

FROM FIRST PRINCIPLES TO INTELLIGENT SOLUTIONS

BOOK 01

Python Programming from First Principles

Learn to Code, Think Computationally,
and Solve Real-World Problems



Prof. Philippe Tamla

Understand. Experiment. Build.

REVIEW EDITION · 26 OCTOBER 2026

Python Programming from First Principles

Learn to Code, Think Computationally, and Solve Real-World Problems

Prof. Philippe Tamla

From First Principles to Intelligent Solutions

Review edition · 26 October 2026

Copyright and edition

Python Programming from First Principles

Learn to Code, Think Computationally, and Solve Real-World Problems

Copyright © 2026 Philippe Tamla. All rights reserved.

Author: Prof. Philippe Tamla

Series: From First Principles to Intelligent Solutions

Review edition: 26 October 2026

This edition is prepared for author review. It is not identified as a commercially released edition. Python and other product names belong to their respective owners; their use does not imply endorsement.

The examples are written for learning. Run them with the stated inputs, examine their results and test changes before applying them to other work. Online services, software interfaces and access conditions can change.

Keep any copyright and license notices supplied with companion materials. Source references and third-party credits appear alongside the relevant material.

Contributors

Author

Prof. Philippe Tamla

Prof. Philippe Tamla is the author of *Python Programming from First Principles*, part of the *From First Principles to Intelligent Solutions* series. This book develops programming through explanations, small experiments and practical projects, with applications that connect computation to human questions and everyday tasks.

The teaching approach places understanding before automation: read the code, predict its behavior, run it, change it and explain what you observe. AI assistance is optional and should support the reader's reasoning.

Acknowledgments

To every reader who takes the time to work through an example, question an explanation or share a correction: thank you. Careful questions and concrete feedback help make a programming book more useful.

Table of Contents

Preface	x
Get in touch	xii
Share your thoughts	xiii
1. Introduction to Python	1
What is Python?	1
Installing Python	3
Python Interpreter	4
First Programs	5
Jupyter Notebook	6
Scripts and magic commands	8
Packages and installation	10
2. Basic Concepts	12
Python basics	12
Numbers	13
Strings	16
Variables	20
Indentation, Semicolons, and Comments	22
Input and Output	24
Type Casting (Type Conversion)	26
Boolean Values and Logical Expressions	29
Operators in Python	32
String Formatting Techniques	36

3. Collection Data Types	38
Data Types and Collections	38
Lists	39
Tuples	46
Sets	49
Dictionaries	55
Working with Nested Data Structures	60
4. Control Flow	63
Control Flow	63
Conditional Statements (if, elif, else)	64
The match Statement	69
While Loops	73
For Loops	75
5. Functions	78
Python Functions	78
Function Arguments and Return Values	79
Flexible Arguments: *args and **kwargs	82
Scope and Decorators	85
Python Built-in Functions	88
Lambda Functions, Recursion, and Generators	93
6. Modules	97
Introduction to Modules	97
The datetime Module	99

The math Module	101
The json Module	106
The re (Regular Expressions) Module	108
Python PIP	113
The try and except Blocks	114
Virtual environments	115
7. Object-Oriented Programming	117
7.1 Classes & Objects	117
What Is OOP?	117
Creating Classes and Objects	119
Object Attributes and Methods	122
Initializing Objects with <code>__init__</code> and <code>self</code>	125
Identity, State, and Behavior	129
7.2 Instance & Class Members	132
Instance and Class Attributes	132
Instance and Class Methods	135
Coding Challenge: Tracking Books Across All Libraries	139
7.3 Object Relationships	141
Encapsulation	141
Composition: “Has-a” Relationships	146
Inheritance	149
“Is-a” and “Has-a” Relationships	153
Polymorphism	154
7.4 Interfaces & Abstraction	160
Interfaces and Behavioral Contracts	160
Abstract Classes	163

Multiple Inheritance	167
7.5 UML Modeling	170
Introduction to UML	170
Information or Data Model	171
Class Diagram	172
Object Diagram	174
From Diagram to Code	175
7.6 Application Design	178
Use Case Diagram	178
Activity Diagram	179
Sequence Diagram	180
Structuring Code in OOP	182
Component Diagram	183
Model–View–Controller (MVC) Architecture	184
7.7 University Course Project	186
Project: University Course System	186
8. Advanced Python Concepts	189
Introduction	189
Specialized Data Structures	190
Counting with Counter	191
Grouping with defaultdict	192
Lightweight Records with namedtuple	193
Double-Ended Queues with deque	194
Iteration Tools with itertools	195
Priority Queues with heapq	198

Categorical Data with Enum	200
Summary and Coding Challenges	201
File Handling & Serialization	206
File Handling in Python	207
File Serialization	211
Error Handling	216
Exercise 1 — Building a Safe Calculator (Input Validation + Specific Exceptions)	221
Exercise 2 — Student Registration Validator (Raising & Custom Exceptions)	222
Python Testing	223
Unit Testing with unittest	226
Unit Testing with pytest	230
Concurrency & Parallelism	237
The Global Interpreter Lock (GIL)	238
Threads: Best for I/O-Bound Tasks	239
Processes: Best for CPU-Bound Tasks	241
Async I/O with asyncio	243
Project: Parallel Word Counter	245
APIs & Web Scraping	247
Understanding HTTP	248
HTML Basics	250
Fetching and Parsing Online Data	252
Ethics & Responsible Web Scraping	254
Project: Fetch & Parse Real Website Data	255

Preface

Learning to program means learning to turn a question into steps a computer can carry out. Python gives us a readable way to express those steps, but understanding comes from examining what a program does and testing why it works. This book starts with small instructions and builds toward functions, objects and applications you can explain and adapt.

Who this book is for

This book is for beginners, university students, independent learners and researchers who want a practical foundation in Python. No previous programming experience is required. You should be able to use a computer, edit text and save files; the opening material introduces the tools needed to run Python.

Examples connect programming with familiar tasks, including text processing, library records and a university course system. Readers interested in the humanities, social sciences and data-driven work can use these examples as starting points without needing specialist knowledge of those fields.

What this book covers

The book has eight parts. Work through them in order when the concepts are new to you; use the contents to return to a specific topic later.

- **Part 1 - Introduction to Python:** prepare your tools, run your first programs, and learn how scripts and notebooks fit into a Python workflow.
- **Part 2 - Basic Concepts:** work with numbers, strings, names, input, conversions, Boolean expressions and operators.
- **Part 3 - Collections:** organize related values using lists, tuples, sets, dictionaries and nested records.
- **Part 4 - Control Flow:** choose between actions and repeat work using conditions, pattern matching and loops.
- **Part 5 - Functions:** organize reusable behavior with parameters, return values and scope, then explore flexible arguments, decorators, recursion and generators.
- **Part 6 - Modules:** separate code into files and use standard-library tools for dates, mathematics, JSON and regular expressions.
- **Part 7 - Object-Oriented Programming:** create classes and objects, distinguish instance and class members, examine relationships and contracts, and model designs with UML. The University Course System project brings these ideas together.
- **Part 8 - Advanced Python Concepts:** apply specialized collections, file handling, serialization, exceptions, tests, concurrency, HTTP requests and HTML parsing.

To get the most out of this book

Read an example before running it. Predict what it will do, run it with the stated inputs, then change one thing and compare the result with your prediction. Explain the change in your own words before moving on. Complete the coding challenges to practise using an idea without copying a worked example.

Use a computer on which you can install and run Python and save your own files. Follow the platform and environment guides in Getting Started. The `match` statement needs Python 3.10 or later. Some later examples require separately installed packages; each relevant section identifies them. Terminal commands, multi-file tests and process pools need a local environment rather than only a browser workspace.

Run examples in their stated order when they share definitions. A continuation identifies the earlier class, function or file it needs. Intentional errors explain a failure; exercise starters need completion. Live-service results, dates, random values and timings can vary.

If you use an AI assistant, begin with your own attempt. Ask for a focused hint or an explanation, then check the answer against the code and results you can reproduce. A live AI account is not required to study the book.

Download code files

Open **Companion resources** in the online book to download the code, notebooks and supporting files. Extract the complete bundle into a new practice folder and keep its directory structure. Read the setup instructions for the example you want to run; a notebook is a practice copy, not a replacement for the book's explanation.

<https://first-principles-book-catalogue.tamlafamily.chatgpt.site/>.

Essential examples and output remain in the printed text, so you can reason about them before opening a computer.

Conventions used

Bold introduces an important term or highlights a screen label. Inline monospace identifies code, filenames and commands, such as `Book` and `python example.py`. Italics identify titles or provide light emphasis.

A **Listing** identifies a code example. **Output** shows what it prints and is not code to type. **Expected output** labels behavior that has not been executed; **target output** belongs to a task you must complete. Templates and signatures illustrate syntax rather than complete programs.

Figure and **Table** captions identify visuals and reference data. UML symbols are explained before use. **Try it** asks for a bounded change; a **Coding Challenge** asks you to apply the idea independently. Online prediction answers can be opened after your attempt; `print` shows them directly. Do not type terminal prompts or notebook input labels.

Get in touch

Questions and corrections are welcome. Visit kcslab.org and use the contact options provided on the website. When sending a question or correction, include the details below so the issue is easy to understand and reproduce.

Ask a useful question

Include the book title, the section heading and the listing or figure number. For a code question, show the smallest example that reproduces the problem, your Python version, the input you used and the complete error message. Explain what you expected and what happened instead.

Report a correction

Identify the passage or example and describe the proposed correction. Include enough detail for another reader to reproduce the issue. Page numbers can change between editions, so a section heading and the review date are especially helpful.

Do not post passwords, API keys, private records or other people's personal information in a public discussion. Replace sensitive values with small fictional examples that still demonstrate the problem.

Share your thoughts

After working through the book, share an honest account of your experience with other learners or through [KCSLAB](#). Tell readers what you learned, which example helped you understand an idea, and where an explanation could be clearer.

Specific feedback is more useful than a rating alone. Mention your starting experience and the part of the book you used so others can judge whether it suits their needs. If you built a project, explain what you changed and how you checked the result.

You do not need to give a positive review. Thoughtful criticism and reproducible corrections help improve future editions. Use [Get in touch](#) for questions or errors that need a detailed response.

1. Introduction to Python

What is Python?

Python is a programming language: a way to express instructions that a computer can execute. It allows us to begin with small programs and develop more complex solutions as our understanding grows.

What is programming?

Programming means describing a task precisely enough for a computer to carry it out. To put historical events in order, for example, a program needs a representation of each event, a rule for ordering the events and instructions for displaying them.

The programmer defines those choices and checks the result. A computer can follow instructions quickly, but speed does not guarantee that the instructions solve the intended problem.

Why programming languages are needed

Processors operate on **machine instructions** represented in binary. A programming language lets us describe a computation using meaningful names and operations instead of writing those instructions directly.

We write source code in Python. A Python implementation handles that code, and its **interpreter** executes the program. This gives us a readable way to express a solution while the implementation manages the lower-level work. Readability still depends on precision: Python's punctuation and structure have defined meanings.

Why Python?

Several features make Python a useful starting point for the work in this book:

- **Readable notation.** Meaningful names and a consistent structure help reveal what a program does.
- **A gradual starting point.** A short program can use only a few language features; more can be introduced as the problem requires them.
- **A range of applications.** Python supports automation, data analysis, web applications and machine learning.
- **Libraries and learning resources.** Existing packages extend the language, while documentation and community resources help answer unfamiliar questions.

The examples here begin with greetings and historical timelines before developing the ideas needed to work with data and more complex programs.

A first glimpse of Python

This short example records a historical event and displays it:

```
year = 1969
event = "First humans land on the Moon"
print(year, event)
```

Listing 1 — Naming two values and displaying a historical event.

Three parts are worth noticing:

- **Names:** `year` and `event` refer to information used by the program.
- **Values:** `1969` is a number; the words inside quotation marks are text.
- **Output:** `print()` displays the two values in the order given.

The result is:

```
1969 First humans land on the Moon
```

Listing 2 — Output from the historical-event example.

Variables, text and the `print()` function will be explained more fully in the following chapters. For now, the example shows how a few readable instructions produce a visible result.

Python versions

- Python 3 is the version family used throughout this book. The Python Developer's Guide¹ lists the releases that are currently supported.
- Python 2 is an earlier, incompatible generation. Its official support ended on 1 January 2020², and it is not used here.

In a version number such as `3.14.3`, the numbers identify the **major version**, **feature release** and **maintenance release**. The introductory examples do not require the exact version shown in a screenshot; they require a supported Python 3 release compatible with the chosen tools.

Preparing to run Python

The next step is to make Python available on the computer. [Installing Python](#) establishes the working environment in which these instructions can be saved, executed and examined.

- [Python Developer's Guide](#).☐☐
- [ended on 1 January 2020](#).☐☐

Installing Python

Python must be installed on the computer before it can run the book's examples. The installation procedure depends on the operating system.

This book uses **Anaconda Distribution**, which includes Python and scientific **packages**—collections of reusable code. It also includes **conda**, a tool for installing packages and managing **environments**, each with its own Python installation and packages.¹

- Anaconda, [Installation documentation](#). The platform guides describe the installation choices for each operating system. □ □

Python Interpreter

The Python interpreter is a program that reads and executes Python code. It provides an interactive environment for writing and testing Python code.

Default Python Interpreter

To access the Python interpreter by default, open a terminal or command prompt and type `python` in your activated environment (`python3` may be the command on your system). This command will launch the Python interpreter in interactive mode, displaying the familiar `>>>` prompt within your terminal where you can then type code expressions.

In interactive mode, you can type Python code directly into the interpreter, and it will execute the code immediately. For example:

```
>>> print("Hello, World")
Output:
Hello, World
```

Listing 441 — Default Python Interpreter

To exit the Python interpreter, enter `exit()` or press `Ctrl + D` on Unix-like systems or `Ctrl + Z` followed by `Enter` on Windows.

Executing Python Programs

Executing Python programs is straightforward; you simply call the `python` command followed by the name of the Python file (with a `.py` extension) as its first argument. For example, let's say we have a file named `hello_world.py` containing the following code:

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

You can execute it using the following command (assuming `hello_world.py` resides in your current working directory):

```
$ python hello_world.py
Hello, World
```

Listing 442 — Executing Python Programs

While the standard interpreter is fine for quick scripts, more advanced tasks (such as data analysis or scientific computing) often use **IPython**, an enhanced interactive shell, or **Jupyter Notebooks**, a web-based environment originally developed as part of the IPython project.

First Programs

You can now run code in a notebook or a saved Python file. Use that skill to display several lines of information. Both projects below use `print()`, which displays the text you put inside its quotation marks. Each call produces a line of output. You do not need variables or loops yet.

Project 1 — Cultural Greeting Generator

Start with a program that displays greetings in different languages. Each printed greeting should appear on its own line.

Task:

Write a program that prints greetings in at least three languages (for example, English, German and Spanish). The output below shows one possible selection.

```
# Write your code under this line
```

Listing 35 — Project 1 — Cultural Greeting Generator

Expected Output:

```
Hello!
Hallo!
¡Hola!
Bonjour!
```

Listing 36 — Project 1 — Cultural Greeting Generator

After printing greetings, the next project adds a simple structure by combining years with events in a timeline.

Project 2 — Timeline Printer

Next, use separate lines to present events in chronological order. Each line should pair a year with a short description.

Task Choose three or four important years and print each year with a short description.

Expected Output:

```
1492: Columbus reaches the Americas
1789: French Revolution begins
1969: First humans land on the Moon
1989: Fall of the Berlin Wall
```

Listing 37 — Project 2 — Timeline Printer

Continue to Basic Concepts

Compare each result with the expected output, then change a greeting or add another event. Explain which instruction produces each line. When you are comfortable saving and running these programs, continue to [Basic Concepts](#) to learn about values, variables and operators.

Jupyter Notebook

Choose this route if you want to work through the book's companion notebooks. A notebook keeps code cells, explanations and output in a document saved with the `.ipynb` extension. If you prefer saved `.py` files, continue to [VS Code](#) instead.

Open Jupyter Notebook

After completing your platform's verification, open Anaconda Prompt on Windows or a terminal configured for conda on macOS/Linux. Activate the environment you verified, then start Notebook:

```
conda activate base
jupyter notebook
```

Listing 20 — Start Jupyter Notebook in the selected Anaconda environment.

The command starts a local server and normally opens a browser. If the browser does not open, use the complete local address printed in the terminal. Keep that terminal running while you work. If the command is unavailable, consult the Jupyter (software for working with documents that combine code, text and output) installation instructions¹ for the selected environment.

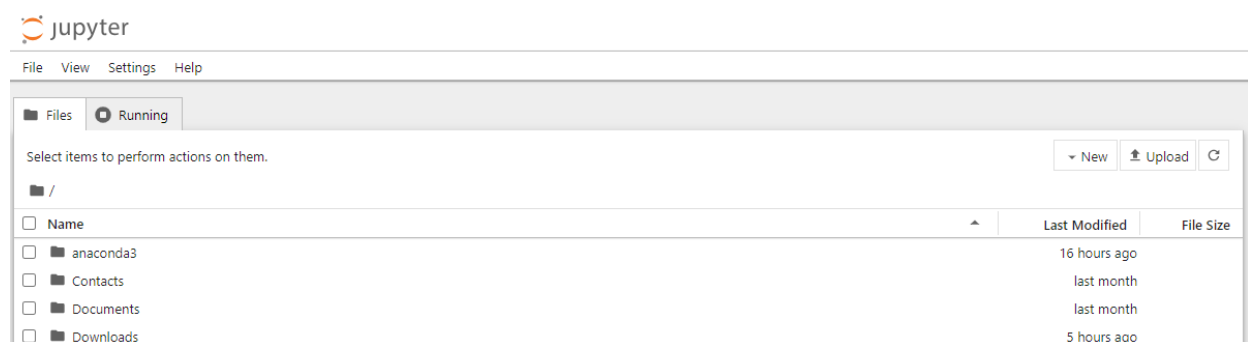


Figure 4 — The notebook file browser. This original book screenshot shows an older interface; current menus may differ.

Create, run and save a notebook

Create a notebook and choose a Python kernel when prompted. The kernel executes the code in its cells. Enter the following instruction in a code cell, then press **Shift+Enter**. The output below the cell should read `Notebook ready`.

```
print("Notebook ready")
```

Listing 21 — Run one instruction in a notebook cell.

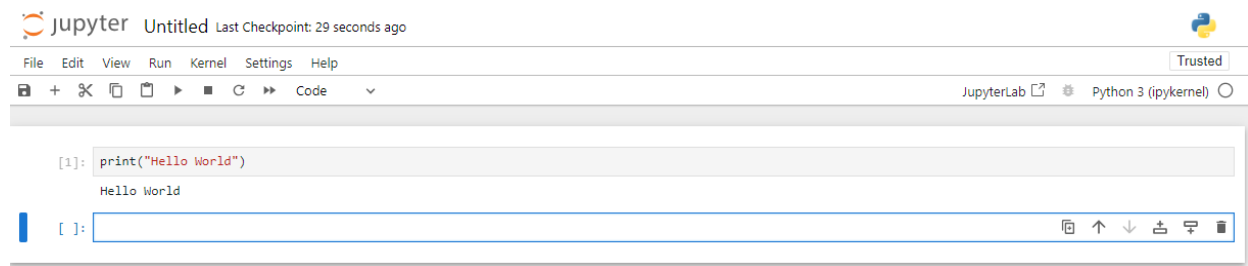


Figure 5 — A code cell with its output. The original screenshot uses Hello World; your Notebook ready message follows the same cell-execution process.

Give the notebook a meaningful name and use the application’s save command. Saving preserves the document; running a cell executes that cell. They are separate actions. When reopening a notebook, run its cells in order so that later cells can use values created earlier.

Continue with the book

You can now use the companion notebook linked from [Your first Python programs](#). You do not need to complete the VS Code (Visual Studio Code, an application for writing and organising program files) setup to follow that route.

See the Jupyter Notebook documentation² for current interface details.

- [Jupyter installation instructions](#).☐☐
- [Jupyter Notebook documentation](#).☐☐

Scripts and magic commands

A **script** is a saved file of Python instructions, such as `hello.py`. An interactive session instead accepts instructions one at a time and keeps their results available for exploration. IPython adds convenience commands to this interactive style of working.

These conveniences are optional. Ordinary Python files remain the main way to save programs that can be run again.

Run a saved script

In a terminal opened in the folder containing `hello.py`, the standard command is:

```
python hello.py
```

Listing 45 — Run a saved script with the active Python.

This starts the program and displays its output. The file must exist in that folder, or the command must include its path.

To explore the same file interactively, activate the intended conda environment and start IPython by entering `ipython` in the terminal. If that environment does not contain IPython, install it there with `conda install ipython` first.

At IPython’s In []: prompt, enter:

```
%run hello.py
```

Listing 46 — Run a saved file from an IPython session.

The script executes, and its names remain available for inspection in that session. This is useful when a program calculates values that you want to examine after it finishes.

What makes a command “magic”?

A **magic command** is an IPython instruction for working with the interactive session. A leading `%` marks a line command. These commands also work in Jupyter notebooks using the IPython Python kernel; they are not ordinary Python syntax and should not be copied into a `.py` file.¹

Command	Purpose
<code>%pwd</code>	Show the session’s current working folder.
<code>%run hello.py</code>	Execute a saved Python file from the session.
<code>%timeit 2 + 2</code>	Repeatedly measure the execution time of an expression.

Table 1 — A few useful IPython line magics.

`%pwd` helps explain a file-not-found error: it shows where the session is looking. `%timeit` is useful when comparing ways of performing a calculation. Its timing varies between computers and runs, so it is a measurement rather than a fixed program result.

Keep the three places distinct

- Terminal: enter commands such as `python hello.py`, `conda install numpy` or `ipython`.
- Python file or ordinary `>>>` prompt: enter Python statements such as `print("Hello World!")`.
- IPython prompt or a notebook cell using its Python kernel: enter Python statements and supported magic commands such as `%run hello.py`.

The distinction prevents a common source of confusion: a valid instruction entered in the wrong place. Exit a terminal IPython session with `exit()` when finished. With the tools prepared, the language chapters can concentrate on what the programs mean.

- IPython, [Built-in magic commands](#). Syntax and kernel-specific behaviour checked on 22 October 2026. □ □

Packages and installation

Python already supplies tools for everyday tasks, such as displaying text and working with files. Other tasks benefit from additional packages. Installing a package makes its code available to a particular Python environment; importing it makes that code available inside a program.

Why add a package?

NumPy is a package for numerical work. Its central structure is an array: an organised collection of values on which calculations can be performed together. NumPy is useful for measurements, numerical datasets and calculations used in scientific computing.¹

A greeting program needs no NumPy installation. A program that calculates with many numerical values may benefit from it. Packages are added when the problem calls for them, rather than simply because they exist.

Install in the environment you use

With Anaconda, open Anaconda Prompt on Windows or the configured Terminal on macOS/Linux. Select the intended environment, then use `conda` to install NumPy if it is needed:

```
conda install numpy
```

Listing 38 — Install NumPy in the active conda environment.

Conda displays a proposed package plan. Review it and confirm to proceed. It also handles dependencies—other packages needed by the requested package. If NumPy is already present, another installation is unnecessary.²

pip is Python's package installer, commonly used to obtain packages from the Python Package Index (PyPI). For a Python environment managed with `pip`, the equivalent command is:

```
python -m pip install numpy
```

Listing 39 — Install NumPy using `pip` associated with the selected Python.

`python -m pip` runs `pip` through that interpreter, helping avoid installing into a different Python by mistake. Use the appropriate route; do not run both commands for the same installation. In a conda environment, prefer conda packages first and use `pip` for additional needs, rather than repeatedly mixing the two tools.³

Installation commands belong in the terminal, not in a `.py` file or at the `>>>` prompt.

Use the installed package

In a Python file or notebook using that same environment, enter:

```
import numpy as np

values = np.array([10, 20, 30])
print(values.mean())
```

Listing 40 — Calculate the mean of three values with NumPy.

`import` loads NumPy for this program. `as np` gives it the conventional short name `np`. The next line creates an array, and `mean()` calculates its arithmetic average. The output is `20.0`.

If Python reports `ModuleNotFoundError`, check that the editor or notebook uses the environment in which NumPy was installed. To update NumPy later, use `conda update numpy` in the conda route, or `python -m pip install --upgrade numpy` in the pip route.

As projects acquire different package requirements, keeping their environments separate becomes useful.

- NumPy, [What is NumPy?](#). □ □
- conda, [Managing packages](#). □ □
- pip, [User guide](#). Installation and upgrade syntax checked on 22 October 2026. □ □

2. Basic Concepts

Python basics

A working installation lets us execute instructions; understanding those instructions begins with values. Numbers, text and names provide the material from which Python programs are built.

A Python program begins with values and instructions. Numbers and text represent information; variables give that information names. Operators combine values, while input and output connect the program with its user.

As a program grows, control flow determines which instructions run and when they repeat. Data structures keep related values together. Functions give reusable operations a name, and modules organise those definitions across files. These foundations prepare the way for classes and objects in the following part of the book.

Numbers

Numbers are a core part of every Python program. They let us count, calculate, and measure values used in everything from budgeting to data analysis and science. Python supports several numeric types and powerful operations.

In this section, you will learn about **integers** for whole numbers, **floats** for decimal values, **operators** for performing calculations, and mathematical expressions for combining them to solve real problems.

