main.py
# $ git add main.py
# $ git commit -m "First commit: main script"
# [master (root-commit) a1b2c3d] First commit: main script
#
# $ echo "# My Project" > README.md
# $ git add README.md
# $ git commit -m "Add README"
#
```

```
# $ git log --oneline
# b4e5f6g Add README
# a1b2c3d First commit: main script
```

11.5 Virtual environments

Definition 11.5 (Virtual environment)**.** A **virtual environment** is an isolated directory containing an independent Python installation. Each project can thus have its own dependencies without conflicts with other projects.

Python

```
# Create a virtual environment
# $ python -m venv my_env

# Activate it
# Linux/Mac:   $ source my_env/bin/activate
# Windows:     $ my_env\Scripts\activate

# Install packages
# (my_env) $ pip install numpy scipy matplotlib

# Save dependencies
# (my_env) $ pip freeze > requirements.txt

# Reproduce the environment elsewhere
# $ pip install -r requirements.txt
```

Best Practice

Always create a virtual environment for each project. Add the environment folder (`my_env/`) to the `.gitignore` file so it is not tracked by version control.

11.6 Structure of a scientific project

Python

```
# Recommended structure for a scientific project
#
# my_project/
# |-- README.md              # Project description
# |-- requirements.txt       # Dependencies
# |-- .gitignore             # Files to ignore
# |-- src/
# |   |-- __init__.py
# |   |-- analysis.py        # Analysis functions
```

```
# |   |-- visualization.py   # Plotting functions
# |-- tests/
# |   |-- test_analysis.py   # Unit tests
# |-- data/
# |   |-- measurements.csv   # Raw data
# |-- notebooks/
# |   |-- exploration.ipynb  # Jupyter notebooks
# |-- results/
#     |-- figures/            # Generated plots
```

Python

```
# Example .gitignore file
#
# __pycache__/
# *.pyc
# my_env/
# .ipynb_checkpoints/
# results/figures/*.png
```

Common Pitfalls

Common errors in best practices

1. **Not testing your code** — "It seems to work" is not a test. Even a simple `assert` is better than nothing. Silent errors are the most dangerous in science.
2. **Not using version control** — Without Git, an unfortunate modification can destroy hours of work. Make **regular** commits with **descriptive** messages.
3. **Obscure variable names** — `x1`, `tmp`, `data2` convey nothing. Prefer `temperature\_kelvin`, `molar\_mass`, `initial\_concentrations`.
4. **Undocumented code** — In six months, you will no longer remember what your function does. Add a docstring to every function.
5. **Everything in a single file** — Split your project into logical modules. A 2000-line file is difficult to maintain.

Scientific Mini-Project

Organizing a mini scientific project Create a complete project following best practices:

1. Create the directory structure described above.
2. Initialize a Git repository and create a `.gitignore`.
3. In `src/analysis.py`, write three documented functions (for example: data loading, statistics computation, fitting).

4. In `tests/test_analysis.py`, write at least 5 tests with `assert`.
5. Create a virtual environment and a `requirements.txt`.
6. Make at least 3 Git commits with clear messages.
7. Check your history with `git log --oneline`.

11.7 Exercises

Exercise 11.1 ($\star$)**.** Fix the following code to comply with PEP 8. Rename the variables, add spaces, and a docstring.

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

Exercise 11.2 ($\star$)**.** Write a function `is\_prime(n)` that tests whether an integer is prime. Add a NumPy-style docstring and at least 5 tests with `assert` (including edge cases: 0, 1, 2, negative numbers).

Exercise 11.3 ($\star\star$)**.** Create a module `physics.py` containing three documented functions: `gravitational\_force(m1, m2, d)`, `potential\_energy(m, h)`, and `pendulum\_period(L)`. Write a separate test file verifying each function with known values.

Exercise 11.4 ($\star\star$)**.** Initialize a Git repository in a new directory. Create a file `calculations.py`, make a first commit. Modify the file, make a second commit. Use `git diff HEAD~1` to see the differences. Use `git log --oneline` to display the history.

Exercise 11.5 ($\star\star\star$)**.** Take one of your projects from previous chapters (for example the Pandas data analysis or the SciPy fitting). Reorganize it as a structured project: separate the code into modules, add docstrings to all functions, write tests, create a virtual environment with `requirements.txt`, and version everything with Git (at least 5 commits).

Key Functions

Chapter	11	—	Best	practices:	the	essentials

Practice	Description
PEP 8	Official Python style: `snake\_case`, 4 spaces, 79 chars
Docstrings	Documentation embedded in the code (NumPy style)
`assert`	Verification: `assert obtained == expected`
`math.isclose`	Floating-point comparison with tolerance
`git init`	Initialize a repository
`git add + commit`	Record changes
`git log`	View history
`python -m venv`	Create a virtual environment
`pip freeze`	Save dependencies
`.gitignore`	Exclude files from Git tracking

Python Quick Reference

.1 Types and Operations

Type	Description
`int`	Integer
`float`	Floating-point number
`str`	String
`bool`	Boolean (`True`/`False`)
`list`	List (mutable)
`tuple`	Tuple (immutable)
`dict`	Dictionary (key-value)
`set`	Set (no duplicates)

.2 Built-in Functions

Function	Description
`print()`	Display
`len()`	Length
`type()`	Type of an object
`range()`	Integer sequence
`input()`	Read user input
`int()`, `float()`, `str()`	Type conversion
`abs()`, `round()`, `max()`, `min()`	Basic math
`sorted()`, `reversed()`	Sorting and reversing
`enumerate()`, `zip()`	Advanced iteration

.3 NumPy

Function	Description
`np.array()`	Create array
`np.zeros()`, `np.ones()`	Pre-filled arrays
`np.linspace()`, `np.arange()`	Sequences
`np.sin()`, `np.cos()`, `np.exp()`	Math functions
`np.mean()`, `np.std()`	Statistics
`np.dot()`, `np.linalg.solve()`	Linear algebra

.4 R Quick Reference

R Function	Description
`c()`	Create vector
`mean()`, `sd()`, `var()`	Statistics
`t.test()`, `cor.test()`	Statistical tests
`lm()`	Linear regression
`plot()`, `hist()`, `boxplot()`	Plots
`read.csv()`, `data.frame()`	Data

Bibliography

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

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

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

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