Integers

Integers represent whole numbers without a decimal point. They can be positive, negative, or zero. You might use integers to count students in a class, track days of the week, or represent a score in a game.

```
print(-25)
print(-7)
print(0)
```

Listing 58 — Integers

Output

```
-25
-7
0
```

Integers are the simplest form of numeric data. You'll find them useful in situations such as counting items, measuring scores, or indexing data in lists and tables.

Floats

Floats (short for *floating-point numbers*) approximate real-valued measurements, such as distance, weight or time. A decimal point is one way to write a float. Many decimal fractions cannot be represented exactly, so small rounding differences can occur.

```
print(3.14)
print(-0.75)
print(2.0)
```

Listing 59 — Floats

Output

```
3.14
-0.75
2.0
```

Floats let a calculation represent fractional values; they do not guarantee exact decimal arithmetic.

Operators

Python can perform arithmetic operations just like a calculator.

The basic operators are:

- + for addition
- – for subtraction
- * for multiplication
- / for division

```
print(8 + 4)
print(10 - 6)
print(5 * 3)
print(12 / 4)
```

Listing 60 — Operators

Output

```
12
4
15
3.0
```

Division always produces a float, even when the result is a whole number. Python also includes two more useful operators:

- // for floor division (rounds the quotient down, toward negative infinity)
- % for modulo (returns the remainder)

```
print(17 // 3)
print(17 % 3)
```

Listing 61 — Operators

Output

```
5
2
```

You can also raise numbers to a power using ** (the exponentiation operator):

```
print(2 ** 5)
print(10 ** 2)
```

Listing 62 — Operators

Output

```
32
100
```

Mathematical Expressions

A mathematical expression combines numbers and operators to form a complete calculation. Parentheses () can be used to control the order of operations.

```
print((8 + 2) * 5 - 10 / 2)
```

Listing 63 — Mathematical Expressions

Output

```
45.0
```

Python follows the same order of operations as regular mathematics—parentheses first, then exponents, multiplication and division, and finally addition and subtraction.

Mini Project: Simple Arithmetic Program

Write a program that performs and displays the results of a few arithmetic operations. It should calculate and print the following directly:

- The sum of 15 and 7
- The result of 9 divided by 2
- The remainder when 19 is divided by 5
- The square of 6 (use the exponent operator `**`)

Coding Challenge

Create your own short arithmetic expression that combines at least **three different operators**. Try to predict the result before running it, then test your answer in Python. Experiment with parentheses to see how the order of operations affects the final value.

Strings

Strings are used to represent text in Python—words, sentences, or any sequence of characters. They are one of the most common data types in every program because most software needs to display or process text: names, messages, file names, or even data from users.

In this section, you will learn how to create, combine, and manipulate strings, as well as how to access specific characters and apply useful operations to make your text handling flexible and powerful.

Creating Strings

A string in Python is simply text enclosed within quotation marks. It can be written with either **single quotes** (`'...'`) or **double quotes** (`"..."`). Both are valid and behave the same.

A **variable** is a name for a value; `=` assigns the value on its right to the name on its left. The Variables section develops this idea.

```
name = "Ada Lovelace"
greeting = 'Hello, world!'
```

Listing 64 — Creating Strings

In both cases, Python treats the value as text.

For longer text that spans multiple lines, triple quotes (`"""..."""` or `'''...'''`) can be used.

```
print("""This is
a multi-line
string.""")
```

Listing 65 — Creating Strings

Output

```
This is
a multi-line
string.
```

Strings are everywhere—used in displaying messages, storing user names, saving file paths, or even writing logs and labels.

Combining and Repeating Strings

Strings can be combined or multiplied to create new ones.

To join two or more strings, use the `+` operator. This is called string concatenation. To repeat a string several times, use the `*` operator.

```
print("Hello" + " " + "Python")
print("Hi! " * 3)
```

Listing 66 — Combining and Repeating Strings

Output

```
Hello Python
Hi! Hi! Hi!
```

This ability is especially useful when constructing messages or formatting text dynamically. For instance, printing a welcome message multiple times or joining words to form sentences.

Accessing Characters

Strings are sequences of characters, and each character has a position known as an index. Indexing allows you to access specific characters within a string. In Python, indexing starts at 0 for the first character.

```
print("Python"[0]) # first character
print("Python"[1]) # second character
print("Python"[-1]) # last character
```

Listing 67 — Accessing Characters

Output

```
P
y
n
```

The negative index -1 refers to the last character, -2 to the second last, and so on. This makes it easy to extract or inspect parts of a string.

You can also take slices of a string to get a range of characters.

```
print("Programming"[0:6]) # characters 0 to 5
print("Programming"[3:]) # from index 3 to end
print("Programming"[:5]) # from start to index 4
```

Listing 68 — Accessing Characters

Output

```
Progra
gramming
Progr
```

Slicing is very powerful when processing text, like extracting usernames from email addresses or abbreviations from words.

Useful String Operations

Python provides many operations and functions to make string handling easier.

Changing Case

A **method** is an operation called through a value using a dot and parentheses, as in `"ADA".lower()`. Here the operation returns new text.

You can convert text to **lowercase**, **uppercase**, or **title case** depending on the formatting you need. For example, when collecting names from users, you might convert them all to a consistent format so that entries like "ADA LOVELACE", "ada lovelace", or "aDa lOvElAcE" appear neatly as "Ada Lovelace".

```
print("ADA".lower())
print("ada lovelace".upper())
print("ada lovelace".title())
```

Listing 69 — Changing Case

Output

```
ada
ADA LOVELACE
Ada Lovelace
```

This is often used for cleaning or standardizing user input.

Counting and Finding

Python can count occurrences of characters or find where a substring appears.

```
print("banana".count("a"))
print("banana".find("na"))
```

Listing 70 — Counting and Finding

Output

```
3
2
```

The `count()` method tells how many times something appears in the string, while `find()` gives the index of the first match (or `-1` if not found).

Replacing and Checking

Strings in Python are **immutable**, that is, they cannot be changed directly after they are created. This means you cannot modify individual characters inside a string.

However, you can easily create a **new, modified version** using the `replace()` method.

```
print("I like Java".replace("Java", "Python"))
```

Listing 71 — Replacing and Checking

Here, the original string remains unchanged; instead, Python returns a new string where “Java” has been replaced with “Python”. This approach is useful when you need to clean, update, or customize text before displaying or saving it.

Mini Project: Welcome Message Generator

Write a short program that displays a friendly welcome message using string operations. It should take a user’s first and last name (entered as text) and display a properly formatted greeting. Start with first and last names assigned in the code. After Input and Output introduces `input()`, extend the program to ask for them. Use `title()` and string concatenation to format the greeting.

Example Run:

```
Enter your first name: jon  
Enter your last name: doe  
Hello, Jon Doe! Welcome to Python programming.
```

Listing 72 — Mini Project: Welcome Message Generator

Variables

Programs become powerful when they can remember information and use it later. That's exactly what **variables** do. A variable is a **name that refers to a value**. Assignment connects a name to a value, and later code can use that name. Assigning a value does not automatically copy the object.

Creating and Using Variables

To create a variable, use the assignment operator (=). The name appears on the left, and the value you want to store appears on the right.

```
age = 25
city = "London"
print(age)
print(city)
```

Listing 51 — Creating and Using Variables

Output

```
25
London
```

When you print `age` or `city`, Python retrieves the stored values.

Unlike mathematics, the `=` symbol in Python does not mean equality—it means *assignment*: make the name on the left refer to the value on the right.

Changing Values

Once a variable is created, you can update its value at any time. The new assignment overwrites the old one.

```
temperature = 20
print(temperature)

temperature = 25
print(temperature)
```

Listing 52 — Changing Values

Output

```
20
25
```

Variables are flexible. The same name can store different kinds of data at different points in a program, though doing so carelessly can make code confusing.

Naming Conventions for Variables

Good names make code easier to understand. Variable names must start with a **letter** or **underscore**, followed by letters, numbers, or underscores. They are also **case-sensitive**, so `Name` and `name` are different.

❑ Valid names:

name, age_2025, _score

❑ Invalid names:

2nd_place, first-name, total\$

It's a good habit to use descriptive names that explain the variable's purpose. For example:

```
user_name = "Alice"
item_price = 19.99
```

Listing 53 — Naming Conventions for Variables

Combining Variables and Expressions

Variables can store values that you use in calculations or combine into new results.

```
price = 10
quantity = 3
total = price * quantity
print(total)
```

Listing 54 — Combining Variables and Expressions

Python evaluates the expression on the right first ($10 * 3$), then stores the result (30) in `total`. This allows you to perform operations dynamically, rather than typing every number directly into your code.

Mini Project — Travel Distance Calculator

Write a program that calculates how far a car travels based on speed and time.

Instructions:

- Create two variables: speed (in kilometers per hour) and time (in hours).
- Compute the distance using the formula `distance = speed * time`.
- Display the result with a descriptive message.

Example Run:

```
Speed: 90
Time: 2
The car traveled 180 kilometers.
```

Listing 55 — Mini Project — Travel Distance Calculator

Indentation, Semicolons, and Comments

This section introduces you to how Python structures code. Three key rules shape every Python script: **indentation**, **semicolons**, and **comments**.

Indentation

Python uses indentation (spaces or tabs) instead of curly braces `{}` to group code blocks. Lines with the same indentation level belong together, while deeper indentation shows nested code.

```
x = 10
if x > 5:
    print("x is greater than 5")
else:
    print("x is 5 or less")
```

Listing 47 — Indentation

Output

```
x is greater than 5
```

Indentation is mandatory in Python, and the interpreter will raise an `IndentationError` if it is missing or inconsistent. This rule keeps code structured, clean, and easy to read.

Semicolons

Unlike many other languages, Python does not require semicolons at the end of lines. A newline normally ends a simple statement; expressions can continue inside parentheses, brackets or braces. You can use semicolons to write multiple statements on one line, but this is rarely done:

```
a = 5; b = 10; print(a + b)
```

Listing 48 — Semicolons

Output

```
15
```

Comments

Comments explain code and are ignored by Python when it runs.

Single-line comments start with `#`:

```
# This checks if a number is positive
if x > 0:
    print("Positive number")
```

Listing 49 — Comments

Output

```
Positive number
```

For a comment spanning several lines, begin each line with `#`. Triple quotes (`'''` or `"""`) create **strings**, not comments. A string at the beginning of a module, class or function can be a **docstring**, which documents that unit. The example below is a string expression and prints nothing.

```
"""
This program compares a number with 5.
It prints whether the number is greater or smaller.
"""
```

Listing 50 — Comments

Understanding indentation, semicolons, and comments ensures that your Python code is clear, correct, and easy to maintain.

Input and Output

A program becomes truly useful when it can communicate with people. Communication happens in two directions: the program shows results (**output**) and receives information from the user (**input**).

Python makes this simple with two built-in functions:

- `print()` for showing messages or results
- `input()` for reading data that the user types

Output with `print()`

The `print()` function displays text or values on the screen. You can print messages, numbers, or even the results of calculations.

```
print("Hello, world!")
print("2 + 3 =", 2 + 3)
```

Listing 56 — Output with `print()`

Output

```
Hello, world!
2 + 3 = 5
```

This is how Python communicates results. Whether you are displaying a summary, a report, or a message, `print()` sends information to the user.

Input with `input()`

The `input()` function allows the program to ask the user for information. When it runs, it pauses and waits for the user to type something. Whatever the user enters is stored as text and can be used later in the program.

```
name = input("Enter your name: ")
print("Hello,", name)
```

Listing 57 — Input with `input()`

Example run (the text after the prompt is what the user types):

```
Enter your name: Ada
Hello, Ada
```

In this short example, the program asks for the user's name, stores it in the variable `name`, and then prints a friendly greeting.

The text inside the parentheses of `input()` acts as a prompt message. It tells the user what to type. This is the same principle behind online forms, quizzes, or command-line tools where the user must enter details before continuing.

Mini Project: Greeting and Age Program

Write a short program that asks for a name and age, then displays how old the person will be next year. This uses `int()` to convert the age text into an integer; Type Conversion explains that step next.

Example run:

```
What is your name? Leo
How old are you? 19
Hello, Leo! Next year you will be 20.
```

The program gathers information from the user, performs a small calculation, and displays the result clearly. This same pattern appears in real systems, for example while filling a form online, updating a profile, or answering questions in a chatbot.

Type Casting (Type Conversion)

In Python, every piece of data has a **type**. Numbers, text, and lists are all different types of data, and each type behaves in its own way. Sometimes, a program needs to convert a value from one type to another—for example, when reading user input (which always arrives as text) but needing to perform a calculation. This process is called **type casting** or **type conversion**.

Understanding Type Conversion

Type conversion means transforming one data type into another. Python provides simple built-in functions for this:

- `int()` converts a value into an integer
- `float()` converts a value into a floating-point number
- `str()` converts a value into a string

Here's a simple example showing how conversions work:

```
print(int("42"))
print(float("3.14"))
print(str(25))
```

Listing 78 — Understanding Type Conversion

Output

```
42
3.14
25
```

In the first two lines, text values are turned into numbers that Python can use for arithmetic. In the third line, a number is turned into text that can be printed or combined with other strings.

Why Type Conversion Matters

When data is entered through the `input()` function, Python treats it as a string, even if the user types numbers. Trying to use that string in a calculation will result in an error unless it's converted.

```
age = input("Enter your age: ")
next_year = age + 1
```

Listing 79 — Why Type Conversion Matters

This leads to this error:

```
TypeError: can only concatenate str (not "int") to str
```

Here, Python tries to add a string to a number, which is not allowed.

To fix this, convert `age` to an integer before performing arithmetic:

```
age = int(input("Enter your age: "))
next_year = age + 1
print("Next year you will be", next_year)
```

Listing 80 — Why Type Conversion Matters

Output:

```
Enter your age: 25
Next year you will be 26
```

Now Python knows that age is a number, so it can perform the calculation correctly.

Converting Between Numbers and Strings

Conversion works both ways. You can also turn numbers into strings, which is useful when combining them with text in messages or output.

```
temperature = 25
print("The temperature is " + str(temperature) + "°C today.")
```

Listing 81 — Converting Between Numbers and Strings

Without `str()`, Python would raise an error because it cannot combine text and numbers directly in the same expression.

Type Conversion with Conditions

Sometimes, after converting input to numbers, you might want your program to **react differently** based on the result. The `if...else` statement helps with this by checking a condition and deciding which code to run.

Example Run:

```
Enter a number: 12
Positive number
```

Listing 82 — Type Conversion with Conditions

```
number = int(input("Enter a number: "))
if number > 0:
    print("Positive number")
else:
    print("Zero or negative number")
```

Listing 83 — Type Conversion with Conditions

Mini Project: BMI Calculator

Write a program that calculates a person's **Body Mass Index (BMI)** based on their **weight** and **height**.

- Ask the user to enter their **weight** in kilograms and **height** in meters using the `input()` function.
- Remember that `input()` returns a string.
- Apply the BMI formula: $BMI = \frac{weight}{height^2}$

- Use `round(bmi, 1)` to display one decimal place. `round(value, digits)` rounds a number to the requested precision.

After Conditional Statements, extend this arithmetic exercise with adult BMI bands: below 18.5, 18.5 to below 25, 25 to below 30, and 30 or above. The sample labels follow the [CDC adult BMI categories](#). These are screening categories, not a health assessment on their own.

Example Runs:

Run 1:

```
Enter your weight (kg): 50
Enter your height (m): 1.75
Your BMI is: 16.3
BMI category: underweight.
```

Run 2:

```
Enter your weight (kg): 68
Enter your height (m): 1.75
Your BMI is: 22.2
BMI category: healthy weight range.
```

Run 3:

```
Enter your weight (kg): 90
Enter your height (m): 1.70
Your BMI is: 31.1
BMI category: obesity.
```

Listing 84 — Mini Project: BMI Calculator

Boolean Values and Logical Expressions

A **Boolean** has one of two values, True or False. Comparisons and logical operators let a program combine answers to questions such as “is the age at least 18?”

Boolean Values: True and False

Python uses the keywords True and False (with capital letters) to represent Boolean values. They can appear directly or result from comparisons and logical operations.

```
print(True)
print(False)
```

Listing 73 — Boolean Values: True and False

Output

```
True
False
```

Booleans may seem simple, but they are powerful. They act as the foundation for conditions in if statements and loops that drive program logic.

Comparison Operators

A Boolean value often comes from **comparing two expressions**. Python provides several comparison operators that return either True or False.

Operator	Meaning	Example	Result
==	Equal to	5 == 5	True
!=	Not equal to	5 != 3	True
>	Greater than	7 > 10	False
<	Less than	2 < 5	True
>=	Greater than or equal to	10 >= 10	True
<=	Less than or equal to	4 <= 3	False

Table 2 — Comparison Operators

Try running a few comparisons:

```
print(10 > 3)
print(8 == 5)
print(7 != 2)
```

Listing 74 — Comparison Operators

Output

```
True
False
```

True

Each expression evaluates to a Boolean value, allowing programs to respond differently depending on whether a condition is true or false.

Logical Operators

Sometimes you need to combine multiple conditions. For the Boolean values used here, `and`, `or` and `not` give the results in the table. More generally, `and` and `or` return one of their operands, while `not` always returns a Boolean:

Operator	Meaning	Example	Result
<code>and</code>	True if both conditions are True	<code>(5 > 3) and (8 > 6)</code>	True
<code>or</code>	True if at least one condition is True	<code>(5 > 10) or (8 > 6)</code>	True
<code>not</code>	Inverts the Boolean value	<code>not (5 > 3)</code>	False

Table 3 — Logical Operators

Example:

```
age = 20
has_ticket = True

print(age >= 18 and has_ticket)
print(age < 18 or has_ticket)
print(not has_ticket)
```

Listing 75 — Logical Operators

Output

```
True
True
False
```

In the first line, both conditions are true, so the result is True.

In the second, only one is true—but because we used `or`, the overall result is still True.

Finally, `not` reverses the truth value.

Using Booleans in Conditions

The following is a preview of conditional statements, developed in Control Flow. An `if` line tests a condition, and an `else` block runs when it is false. Indented statements form each block. They are most often used in **if statements**, where the program checks whether a condition is true before performing an action.

```
is_raining = False

if is_raining:
    print("Take an umbrella!")
else:
    print("Enjoy the sunshine!")
```

Listing 76 — Using Booleans in Conditions

Here, the variable `is_raining` holds a Boolean value.

Python checks it and chooses which message to display based on the result.

Mini Project — Temperature Checker

Write a program that helps decide what to wear based on temperature.

Ask the user for today's temperature and then use Boolean logic to print a suggestion.

Instructions:

- Convert the input to a number.
- Use conditions to check if it is cold (below 15°C), warm (15–25°C), or hot (above 25°C).
- Print a short message for each case.

Example Run:

```
Example Run 1:
Enter the temperature (°C): 12
It's cold today. Wear a jacket!

Example Run 2:
Enter the temperature (°C): 20
The weather is warm and pleasant.

Example Run 3:
Enter the temperature (°C): 30
It's hot today. Stay hydrated and wear light clothes!
```

Listing 77 — Mini Project — Temperature Checker

This project introduces the idea of Boolean-driven decisions and prepares you for the control flow chapter that follows.

Operators in Python

Operators are special symbols that tell Python to perform specific actions on values. They are the building blocks of all expressions — combining variables, numbers, and text to produce results.

In this section, you will learn about the main categories of operators in Python: **arithmetic**, **comparison**, **logical**, **assignment**, **identity**, and **membership**.

Arithmetic Operators

Arithmetic operators perform basic mathematical calculations. They are among the most frequently used operators in any program — from scientific analysis to financial applications.

```
print(10 + 5)    # Addition
print(10 - 5)    # Subtraction
print(10 * 5)    # Multiplication
print(10 / 3)    # Division (returns float)
print(10 // 3)   # Floor division (whole number)
print(10 % 3)    # Modulus (remainder)
print(2 ** 3)    # Exponentiation (power)
```

Listing 85 — Arithmetic Operators

Output

```
15
5
50
3.3333333333333335
3
1
8
```

These operators follow standard mathematical rules, including parentheses () for grouping, so you can write expressions such as (a + b) * c.

Comparison Operators

Comparison operators are used to compare values. They always return a Boolean result — either True or False. These are essential for decision-making in your programs.

Operator	Meaning	Example	Result
==	Equal to	5 == 5	True
!=	Not equal to	5 != 3	True
>	Greater than	7 > 10	False
<	Less than	2 < 5	True
>=	Greater than or equal to	10 >= 10	True
<=	Less than or equal to	4 <= 3	False

Table 4 — Comparison Operators

Example:

```
x = 10
y = 7
print(x > y)
print(x == 10)
```

Listing 86 — Comparison Operators

Output

```
True
True
```

Logical Operators

Logical operators combine multiple Boolean expressions to form complex conditions. They are especially useful when you need to check several things at once.

Operator	Meaning	Example	Result
and	True if both conditions are True	(5 > 3) and (8 > 6)	True
or	True if at least one condition is True	(5 > 10) or (8 > 6)	True
not	Inverts the Boolean value	not (5 > 3)	False

Table 5 — Logical Operators

Example:

```
age = 20
has_id = True
print(age >= 18 and has_id)
```

Listing 87 — Logical Operators

Output

```
True
```

This shows how logical operators can combine conditions in real-world decision-making — like verifying if someone can enter an event.

Assignment Operators

Assignment operators store values in variables and can also perform arithmetic in the same step. They make your code shorter and more expressive.

```
x = 10      # simple assignment
x += 5      # same as x = x + 5
x -= 3      # same as x = x - 3
x *= 2      # same as x = x * 2
x /= 4      # same as x = x / 4
print(x)
```

Listing 88 — Assignment Operators

Output

```
6.0
```

Each operation calculates a result and assigns it back to `x`, which is useful for totals and incremental changes.

Identity Operators

Identity operators check whether two names refer to the same object. This differs from comparing their values. They are often used when working with objects and references.

```
a = [1, 2, 3]
b = a
c = [1, 2, 3]

print(a is b)      # True - both refer to the same list
print(a is c)      # False - same contents, different objects
print(a is not c)  # True
```

Listing 89 — Identity Operators

Output

```
True
False
True
```

The keyword `is` tests for identity (same object), while `==` tests for equality (same value).

Membership Operators

Membership operators test whether a value is present inside a sequence like a list or a string. They return either `True` or `False`.

```
fruits = ["apple", "banana", "cherry"]

print("banana" in fruits)    # True
print("orange" not in fruits) # True
```

Listing 90 — Membership Operators

This is useful in all sorts of data-related tasks — for example, checking if a record exists in a dataset.

Mini Project — Discount Eligibility Checker

Write a program that determines if a customer qualifies for a discount based on their **age** and **membership** status.

Instructions:

- Ask for the customer's age and whether they are a member (yes or no).
- Apply the rule: If the customer is **under 18** or a **member**, they get a discount.
- Display the result.

Example Run:

```
Enter your age: 16  
Are you a member? (yes/no): no  
You qualify for a discount!
```

Listing 91 — Mini Project — Discount Eligibility Checker

String Formatting Techniques

When programs display text mixed with numbers or variables, it is important to keep the output clean and readable. String formatting provides advanced ways to build such messages as it goes beyond simple concatenation and allows values to be inserted neatly inside sentences.

Python offers several approaches to do this, from the older `.format()` method to the modern and elegant **f-strings**.

Concatenation

Concatenation joins pieces of text together using the `+` operator. It is the simplest form of formatting, but it only works with strings — so numbers must be converted first using `str()`.

```
name = "Alice"
age = 25
print("Hello, my name is " + name + " and I am " + str(age) + " years old.")
```

Listing 92 — Concatenation

Output

```
Hello, my name is Alice and I am 25 years old.
```

While this works for short messages, longer sentences can become difficult to read and maintain.

The `.format()` Method

The `.format()` method provides a cleaner way to insert values into placeholders `{}` within a string. This avoids explicit conversions and keeps the message easier to manage.

```
name = "Bob"
city = "Paris"
print("Hello, {}! Welcome to {}".format(name, city))
```

Listing 93 — The `.format()` Method

Output

```
Hello, Bob! Welcome to Paris.
```

Placeholders can also be ordered or named for greater clarity:

```
print("{1} is older than {0}".format("Alice", "Bob"))
print("Name: {n}, Age: {a}".format(n="Charlie", a=30))
```

Listing 94 — The `.format()` Method

Output

```
Bob is older than Alice
Name: Charlie, Age: 30
```

The `.format()` method is flexible and remains useful in many projects, though newer Python versions favor f-strings for simplicity.

F-Strings — Modern and Readable

F-strings, introduced in Python 3.6, let you place expressions directly inside text. By prefixing a string with `f`, you can directly include variables or even expressions inside `{}`.

```
name = "Diana"
height = 1.70
weight = 60
bmi = weight / (height ** 2)

print(f"{name}'s BMI is {bmi:.2f}")
```

Listing 95 — F-Strings — Modern and Readable

Output

```
Diana's BMI is 20.76
```

The `{bmi:.2f}` syntax formats the number to two decimal places.

You can also perform calculations or call functions inside an f-string:

```
print(f"In five years, {name} will be {25 + 5} years old.")
```

Listing 96 — F-Strings — Modern and Readable

Use an f-string when the values are available where the message is written; `.format()` remains useful for templates stored separately.

Mini Project — Bill Splitter

This final project of the Basic Concepts module combines several ideas learned so far: input, type conversion, arithmetic, and f-string formatting.

Write a program that helps a group of friends split a restaurant bill evenly.

The program should: - Ask for the **total bill** amount. - Ask for the **number of people**. - Require a number of people greater than zero, then calculate how much each person should pay. - Display the result neatly formatted to two decimal places using an f-string.

Example Run:

```
Enter the total bill: 120
Enter the number of people: 4
Each person should pay: 30.00
```

Listing 97 — Mini Project — Bill Splitter

3. Collection Data Types

Data Types and Collections

Individual values are enough for a short calculation or message. A program that represents several events, people or results needs a way to organise those values together.

Most programs need to handle more than one value at a time — a list of students, a set of unique tags, or a record of scores. Python provides several built-in data types to organize such data efficiently.

In this chapter, you will explore the main collection types: **lists**, **tuples**, **sets**, and **dictionaries**.

You will learn how to store, access, and manage groups of values, as well as how to combine these structures to model more complex data.

By the end of this chapter, you will know how to choose the right data type for a task and apply it in practical examples and small projects.

Lists

Programs often need several related values: names, grades, or prices. A list stores these values together, so you do not need a separate variable for every item.

A list is an ordered collection of items enclosed in square brackets `[ ]`, where each item is separated by a comma. Lists are flexible: they can store numbers, text, or even a mix of different data types.

Creating a List

A simple list can be made by placing several values inside brackets:

```
fruits = ["apple", "banana", "cherry"]
print(fruits)
```

Listing 127 — Creating a List

Output

```
['apple', 'banana', 'cherry']
```

Here, `fruits` contains three strings. Lists can also store numbers or mixed types:

```
mixed = [10, "hello", 3.14]
print(mixed)
```

Listing 128 — Creating a List

Output

```
[10, 'hello', 3.14]
```

Lists are used whenever a program needs to handle collections that may change over time, such as adding or removing items from a shopping list.

Accessing List Elements

Since lists are ordered, each item has a position called an **index**. Python counts from zero, so the first element is at position 0.

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0]) # first element
print(fruits[2]) # third element
```

Listing 129 — Accessing List Elements

Output

```
apple
cherry
```

You can also use **negative indexing** to access items from the end:

```
print(fruits[-1]) # last element
```

Listing 130 — Accessing List Elements

Output

```
cherry
```

Slicing a List

Slicing allows you to extract a sublist by specifying a range of indices inside square brackets, separated by a colon `:`. The slice starts from the first index and goes up to, but does not include, the second one.

```
numbers = [10, 20, 30, 40, 50]
print(numbers[1:4])  # elements from index 1 to 3
```

Listing 131 — Slicing a List

Output

```
[20, 30, 40]
```

Omitting one side of the slice selects from the start or to the end:

```
print(numbers[:3])  # first three
print(numbers[2:])  # from index 2 onward
```

Listing 132 — Slicing a List

Output

```
[10, 20, 30]
[30, 40, 50]
```

Slicing can be very useful when working with subsets of data, such as selecting only part of a list for display or calculation.

Combining Lists (Concatenation)

Python allows two or more lists to be joined together using the `+` operator. This process is known as concatenation.

```
a = [1, 2, 3]
b = [4, 5]
combined = a + b
print(combined)
```

Listing 133 — Combining Lists (Concatenation)

Output

```
[1, 2, 3, 4, 5]
```

Concatenation is essential when merging data from different sources or when collecting results from multiple operations into a single sequence.

Modifying a List

Unlike strings, lists are **mutable**, meaning their contents can be changed. You can replace elements or add new ones using assignment and built-in methods.

```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "mango"
print(fruits)
```

Listing 134 — Modifying a List

Output

```
['apple', 'mango', 'cherry']
```

To add items, use `.append()` to add at the end or `.insert()` to add at a specific position:

```
fruits.append("orange")
fruits.insert(1, "kiwi")
print(fruits)
```

Listing 135 — Modifying a List

Output

```
['apple', 'kiwi', 'mango', 'cherry', 'orange']
```

To remove items, use `.remove()` by value or `.pop()` by index:

```
fruits.remove("mango")
fruits.pop(2)
print(fruits)
```

Listing 136 — Modifying a List

Output

```
['apple', 'kiwi', 'orange']
```

Copying and References

An important detail about lists in Python is that **variables hold references, not copies**. Assigning one list to another variable does not make a new list; it simply gives another name to the same list.

```
rgb = ["Red", "Green", "Blue"]
rgba = rgb # both variables point to the same list
rgba.append("Alph")
print(rgb)
```

Listing 137 — Copying and References

Output

```
['Red', 'Green', 'Blue', 'Alph']
```

Notice how changing `rgba` also affected `rgb`. They are two names for the same list object.

If you really need a new list, you can make a **shallow copy** by slicing. This creates a new outer list; any mutable items nested inside it would still be shared:

```
correct_rgba = rgba[:] # copy the list
correct_rgba[-1] = "Alpha" # fix spelling
print(correct_rgba)
print(rgba)
```

Listing 138 — Copying and References

Output

```
['Red', 'Green', 'Blue', 'Alpha']  
['Red', 'Green', 'Blue', 'Alph']
```

Now, the correction only applied to the new list, leaving the original unchanged.

Assigning to Slices

Lists allow you to modify or delete several items at once through slice assignment. This operation can replace, remove, or even clear the entire list.

```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']  
letters[2:5] = ['C', 'D', 'E'] # replace a range  
print(letters)  
  
letters[2:5] = [] # remove items  
print(letters)  
  
letters[:] = [] # clear the list  
print(letters)
```

Listing 139 — Assigning to Slices

Output

```
['a', 'b', 'C', 'D', 'E', 'f', 'g']  
['a', 'b', 'f', 'g']  
[]
```

Slice assignment updates several list items without a loop.

Other Useful Features

The following functions and methods help you inspect, copy, and organize lists.

Measuring Length with `len()`

The `len()` function returns the number of elements in a list.

```
letters = ['a', 'b', 'c', 'd']  
print(len(letters))
```

Listing 140 — Measuring Length with `len()`

Output

```
4
```

Here, `len(letters)` returns 4 because the list has four elements.

Removing Items by Value with `remove()`

Sometimes the value itself is known, but not its position. The `.remove()` method deletes the first occurrence of the specified value from the list.

```
numbers = [10, 20, 30, 20]
numbers.remove(20)
print(numbers)
```

Listing 141 — Removing Items by Value with `remove()`

Output

```
[10, 30, 20]
```

Notice that only the first `20` was removed, while the second remained. This can be useful when cleaning data (e.g., removing an unwanted value from a dataset).

Removing Items by Index with `pop()`

If the position of the element is known, `.pop()` is often the better choice. It removes the item at a given index and also returns it, which can be stored in a variable.

```
numbers = [1, 2, 3, 4]
removed = numbers.pop(2)
print("Removed:", removed)
print("Remaining:", numbers)
```

Listing 142 — Removing Items by Index with `pop()`

Output

```
Removed: 3
Remaining: [1, 2, 4]
```

Here, the element at index `2` (the third element) was removed and stored in the variable `removed`. If no index is given, `.pop()` removes the last element.

Sorting a List with `sort()`

Often, lists need to be ordered — alphabetically, numerically, or even by custom rules. Python's `.sort()` method rearranges the list in place.

```
grades = [88, 75, 92, 100, 67]
grades.sort()
print(grades)
```

Listing 143 — Sorting a List with `sort()`

Output

```
[67, 75, 88, 92, 100]
```

Here the grades are ordered from lowest to highest. The `.sort()` method modifies the original list instead of creating a new one. Sorting is crucial in tasks such as ranking, searching, or preparing data for analysis.

Copying Lists Safely with `copy()`

Assigning one list to another does not create a new copy — it creates a second name for the same list. To avoid accidental changes, the `.copy()` method creates a shallow copy.

```
colors = ["red", "blue", "green"]
new_colors = colors.copy()

new_colors.append("yellow")
print("Original:", colors)
print("Copy:", new_colors)
```

Listing 144 — Copying Lists Safely with copy()

Output

```
Original: ['red', 'blue', 'green']
Copy: ['red', 'blue', 'green', 'yellow']
```

Here the new list `new_colors` was extended, but the original `colors` stayed unchanged.

Nested Lists

Lists can even contain other lists. This allows us to model more complex data structures such as tables, grids, or matrices.

```
matrix = [[1, 2], [3, 4], [5, 6]]

print(matrix[1])      # the second row
print(matrix[1][0])   # the first element of the second row
```

Listing 145 — Nested Lists

Output

```
[3, 4]
3
```

The outer list has three elements, and each element is itself a list of two numbers. Accessing `matrix[1][0]` drills down: first into the second row, then into its first element. Nested lists are the foundation for working with **2D data**, such as spreadsheets, images, or chessboards.

Mini Project — Shopping List Manager

Write a program that allows users to build and edit a shopping list interactively.

Instructions:

- Start with an empty list.
- Ask the user to enter three items.
- Display the full list and the number of items.
- Replace one item, insert another, remove one by value, and finally sort the list alphabetically.
- Make a safe copy of the final list to confirm independence.

Example Run:

```
Enter an item: Milk
Enter an item: Bread
Enter an item: Apples
Your shopping list (3 items): ['Milk', 'Bread', 'Apples']

Updating list...
Replacing 'Bread' with 'Oat Bread'
Inserting 'Eggs' at the start
Removing 'Apples'
Sorted list: ['Eggs', 'Milk', 'Oat Bread']
Safe copy: ['Eggs', 'Milk', 'Oat Bread']
Original remains: ['Eggs', 'Milk', 'Oat Bread']
```

Listing 146 — Mini Project — Shopping List Manager

This project demonstrates how lists can store, modify, and manage data dynamically — a key concept for applications like inventories, user inputs, or data processing pipelines.

Tuples

Lists suit items that may change. For a fixed group of values, use a tuple. A **tuple** is an ordered sequence whose element references cannot be replaced, added or removed after creation. This makes the tuple **immutable**. If an element is itself a mutable object, such as a list, that object can still change.

Tuples are ideal for storing fixed collections of items, such as coordinates, months of the year, or settings that should not be altered. Tuples are commonly written with parentheses (); for a nonempty tuple, the commas create the tuple.

Creating a Tuple

To define a tuple, place several values inside parentheses, separated by commas.

```
fruits = ("apple", "banana", "cherry")
print(fruits)
```

Listing 161 — Creating a Tuple

Output

```
('apple', 'banana', 'cherry')
```

Even though they look similar to lists, tuples are much stricter — once created, their elements cannot be changed, added, or removed.

If a tuple has only one element, you must include a comma after it; otherwise, Python will not recognize it as a tuple:

```
single = ("apple", )
print(type(single))
```

Listing 162 — Creating a Tuple

Output

```
<class 'tuple'>
```

Accessing Tuple Elements

You can access tuple elements in the same way as lists, using indexing.

```
fruits = ("apple", "banana", "cherry")
print(fruits[0]) # first element
print(fruits[-1]) # last element
```

Listing 163 — Accessing Tuple Elements

Output

```
apple
cherry
```

Slicing also works with tuples, letting you extract parts of a sequence.

Updating Tuples

Tuples are **immutable**, meaning they cannot be directly modified. However, there are cases where you can simulate updates by **converting a tuple into a list**, changing it, and then converting it back.

```
fruits = ("apple", "banana", "cherry")

# Convert to list
temp_list = list(fruits)
temp_list[1] = "mango"

# Convert back to tuple
fruits = tuple(temp_list)
print(fruits)
```

Listing 164 — Updating Tuples

Output

```
('apple', 'mango', 'cherry')
```

This approach is common when you want the safety of tuples but still need occasional updates.

Unpacking Tuples

Unpacking means assigning each element of a tuple to separate variables in one step. This gives each value a useful name without separate indexing operations.

```
student = ("Alice", 21, "History")
name, age, course = student
print(name)
print(age)
print(course)
```

Listing 165 — Unpacking Tuples

Output

```
Alice
21
History
```

Unpacking makes code cleaner and easier to read, especially when working with small groups of related values.

Joining Tuples

Just like lists, tuples can be combined using the + operator. This creates a new tuple containing elements from both.

```
a = (1, 2, 3)
b = (4, 5)
c = a + b
print(c)
```

Listing 166 — Joining Tuples

Output

```
(1, 2, 3, 4, 5)
```

Because tuples cannot be changed, joining them always creates a new tuple rather than altering an existing one.

Tuple Methods

Although tuples are immutable, Python still provides a few useful built-in methods for working with them.

The `count()` Method

`count()` tells you how many times a specific value appears in the tuple.

```
numbers = (1, 2, 2, 3, 2, 4)
print(numbers.count(2))
```

Listing 167 — The `count()` Method

Output

```
3
```

The `index()` Method

`index()` returns the position of the first occurrence of a value.

```
numbers = (10, 20, 30, 40)
print(numbers.index(30))
```

Listing 168 — The `index()` Method

Output

```
2
```

These methods are especially helpful when analyzing or validating tuple data.

Why Use Tuples?

Tuples offer several advantages:

- They protect data from accidental changes.
- Tuples can use less memory than comparable lists. Choose them to express fixed structure rather than assuming every operation will be faster.
- Tuples containing suitable immutable values can be dictionary keys, introduced later. A tuple containing a list cannot be a key.

In practice, tuples are perfect for fixed collections of items that define a single entity, like `(latitude,longitude)` for a location or `(day,month,year)` for a date.

Sets

A **set** in Python is a collection of unique and unordered items. Unlike lists or tuples, sets automatically remove duplicates and do not preserve the order of elements. They are useful when you need to store distinct values, test membership quickly, or perform mathematical operations such as union and intersection.

Printed examples show one possible ordering. Your order may differ; compare the members rather than the printed positions. A `pop()` call can remove a different member in another run.

Creating Sets

Sets are defined using curly braces `{}` or the `set()` function.

```
fruits = {"apple", "banana", "cherry"}
print(fruits)
```

Listing 169 — Creating Sets

Output

```
{'apple', 'cherry', 'banana'}
```

You can also create an empty set, but remember to use `set()`, not `{}`, since `{}` creates an empty dictionary

```
empty_set = set()
print(type(empty_set))
```

Listing 170 — Creating Sets

Output

```
<class 'set'>
```

Accessing Set Items

Sets are **unordered**, so you cannot access items using an index like in lists or tuples. However, you can loop through them or check if an element exists using the `in` keyword.

```
fruits = {"apple", "banana", "cherry"}

print("banana" in fruits)  # check membership
print("orange" in fruits)
```

Listing 171 — Accessing Set Items

Output

```
True
False
```

Add Set Items

Sets provide methods for adding items. Use `add()` for adding a single element to a set and `update()` for adding multiple elements at once.

```
fruits = {"apple", "banana"}
fruits.add("cherry")
print(fruits)

fruits.update(["mango", "orange"])
print(fruits)
```

Listing 172 — Add Set Items

Output

```
{'apple', 'cherry', 'banana'}
{'apple', 'banana', 'cherry', 'mango', 'orange'}
```

Remove Set Items

There are several ways to remove items from a set:

- `.remove()` – removes the item but raises an error if it does not exist.
- `.discard()` – removes the item if present, but does nothing if not.
- `.pop()` – removes and returns an arbitrary element; you cannot choose which one.
- `.clear()` – removes all items.
- `del` – removes a variable name; other names may still refer to the set.

```
fruits = {"apple", "banana", "cherry"}
fruits.remove("banana")
print(fruits)

fruits.discard("pear") # no error if missing
print(fruits)

removed = fruits.pop()
print("Removed:", removed)
print("Remaining:", fruits)

fruits.clear()
print("Cleared set:", fruits)
```

Listing 173 — Remove Set Items

Output

```
{'apple', 'cherry'}
{'apple', 'cherry'}
Removed: apple
Remaining: {'cherry'}
Cleared set: set()
```

Loop Sets

You can loop through a set using a simple for loop to access each element (we'll explore loops in detail later).

```
colors = {"red", "green", "blue"}
for color in colors:
    print(color)
```

Listing 174 — Loop Sets

Output

```
blue
red
green
```

The order may vary because sets do not store elements in a specific sequence.

Join Sets

In Python, **joining sets** means combining or comparing their contents. This works exactly like mathematical set operations — allowing you to find elements that appear in one set, both sets, or only in one of them. Python provides several methods and operators to perform these joins efficiently.

Let's start with two sets:

```
a = {1, 2, 3}
b = {3, 4, 5}
```

Listing 175 — Join Sets

Union – Combine All Unique Elements

The union of two sets returns a new set containing *all unique elements* from both. Duplicates are automatically removed.

```
print(a.union(b))
print(a | b)    # shorthand operator
```

Listing 176 — Union – Combine All Unique Elements

Output

```
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
```

The `union()` method and `|` operator produce the same result. Union is commonly used when merging datasets or aggregating results without duplicates — for example, combining customer lists from two regions.

Intersection – Find Common Elements

The **intersection** returns only elements that appear in both sets.

```
print(a.intersection(b))
print(a & b)    # shorthand operator
```

Listing 177 — Intersection – Find Common Elements

Output

```
{3}
{3}
```

This is useful when identifying overlap — such as shared courses between students or mutual friends between two users.

Difference – Find Unique Elements in One Set

The difference returns elements that are in the first set but not in the second.

```
print(a.difference(b))
print(a - b)    # shorthand operator
```

Listing 178 — Difference – Find Unique Elements in One Set

Output

```
{1, 2}
{1, 2}
```

If you reverse the order, the result changes:

```
print(b.difference(a))
```

Listing 179 — Difference – Find Unique Elements in One Set

Output

```
{4, 5}
```

Symmetric Difference – Find Unshared Elements

The **symmetric difference** returns all elements that are in *either* of the sets, but not in both.

```
print(a.symmetric_difference(b))
print(a ^ b)    # shorthand operator
```

Listing 180 — Symmetric Difference – Find Unshared Elements

Output

```
{1, 2, 4, 5}
{1, 2, 4, 5}
```

Use this method when you want to find mismatched or non-overlapping data — such as products unique to each store or topics not shared between two lists.

Updating Sets In Place

Instead of creating new sets, you can update an existing one using methods like:

- `update()` — adds all elements from another set.
- `intersection_update()` — keeps only elements found in both.
- `difference_update()` — removes elements found in another set.
- `symmetric_difference_update()` — keeps only unshared elements.

```
x = {1, 2, 3}
y = {3, 4, 5}

x.update(y)
print("After update:", x)

x = {1, 2, 3}
x.intersection_update(y)
print("After intersection_update:", x)
```

Listing 181 — Updating Sets In Place

Output

```
After update: {1, 2, 3, 4, 5}
After intersection_update: {3}
```

These in-place updates are efficient when working with large data collections where you do not need to keep the original set.

Frozen Sets

A **frozenset** is an immutable version of a set. Once created, its contents cannot be changed, making it safe for use as dictionary keys or fixed configurations.

```
numbers = frozenset({1, 2, 3, 4})
print(numbers)
```

Listing 182 — Frozen Sets

Output

```
frozenset({1, 2, 3, 4})
```

Attempting to modify it (for example, `numbers.add(5)`) will raise an error. Frozensets are especially useful in data integrity contexts where values must remain constant.

Mini Project — Course Enrollment Analyzer

Write a program that analyzes course enrollments for two students and reports **shared**, **unique**, and **total** courses using Python sets. This project puts set operations into practice to solve a realistic problem in academic data management.

Tasks: 1. Ask the user to enter the list of courses for **Student A** and **Student B**, separated by commas. 2. Convert each list into a set (to automatically remove duplicates). 3. Display: - The courses both students share (intersection). - The courses unique to each student (difference). - The total number of distinct courses offered between them (union).

For this starter, separate course names with a comma and one space. The string method `split(", ")` turns that text into a list of course names; `set(...)` removes duplicates. Complete the remaining reports using the operations above. Set display order may vary.

Code to complete:

```
a_courses = input("Enter Student A's courses (comma separated): ")
b_courses = input("Enter Student B's courses (comma separated): ")
a_courses = set(a_courses.split(", "))
b_courses = set(b_courses.split(", "))
# TODO: print the intersection, differences and union
```

Listing 183 — Mini Project — Course Enrollment Analyzer

Example Run:

```
Enter Student A's courses (comma separated): Math, Biology, History, English
Enter Student B's courses (comma separated): Math, Chemistry, English, Art

Shared courses: {'Math', 'English'}
Unique to Student A: {'Biology', 'History'}
Unique to Student B: {'Chemistry', 'Art'}
All distinct courses: {'Biology', 'Art', 'History', 'English', 'Chemistry', 'Math'}
```

Listing 184 — Mini Project — Course Enrollment Analyzer

Dictionaries

A **dictionary** in Python is a collection that stores data in **key–value pairs**. Unlike lists or tuples, where elements are accessed by position, dictionaries let you access values by their associated keys. Use them to represent structured data such as user profiles, product catalogues, or database records.

Dictionaries preserve insertion order. They are **mutable** and **dynamic**: you can change their values and add or remove entries after creation.

Creating Dictionaries

Dictionaries are created using curly braces `{}` with each pair written as `key: value`.

```
student = {  
    "name": "Alice",  
    "age": 22,  
    "major": "Computer Science"  
}  
  
print(student)
```

Listing 147 — Creating Dictionaries

Output

```
{'name': 'Alice', 'age': 22, 'major': 'Computer Science'}
```

Access Items

You can access a dictionary value by using its key inside square brackets `[]` or with the safer `.get()` method.

```
print(student["name"])  
print(student.get("age"))
```

Listing 148 — Access Items

Output

```
Alice  
22
```

If a key does not exist, `student["grade"]` would raise an error, while `student.get("grade")` would return `None`. This makes `.get()` safer for uncertain keys.

Change Items

Dictionaries are mutable, so you can update the value of an existing key easily.

```
student["age"] = 23  
student["major"] = "Data Science"  
print(student)
```

Listing 149 — Change Items

Output

```
{'name': 'Alice', 'age': 23, 'major': 'Data Science'}
```

To update multiple fields at once, use the `.update()` method:

```
student.update({'age': 24, 'year': 'Final'})
print(student)
```

Listing 150 — Change Items

Output

```
{'name': 'Alice', 'age': 24, 'major': 'Data Science', 'year': 'Final'}
```

Add Items

Adding a new key–value pair is as simple as assigning a new key:

```
student["GPA"] = 3.9
print(student)
```

Listing 151 — Add Items

Output

```
{'name': 'Alice', 'age': 24, 'major': 'Data Science', 'year': 'Final', 'GPA': 3.9}
```

The `.update()` method can also be used to add multiple items at once.

Remove Items

You can remove dictionary items in several ways: by using `pop()` to remove a specific key, `popitem()` to remove the last inserted pair, `del` to delete a key entirely, or `clear()` to remove all items from the dictionary.

```
student.pop("year")      # removes item by key
print(student)
```

Listing 152 — Remove Items

Output

```
{'name': 'Alice', 'age': 24, 'major': 'Data Science', 'GPA': 3.9}

student.popitem()        # removes the last inserted key–value pair
print(student)
```

Listing 153 — Remove Items

Output

```
{'name': 'Alice', 'age': 24, 'major': 'Data Science'}

del student["age"]       # removes key entirely
print(student)
```

Listing 154 — Remove Items

Output

```
{'name': 'Alice', 'major': 'Data Science'}
```

```
student.clear()          # removes all items
print(student)
```

Listing 155 — Remove Items

Output

```
{}
```

Loop Through Dictionaries

This previews the `for` loop, developed in Control Flow: it takes one item at a time and runs the indented body for that item. `values()` supplies values, and `items()` supplies key–value pairs that can be unpacked into two names.

```
student = {"name": "Alice", "age": 24, "major": "Data Science"}

# Loop through keys
for key in student:
    print(key)

# Loop through values
for value in student.values():
    print(value)

# Loop through both keys and values
for key, value in student.items():
    print(key, ":", value)
```

Listing 156 — Loop Through Dictionaries

Output

```
name
age
major
Alice
24
Data Science
name : Alice
age : 24
major : Data Science
```

This technique is useful for displaying structured data or processing user records.

Copy Dictionaries

Assigning one dictionary to another variable (`b = a`) doesn't create a new copy—it creates a reference. Use `.copy()` or `dict()` for a shallow copy: the outer dictionary is new, but mutable values nested inside it are still shared.

```
student = {"name": "Alice", "age": 24}
copy_student = student.copy()
copy_student["name"] = "Eve"

print(student)
print(copy_student)
```

Listing 157 — Copy Dictionaries

Output

```
{'name': 'Alice', 'age': 24}
{'name': 'Eve', 'age': 24}
```

Nested Dictionaries

Dictionaries can also hold other dictionaries as values, allowing more complex structures. This is common when modeling related data.

```
students = {
    "A1": {"name": "Alice", "age": 22},
    "B2": {"name": "Bob", "age": 23},
    "C3": {"name": "Charlie", "age": 21}
}

print(students["A1"]["name"])
```

Listing 158 — Nested Dictionaries

Output

```
Alice
```

You can loop through nested dictionaries to display all details:

```
for student_id, info in students.items():
    print(student_id, "=>", info["name"], "-", info["age"])
```

Listing 159 — Nested Dictionaries

Output

```
A1 => Alice - 22
B2 => Bob - 23
C3 => Charlie - 21
```

Find more dictionary methods on the [Python official documentation](#).

Mini Project — Interactive Student Gradebook

Write a program that manages a small **student gradebook** using Python dictionaries. The program should allow users to **add students and their grades**, **update existing grades**, and **calculate the class average**. Complete the interactive version after For Loops and Built-in Functions introduce repetition and `sum()`. For now, trace updates to the three sample records.

Example Run:

```
How many students are in the class? 3
Enter the name of student 1: Alice
Enter Alice's grade: 92
Enter the name of student 2: Bob
Enter Bob's grade: 85
Enter the name of student 3: Clara
Enter Clara's grade: 78

Current Gradebook:
Alice: 92.0
Bob: 85.0
Clara: 78.0

Would you like to update a student's grade? (yes/no): yes
Enter the student's name to update: Bob
Enter Bob's new grade: 88
Bob's grade has been updated.

Final Gradebook:
Alice: 92.0
Bob: 88.0
Clara: 78.0

Class average: 86.0
```

Listing 160 — Mini Project — Interactive Student Gradebook

Working with Nested Data Structures

Combining lists, tuples, and dictionaries for structured data

Real-world data is rarely flat. A class has many students; each student has several courses; each course has a grade. To model this cleanly, we **nest** Python's core collections: lists, tuples, and dictionaries. Nesting combines familiar collections into structured records. We will later use JSON to exchange such records with other programs.

List of Dictionaries — A Collection of Records

A **list of dictionaries** is often used when you have several similar items, each described by key–value pairs. This is common when storing records like users, books, or products.

```
books = [
    {"title": "To Kill a Mockingbird", "author": "Harper Lee", "year": 1960},
    {"title": "1984", "author": "George Orwell", "year": 1949},
    {"title": "The Great Gatsby", "author": "F. Scott Fitzgerald", "year": 1925}
]

print(books[0]["title"]) # Access the first book's title
print(books[1]["author"]) # Access the second book's author
```

Listing 185 — List of Dictionaries — A Collection of Records

Output

```
To Kill a Mockingbird
George Orwell
```

This format is flexible — you can easily add, remove, or filter items, making it useful for things like contact lists, library catalogs, or e-commerce products.

Dictionary of Lists — Grouping Related Values

Sometimes, you need to group multiple items under one category. For example, imagine a grocery store that keeps track of products available in each section.

```
store_sections = {
    "Fruits": ["Apples", "Bananas", "Mangoes"],
    "Vegetables": ["Carrots", "Broccoli", "Spinach"],
    "Bakery": ["Bread", "Croissant"]
}

# Add a new item
store_sections["Bakery"].append("Muffins")
print(store_sections["Bakery"])
```

Listing 186 — Dictionary of Lists — Grouping Related Values

Output

```
['Bread', 'Croissant', 'Muffins']
```

This structure is perfect when one key naturally points to a group of related items — like playlists in a music app or classes offered in a department.

Dictionary of Dictionaries — Nested Information

You can also create a dictionary where each value is another dictionary. This allows for more detailed data — like customer information in an online store.

```
customers = {
    "C001": {"name": "Alice", "email": "alice@example.com", "points": 1200},
    "C002": {"name": "Bob", "email": "bob@example.com", "points": 800},
}

print(customers["C001"]["name"]) # Access Alice's name
customers["C002"]["points"] += 200 # Update Bob's points
print(customers["C002"])
```

Listing 187 — Dictionary of Dictionaries — Nested Information

Output

```
Alice
{'name': 'Bob', 'email': 'bob@example.com', 'points': 1000}
```

This kind of structure is widely used in **school systems** where each *student* has multiple *courses*, and each course stores a *grade*, as well as in **business databases** where each *customer* may have multiple *orders*, and each order contains several *products* with their details.

Tuples Inside Collections — Storing Fixed Information

Tuples are useful when you want to store fixed, ordered data that should not change — such as coordinates, prices, or dimensions.

```
cities = {
    "Paris": (48.8566, 2.3522),
    "London": (51.5074, -0.1278),
    "Tokyo": (35.6895, 139.6917)
}

lat, lon = cities["Tokyo"]
print(f"Tokyo is located at {lat}°N, {lon}°E")
```

Listing 188 — Tuples Inside Collections — Storing Fixed Information

Output

```
Tokyo is located at 35.6895°N, 139.6917°E
```

Tuples provide structure and stability to small data elements that belong together and should remain unchanged.

Accessing and Updating Nested Data

To access or update nested data, you simply chain together the right indices or keys. For example, consider a school record where students are stored by name, and each student has grades for multiple subjects:

```
students = {
    "Alice": {"Math": 88, "English": 92},
    "Bob": {"Math": 75, "English": 85}
}

# Access data
print(students["Alice"]["English"]) # 92

# Update data
students["Bob"]["Math"] = 80
print(students["Bob"])
```

Listing 189 — Accessing and Updating Nested Data

Output

```
92
{'Math': 80, 'English': 85}
```

Nested structures make it easy to represent detailed, real-world relationships — like a person with multiple addresses, or a restaurant with menus by category.

Mini Project — City Weather Tracker

Write a program that tracks **average temperatures** for several cities. Each city will have monthly data stored in a **nested dictionary**. The user can add new temperature data and view city summaries.

Tasks:

- Create a dictionary where each key is a city name and each value is another dictionary of months and average temperatures.
- Start with Paris: January 5°C and July 25°C; London: January 6°C and July 20°C.
- Add a city, update a month, display the records, and calculate the mean of the months recorded for each city. A mean of a few selected months is not an annual mean.
- Build the interactive version after For Loops and Built-in Functions. Until then, practise direct updates to these nested records.

Example Run:

```
Enter a city name: Paris
Enter the month: March
Enter the average temperature: 12

Updated weather data for Paris: {'January': 5, 'July': 25, 'March': 12}

Mean of recorded monthly temperatures:
Paris: 14.0°C
London: 13.0°C
```

Listing 190 — Mini Project — City Weather Tracker

4. Control Flow

Control Flow

Collections organise the information in a program. Control flow determines what the program does with that information: which instructions run, under which conditions, and how often.

Control flow determines which statements run and in what order. Conditions choose a path; loops repeat actions. For example, a weather app uses control flow to decide whether to display a rain alert, a sunshine icon, or a temperature warning based on live weather data. The order can therefore depend on the data rather than always following one fixed sequence. In this module, you'll learn how Python evaluates conditions, executes specific code blocks, and loops through data until certain criteria are met.

In this module, you will learn how to:

- Make decisions using **if**, **elif**, and **else** statements.
- Simplify conditional logic with the **match** statement (Python 3.10+).
- Repeat actions using **while** and **for loops**.
- Manage loop behavior effectively with **break**, **continue**, and **pass**.

By the end of this chapter, you will understand how to control the order in which your code runs, which is a key step toward building interactive and logical Python programs.

Conditional Statements (if, elif, else)

In everyday life, decisions drive our actions: *If it rains, take an umbrella; if it's sunny, wear sunglasses.* In programming, conditional statements let Python do the same — execute different code depending on whether a condition is **True** or **False**. This gives your programs logic and adaptability.

The `if` Statement

The `if` statement tests a condition. If it is **True**, Python executes the indented block beneath it. Parentheses around the condition are optional, but **indentation is required**.

```
temperature = 30

if temperature > 25:
    print("It's a hot day!")
```

Listing 98 — The `if` Statement

Output

```
It's a hot day!
```

Here, the program checks `if temperature > 25`. Since the condition is true, the `print()` statement runs. If the condition were false, Python would skip the block entirely.

The `else` Statement

The `else` clause adds an alternative — it runs when the condition in `if` is false.

```
temperature = 18

if temperature > 25:
    print("It's a hot day!")
else:
    print("It's a cool day.")
```

Listing 99 — The `else` Statement

Output

```
It's a cool day.
```

Now Python checks the first condition; since it's false, it executes the `else` block instead.

The `elif` Statement

When multiple conditions are possible, `elif` (short for *else if*) helps create a chain of checks. Python evaluates each condition in order, and once one is true, it skips the rest.

```
temperature = 12

if temperature > 25:
    print("It's hot outside.")
elif temperature > 15:
    print("It's warm outside.")
else:
    print("It's cold outside.")
```

Listing 100 — The elif Statement

Output

```
It's cold outside.
```

This example has three possibilities, but only one block will run.

Combining Conditions with Logical Operators

In many real situations, decisions depend on more than one condition. For example, you might let someone enter a concert only if they are old enough **and** have a ticket, or send a reminder if it is late **or** a deadline is approaching.

Python's logical operators —and,or, and not — allow you to combine or modify conditions to reflect these real-world decisions.

The and Operator

The and operator connects two or more conditions and returns **True** only if **all** conditions are true. If any of them is false, the entire expression becomes false.

This is useful when multiple requirements must be met at once.

```
age = 20
has_ticket = True

if age >= 18 and has_ticket:
    print("You may enter the concert.")
else:
    print("Access denied.")
```

Listing 101 — The and Operator

Output

```
You may enter the concert.
```

Here, both conditions are true, so the program allows entry. If the person did not have a ticket, the second condition would fail, and access would be denied.

The or Operator

The or operator returns **True** if **at least one** of the conditions is true. It only becomes false when all conditions are false.

This is helpful when any one condition is enough to take action.

```
rainy = False
cold = True

if rainy or cold:
    print("Better take a jacket.")
else:
    print("The weather looks fine.")
```

Listing 102 — The or Operator

Output

Better take a jacket.

Even though it isn't raining, it's cold — and that is enough for the program to print the message.

The not Operator

The not operator reverses a condition. If something is true, not makes it false, and vice versa. This is often used when you want to act only if something is **not** the case.

```
logged_in = False

if not logged_in:
    print("Please log in to continue.")
else:
    print("Welcome back!")
```

Listing 103 — The not Operator

Output

Please log in to continue.

not can also be combined with comparisons:

```
temperature = 12

if not (temperature > 15):
    print("It's cold outside.")
```

Listing 104 — The not Operator

Output

It's cold outside.

This example reads almost like English: "If not temperature greater than 15" — meaning the temperature is 15 or below.

Nested if Conditions

Sometimes one decision depends on another. In such cases, you can place an if statement inside another if. This is called **nesting**. It helps express multi-level logic, where one condition must be true before another is checked.

```
age = 25
student = True

if age < 30:
    if student:
        print("You qualify for the young student discount.")
    else:
        print("You are young, but not a student.")
else:
    print("You do not qualify for the discount.")
```

Listing 105 — Nested if Conditions

Output

```
You qualify for the young student discount.
```

Here, Python first checks whether the age is under 30. Only if that's true does it check whether the person is a student. Nested conditions help control complex decisions, but they should be used carefully to keep the code readable.

The `pass` Statement

Sometimes you may need a conditional structure before you've decided what to put inside it. Python does not allow an empty block, so the `pass` statement acts as a placeholder — telling Python to *do nothing for now*.

```
age = 18

if age >= 18:
    pass # logic will be added later
else:
    print("Underage user.")
```

Listing 106 — The `pass` Statement

Without `pass`, leaving the `if` block empty would cause an error. It's often used as a temporary placeholder while developing or when a condition is needed syntactically but not functionally.

Mini Project — Movie Ticket Price Calculator

Write a program that determines the ticket price based on age.

Ask the user for their age and use conditions to decide the price:

- Children under 12 pay 5 euros
- People aged 12–17 pay 7 euros
- Adults (18 and above) pay 10 euros

Example Run:

```
Enter your age: 10  
Ticket price: 5 euros
```

```
Enter your age: 15  
Ticket price: 7 euros
```

```
Enter your age: 30  
Ticket price: 10 euros
```

Listing 107 — Mini Project — Movie Ticket Price Calculator

This exercise reinforces decision-making logic and prepares you to build interactive programs that respond to user input.

The match Statement

The `match` statement, available from Python 3.10, compares a value with patterns and runs the first matching case. It can match both simple values and the structure of data. Here we begin with literal values.

It's especially useful when you want to match patterns — like user choices, menu commands, or specific data values — without writing several `if` and `elif` statements.

Basic Example

Each case introduces a pattern. A pipe (`|`) combines alternatives; `_` is a wildcard used here as a fallback. The input must match the spelling and capitalization shown:

Let's start with a day of the week:

```
day = input("Enter a day of the week: ")

match day:
    case "Monday":
        print("Start of the work week!")
    case "Friday":
        print("Almost the weekend!")
    case "Saturday" | "Sunday":
        print("It's the weekend, enjoy your rest!")
    case "Tuesday" | "Wednesday" | "Thursday":
        print("Midweek day – keep going!")
    case _:
        print("Unrecognized day.")
```

Listing 108 — Basic Example

Example Runs:

```
Enter a day of the week: Friday
Almost the weekend!

Enter a day of the week: Sunday
It's the weekend, enjoy your rest!
```

Listing 109 — Basic Example

In this example, Python compares the input against each case in order. Once a match is found, it executes the corresponding block and skips the rest.

The underscore `_` acts as a **default case**, similar to `else`, for situations where no other case applies.

Matching Multiple Patterns

Sometimes, different inputs should lead to the same result. Instead of repeating code, you can use the **pipe symbol** (`|`) to match multiple patterns at once.

Here's an example where we group fruits into categories:

```
fruit = input("Enter a fruit name: ")

match fruit:
    case "Apple" | "Pear" | "Banana":
        print("Common fruit.")
    case "Mango" | "Pineapple":
        print("Tropical fruit.")
    case _:
        print("Unrecognized fruit.")
```

Listing 110 — Matching Multiple Patterns

Output:

```
Enter a fruit name: Mango
Tropical fruit.
```

The program checks each pattern in turn and executes the first one that matches, keeping your logic compact and clear.

Using match with Numbers

You can also use match for numeric input — such as menu selections or response codes — to make programs easier to navigate.

In the example below, the program reacts based on the number entered by the user:

```
choice = int(input("Enter a menu option (1–3): "))

match choice:
    case 1:
        print("You selected: Start new game")
    case 2:
        print("You selected: Load game")
    case 3:
        print("You selected: Exit")
    case _:
        print("Invalid option.")
```

Listing 111 — Using match with Numbers

Example Run:

```
Enter a menu option (1–3): 2
You selected: Load game
```

Listing 112 — Using match with Numbers

Mini Project — Simple Menu System

Write a program that displays a small restaurant menu and lets the user place an order.

Use a match statement to select the meal based on the user's choice, and apply simple conditions for size, student discount, takeaway, and vegetarian preference.

Display a short summary with the total price at the end. Apply the size multiplier first, then a 10% student discount, then a €0.50 takeaway fee if applicable. Format only the final total to two decimal places. Reject an unknown meal or size; reject a nonvegetarian dish when vegetarian food is

requested. `strip()` removes surrounding whitespace and `lower()` normalizes the yes/no input.

Here's the starting structure — complete it by adding the control flow using `match`, `if`, and `else` statements:

```
# Menu data (option: name, base price, vegetarian)
menu = {
    1: ("Pizza Margherita", 9.50, True),
    2: ("Chicken Pasta", 11.00, False),
    3: ("Veggie Salad", 7.50, True),
    4: ("Tomato Soup", 6.00, True),
}

# Size multipliers
size_factor = {"small": 0.90, "regular": 1.00, "large": 1.25}

# Display menu
print("Menu:")
print("1) Pizza Margherita   9.50")
print("2) Chicken Pasta     11.00")
print("3) Veggie Salad       7.50")
print("4) Tomato Soup        6.00")

# Collect user input
choice = int(input("Enter your choice (1-4): "))
size = ...
is_student = ...
takeaway = input("Takeaway? (yes/no): ").strip().lower() == "yes"
wants_veg = ...

# TODO: Complete the program using match, if, and else statements
# Hint: Use match to check choice, if to validate vegetarian options, and conditions to apply
discounts and fees.
```

Listing 113 — Mini Project — Simple Menu System

Example Runs:

Example 1:

```
Enter your choice (1-4): 1
Size (small, regular, large): large
Student? (yes/no): yes
Takeaway? (yes/no): yes
Vegetarian preference? (yes/no): yes
```

```
--- Order Summary ---
Dish: Pizza Margherita
Size: Large
Vegetarian: Yes
Student: Yes
Takeaway: Yes
Total: €11.19
```

Example 2:

```
Enter your choice (1-4): 2
Size (small, regular, large): regular
Student? (yes/no): no
Takeaway? (yes/no): yes
Vegetarian preference? (yes/no): yes
Selected dish 'Chicken Pasta' is not vegetarian.
Suggestion: try 'Veggie Salad' or 'Tomato Soup'.
```

Listing 114 — Mini Project — Simple Menu System

While Loops

A while loop repeats a block of code as long as a given condition remains true. It is especially useful when the number of repetitions is not known in advance. For example, it is used to wait for user input or process data until a certain value is reached.

Basic Example

Let's start with a simple counter:

```
count = 1

while count <= 5:
    print("Count:", count)
    count += 1

print("Loop finished.")
```

Listing 115 — Basic Example

Output

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
Loop finished.
```

Here, Python checks the condition `count <= 5` before every iteration.

The loop continues while it's `True`, and stops when it becomes `False`.

Without updating `count`, the loop would run forever — so remember to include a statement that changes the condition inside the loop.

Example with User Input

While loops are especially useful for interactive programs that depend on user input. Let's see an example where the program repeatedly asks the user for a password until they type the correct one.

```
# Ask for a password until the user enters the correct one
password = ""

while password != "python123":
    password = input("Enter the password: ")

print("Access granted!")
```

Listing 116 — Example with User Input

Example Run:

```
Enter the password: hello
Enter the password: 123
Enter the password: python123
Access granted!
```

Listing 117 — Example with User Input

Here, the loop keeps running **as long as** the password entered is not equal to "python123".

Once the user types the correct password, the condition becomes **False** and the loop exits.

Using `break` and `continue`

You can control how a loop behaves using two special keywords: - `break` — exits the loop immediately
- `continue` — skips the rest of the loop body and moves to the next iteration

```
number = 0

while number < 10:
    number += 1
    if number == 5:
        continue # skip printing 5
    if number == 8:
        break # stop the loop at 8
    print(number)
```

Listing 118 — Using `break` and `continue`

Output

```
1
2
3
4
6
7
```

This example shows how to control a loop and skipping certain values or stopping early when needed.

Mini Project — Number Guessing Game

Let's apply what we've learned about while loops to create a small interactive game.

Write a program that asks the user to guess a secret number between 1 and 10.

The program should keep asking until the user guesses correctly.

After each wrong guess, it gives a hint — "too high" or "too low."

Example Run:

```
Guess a number between 1 and 10: 3
Too low! Try again.
Guess a number between 1 and 10: 9
Too high! Try again.
Guess a number between 1 and 10: 7
Congratulations! You guessed it right.
```

Listing 119 — Mini Project — Number Guessing Game

For Loops

A **for loop** is used to repeat a block of code a specific number of times or to go through each item in a sequence, such as a list, tuple, or string.

A for loop requests one item at a time from an **iterable**, something that supplies successive items, such as a list or string. It stops when no items remain; it does not need to know the number of iterations in advance.

Basic Example

Let's begin with a simple example that prints each fruit in a list:

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print("I like", fruit)
```

Listing 120 — Basic Example

Output

```
I like apple
I like banana
I like cherry
```

Here, Python goes through the list `fruits` one element at a time, temporarily storing each value in the variable `fruit`.

The loop ends automatically after the last item.

This type of loop is very common in data processing, where you need to perform the same action for each element in a dataset.

Working with Strings

A for loop can also be used to go through each character in a string:

```
for letter in "Python":
    print(letter)
```

Listing 121 — Working with Strings

Output

```
P
y
t
h
o
n
```

This is often used when you want to check or modify characters individually - for example, counting vowels or formatting text.

Using `range(stop)` Function

When you need to repeat an action a specific number of times, the built-in `range(stop)` function is ideal.

It creates a sequence of numbers for the loop to iterate over, stopping just before the value defined by `stop`.

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

Listing 122 — Using `range(stop)` Function

Output

```
Iteration 0  
Iteration 1  
Iteration 2  
Iteration 3  
Iteration 4
```

The function `range(5)` creates numbers from **0 to 4**, stopping before 5.

Using `range(start, stop, step)` Function

You can also start at a specific number or set a custom step value using `range(start, stop, step)`, which lets you control where the sequence begins, where it ends, and how much it increases with each loop iteration.

```
for i in range(2, 10, 2):  
    print(i)
```

Listing 123 — Using `range(start, stop, step)` Function

Output

```
2  
4  
6  
8
```

This prints only the even numbers from 2 up to, but not including, 10.

Example with User Input

Now, let's make the loop interactive.

This program asks the user for a number and prints all numbers from 1 up to that number.

```
limit = int(input("Enter a number: "))  
  
for i in range(1, limit + 1):  
    print(i)
```

Listing 124 — Example with User Input

Example Run:

```
Enter a number: 5
1
2
3
4
5
```

Listing 125 — Example with User Input

Here, the program uses the value provided by the user to control how many times the loop runs.

It demonstrates how `for` and `range()` can be used together to create flexible repetition.

Mini Project — Multiplication Table Generator

Write a program that displays the multiplication table for a number entered by the user. The program should print the results from $1 \times \text{number}$ up to $10 \times \text{number}$.

Example Run:

```
Enter a number: 4

Multiplication table for 4:
4 × 1 = 4
4 × 2 = 8
4 × 3 = 12
4 × 4 = 16
4 × 5 = 20
4 × 6 = 24
4 × 7 = 28
4 × 8 = 32
4 × 9 = 36
4 × 10 = 40
```

Listing 126 — Mini Project — Multiplication Table Generator

This small project demonstrates how `for` loops can automate repetitive calculations, a key skill for tasks such as data processing, analysis, and report generation.

5. Functions

Python Functions

Conditions and loops describe the steps of a task. When those steps form a useful operation, a function gives that operation a name so it can be called again.

A function is a reusable block of code that performs a specific task. It only runs when called, which helps avoid repeating the same instructions throughout a program. Functions also make code easier to organize and maintain.

Creating a Function

In Python, a function is defined using the `def` keyword, followed by its name and parentheses `()`.

```
def say_hello():  
    print("Hello, welcome to Python!")
```

Listing 191 — Creating a Function

This creates a function named `say_hello` that prints a greeting message. Indentation is here important as it shows which lines belong inside the function.

Calling a Function

To run the function, simply write its name followed by parentheses:

```
say_hello()
```

Listing 192 — Calling a Function

Output

```
Hello, welcome to Python!
```

You can call it as many times as needed:

```
say_hello()  
say_hello()  
say_hello()
```

Listing 193 — Calling a Function

Output

```
Hello, welcome to Python!  
Hello, welcome to Python!  
Hello, welcome to Python!
```

Functions are one of Python's most useful tools — they make your code cleaner, shorter, and easier to reuse.

Function Arguments and Return Values

Functions become much more useful when they can receive information, process it, and send something back.

Arguments allow data to be passed into a function, while the `return` statement sends data out of it.

Together, they make functions powerful tools for computation and code reuse.

Passing Information with Arguments

Arguments are values you provide inside the parentheses when calling a function.

They allow the same function to behave differently depending on the input.

Let's start with a simple example:

```
def greet(name):  
    print("Hello,", name)
```

Listing 194 — Passing Information with Arguments

Here, `name` is a parameter or a placeholder for any value you provide when calling the function.

```
greet("Alice")  
greet("Bob")
```

Listing 195 — Passing Information with Arguments

Output

```
Hello, Alice  
Hello, Bob
```

Each time the function runs, the value passed to `name` changes the output.

Passing Multiple Arguments

A function can take more than one argument.

In this example, the function adds two numbers and prints the result.

```
def add_numbers(a, b):  
    print("The sum is:", a + b)
```

Listing 196 — Passing Multiple Arguments

Calling it with different numbers produces different results:

```
add_numbers(3, 7)  
add_numbers(10, 25)
```

Listing 197 — Passing Multiple Arguments

Output

```
The sum is: 10  
The sum is: 35
```

Each argument represents one piece of information the function needs to perform its job.

Default Arguments

Sometimes, you may want a function to work even when no argument is provided. Give a parameter a default value so that callers can omit its corresponding argument.

```
def greet(name="Guest"):
    print("Welcome,", name)
```

Listing 198 — Default Arguments

Calling it with or without a name gives both possibilities:

```
greet("Anna")
greet()
```

Listing 199 — Default Arguments

Output

```
Welcome, Anna
Welcome, Guest
```

Default arguments make your functions flexible and prevent errors when certain inputs are missing.

Returning Values

So far, our functions have printed results directly.

However, you can also make a function **return** a value so that the result can be stored, reused, or combined with other expressions.

Here's a simple function that returns the sum of two numbers:

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

Listing 200 — Returning Values

The function gives back the result when it is called:

```
total = add(5, 8)
print(total)
```

Listing 201 — Returning Values

Output

```
13
```

`print()` displays information. `return` gives a value back to the caller; a function that reaches its end without `return` returns `None`.

Using Returned Values

Returned values can be used in further calculations or stored for later use.

```
def square(x):  
    return x * x  
  
result = square(4) + square(3)  
print(result)
```

Listing 202 — Using Returned Values

Output

25

Here, each call to `square()` returns a number, and the results are added together.

Returning Multiple Values

A function can also send back more than one result by separating them with commas.

Python automatically groups them into a **tuple**.

```
def calculate(a, b):  
    return a + b, a - b
```

Listing 203 — Returning Multiple Values

```
sum_result, difference = calculate(10, 4)  
print("Sum:", sum_result)  
print("Difference:", difference)
```

Listing 204 — Returning Multiple Values

Returning multiple values helps when a function needs to provide several related results at once.

Mini Project — Currency Converter

Write a program that converts an amount from one currency to another using the illustrative fixed rate 0.92 EUR per USD; this is exercise data, not a current exchange-rate quote. Create a function that takes the amount and conversion rate as arguments, returns the converted value, and prints a message showing the result.

Example Run:

```
Enter amount in USD: 100  
100.0 USD is equal to 92.00 EUR
```

Listing 205 — Mini Project — Currency Converter

Flexible Arguments: *args and **kwargs

Sometimes, you may not know in advance how many arguments a function will receive.

Python provides two special tools —`*args` and `**kwargs`— that allow functions to handle a variable number of inputs gracefully.

A **positional argument** is matched by its position in a call; a **keyword argument** is passed as `name=value`. The names `args` and `kwargs` are conventions; the `*` and `**` provide the behavior.

Using `*args` for Multiple Positional Arguments

The `*args` syntax lets a function accept any number of **positional arguments**.

Inside the function, all the extra values are stored together in a **tuple**.

The built-in `sum()` adds the numbers in a collection; an empty collection has a sum of zero. Let's use it in a function.

```
def add_numbers(*args):
    print("Received:", args)
    print("Sum:", sum(args))
```

Listing 206 — Using `*args` for Multiple Positional Arguments

When you call the function, you can pass **as many numbers** as you want:

```
add_numbers(2, 4, 6)
add_numbers(5, 10, 15, 20)
```

Listing 207 — Using `*args` for Multiple Positional Arguments

Output

```
Received: (2, 4, 6)
Sum: 12
Received: (5, 10, 15, 20)
Sum: 50
```

Here, `*args` collects all arguments into a tuple named `args`, allowing the function to handle flexible input lengths.

Using `**kwargs` for Keyword Arguments

While `*args` gathers unnamed (positional) values, `**kwargs` collects **named arguments** into a **dictionary**.

This is useful when you want to handle labeled information.

```
def describe_person(**kwargs):
    print("Person details:")
    for key, value in kwargs.items():
        print(f"{key}: {value}")
```

Listing 208 — Using `**kwargs` for Keyword Arguments

You can now call the function with any combination of keyword arguments:

```
describe_person(name="Alice", age=28, city="Paris")

describe_person(name="Bob", job="Engineer")
```

Listing 209 — Using **kwargs for Keyword Arguments

Output

```
Person details:
name: Alice
age: 28
city: Paris
Person details:
name: Bob
job: Engineer
```

Each argument becomes a key–value pair inside the kwargs dictionary, giving you full flexibility to pass variable information.

Combining *args and **kwargs

You can even use both in the same function to handle any number of positional and keyword arguments together.

This makes your functions adaptable to many scenarios.

```
def show_info(*args, **kwargs):
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)
```

Listing 210 — Combining *args and **kwargs

```
show_info("Python", "Data Science", version=3.12, difficulty="Intermediate")
```

Listing 211 — Combining *args and **kwargs

Output

```
Positional arguments: ('Python', 'Data Science')
Keyword arguments: {'version': 3.12, 'difficulty': 'Intermediate'}
```

This combination is powerful when building reusable functions that must handle both kinds of data flexibly.

Mini Project — Student Record Formatter

Write a program that collects information about students using flexible arguments.

The function should accept **any number** of names and shared details (like course or year) and print them neatly. In `','.join(names)`, the separator joins the strings in names into one string. These details describe the group, not a separate record for each student.

```
def student_info(*names, **details):
    print("Students:", ", ".join(names))
    print("Details:")
    for key, value in details.items():
        print(f"  {key}: {value}")

student_info("Alice", "Bob", "Clara", course="Python", year=2025, instructor="Dr. Smith")
```

Listing 212 — Mini Project — Student Record Formatter

Output

```
Students: Alice, Bob, Clara
Details:
  course: Python
  year: 2025
  instructor: Dr. Smith
```

This small project demonstrates how *args and **kwargs let functions adapt to changing requirements — an essential skill in writing flexible and professional Python programs.

Scope and Decorators

As your programs grow larger, understanding where and how variables exist becomes important.

This idea is known as **scope** — the part of a program where a variable can be seen or used.

Once you understand scope, we'll move to **decorators**, a feature that lets you extend or modify the behavior of functions without changing their code.

Understanding Variable Scope

A **namespace** maps names to objects. **Scope** determines where Python looks for a name.

We begin with local and global scope, then examine an enclosing function's scope: - **Local scope**: variables created inside a function, available only there. - **Global scope**: variables created outside any function, available throughout the file.

Let's see a simple example:

```
message = "Hello from the global scope"

def greet():
    local_message = "Hello from inside the function"
    print(local_message)    # accessible only inside the function
    print(message)         # accessing the global variable

greet()
print(message) # accessible anywhere
```

Listing 213 — Understanding Variable Scope

Output

```
Hello from inside the function
Hello from the global scope
Hello from the global scope
```

If you try to print `local_message` outside the function, Python will raise an error because it only exists within `greet()`.

The `global` Keyword

Sometimes, you may want to modify a global variable inside a function.

To do this, you must declare it as `global`:

```
count = 0

def increase_count():
    global count
    count += 1
    print("Inside function:", count)

increase_count()
print("Outside function:", count)
```

Listing 214 — The `global` Keyword

Output

```
Inside function: 1
Outside function: 1
```

Without the `global` keyword, Python would assume `count` is a new local variable and raise an error.

The `nonlocal` Keyword and Nested Functions

A function can also contain another function.

An inner function can read names from an enclosing function without `nonlocal`. Use `nonlocal` when the inner function needs to assign to an existing name in that enclosing function.

```
def outer():
    x = "outer value"

    def inner():
        nonlocal x
        x = "inner value"
        print("Inside inner:", x)

    inner()
    print("Inside outer:", x)

outer()
```

Listing 215 — The `nonlocal` Keyword and Nested Functions

Output

```
Inside inner: inner value
Inside outer: inner value
```

Here, `nonlocal` allows the inner function to modify a variable that belongs to the enclosing function.

Function Decorators

Now that you know about scope, decorators will make more sense.

A **decorator** is a function that takes another function as input, adds some extra behavior, and returns it. They are often used for logging, access control, timing, or formatting output.

Let's start with a simple example.

```
def simple_decorator(func):
    def wrapper():
        print("Before the function runs")
        func()
        print("After the function runs")
    return wrapper
```

Listing 216 — Function Decorators

You can apply the decorator to another function using the `@` symbol:

```
@simple_decorator
def greet():
    print("Hello, Python!")

greet()
```

Listing 217 — Function Decorators

Output

```
Before the function runs
Hello, Python!
After the function runs
```

Here's what happens:

- The `@simple_decorator` line tells Python to pass the function `greet` to `simple_decorator` and replace `greet` with the returned wrapper.
- The `wrapper()` function adds code before and after the original function.
- When `greet()` is called, it runs inside the decorator.

Decorators with Arguments

You can make decorators more flexible by allowing them to work with any function arguments using `*args` and `**kwargs`.

```
def logger(func):
    def wrapper(*args, **kwargs):
        print(f"Running {func.__name__} with arguments {args} {kwargs}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} finished with result {result}")
        return result
    return wrapper

@logger
def add(a, b):
    return a + b

add(3, 5)
```

Listing 218 — Decorators with Arguments

Output

```
Running add with arguments (3, 5) {}
add finished with result 8
```

This pattern is often used in real-world applications — for example, in web frameworks, decorators can check user permissions, handle errors, or measure performance.

Python Built-in Functions

You have already defined functions. Python also provides **built-in functions**, available without an import, that can help you process values and collections.

They make it easier to perform everyday tasks such as generating sequences, transforming data, or working with collections.

This section focuses on five commonly used built-in functions that appear throughout real-world programs: `range()`, `sorted()`, `map()` and `filter()`. We then introduce the separate standard-library module `random`, which must be imported.

The `range()` Function

The `range()` function generates a sequence of numbers. It is commonly used in loops or whenever you need to repeat something a specific number of times.

It can be called in three forms:

- `range(stop)` □ starts at 0, ends before stop
- `range(start, stop)` □ starts at start, ends before stop
- `range(start, stop, step)` □ adds a step between numbers

Let's see it in action:

```
print(list(range(5)))      # 0 to 4
print(list(range(2, 8)))   # 2 to 7
print(list(range(0, 10, 2))) # even numbers
```

Listing 219 — The `range()` Function

Output — one possible run

```
[0, 1, 2, 3, 4]
[2, 3, 4, 5, 6, 7]
[0, 2, 4, 6, 8]
```

The first example starts from 0 by default, while the second begins at 2.

The third example introduces a step of 2, producing only even numbers.

The stop value is always excluded, meaning the sequence ends just before it.

The `sorted()` Function

The `sorted()` function arranges items in order and returns a new list, leaving the original unchanged. You can also specify whether sorting should be ascending, descending, or follow a custom rule.

```
numbers = [3, 1, 4, 2]
print(sorted(numbers))      # ascending order
print(sorted(numbers, reverse=True)) # descending order
```

Listing 220 — The `sorted()` Function

Output — one possible run

```
[1, 2, 3, 4]
[4, 3, 2, 1]
```

Here, `sorted()` first arranges numbers from smallest to largest, then reverses the order when `reverse=True`.

You can even sort text by a custom rule using the `key` parameter.

In this example, words are sorted by length:

```
words = ["banana", "fig", "apple"]
print(sorted(words, key=len))
```

Listing 221 — The `sorted()` Function

Output — one possible run

```
['fig', 'apple', 'banana']
```

The `map()` Function

The `map()` function *applies* another function *to each element in a sequence*.

It takes two arguments: a function and an iterable.

Let's use it to square all numbers in a list:

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

numbers = [1, 2, 3, 4]
result = map(square, numbers)
print(list(result))
```

Listing 222 — The `map()` Function

Output — one possible run

```
[1, 4, 9, 16]
```

`map()` returns an **iterator**, which supplies results one at a time as requested. `list(result)` collects those results into a list. This expresses a transformation without an explicit loop.

The `filter()` Function

The `filter()` function extracts only the elements that satisfy a condition.

It also takes two arguments: a function that returns `True` or `False`, and an iterable.

In the example below, only even numbers are kept.

```
def is_even(x):
    return x % 2 == 0

numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(is_even, numbers)
print(list(even_numbers))
```

Listing 223 — The filter() Function

Output — one possible run

[2, 4, 6]

The random Module

The random module lets Python generate **pseudorandom numbers** and choices, which are essential for games, simulations, and sampling data.

To use it, you must first import it:

```
import random
```

Listing 224 — The random Module

Let's look at the most useful random functions and how their parameters work.

```
###random.random()
```

This function returns a **floating-point number** between 0.0 (inclusive) and 1.0 (exclusive).

It takes no arguments.

```
print(random.random())
```

Listing 225 — The random Module

Output — one possible run

0.8533700289991242

Each call generates a new random value, perfect for simulations or percentages.

```
###random.randint(a, b)
```

This function returns a **random integer** between a and b, both included.

It takes **two integer arguments**:

- a: the lowest possible value
- b: the highest possible value

```
print(random.randint(1, 6)) # simulating a dice roll
```

Listing 226 — The random Module

Output — one possible run

2

Each time you run it, you'll get an integer from 1 to 6.

```
###random.choice(sequence)
```

This function returns one random element from a given sequence (like a list or tuple).

It takes sequence as argument for the collection to choose from.

```
fruits = ["apple", "banana", "cherry", "date"]
print(random.choice(fruits))
```

Listing 227 — The random Module

Output — one possible run

```
banana
```

Each time it randomly selects one fruit.

```
###random.shuffle(list)
```

This function rearranges the elements of a list in place, meaning it modifies the original list.

It takes a list as its single argument.

```
cards = ["Ace", "King", "Queen", "Jack"]
random.shuffle(cards)
print(cards)
```

Listing 228 — The random Module

Output — one possible run

```
['King', 'Ace', 'Jack', 'Queen']
```

The order may change; a shuffle can also leave a list in the same order.

Combined Random Functions — Random Prize Generator

Let's now combine these random functions into a short program that simulates a prize draw at a fair. A few random choices, a number roll, and a lucky probability all come together in one example.

```
import random

# Step 1: shuffle participant list
participants = ["Anna", "Ben", "Clara", "David", "Ella"]
random.shuffle(participants)
print("Shuffled participants:", participants)

# Step 2: pick a random winner
winner = random.choice(participants)

# Step 3: assign a random prize value between 10 and 100
prize = random.randint(10, 100)

# Step 4: decide if there's a bonus prize based on probability
bonus_chance = random.random()
if bonus_chance > 0.8:
    print(f"🎉 Congratulations {winner}! You won ${prize} plus a bonus gift!")
else:
    print(f"🎉 Congratulations {winner}! You won ${prize}.")
```

Listing 229 — Combined Random Functions — Random Prize Generator

Output — one possible run

```
Shuffled participants: ['Clara', 'Ella', 'David', 'Anna', 'Ben']  
☐ Congratulations Anna! You won $94.
```

Summary

Built-in functions are the building blocks of many Python programs. They allow you to generate sequences, sort or transform data, while the separate `random` module supplies pseudorandom values.

You'll soon combine many of these functions with **lambda expressions** and **user-defined functions** to build smart, flexible programs.

Lambda Functions, Recursion, and Generators

As you become more comfortable with Python functions, you'll discover that they can be written and used in different ways.

In this section, we explore three powerful techniques that make Python functions more expressive and efficient: **lambda functions**, **recursion**, and **generators**.

Lambda Functions — Quick, Anonymous Functions

Sometimes, you need a small function only once, such as when sorting data or filtering a list. Instead of writing a full `def` block, Python allows you to create **anonymous functions** using the `lambda` keyword.

A lambda function can have any number of arguments, but only one expression. The expression is automatically returned.

```
square = lambda x: x ** 2
print(square(5))
```

Listing 230 — Lambda Functions — Quick, Anonymous Functions

Output

```
25
```

This is equivalent to:

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

Listing 231 — Lambda Functions — Quick, Anonymous Functions

Lambda functions are often used with built-in functions like `map()`, `filter()`, or `sorted()`.

For example:

```
numbers = [3, 6, 9, 2, 5]
doubled = list(map(lambda x: x * 2, numbers))
print(doubled)
```

Listing 232 — Lambda Functions — Quick, Anonymous Functions

Output

```
[6, 12, 18, 4, 10]
```

Here, `map()` applies the lambda function to each element of the list — doubling all numbers without needing a separate function definition.

Recursion — A Function That Calls Itself

Recursion is a technique where a function calls itself to solve a smaller version of the same problem. It is often used in mathematical operations or tasks that naturally repeat with smaller inputs — such as calculating factorials or navigating nested data.

For a nonnegative integer n , the factorial is the product of the integers from 1 to n ; $0!$ is 1. This example assumes a nonnegative integer input.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Listing 233 — Recursion — A Function That Calls Itself

```
print(factorial(5))
```

Listing 234 — Recursion — A Function That Calls Itself

Output

```
120
```

Here's what happens:

1. `factorial(5)` calls `factorial(4)`
2. `factorial(4)` calls `factorial(3)` and so on
3. When `n == 0`, the recursion stops (base case)
4. The results are multiplied as the calls return.

It's important to always include a **base case**, otherwise the function will call itself forever and cause a recursion error.

Generators — Producing Data Efficiently

Generators are functions that can pause and resume their execution.

When a generator function is called, it returns a **generator object**, which is an iterator.

Its body has not started running. Execution begins when a value is requested, through iteration or `next()`.

`yield` supplies a value and pauses execution. A return or the end of the body finishes the generator.

Example

Here's an example that generates numbers up to a limit:

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1

# Create a generator
numbers = count_up_to(5)

# Iterate through values one by one
for num in numbers:
    print(num)
```

Listing 235 — Example

Output

```
1
2
3
4
5
```

Here's what happens step by step:

- The function `count_up_to()` is called, but it doesn't run completely—it returns a generator object.
- Each iteration in the `for` loop triggers the next `yield` statement.
- The function pauses after yielding each number, and resumes the next time the loop requests a new value.

Using `next()` Manually

You can also control generators manually with the `next()` function, which retrieves the next yielded value each time it's called.

```
gen = count_up_to(3)
print(next(gen))
print(next(gen))
print(next(gen))
```

Listing 236 — Using `next()` Manually

Output

```
1
2
3
```

Why Use Generators?

Generators are **memory-efficient** because they produce one value at a time instead of creating an entire list in memory. This makes them especially powerful when working with large sequences or infinite streams of data.

For instance, compare a generator to building a list directly:

```
# Regular list - builds all items at once
nums_list = [i for i in range(1000000)]

# Generator - builds items on demand
nums_gen = (i for i in range(1000000))
```

Listing 237 — Why Use Generators?

The list version stores one million numbers in memory.

The generator version, using parentheses `()`, produces numbers **on demand**, saving resources.

Mini Project — Random Quote Stream

Let's use a generator to simulate an ongoing stream of random quotes every time the user requests one. It combines your knowledge of **functions**, **generators**, and the **random module**.

```
import random

def quote_stream():
    quotes = [
        "Keep learning, keep growing.",
        "Code is like humor – when you have to explain it, it's bad.",
        "Simplicity is the soul of efficiency.",
        "In theory, there's no difference between theory and practice."
    ]
    while True:
        yield random.choice(quotes)

# Create a generator
stream = quote_stream()

# Get three random quotes
print(next(stream))
print(next(stream))
print(next(stream))
```

Listing 238 — Mini Project — Random Quote Stream

Output — one possible run

```
Simplicity is the soul of efficiency.
Keep learning, keep growing.
Simplicity is the soul of efficiency.
```

Each call to `next(stream)` retrieves a new random quote, while the generator keeps running indefinitely until stopped.

This demonstrates how generators can power **infinite or dynamic data flows** efficiently.

6. Modules

Introduction to Modules

Functions organise a task within a program. Modules extend that organisation across files, allowing related definitions to be kept together and used elsewhere.

As programs grow, it becomes inefficient to keep all the code in one file. Python solves this with **modules**—separate files that organize reusable code. A module can contain functions, variables, and even entire classes, making your projects cleaner and easier to manage.

Some modules come with Python’s **standard library**, such as `math` and `datetime`; others are files you write or packages you install. Once imported, the functions inside a module can be used anywhere in your program.

Creating and Importing a Module

You can create your own module by saving a Python file (for example, `helper.py`) with some functions inside it.

```
# helper.py
def greet(name):
    return f"Hello, {name}! Welcome to Python."
```

Listing 239 — Creating and Importing a Module

Save the first listing as `helper.py`. Put the following code in `main.py` in the same folder and run `main.py`; the file named `helper.py` supplies the imported module:

```
import helper
print(helper.greet("Alice"))
```

Listing 240 — Creating and Importing a Module

Output

```
Hello, Alice! Welcome to Python.
```

This example is deterministic: the same name produces the same greeting.

You can also import specific items:

```
from helper import greet
print(greet("Bob"))
```

Listing 241 — Creating and Importing a Module

Output

```
Hello, Bob! Welcome to Python.
```

Or use an alias:

```
import helper as h
print(h.greet("Ada"))
```

Listing 242 — Creating and Importing a Module

Output

```
Hello, Ada! Welcome to Python.
```

The datetime Module

The `datetime` module allows you to work with **dates and times**—useful for logs, event scheduling, and timestamps.

You can use it to get the current time, create specific date objects, and format dates in readable ways.

Python Dates

The simplest way to use the module is to get the current date and time.

```
import datetime
now = datetime.datetime.now()
print("Current date and time:", now)
```

Listing 243 — Python Dates

Output — one possible run

```
Current date and time: 2026-10-24 18:13:26.584252
```

Here, `datetime.datetime.now()` returns the local date and time. This call does not attach timezone information; its value changes between runs.

Date Output and Components

You can access each part of a date individually.

```
print("Year:", now.year)
print("Month:", now.month)
print("Day:", now.day)
```

Listing 244 — Date Output and Components

Output — one possible run

```
Year: 2026
Month: 10
Day: 24
```

This is useful for filtering data or formatting custom reports.

Creating Date Objects

To represent specific dates, use `datetime.date(year, month, day)`.

```
birthday = datetime.date(1995, 6, 15)
print("Birthday:", birthday)
```

Listing 245 — Creating Date Objects

Output — one possible run

```
Birthday: 1995-06-15
```

Formatting Dates

Dates can be formatted into readable strings using `.strftime(format)`.

```
formatted = now.strftime("%A, %d %B %Y")
print("Formatted date:", formatted)
```

Listing 246 — Formatting Dates

Output — one possible run

```
Formatted date: Saturday, 24 October 2026
```

Common format codes include:

- `%A` — weekday name
- `%d` — day of month
- `%B` — month name
- `%Y` — four-digit year

The math Module

Mathematics is a crucial part of many programming tasks — from simple calculations to advanced data analysis.

Python's **math module** provides access to mathematical functions that go beyond basic arithmetic.

It includes **constants**, **power** and **logarithmic functions**, **trigonometry**, and **rounding tools** that make it easier to perform scientific and statistical computations.

To use any of these functions, you must first import the module:

```
import math
```

Listing 247 — The math Module

Mathematical Constants

The **math** module includes useful constants like π (pi) and *e* (Euler's number), which are widely used in geometry, physics, and probability.

```
print("Pi constant:", math.pi)
print("Euler's number:", math.e)
```

Listing 248 — Mathematical Constants

Output

```
Pi constant: 3.141592653589793
Euler's number: 2.718281828459045
```

You can use these constants to compute values like the circumference of a circle:

```
radius = 5
circumference = 2 * math.pi * radius
print("Circumference of the circle:", circumference)
```

Listing 249 — Mathematical Constants

Output

```
Circumference of the circle: 31.41592653589793
```

Power and Roots

The **math** module provides functions for **exponentiation** and **square roots**. `math.pow()` converts its arguments to floats; integer exponentiation with `**` can retain exact integer results.

```
print("Square root of 25:", math.sqrt(25))
print("2 raised to the power of 5:", math.pow(2, 5))
print("Exponential of 2:", math.exp(2))
```

Listing 250 — Power and Roots

Output

```
Square root of 25: 5.0
2 raised to the power of 5: 32.0
Exponential of 2: 7.38905609893065
```

Here:

- `sqrt(x)` computes $\sqrt{x}$
- `pow(x, y)` returns x raised to y
- `exp(x)` returns e^x These are fundamental in fields like data science and finance where growth rates and probabilities are calculated.

Logarithmic Functions

Logarithms are used to measure **growth, decay, and changes in scale**, such as population growth, sound intensity, or pH levels in chemistry. The `math` module provides several logarithmic functions for different bases:

- `math.log(x)` returns the **natural logarithm** (base e), where e is a special mathematical constant approximately equal to **2.718**, often used to describe continuous growth or decay.
- `math.log10(x)` returns the **base-10 logarithm**, commonly used in engineering and scientific scales.
- `math.log2(x)` returns the **base-2 logarithm**, often used in computer science for binary calculations.

For example:

```
print("Natural log of e:", math.log(math.e))
print("Log base 10 of 1000:", math.log10(1000))
print("Log base 2 of 8:", math.log2(8))
```

Listing 251 — Logarithmic Functions

Output

```
Natural log of e: 1.0
Log base 10 of 1000: 3.0
Log base 2 of 8: 3.0
```

Here:

- `math.log(10)` means $\ln(10)$, or the power you raise e to get 10.
- `math.log10(1000)` equals 3 because $10^3 = 1000$.
- `math.log2(8)` equals 3 because $2^3 = 8$.

Logarithms appear everywhere — from analyzing exponential growth to compressing data ranges in visualizations.

These are essential when working with sound frequencies, exponential growth models, and data transformations.

Rounding Functions: `floor()`, `ceil()`

The `math` module includes precise rounding functions that control how numbers are approximated.

```
print("Floor of 3.7:", math.floor(3.7)) # Rounds down
print("Ceil of 3.2:", math.ceil(3.2))  # Rounds up
print("Absolute value of -5.3:", math.fabs(-5.3))
```

Listing 252 — Rounding Functions: floor(), ceil()

Output

```
Floor of 3.7: 3
Ceil of 3.2: 4
Absolute value of -5.3: 5.3
```

Use `floor` to count complete units and `ceil` to find the number of containers needed. Monetary rounding requires its own explicit rules.

Trigonometric Functions: `sin()`, `cos()`, `tan()`

The `math` module includes trigonometric functions such as `sin()`, `cos()`, and `tan()`.

These functions expect angles in **radians**, not degrees, and are used to calculate the sine, cosine, and tangent of an angle.

```
import math

angle = math.pi / 4 # 45 degrees in radians

print("Sine:", math.sin(angle))
print("Cosine:", math.cos(angle))
print("Tangent:", math.tan(angle))
```

Listing 253 — Trigonometric Functions: `sin()`, `cos()`, `tan()`

Output

```
Sine: 0.7071067811865475
Cosine: 0.7071067811865476
Tangent: 0.9999999999999999
```

Here:

- `math.sin(x)` returns the sine of the angle `x`.
- `math.cos(x)` returns the cosine of the angle `x`.
- `math.tan(x)` returns the tangent of the angle `x`.

These functions are used in geometry, physics, and computer graphics where angles and rotations need to be measured precisely.

Since trigonometric functions use radians, you can convert degrees to radians using `math.radians()`:

```
degrees = 90
radians = math.radians(degrees)
print(f"{degrees}° in radians is:", radians)
```

Listing 254 — Trigonometric Functions: `sin()`, `cos()`, `tan()`

Output

```
90° in radians is: 1.5707963267948966
```

And to convert back from radians to degrees, use `math.degrees()`:

```
print("Radians to degrees:", math.degrees(math.pi))
```

Listing 255 — Trigonometric Functions: `sin()`, `cos()`, `tan()`

Output

```
Radians to degrees: 180.0
```

Factorials, Combinations, and Permutations

The `math` module also includes tools for **counting and probability**, which are very common in statistics and combinatorics.

- `math.factorial(n)` returns the factorial of a number, which is the product of all positive integers up to `n`.
- `math.comb(n, k)` returns the number of ways to choose `k` items from `n` items (without regard to order).
- `math.perm(n, k)` returns the number of ways to arrange `k` items from `n` items (where order matters).

Let's see these in action:

```
import math

print("Factorial of 5:", math.factorial(5))
print("Combinations of 5 choose 2:", math.comb(5, 2))
print("Permutations of 5 choose 2:", math.perm(5, 2))
```

Listing 256 — Factorials, Combinations, and Permutations

Output

```
Factorial of 5: 120
Combinations of 5 choose 2: 10
Permutations of 5 choose 2: 20
```

Here:

- `factorial(5)` gives $5 \times 4 \times 3 \times 2 \times 1 = \mathbf{120}$
- `comb(5, 2)` counts how many ways we can pick 2 items from 5, giving **10**
- `perm(5, 2)` counts how many ways we can arrange 2 items from 5, giving **20**

These functions are useful in fields like **probability**, **statistics**, and **game design**, where you need to count possible outcomes or arrangements.

Rounding to Whole Numbers

Sometimes you only need the integer part of a number.

The function `math.trunc()` removes everything after the decimal point — it **cuts off** the fractional part instead of rounding up or down.

```
print("Truncate 7.9:", math.trunc(7.9))  
print("Truncate -3.2:", math.trunc(-3.2))
```

Listing 257 — Rounding to Whole Numbers

Output

```
Truncate 7.9: 7  
Truncate -3.2: -3
```

Unlike `math.floor()` or `math.ceil()`, which round down or up, `math.trunc()` simply **drops** the decimal part and keeps the integer portion.

The json Module

The `json` module allows you to work with **JSON data**—a lightweight format widely used for sharing data between systems and APIs.

It helps you convert between Python objects and JSON text easily.

JSON in Python

JSON (JavaScript Object Notation) represents data as text but looks similar to Python dictionaries. This makes conversion between the two very natural in Python.

Parse JSON — Convert from JSON to Python

`json.loads()` parses JSON text into the corresponding Python values: an object becomes a dictionary, an array becomes a list, and scalar values become the corresponding Python scalars. Here the input is an object:

```
import json

data = '{"name": "Alice", "age": 25, "city": "Paris"}'
person = json.loads(data)
print(person["city"])
```

Listing 258 — Parse JSON — Convert from JSON to Python

Output

```
Paris
```

Convert from Python to JSON

You can use `json.dumps()` to turn Python data structures into JSON strings.

```
student = {
    "name": "Bob",
    "age": 22,
    "courses": ["Math", "AI"]
}

json_string = json.dumps(student)
print(json_string)
```

Listing 259 — Convert from Python to JSON

Output

```
{"name": "Bob", "age": 22, "courses": ["Math", "AI"]}
```

Formatting the Result

You can make JSON output more readable by adding indentation and sorting keys.

```
print(json.dumps(student, indent=4, sort_keys=True))
```

Listing 260 — Formatting the Result

Output

```
{
  "age": 22,
  "courses": [
    "Math",
    "AI"
  ],
  "name": "Bob"
}
```

The `json` module is essential for saving configurations, web communication, and data storage.

The re (Regular Expressions) Module

The re module lets you **search**, **match**, and **manipulate text patterns** efficiently using *regular expressions*. They are powerful for validating input, cleaning data, and analyzing text.

RegEx in Python

Before using any regular expressions, you must import Python’s built-in re module:

```
import re
```

Listing 261 — RegEx in Python

When you have imported the re module, you can start using regular expressions.

A **regular expression** describes a text pattern. In the pattern below, ^ marks the start, .* permits zero or more characters other than a newline, and \$ marks the end (or the position before a final newline). re.search() returns a match object when it finds the pattern, or None otherwise.

Let’s search for text that starts with “The” and ends with “Spain”:

```
import re

txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)
if x:
    print("YES! We have a match!")
else:
    print("No match")
```

Listing 262 — RegEx in Python

Output

```
YES! We have a match!
```

RegEx Functions and Syntax Overview

Before diving into examples, let’s review the key **functions and symbols** that make regular expressions so powerful in Python.

Regular expressions (RegEx) work through the re module, which provides several built-in functions for searching, splitting, and replacing text.

RegEx Functions

The re module offers a set of functions that allow you to **search**, **extract**, or **modify** text patterns:

Function	Description
findall()	Returns a list containing all matches
search()	Returns a Match object if a match is found anywhere in the string

Function	Description
<code>split()</code>	Splits a string into a list at each match
<code>sub()</code>	Replaces one or many matches with a specified string

Table 6 — RegEx Functions

These functions form the foundation of most regular expression tasks in Python.

Metacharacters

Metacharacters are **special symbols** with unique meanings in regular expressions. They let you define flexible patterns instead of exact text matches.

Character	Description	Example
<code>[]</code>	A set of characters	<code>[a-m]</code>
<code>\</code>	Signals a special sequence (can also escape special characters)	<code>\d</code>
<code>.</code>	Any character except newline	<code>he..o</code>
<code>^</code>	Matches the start of a string	<code>^hello</code>
<code>\$</code>	Matches the end of a string	<code>planet\$</code>
<code>*</code>	Zero or more occurrences	<code>he.*o</code>
<code>+</code>	One or more occurrences	<code>he.+o</code>
<code>?</code>	Zero or one occurrence	<code>he.?o</code>
<code>{}</code>	Exactly the specified number of occurrences	<code>he.{2}o</code>
<code> </code>	Either/or condition	<code>falls stays</code>
<code>()</code>	Capture and group expressions	<code>(abc)</code>

Table 7 — Metacharacters

Flags

Flags modify how regular expressions are interpreted. They can be combined with functions like `re.search()` or `re.findall()` to change matching behavior.

Flag	Shorthand	Description
<code>re.ASCII</code>	<code>re.A</code>	Use ASCII rules for character classes such as <code>\w</code> , <code>\d</code> and <code>\s</code>
<code>re.DEBUG</code>	—	Print debugging information

Flag	Shorthand	Description
<code>re.DOTALL</code>	<code>re.S</code>	Make <code>.</code> match all characters, including newlines
<code>re.IGNORECASE</code>	<code>re.I</code>	Case-insensitive matching
<code>re.MULTILINE</code>	<code>re.M</code>	Make <code>^</code> and <code>\$</code> match at line boundaries
<code>re.UNICODE</code>	<code>re.U</code>	Default behavior (matches Unicode text)
<code>re.VERBOSE</code>	<code>re.X</code>	Allow spaces and comments in complex patterns for readability

Table 8 — Flags

Special Sequences

Special sequences start with a backslash (`\`) and represent common text patterns like digits, spaces, or word characters. When used inside raw strings (like `r"\d"`), Python reads them literally.

Sequence	Description	Example
<code>\A</code>	Match if characters are at the start of the string	<code>\AThe</code>
<code>\b</code>	Match at the start or end of a word	<code>r"\bain"</code> or <code>r"ain\b"</code>
<code>\B</code>	Match a position that is not a word boundary	<code>r"\Bain"</code>
<code>\d</code>	Match any Unicode decimal digit; use <code>[0–9]</code> for ASCII digits	<code>\d</code>
<code>\D</code>	Match any non-digit	<code>\D</code>
<code>\s</code>	Match any whitespace (space, tab, newline)	<code>\s</code>
<code>\S</code>	Match any non-whitespace character	<code>\S</code>
<code>\w</code>	Match any word character (letters, digits, underscore)	<code>\w</code>
<code>\W</code>	Match any non-word character	<code>\W</code>
<code>\Z</code>	Match if characters are at the end of the string	<code>Spain\Z</code>

Table 9 — Special Sequences

Sets

A **set** is a collection of characters enclosed in square brackets `[ ]`. It matches **any single character** inside the brackets.

Set	Description
<code>[arn]</code>	Matches if one of the characters <code>a</code> , <code>r</code> , or <code>n</code> is present
<code>[a-n]</code>	Matches any lowercase letter between <code>a</code> and <code>n</code>
<code>[^arn]</code>	Matches any character except <code>a</code> , <code>r</code> , and <code>n</code>
<code>[0123]</code>	Matches if one of <code>0</code> , <code>1</code> , <code>2</code> , or <code>3</code> is present
<code>[0-9]</code>	Matches any digit between <code>0</code> and <code>9</code>
<code>[0-5][0-9]</code>	Matches any two-digit number from <code>00</code> to <code>59</code>
<code>[a-zA-Z]</code>	Matches any alphabetic character, lowercase or uppercase
<code>[+]</code>	Matches a literal <code>+</code> (special characters lose meaning inside sets)

Table 10 — Sets

With these tools — functions, metacharacters, flags, special sequences, and sets — you can define almost any pattern imaginable, from finding numbers in a text to validating phone numbers or emails.

Applying RegEx Functions

We will now demonstrate how useful functions such as `findall()`, `search()`, `split()`, and `sub()` can be used with `re`.

Using `findall()` Function

The `findall()` function returns a **list** of all parts of the string that match a pattern.

```
import re

text = "I have 2 apples, 5 bananas, and 12 oranges."
numbers = re.findall(r"\d+", text)
print(numbers)
```

Listing 263 — Using `findall()` Function

Output

```
['2', '5', '12']
```

- `\d+` means “find one or more digits.”
- The function goes through the text and extracts every number it finds.
- The result is a list of numbers written as strings.

- Note: The prefix `r"..."` tells Python to treat the pattern as a **raw string**, meaning backslashes (`\`) are read literally and not as escape characters.

Using `search()` Function

The `search()` function looks for the **first match** of a pattern and returns a match object if found.

```
text = "Hello world"
match = re.search(r"world", text)

if match:
    print("Found:", match.group())
else:
    print("No match")
```

Listing 264 — Using `search()` Function

Output

```
Found: world
```

Explanation:

- The pattern `"world"` is found inside the string `"Hello world"`.
- `match.group()` returns the exact part of the text that matched.

Using `split()` Function

The `split()` function breaks a string into parts wherever the pattern appears.

```
text = "apple,banana;cherry orange"
result = re.split(r"[;, ]", text)
print(result)
```

Listing 265 — Using `split()` Function

Output

```
['apple', 'banana', 'cherry', 'orange']
```

Explanation:

- The pattern `[;, ]` tells Python to split wherever there is a comma, semicolon, or space.
- The result is a clean list of fruit names.

This handles simple delimiter-separated text. For CSV with quoted fields, use the `csv` module introduced in File Handling.

Python PIP

Python includes a standard library. When a program needs another package, use pip to install it into the Python environment that runs the program.

Check the interpreter

A notebook kernel may use a different interpreter from your terminal. Inspect its executable before installing packages.

```
import sys
print(sys.executable)
print(sys.version)
```

Listing 266 — Check the interpreter

Output

```
/opt/anaconda3/bin/python
3.13.5 | packaged by Anaconda, Inc. | (main, Jun 12 2025, 11:23:37) [Clang 14.0.6 ]
```

Install and inspect a package

Run these commands in a terminal with your chosen environment active:

```
python -m pip --version
python -m pip install requests
python -m pip show requests
python -m pip list
```

Listing 267 — Install and inspect a package

`python -m pip` uses pip belonging to that interpreter. After installation, a Python program imports the package with `import requests`. The package name used for installation and the import name are not always identical.

Record dependencies

```
python -m pip freeze > requirements.txt
python -m pip install -r requirements.txt
```

Listing 268 — Record dependencies

The first command records installed versions. The second installs the recorded requirements in another environment. Keep the requirements with the project. Recreate the environment rather than copying its directory between computers.

Reference: [pip user guide](#), checked 20 October 2026.

The try and except Blocks

An expression can be valid Python and still fail when it runs. Converting unsuitable text raises `ValueError`; dividing by zero raises `ZeroDivisionError`.

Handle an expected failure

Put the operation that may fail in `try`. An `except` block handles the named exception. Other exceptions continue to the caller.

```
answer = "not a number"
try:
    value = float(answer)
except ValueError:
    print("Enter a number.")
else:
    print(value)
```

Listing 269 — Handle an expected failure

Output

```
Enter a number.
```

Change `answer` to `"12.5"`. Predict which block runs, then rerun the cell. `else` runs when the `try` block completes without an exception.

Distinguish different errors

```
numerator = "12"
denominator = "0"
try:
    result = float(numerator) / float(denominator)
except ValueError:
    print("Both values must be numbers.")
except ZeroDivisionError:
    print("The divisor must not be zero.")
else:
    print(result)
```

Listing 270 — Distinguish different errors

Output

```
The divisor must not be zero.
```

Use denominator `"4"`, then `"zero"`, then `"0"`. Predict the result each time. Catch the error you can explain and handle; a broad handler can hide an unrelated programming mistake.

Cleanup with `finally`

A `finally` block runs when control leaves the `try` statement, including after an exception. Use it for cleanup. For files, a `with` statement usually expresses resource cleanup more clearly; the file-handling chapter develops that pattern.

Reference: [Python: Errors and Exceptions](#), checked 20 October 2026.

Virtual environments

Installing a package changes the environment in which it is installed. If two projects need different package versions, sharing one environment can create conflicts. A separate project environment keeps one project's requirements from changing another's.

An environment is not the project folder: the folder holds your program files, while the environment supplies Python and its installed packages.

Create a project environment

For the Anaconda route used in this book, conda can create an isolated environment with its own Python and packages. In Anaconda Prompt on Windows or the configured Terminal on macOS/Linux, enter:

```
conda create --name book-practice python numpy
```

Listing 41 — Create a named environment containing Python and NumPy.

Review the proposed installation and confirm it. `book-practice` is the environment's name; it does not create or replace your project's source folder. Naming the packages in the command ensures that this new environment has them, regardless of what is installed in `base.1`

Use it for the project

- Activate the environment in the terminal:

```
conda activate book-practice
```

Listing 42 — Select the project environment for terminal commands.

- Check the selected Python and NumPy:

```
python -c "import sys; print(sys.executable)"  
python -c "import numpy; print(numpy.__version__)"
```

Listing 43 — Check the interpreter location and installed NumPy version.

The path should identify `book-practice`; the second command should display a version number rather than an import error.

- In VS Code, use Python: Select Interpreter and choose `book-practice` for the open project. Terminal activation and editor selection must agree about which Python the project uses.

For a notebook, select the matching Python kernel—the process that executes its cells. Selecting an interpreter for saved files does not change a notebook that is already running with another kernel.

Leave the environment

```
conda deactivate
```

Listing 44 — Leave the active conda environment.

This changes the terminal's active environment; it does not delete the environment or the program files. Activate book-practice again when returning to the project.

How this relates to venv

Python also provides **venv**, a tool that creates an environment from an existing Python installation. Its files commonly live in a project's `.venv` folder. It serves the same isolation goal, but it is created and activated differently from a conda environment. Use one approach for a project; the commands above follow the book's conda route.²

With the environment chosen, both saved scripts and interactive sessions can use the same intended packages.

- conda, [Managing environments](#). Creation, activation and deactivation checked on 22 October 2026.□□
- Python documentation, [venv — Creation of virtual environments](#).□□

7. Object-Oriented Programming

7.1 Classes & Objects

What Is OOP?

OOP stands for **Object-Oriented Programming**, a programming paradigm—an approach to designing and writing programs—that organizes related data and actions into cooperating **objects**. Python supports OOP to help keep code structured, reusable and easier to maintain. Figure 11 shows three objects exchanging messages to handle a borrowing request in a library program.

The diagram is a conceptual sketch: each box represents a cooperating object. Follow a request from the borrowing screen to the loan service and then to the book record; the labels on the return direction show the replies. It introduces responsibilities and communication before we write the classes.

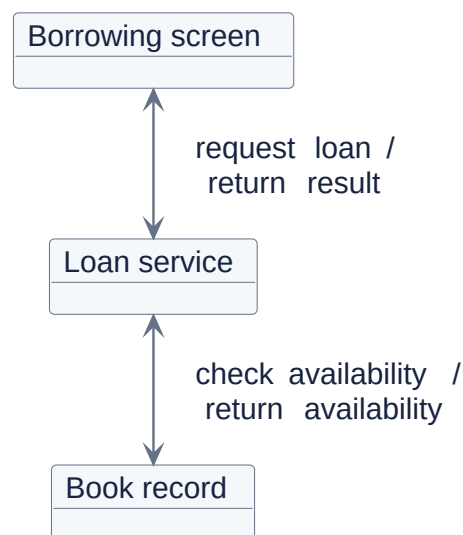


Figure 11 — Conceptual overview — objects cooperate: a borrowing request is checked and its result returned.

OOP offers three practical advantages:

- **Clear structure:** Group related responsibilities into logical parts.
- **Reusability:** Reuse shared logic instead of writing it again.
- **Easier maintenance and debugging:** Keeping responsibilities separate can limit the effects of a change and help you locate problems.

A useful principle for reusable, maintainable code is **DRY**, short for “**Don’t Repeat Yourself.**” When several parts of a program need the same calculation or check, write it once and reuse it instead of copying it. If the rule changes, you only need to update it in one place [1]. For example, write one function that checks whether a book can be borrowed, then call it wherever that check is needed. If

the borrowing rules change, update that function. DRY applies to functions as well as OOP.

We can now put these ideas into practice with [classes and objects](#), the building blocks of OOP. We will start by defining a small class in Python and creating objects from it.

Reference

[1] Thomas, D., & Hunt, A. (2019). *The Pragmatic Programmer: Your Journey to Mastery* (20th anniversary edition). Addison-Wesley. Chapter 2, “DRY—The Evils of Duplication,” Tip 15 (p. 31).

Creating Classes and Objects

A **class** defines a type of object and lets us create separate objects with the same structure. In Figure 12, `MyClass` is the class, while `p1`, `p2` and `p3` are names referring to objects created from it.

UML means **Unified Modeling Language**, a shared set of symbols for describing software. You can read these small diagrams before learning to draw them. A class box names a type; an object box names a particular instance of that type. In the underlined label `p1 : MyClass`, `p1` names the object and `MyClass` names its class. The underline distinguishes this object label from the plain class name. Here, one class definition can produce several separate instances. The later UML lessons introduce each diagram type in more detail.



Figure 12 — One class can create several separate objects.

Define a Class

To create a class, use the keyword `class`, followed by a name and a colon (`:`). Write the class body on the following lines, using indentation. The body must contain a statement, so an empty class uses `pass` as a placeholder. This lets us define `MyClass` before adding any data or actions:

```
class MyClass:
    pass
```

Listing 271 — Define a Class

`pass` performs no action. The class definitions in this section produce no visible output.

A class can have **attributes**: named values attached to the class or its objects. To give `MyClass` a value, replace `pass` with the assignment `x = 5`.

```
class MyClass:
    x = 5
```

Listing 272 — Define a Class

This defines `x` as a **class attribute** of `MyClass`. We will explore attributes in the next section.

Create Objects

Once a class is defined, it can be used to create objects, also called **instances** of the class.

```
p1 = MyClass()
print(p1.x)
```

Listing 273 — Create Objects

Output

```
5
```

In this example, the parentheses in `MyClass()` call the class to create an instance. The assignment gives it the variable name `p1`. The dot in `p1.x` accesses `x`, which the object finds on its class; `print(p1.x)` displays 5.

Predict what the following code will print, then run it:

```
p1 = MyClass()
p2 = MyClass()
p3 = MyClass()

print(p1.x)
print(p2.x)
print(p3.x)
```

Listing 274 — Create Objects

Output

```
5
5
5
```

Each call to `MyClass()` creates a separate instance. All three objects find the same value of `x` on their class. Sharing that value does not make them the same object.

Aliasing: Two Names for One Object

Two variable names can refer to the same object. This is called **aliasing**. In the following example, `alias = p1` gives the object a second name, `alias`; it does not copy the object. The statement `del p1` then removes the name `p1`.

Continue with the `p1` object created above. The final part of this example deliberately uses `p1` after removing it. The `try` block attempts that access; Python raises `NameError` because the name is no longer defined. The `except NameError` block handles the error and prints a message.

```
alias = p1
del p1
print(alias.x)
try:
    print(p1.x)
except NameError:
    print("NameError: p1 is no longer defined")
```

Listing 275 — Aliasing: Two Names for One Object

Output

```
5
NameError: p1 is no longer defined
```

The first line comes from `print(alias.x)`: the object is still available through `alias`, so its value can still be read. The second line is the message printed by the error handler. Removing one name did not remove the object.

Try it: in the class definition above, change `x = 5` to `x = 8` and run that class definition again. Before rerunning the example that creates three objects, predict the three printed values. Then run it to create three new objects and check your prediction. Explain why changing the class affects all three

results.

We can now create objects from a class and access a class attribute through them. The next section develops this idea with data stored on individual objects and methods that work with that data.

Object Attributes and Methods

The previous section used `x`, a value stored on a class. An **instance attribute** instead stores information for one object. A **method** is a function defined in a class that can work with an object's data [1]. A book, for example, can store its title, author and year, and provide a method that returns a description.

Figure 13 shows the `Book` class using UML notation. The middle compartment lists attributes; the bottom lists operations. A `+` marks a public member, a colon introduces its type, and parentheses distinguish an operation from an attribute. Here, `str` means text and `int` means an integer.

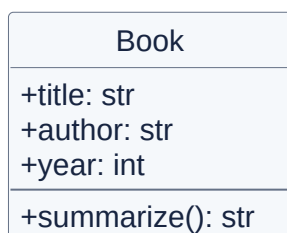


Figure 13 — The `Book` class: three public attributes and its `summarize` operation.

Define the Book

We use `def` to define a method, just as we define a function. Its first parameter, conventionally named `self`, receives the object on which the method is called. Python calls the special method `__init__` after creating an instance; here it stores the supplied starting values. The next section examines this initialization step in detail.

```

class Book:
    def __init__(self, title, author, year):
        self.title = title
        self.author = author
        self.year = year

    def summarize(self):
        return f'{self.title}' by {self.author} ({self.year})"
  
```

Listing 276 — Define the `Book`

The assignments attach `title`, `author` and `year` to the object received as `self`. The `summarize` method reads those attributes and uses an f-string to build a description; expressions inside `{}` supply the values. `return` sends that text back to the caller. Defining this class produces no visible output.

Use Attributes and Methods

Run the class definition above, then create two books. Treat the titles, authors and years as sample inputs. Read an attribute with a dot, as in `b1.title`; add parentheses to call a method, as in `b1.summarize()`.

```

b1 = Book("The Bhagavad Gita", "Vyasa", -400)
b2 = Book("Python Crash Course", "Eric Matthes", 2016)
print(b1.title)
print(b1.summarize())
print(b2.summarize())

```

Listing 277 — Use Attributes and Methods

Output

```

The Bhagavad Gita
'The Bhagavad Gita' by Vyasa (-400)
'Python Crash Course' by Eric Matthes (2016)

```

The first line reads `b1`'s title directly. Each method call then uses the attributes of the object before the dot: `b1` for the first description and `b2` for the second. The method returns text; `print` displays it. This is how one method definition works with different objects without copying its logic.

Example: Temperature Conversion

The same pattern can represent a measurement. `Temperature` stores a Celsius value in `celsius`; its `to_fahrenheit()` method converts that value using $\text{Celsius} \times 9/5 + 32$. In the following UML class diagram, the attribute has no stated type and the operation returns a `float`, a floating-point number.

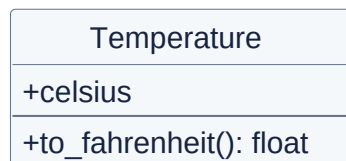


Figure 14 — Temperature stores Celsius and provides an operation returning Fahrenheit.

Predict the two results, then run this complete example:

```

class Temperature:
    def __init__(self, celsius):
        self.celsius = celsius

    def to_fahrenheit(self):
        return self.celsius * 9/5 + 32

temp1 = Temperature(25)
temp2 = Temperature(49)
print(f"{temp1.celsius}°C is {temp1.to_fahrenheit()}°F")
print(f"{temp2.celsius}°C is {temp2.to_fahrenheit()}°F")

```

Listing 278 — Example: Temperature Conversion

Output

```

25°C is 77.0°F
49°C is 120.2°F

```

Each method call reads that object's Celsius value, multiplies it by $9/5$ and adds 32. The surrounding f-string combines the original measurement with the returned result.

Try it: replace 25 with 0 when creating temp1. Predict the Fahrenheit value, run the example and explain the change.

Coding Challenge: Initialize `Movie` Class with `__init__`

Use Book as a model to create a `Movie` class with `title`, `director` and `year` attributes. Create objects with the sample inputs Inception / Christopher Nolan / 2010 and Spirited Away / Hayao Miyazaki / 2001. Print the first movie's title and year, then the second movie's title and director.

Target output for your solution

```
Inception
2010
Spirited Away
Hayao Miyazaki
```

These examples used `__init__` to give each object its starting data. We will now look more closely at when Python calls it and how `self` identifies the object being initialized.

Reference

- Python Software Foundation. [Classes: instance objects and method objects](#).

Initializing Objects with `__init__` and `self`

`__init__` assigns an instance's starting attributes. Python calls it after creating the instance; `self` refers to that instance.

An **activity diagram** is a picture of a workflow: the steps an operation follows. Read this one from top to bottom, following the arrows. The filled circle marks the start, each rounded box names an action, and the bullseye marks completion. Here the workflow creates an instance and then stores its starting attributes. The diagram explains order; the Python definition below supplies the executable statements.

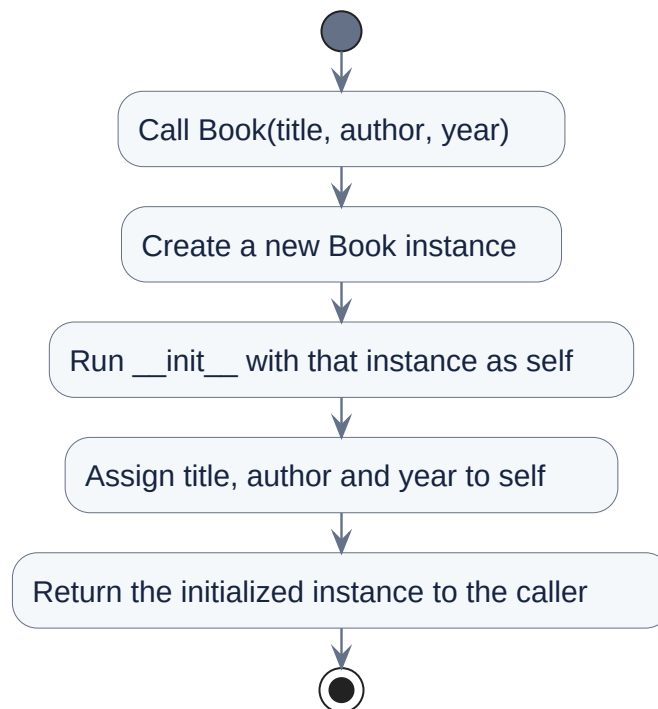


Figure 15 — The initializer stores supplied values on the newly created instance.

Define the Initializer

```
class Book:
    def __init__(self, title, author, year):
        self.title = title
        self.author = author
        self.year = year
```

Listing 279 — Define the Initializer

Each assignment stores an argument on the new book. Defining the class prints nothing.

Run the examples in this section in order.

```
b1 = Book("The Bhagavad Gita", "Vyasa", -400)

print(b1.title)
print(b1.year)
```

Listing 280 — Define the Initializer

Output

```
The Bhagavad Gita
-400
```

Dot access reads the attributes set during initialization.

Why Initialize Attributes Together?

Without an initializer, you must assign attributes manually:

```
class Book:
    pass

b1 = Book()
b1.title = "The Bhagavad Gita"
b1.author = "Vyasa"
b1.year = -400

b2 = Book()
b2.title = "Python Crash Course"
# You may forget to set the author of b2
```

Listing 281 — Why Initialize Attributes Together?

This creates two books but leaves the second without an author. The assignments produce no visible output.

An initializer places the required assignments in one method:

```
class Book:
    def __init__(self, title, author, year):
        self.title = title
        self.author = author
        self.year = year

b1 = Book("The Bhagavad Gita", "Vyasa", -400)
b2 = Book("Python Crash Course", "Eric Matthes", 2016)

print(b1.title)
print(b1.author)
print(b2.title)
print(b2.author)
```

Listing 282 — Why Initialize Attributes Together?

Output

```
The Bhagavad Gita
Vyasa
Python Crash Course
Eric Matthes
```

Both books now receive all three fields. Validation still needs explicit checks; initialization alone does not prove the values are valid.

How `self` Is Supplied

A **sequence diagram** shows which participant calls another and the order of those calls. A participant is an object or part of the program taking part in the interaction. Its dashed vertical line is a **lifeline**: read later messages farther down the page; the spacing does not measure elapsed seconds. Here `Caller` means the code making `Book(...)`. A solid arrow points toward the new `Book` instance, whose box first appears at that creation message. The notes show which instance is passed to `__init__` as `self`.

Each `Book(...)` label abbreviates a call with a title, author and year. The dots mean that the argument details are omitted from the diagram; use the complete calls shown in the code examples. Trace the first call to `b1`: inside its initializer, `self.title = title` stores a value on `b1`. The second call supplies a different instance as `self`, so the same method stores values on `b2`. The caller does not type a `self` argument in an ordinary `Book(...)` call.

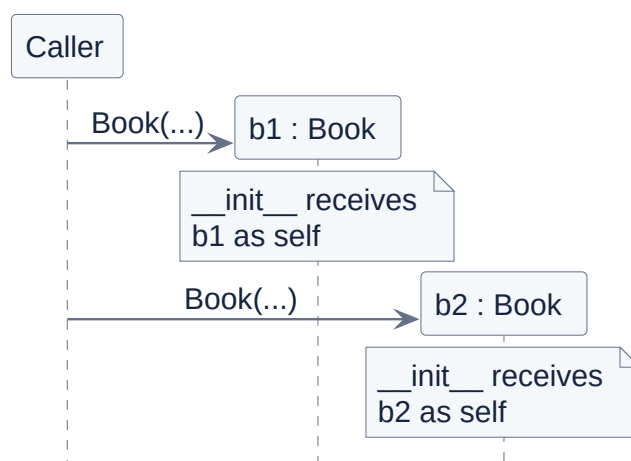


Figure 16 — The same method operates on the instance supplied as `self`.

```
b1 = Book("The Bhagavad Gita", "Vyasa", -400)
```

Listing 283 — How `self` Is Supplied

This ordinary class call creates a book and supplies the instance to its initializer. It prints nothing.

The following explicit call demonstrates argument binding. It reinitializes the existing `b1`; it does not create another object.

```
Book.__init__(b1, "The Bhagavad Gita", "Vyasa", -400)
```

Listing 284 — How `self` Is Supplied

No visible output.

Here you explicitly pass `b1` as the first argument. In an ordinary bound method call, Python supplies that argument.

Try it: add a language parameter and store it on each book. Create two books with different languages and print both values.

With objects initialized, we can distinguish their identity, their stored values and the actions they perform.

Identity, State, and Behavior

An object has an identity, a current state and behavior. Two books may store the same title and still be different objects.

As in the earlier object diagram, underlined labels identify objects; the compartment below each label shows a stored value. No line connects them because equal titles alone do not create a relationship.



Figure 17 — Equal stored titles do not make two Book instances the same object.

Identity

`is` compares identity. `==` uses the type's equality rule; our Book has not defined a custom rule.

```

class Book:
    def __init__(self, title):
        self.title = title

book1 = Book("Hamlet")
book2 = Book("Hamlet")

print(book1 is book2)
print(book1 == book2)
print(book1.title == book2.title)
  
```

Listing 285 — Identity

Output

```

False
False
True
  
```

The books are separate objects, but their title strings are equal. `id()` also returns an identity number; treating it as a memory address is implementation-specific.

State

State is the data held by an object at a particular time. Here, `None` means no courses list was supplied. The conditional expression chooses a new empty list in that case; otherwise `list(courses)` copies the supplied list.

```
class Student:
    def __init__(self, name, courses=None):
        self.name = name
        self.courses = [] if courses is None else list(courses)

ada = Student("Ada")
grace = Student("Grace", ["CS101"])

print(ada.name, ada.courses)
print(grace.name, grace.courses)
```

Listing 286 — State

Output

```
Ada []
Grace ['CS101']
```

Ada has an empty courses list; Grace has CS101. The initializer creates or copies a list, so their collections are independent.

Continue with the students above:

```
ada.courses.append("Math101")
print(ada.courses)
```

Listing 287 — State

Output

```
['Math101']
```

Appending changes Ada's state. Grace's courses are unchanged.

Behavior

A method can update state or return information from it.

```
class Student:
    def __init__(self, name):
        self.name = name
        self.courses = []

    def enroll(self, course):
        self.courses.append(course)

    def full_name(self):
        return self.name

ada = Student("Ada Lovelace")
ada.enroll("CS101")

print(ada.full_name())
print(ada.courses)
```

Listing 288 — Behavior

Output

```
Ada Lovelace
['CS101']
```

`enroll` changes the courses list. `full_name` returns the stored name.

Try it: create another Ada object. Compare the objects with `is`, then compare their names with `==`.

We can now separate values stored on individual objects from defaults supplied by their class.

7.2 Instance & Class Members

Instance and Class Attributes

An **instance attribute** belongs to one object. A **class attribute** is defined on the class and can supply a default to its instances.

The diagram combines a **class view**, showing a definition, with **object views**, showing particular instances. In the Library box, underlining the member `opening_hours` marks a class-level value; `name` is an instance attribute. In an object box, the underlined header `main : Library` names that object and its class; the compartment below lists values stored on that instance. These are two uses of underlining: a class member is shared at class level, whereas an object header identifies a snapshot.

The class-attribute example below first uses `09:00–17:00`, then changes the class default to `08:00–20:00`; this overview shows the latter state. No `opening_hours` slot appears inside the two object boxes because they obtain that value by looking it up on `Library`.

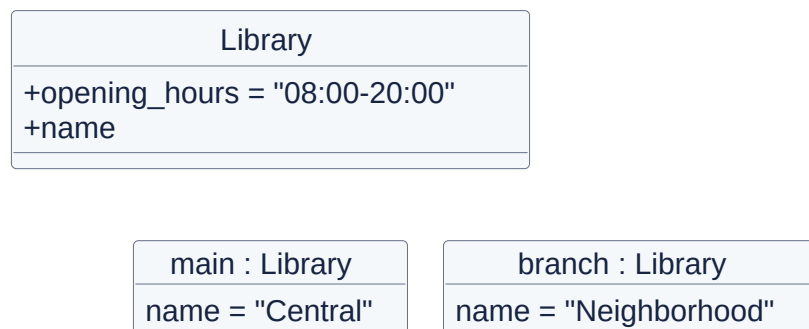


Figure 18 — Instances store their own names and can look up default hours on `Library`.

Instance Attributes

```
class Library:
    def __init__(self, name):
        self.name = name
        self.holdings = []    # instance attribute

central = Library("Central")
branch = Library("Neighborhood")

central.holdings.append("Hamlet")

print("Output:")
print(branch.holdings)
```

Listing 289 — Instance Attributes

Output

```
Output:
[]
```

Each branch receives a separate holdings list. Adding Hamlet to Central does not change the other branch.

Class Attributes

Run these examples in order. The new Library definition adds a class default:

```
class Library:
    opening_hours = "09:00-17:00" # class attribute

    def __init__(self, name):
        self.name = name

main = Library("Central")
branch = Library("Neighborhood")

print("Output: ")
print(main.opening_hours)
print(branch.opening_hours)

Library.opening_hours = "08:00-20:00"
print(main.opening_hours)
print(branch.opening_hours)
```

Listing 290 — Class Attributes

Output

```
Output:
09:00-17:00
09:00-17:00
08:00-20:00
08:00-20:00
```

Both instances initially read the class hours. Changing `Library.opening_hours` changes the default found by unshadowed instances.

Shadowing

Assigning the same name on an instance creates an instance-specific value.

Read the three boxes separately. Library still supplies the class default `08:00-20:00`. The `main` object has no separate hours slot, so lookup reaches that default. The `branch` object now stores `opening_hours = "10:00-18:00"`; its own value is found first. This is **shadowing**: an instance value hides the same-named class value for that instance without changing the class or the other object.

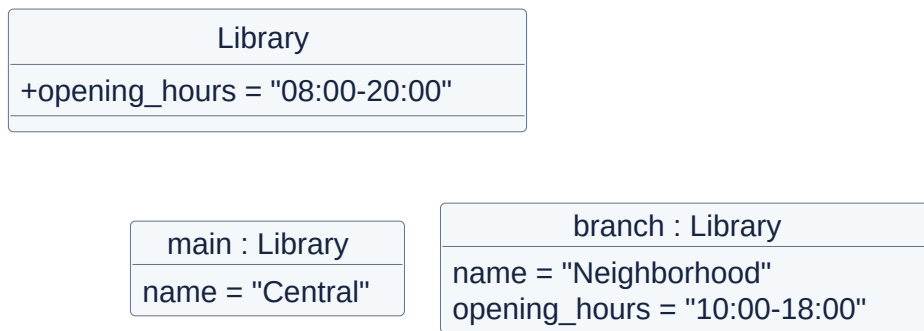


Figure 19 — Main uses the class default; branch has a separate instance value.

```
branch.opening_hours = "10:00-18:00"

print("Output: ")
print(main.opening_hours)
print(branch.opening_hours)
```

Listing 291 — Shadowing

Output

```
Output:
08:00-20:00
10:00-18:00
```

Main still reads the class default; branch reads its own value. Changing the default again would not replace that branch-specific value.

Try it: change the class hours again and print both instances. Explain the different results. Avoid a mutable class list such as `books = []` when every instance should own a separate collection.

The distinction between an instance and its class also guides where methods belong.

Instance and Class Methods

An **instance method** receives the calling object as `self`. A **class method** receives the class as `cls`. Choose according to which data the operation needs.

UML underlines class-level members; ordinary operations belong to instances. Here the underlined operation is implemented with Python's `@classmethod`, rather than `@staticmethod`.

Library
+name +holdings +opening_hours
+add_book(book) +list_books() +set_opening_hours(hours)

Figure 20 — Instance and class methods receive different kinds of receiver.

Instance Methods

```
class Library:
    def __init__(self, name):
        self.name = name
        self.books = [] # unique to each library

    def add_book(self, book): # instance method
        """Add a book to this library."""
        self.books.append(book)

    def list_books(self): # instance method
        """Display all books in this library."""
        print(f"{self.name} Library Collection:")
        for book in self.books:
            print(" -", book)

# Create two libraries
central = Library("Central")
neighborhood = Library("Neighborhood")

# Add books to each branch
central.add_book("Hamlet")
central.add_book("Macbeth")
neighborhood.add_book("Faust")

central.list_books()
neighborhood.list_books()
```

Listing 292 — Instance Methods

Output

```
Central Library Collection:
- Hamlet
- Macbeth
Neighborhood Library Collection:
```

– Faust

`add_book` changes one branch's list. `list_books` prints that branch's heading and books.

Class Methods

A decorator is an `@...` line above a definition that changes how it is used. `@classmethod` binds the class to the first parameter.

The activity diagram shows the call and assignment in order; rounded boxes are actions, with a start circle and an end bullseye.

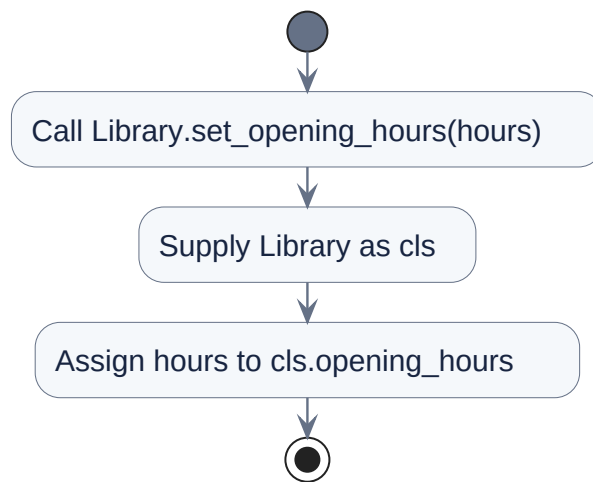


Figure 21 — The decorator supplies the class as `cls` before updating its default.

```

class Library:
    opening_hours = "09:00-17:00" # shared by all libraries

    def __init__(self, name):
        self.name = name
        self.books = []

    @classmethod
    def set_opening_hours(cls, hours): # class method
        """Update opening hours for all libraries."""
        cls.opening_hours = hours
  
```

Listing 293 — Class Methods

This defines a class method; the definition itself prints nothing.

Continue with that `Library` definition:

```

central = Library("Central")
neighborhood = Library("Neighborhood")

print(central.opening_hours)
print(neighborhood.opening_hours)

Library.set_opening_hours("08:00-20:00")

print(central.opening_hours)
print(neighborhood.opening_hours)

```

Listing 294 — Class Methods

Output

```

09:00-17:00
09:00-17:00
08:00-20:00
08:00-20:00

```

The class method changes the default hours used by both unshadowed instances.

Use Both When Needed

```

class Library:
    opening_hours = "09:00-17:00"

    def __init__(self, name):
        self.name = name
        self.books = []

    def add_book(self, book):
        self.books.append(book)

    def list_books(self):
        print(f"{self.name} Library Collection:")
        for book in self.books:
            print(" -", book)

    @classmethod
    def set_opening_hours(cls, hours):
        cls.opening_hours = hours

# Example usage
central = Library("Central")
branch = Library("Neighborhood")

central.add_book("Hamlet")
branch.add_book("Faust")

central.list_books()
branch.list_books()

Library.set_opening_hours("08:00-20:00")
print("Central hours:", central.opening_hours)
print("Branch hours:", branch.opening_hours)

```

Listing 295 — Use Both When Needed

Output

```

Central Library Collection:
- Hamlet

```

```
Neighborhood Library Collection:  
- Faust  
Central hours: 08:00–20:00  
Branch hours: 08:00–20:00
```

The instance methods manage separate collections; the class method changes their shared default. A class does not need both kinds of method unless its responsibilities require them.

Try it: add a book to Central and confirm that the branch's list stays unchanged.

Apply these two kinds of method in the library challenge, combining separate collections with a shared total.

Coding Challenge: Tracking Books Across All Libraries

Combine the branch-specific book lists with a class-wide counter.

This UML snapshot shows the target state: underlining marks the class-wide counter, while each labeled object stores its own collection.

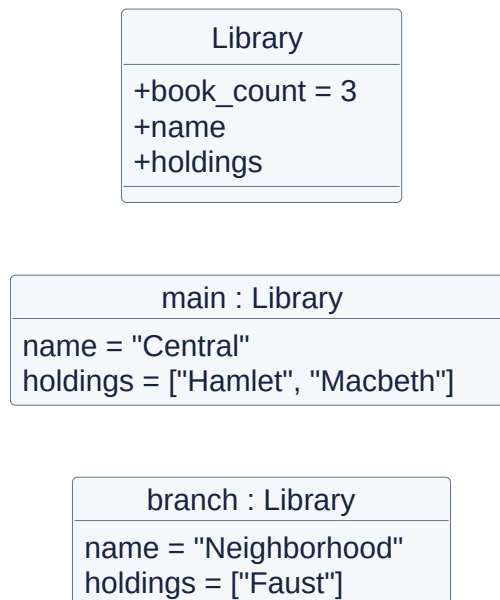


Figure 22 — The exercise combines separate branch collections with a class-wide total.

The Task

Extend the Library from Instance and Class Methods:

- Store the count in a class attribute named `book_count`.
- Update it on every successful `add_book` call. Duplicate titles represent separate copies.
- Return it from a class method named `total_books()`.
- Start from zero on every fresh run. Removal and persistence are outside this exercise.

Add Hamlet and Macbeth to Central, and Faust to Neighborhood. Print each collection and the total.

Bonus: keep a registry of branches and implement `library_summary()`. Show each branch's count and percentage, formatted to one decimal place. An empty system must not divide by zero.

Target output for your solution

```

Central Library Collection:
- Hamlet
- Macbeth
Neighborhood Library Collection:
- Faust

Total books in system: 3
Central: 2 books (66.7% of total)
  
```

Neighborhood: 1 book (33.3% of total)

Test an empty system, one branch and the two-branch example. Explain why the counter and the method need different names.

Managing a shared total also raises a design question: which operations should be allowed to change it? Encapsulation addresses that question.

7.3 Object Relationships

Encapsulation

Encapsulation keeps data and its operations together and provides a clear public interface for using them. Python uses naming conventions and managed attributes rather than enforced public/protected/private access levels.

In this UML activity diagram, rounded boxes are actions and the diamond checks a condition. The accepted path changes state; the rejected path does not.

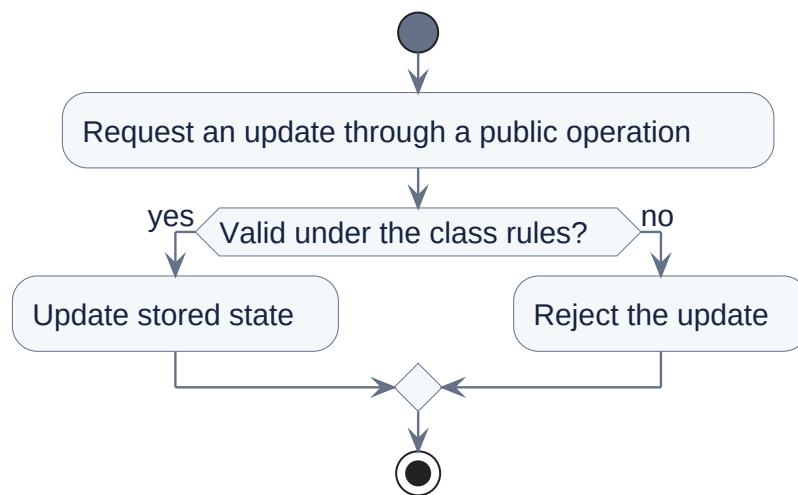


Figure 23 — Encapsulation routes controlled changes through a class’s public operations.

Public Attributes

```

class User:
    def __init__(self, name):
        self.name = name      # public attribute

u = User("Anna")
print(u.name)
  
```

Listing 296 — Public Attributes

Output

Anna

A public attribute can be read directly with dot notation.

Internal Attributes: `_name`

A single underscore means “internal use” by convention. It does not block access.

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self._salary = salary    # protected
```

Listing 297 — Internal Attributes: `__name`

The class definition creates no `Employee` objects yet.

Continue with that definition:

```
e = Employee("Mila", 50000)
print(e._salary)    # allowed, but discouraged
```

Listing 298 — Internal Attributes: `__name`

Output

```
50000
```

The lookup is allowed, but ordinary client code should use the public interface.

Name-Mangled Attributes: `__name`

Two leading underscores, without two trailing underscores, trigger name mangling. This reduces accidental name clashes in subclasses; it is not a security boundary.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.__age = age    # private

p = Person("Sam", 30)
# print(p.__age)    # Error
```

Listing 299 — Name-Mangled Attributes: `__name`

The commented lookup would raise `AttributeError`. As written, the example creates `p` and prints nothing.

Getter and Setter Methods

A getter reads a value. A setter checks an update before storing it.

In the UML class view, `-` marks the intended private implementation and `+` marks public operations. Python's double underscore uses name mangling; it does not enforce access security.

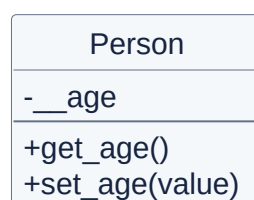


Figure 24 — A getter reads age; a setter checks an update before storing it.

```
class Person:
    def __init__(self, age):
        self.__age = age

    def get_age(self):
        return self.__age

    def set_age(self, value):
        if value > 0:
            self.__age = value
        else:
            print("Age must be positive")
```

Listing 300 — Getter and Setter Methods

Defines Person; no visible output.

This initializer assumes a valid starting age; the setter checks later updates.

Continue with that Person definition:

```
p = Person(25)
print("Current age:", p.get_age())

p.set_age(30)
print("Updated age:", p.get_age())

p.set_age(-5)      # invalid update
print("Final age:", p.get_age())
```

Listing 301 — Getter and Setter Methods

Output

```
Current age: 25
Updated age: 30
Age must be positive
Final age: 30
```

The invalid update is rejected, so the final age remains 30.

Managed Attributes with `property`

`@property` makes a getter callable through attribute syntax. `@celsius.setter` handles assignment to that attribute.

The UML slash marks a value obtained through an attribute interface: `/celsius` reads the stored `__celsius` value. The private marker describes the design intent; the Python property controls reads and assignments.

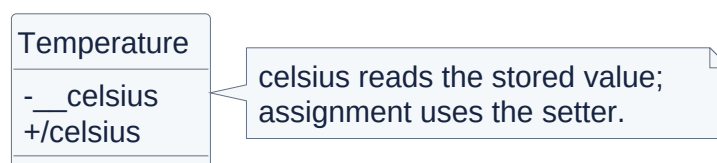


Figure 25 — Property access uses attribute syntax while calling getter and setter code.

```

class Temperature:
    def __init__(self, celsius):
        if celsius < -273.15:
            raise ValueError("Temperature below absolute zero")
        self.__celsius = celsius

    @property
    def celsius(self):
        return self.__celsius

    @celsius.setter
    def celsius(self, value):
        if value >= -273.15:
            self.__celsius = value
        else:
            print("Temperature below absolute zero is not allowed.")

```

Listing 302 — Managed Attributes with property

Defines `Temperature`; no visible output.

The initializer rejects an impossible starting temperature. The setter reports invalid later updates without changing the stored value.

```

t = Temperature(20)
print(t.celsius)

t.celsius = -300    # validation kicks in

```

Listing 303 — Managed Attributes with property

Output

```

20
Temperature below absolute zero is not allowed.

```

Reading `t.celsius` calls the getter; assigning `-300` calls the setter and prints its rejection.

Internal Helper Methods

Name mangling also applies to methods:

```

class Calculator:
    def __init__(self):
        self.result = 0

    def __validate(self, n):
        return isinstance(n, (int, float))

    def add(self, n):
        if self.__validate(n):    # internal use only
            self.result += n

```

Listing 304 — Internal Helper Methods

Defines `Calculator`; no visible output.

`add` calls an internal helper. `isinstance` checks whether a value has one of the specified types; this small check also accepts Boolean values because `bool` is a subclass of `int`.

The mangled spelling remains accessible:

```
class Person:
    def __init__(self, age):
        self.__age = age

p = Person(30)
print(p._Person__age)  # technically possible, but discouraged
```

Listing 305 — Internal Helper Methods

Output

30

This deliberately demonstrates outside access. Do not treat mangling as secrecy or validation.

Coding Challenge 1: Encapsulation – BankAccount

Complete this class with methods to read the balance, deposit and withdraw. Reject a negative initial balance, non-positive amounts and withdrawals exceeding the balance. Rejected operations must leave the balance unchanged.

Exercise starter: complete before running.

```
class BankAccount:
    def __init__(self, initial_balance=0):
        self.__balance = initial_balance  # private

    # add methods here
```

Listing 306 — Coding Challenge 1: Encapsulation – BankAccount

Start at 1000, deposit 300, withdraw 200, then attempt to withdraw 2000.

Target output for your solution

```
Current balance: 1000
After deposit: 1300
After withdrawal: 1100
Cannot withdraw 2000: insufficient funds
Final balance: 1100
```

These are teaching amounts; production currency code needs explicit rounding rules.

SavingsAccount Follow-up

Complete the [SavingsAccount exercise in Inheritance](#), after learning how one class extends another.

Once an object has clear public operations, other objects can use them to work together. This leads to composition.

Composition: “Has-a” Relationships

Composition builds an object from other objects. A Course **has a** Schedule and delegates meeting-related work to it.

The UML association arrow points from Course to its Schedule. The 1 means each course refers to one schedule; 0..* allows a schedule to be shared by several courses, which this constructor permits. A filled composition diamond would additionally require exclusive ownership.

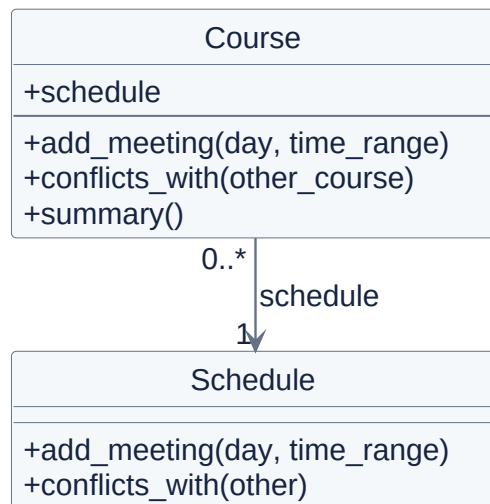


Figure 26 — A Course uses a Schedule object instead of being a Schedule.

Define the Schedule

`any(...)` checks whether at least one generated condition is true. `__str__` supplies the display text used when a schedule is formatted.

```

class Schedule:
    """Holds weekly meeting times like ('Mon', '09:00-10:30')."""
    def __init__(self):
        self._slots = [] # list of (day, time_range) tuples

    def add_meeting(self, day, time_range):
        entry = (day, time_range)
        if entry not in self._slots:
            self._slots.append(entry)

    def conflicts_with(self, other):
        return any(slot in other._slots for slot in self._slots)

    def __str__(self):
        return "no meetings scheduled" if not self._slots else ", ".join(f"{d} {t}" for d, t in
self._slots)
  
```

Listing 307 — Define the Schedule

Defines Schedule; no visible output.

`add_meeting` stores a unique (day, time_range) tuple; `conflicts_with` tests those exact pairs.

Give Each Course a Schedule

Continue with the Schedule definition:

```
class Course:
    """A course that has a schedule."""
    def __init__(self, code, title, credits, schedule=None):
        self.code = code
        self.title = title
        self.credits = int(credits)
        self.schedule = schedule or Schedule() # composition

    def add_meeting(self, day, time_range):
        self.schedule.add_meeting(day, time_range)

    def conflicts_with(self, other_course):
        return self.schedule.conflicts_with(other_course.schedule)

    def summary(self):
        return f"{self.code} – {self.title} ({self.credits} cr) [{self.schedule}]"
```

Listing 308 — Give Each Course a Schedule

Defines Course; no visible output.

The course creates a Schedule when none is supplied and forwards calls to it.

Use the Collaborating Objects

```
cs101 = Course("CS101", "Intro to CS", 6)
cs101.add_meeting("Mon", "09:00-10:30")
cs101.add_meeting("Wed", "09:00-10:30")

dh200 = Course("DH200", "Methods in DH", 4)
dh200.add_meeting("Wed", "09:00-10:30") # overlaps with CS101

his110 = Course("HIS110", "World History", 3)
his110.add_meeting("Tue", "14:00-15:30")

print(cs101.summary())
print(dh200.summary())
print(his110.summary())

print("CS101 vs DH200 conflict?", cs101.conflicts_with(dh200))
print("CS101 vs HIS110 conflict?", cs101.conflicts_with(his110))
```

Listing 309 — Use the Collaborating Objects

Output

```
CS101 – Intro to CS (6 cr) [Mon 09:00-10:30, Wed 09:00-10:30]
DH200 – Methods in DH (4 cr) [Wed 09:00-10:30]
HIS110 – World History (3 cr) [Tue 14:00-15:30]
CS101 vs DH200 conflict? True
CS101 vs HIS110 conflict? False
```

CS101 and DH200 contain the same Wednesday slot; HIS110 shares none. This checks identical slots, not overlapping time intervals such as 09:00–10:30 and 10:00–11:00.

Try it: add the same meeting twice and print the summary. Then add a different slot and explain the result.

A reference to another object does not by itself enforce exclusive ownership. The UML sections will distinguish association from composite ownership. Composition combines objects; inheritance, introduced next, defines a specialized type from an existing class.

Inheritance

Inheritance defines a specialized class from an existing one. The **base class** supplies reusable behavior; a **subclass** can add attributes or override methods.

In UML, a solid line with a hollow triangle points from a subclass toward its base class. The compartments list attributes and operations; + marks public members.

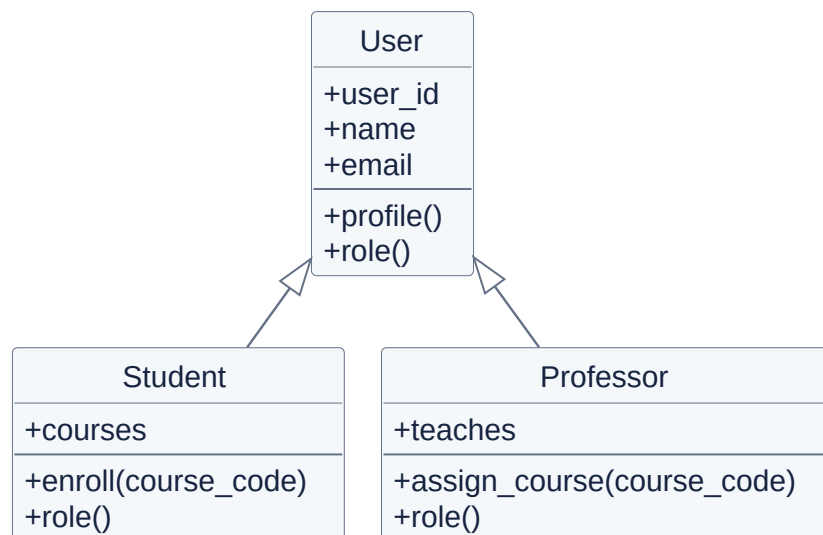


Figure 27 — The subclasses reuse User behavior and add role-specific data and methods.

Define the Base Class

```

class User:
    def __init__(self, user_id, name, email):
        self.user_id = str(user_id)
        self.name = name
        self.email = email

    def profile(self):
        return f"{self.name} <{self.email}>"

    def role(self):
        return "user"
  
```

Listing 310 — Define the Base Class

Defines `User`; no visible output.

`User` stores the common fields and provides `profile` and `role`.

Add a Student Subclass

`Student(User)` names the base class. `super()` finds the next implementation in Python's method resolution order; in this hierarchy, `super().__init__` reaches `User`'s initializer.

```
class Student(User):
    def __init__(self, user_id, name, email):
        super().__init__(user_id, name, email)
        self.courses = []

    def role(self):
        return "student"

    def enroll(self, course_code):
        if course_code not in self.courses:
            self.courses.append(course_code)
```

Listing 311 — Add a Student Subclass

Defines Student; no visible output.

The same student object receives the common fields and its own courses list. Overriding `role` gives it a more specific label.

Add a Professor Subclass

```
class Professor(User):
    def __init__(self, user_id, name, email):
        super().__init__(user_id, name, email)
        self.teaches = []

    def role(self):
        return "professor"

    def assign_course(self, course_code):
        if course_code not in self.teaches:
            self.teaches.append(course_code)
```

Listing 312 — Add a Professor Subclass

Defines Professor; no visible output.

Professor reuses User's initialization and adds its own teaching list.

Use Inherited Classes

Run the definitions above, then:

```
s1 = Student(101, "Anna", "anna@example.com")
p1 = Professor(44, "Dr. Singh", "singh@example.com")

print(s1.profile())
print(p1.profile())

print(s1.role())
print(p1.role())
```

Listing 313 — Use Inherited Classes

Output

```
Anna <anna@example.com>
Dr. Singh <singh@example.com>
student
professor
```

Both objects use the inherited `profile` method, but their `role` implementations return different labels.

Reuse an Internal Attribute

The underscore convention was introduced under encapsulation. It also applies when a subclass works with a base class's internal data.

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self._salary = salary    # protected

class Manager(Employee):
    def give_raise(self, amount):
        self._salary += amount    # allowed: subclass can access protected members
```

Listing 314 — Reuse an Internal Attribute

Defines `Employee`, `Manager`; no visible output.

`Manager` adds behavior to `Employee`; it does not introduce enforced access protection.

```
m = Manager("Alex", 60000)
m.give_raise(5000)
print(m._salary)    # still accessible, but client code should use a public operation
```

Listing 315 — Reuse an Internal Attribute

Output

```
65000
```

The inherited initializer sets 60000; `give_raise(5000)` changes it to 65000.

Coding Challenge: SavingsAccount

After completing the `BankAccount` exercise in Encapsulation and running your completed class definition, create a subclass storing an interest rate. Use the inherited balance query and deposit method to add interest; do not bypass them to change the balance.

Exercise starter: complete before running.

```
class SavingsAccount(BankAccount):
    def __init__(self, initial_balance=0, interest_rate=0.01):
        # call parent constructor and store a private interest rate
        pass

    # add a method to apply interest
```

Listing 316 — Coding Challenge: SavingsAccount

For the target below, start at 1000, pass `interest_rate=0.05`, deposit 500 and apply interest once. The starter default of 0.01 would give a different result.

Target output for your solution

```
Starting balance: 1000  
After deposit: 1500  
After interest: 1575.0
```

Explain which methods are inherited and which responsibility the subclass adds.

We can now compare inheritance with composition and decide when a relationship means “is a” or “has a.”

“Is-a” and “Has-a” Relationships

Inheritance describes **is-a**: a Student is a kind of User. Object collaboration describes **has-a**: a Course uses a Schedule and has enrollment links to students.

Read the symbols introduced here:

- A hollow triangle points to the more general class.
- A plain line shows an association.
- A filled diamond shows exclusive composite ownership; it is a design constraint, not something Python references enforce automatically.
- 1 means exactly one; 0..* means zero or more.

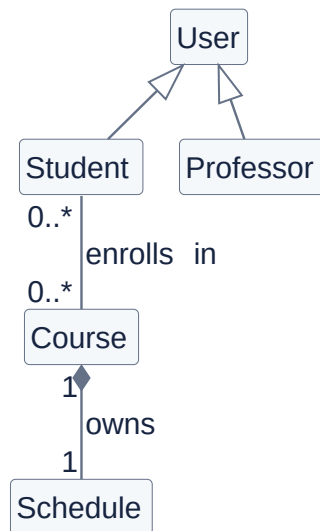


Figure 28 — Inheritance, association and the stated Schedule ownership constraint.

Students and courses have independent lifetimes, so their connection is an association. The Schedule diamond expresses an exclusive-ownership design; the earlier code permits a supplied Schedule, so you would need to enforce that constraint or use a plain association instead.

Apply it: should a Library inherit from Book, or hold Book objects? Justify the choice using the operations each type should support.

These relationships describe structure. Polymorphism shows how different objects can respond to the same operation.

Polymorphism

Polymorphism lets a caller use the same operation with different kinds of object. Each object supplies the behavior needed for that operation.

The UML class boxes show three separate classes with the same public operation. No inheritance line is drawn: these classes need not share a base to support `status()`.

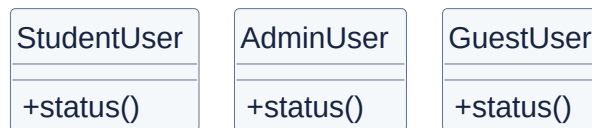


Figure 29 — One operation name reaches the implementation supplied by each object.

Built-in Example

```

print(len("Hello"))           # counts characters
print(len(["a", "b", "c"]))  # counts list items
print(len({"a": 1, "b": 2}))  # counts dictionary pairs
  
```

Listing 317 — Built-in Example

Output

```

5
3
2
  
```

`len` counts characters, list elements or dictionary entries according to the object.

Unrelated Classes

```

class StudentUser:
    def status(self):
        print("Attends classes.")

class AdminUser:
    def status(self):
        print("Manages the system.")

class GuestUser:
    def status(self):
        print("Limited access.")
  
```

Listing 318 — Unrelated Classes

Defines `StudentUser`, `AdminUser`, `GuestUser`; no visible output.

All three classes provide `status`. They need no common base class for this use.

```

users = [StudentUser(), AdminUser(), GuestUser()]

for u in users:
    u.status()
  
```

Listing 319 — Unrelated Classes

Output

```
Attends classes.  
Manages the system.  
Limited access.
```

The loop calls the same method; each object prints its own response.

Overridden Methods

Run these definitions in order:

```
class User:  
    def __init__(self, user_id, name, email):  
        self.user_id = str(user_id)  
        self.name = name  
        self.email = email  
  
    def role(self):  
        return "user"
```

Listing 320 — Overridden Methods

Defines User; no visible output.

```
class Student(User):  
    def __init__(self, user_id, name, email):  
        super().__init__(user_id, name, email)  
    def role(self):  
        return "student"
```

Listing 321 — Overridden Methods

Defines Student; no visible output.

```
class Professor(User):  
    def __init__(self, user_id, name, email):  
        super().__init__(user_id, name, email)  
        self.teaches = []  
  
    def role(self):  
        return "professor"
```

Listing 322 — Overridden Methods

Defines Professor; no visible output.

```
u1 = Student(101, "Anna", "anna@example.com")  
u2 = Professor(44, "Dr. Singh", "singh@example.com")  
  
for u in (u1, u2):  
    print(u.role())
```

Listing 323 — Overridden Methods

Output

```
student  
professor
```

The caller uses `role()` without checking whether the object is a student or professor.

Swappable Behavior with Policies

A policy is an object to which another object delegates a decision. First run the `Schedule` and `Course` definitions from `Composition`. The following classes reuse `Course` and provide two compatible `format(course)` operations.

The hollow triangle marks inheritance from `Course`; dashed arrows show possible dependencies on formatting policies. The two policy classes provide the same operation without inheriting from one another.

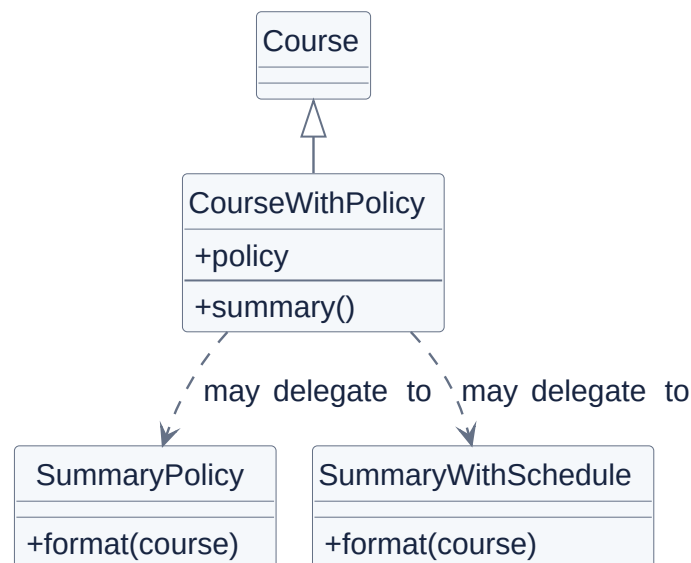


Figure 30 — A course changes its summary behavior by using a different formatting policy.

```

class SummaryPolicy:
    def format(self, course):
        return f"{course.code} - {course.title} ({course.credits} cr)"

class SummaryWithSchedule:
    def format(self, course):
        return f"{course.code} - {course.title} ({course.credits} cr) [{course.schedule}]"

class CourseWithPolicy(Course):
    def __init__(self, code, title, credits, schedule=None, policy=None):
        super().__init__(code, title, credits, schedule)
        self.policy = policy or SummaryPolicy() # composed behavior

    def summary(self):
        return self.policy.format(self)
  
```

Listing 324 — Swappable Behavior with Policies

Defines `SummaryPolicy`, `SummaryWithSchedule`, `CourseWithPolicy`; no visible output.

`CourseWithPolicy` inherits `Course`'s data handling but delegates summary formatting to its policy.

```

plain = CourseWithPolicy("NLP220", "Natural Language Processing", 6)
fancy = CourseWithPolicy("NLP220", "Natural Language Processing", 6,
                        policy=SummaryWithSchedule())

plain.add_meeting("Thu", "10:00-11:30")
fancy.add_meeting("Thu", "10:00-11:30")

print(plain.summary())
print(fancy.summary())

```

Listing 325 — Swappable Behavior with Policies

Output

```

NLP220 – Natural Language Processing (6 cr)
NLP220 – Natural Language Processing (6 cr) [Thu 10:00-11:30]

```

The plain policy omits meeting information; the other includes it. Matching names alone are insufficient: both policies must also accept the expected arguments and return suitable text.

Coding Challenge 1 - Class Polymorphism

Complete three handlers: print to the console, append to `self.messages`, and simulate remote delivery by printing. In `process_logs`, loop over handlers first, then messages. Call `handle(message)` without type checks.

Exercise starter: complete before running.

```

# Coding Challenge 1: Class Polymorphism (Strategy-style Logging)
class ConsoleHandler:
    def handle(self, message):
        # TODO0: print message to console
        pass

class MemoryHandler:
    def __init__(self):
        # TODO0: initialize self.messages to an empty list
        pass

    def handle(self, message):
        # TODO0: append to internal list
        pass

class RemoteHandler:
    def handle(self, message):
        # TODO0: "send" message to a remote server (simulate with print)
        pass

def process_logs(handlers, messages):
    # TODO0: outer loop over handlers, inner loop over messages; call handle(message)
    pass

```

Listing 326 — Coding Challenge 1 - Class Polymorphism

Exercise starter: complete before running.

```
handlers = [ConsoleHandler(), MemoryHandler(), RemoteHandler()]
messages = ["User logged in", "Error: invalid password"]

process_logs(handlers, messages)
print("Stored messages:", handlers[1].messages)
```

Listing 327 — Coding Challenge 1 - Class Polymorphism

Target output for your completed solution

```
[Console] User logged in
[Console] Error: invalid password
[Remote] Sending: User logged in
[Remote] Sending: Error: invalid password
Stored messages: ['User logged in', 'Error: invalid password']
```

Coding Challenge 2 – Polymorphism with Inheritance (Shapes)

Implement `area()` for `Circle`, `Rectangle` and `Triangle`. Require non-negative dimensions. Make `total_area` sum the returned areas without inspecting concrete types.

Exercise starter: complete before running.

```
# Coding Challenge 2: Polymorphism with Inheritance (Shapes)
import math

class Shape:
    def area(self):
        # TODO: subclass should override
        raise NotImplementedError("Subclasses must implement area()")

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    # TODO: implement area()

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    # TODO: implement area()

class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    # TODO: implement area()

def total_area(shapes):
    # TODO: sum up areas of all shapes in the list
    pass
```

```
# demo
shapes = [
    Circle(3), Rectangle(6, 5), Triangle(4, 8)
]

print("Total area:", round(total_area(shapes), 2))
```

Listing 328 — Coding Challenge 2 – Polymorphism with Inheritance (Shapes)

Target output for your completed solution

```
Total area: 74.27
```

The fixture has areas $9 * \text{math.pi}$, 30 and 16. `NotImplementedError` reports a missing operation when called; it does not make `Shape` an abstract base class. Interfaces and behavioral contracts, discussed in the next chapter, make these expectations explicit.

7.4 Interfaces & Abstraction

Interfaces and Behavioral Contracts

An interface states what an object must provide: operations, accepted inputs, results and failure behavior. Callers can depend on that contract rather than one concrete class.

The UML model shows this implementation as an abstract base class, with an italic required operation. Solid hollow-triangle arrows represent the subclasses' inheritance from it.

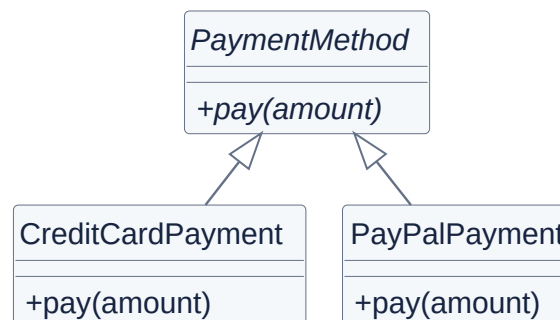


Figure 31 — Compatible objects provide the operation and behavior expected by the caller.

An Explicit Contract

Python's `abc` module provides ABC and the `@abstractmethod` decorator. A subclass must override unresolved abstract operations before it can be instantiated. The next section develops this mechanism.

```

from abc import ABC, abstractmethod

class PaymentMethod(ABC):
    @abstractmethod
    def pay(self, amount):
        pass
  
```

Listing 329 — An Explicit Contract

Defines `PaymentMethod`; no visible output.

```

class CreditCardPayment(PaymentMethod):
    def pay(self, amount):
        print(f"Charging ${amount} to credit card.")

class PayPalPayment(PaymentMethod):
    def pay(self, amount):
        print(f"Processing PayPal payment of ${amount}.")
  
```

Listing 330 — An Explicit Contract

Defines `CreditCardPayment`, `PayPalPayment`; no visible output.

The implementations simulate payments by printing; neither contacts a payment service.

```

methods = [CreditCardPayment(), PayPalPayment()]

for m in methods:
    m.pay(49.99)

```

Listing 331 — An Explicit Contract

Output

```

Charging $49.99 to credit card.
Processing PayPal payment of $49.99.

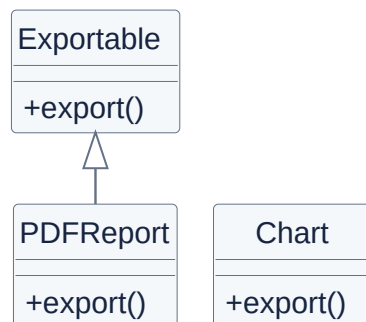
```

Both classes accept `pay(amount)`. An abstract declaration does not prove that their signatures or business rules are correct.

Duck Typing

Duck-typed code calls the required operation without first checking a concrete type. A missing method or incompatible signature fails at runtime.

The UML model shows the inheritance actually present in the code: `PDFReport` extends `Exportable`; `Chart` is independent. Both supply the operation the caller needs.

Figure 32 — The caller uses `export` without requiring both objects to inherit from one base.

```

class Exportable:
    def export(self):
        pass # expected to be overridden

```

Listing 332 — Duck Typing

Defines `Exportable`; no visible output.

This placeholder does not enforce an abstract contract.

```

class PDFReport(Exportable):
    def export(self):
        print("Exporting report as PDF...")

class Chart:
    def export(self):
        print("Exporting chart image...")

```

Listing 333 — Duck Typing

Defines `PDFReport`, `Chart`; no visible output.

```
items = [PDFReport(), Chart()]

for item in items:
    item.export()    # works as long as the method exists
```

Listing 334 — Duck Typing

Output

```
Exporting report as PDF...
Exporting chart image...
```

These export methods print messages only; they create no PDF or image files.

Try it: rename Chart’s export method. Predict which call fails, then restore the shared operation.

We will now examine abstract classes more closely to see how Python can require subclasses to provide specific operations.

Abstract Classes

An abstract base class can combine required operations with reusable behavior. A class with unresolved abstract members cannot be instantiated.

UML italicizes the abstract class and required operation. The hollow triangle shows that `Dog` inherits from `Animal` and supplies `sound()`.

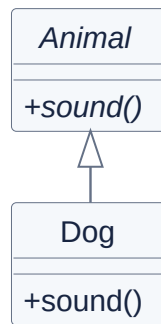


Figure 33 — Implementing the required operation makes `Dog` concrete.

Require an Operation

```

from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass
  
```

Listing 335 — Require an Operation

Defines `Animal`; no visible output.

```

class Dog(Animal):
    def sound(self):
        return "Bark"

dog = Dog()
print(dog.sound())
  
```

Listing 336 — Require an Operation

Output

```
Bark
```

`Dog` supplies `sound`, so it is concrete and can be instantiated.

Attempt Abstract Instantiation

```

from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

try:
    animal = Animal()
except TypeError:
    print("TypeError: Animal still has an abstract method")

```

Listing 337 — Attempt Abstract Instantiation

Output

```
TypeError: Animal still has an abstract method
```

The intentional `TypeError` is caught. No `Animal` instance is created. An abstract method may contain implementation code and still require an override.

Share Concrete Behavior

```

from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

    def move(self):
        return "The animal moves."

```

Listing 338 — Share Concrete Behavior

Defines `Animal`; no visible output.

```

class Dog(Animal):
    def make_sound(self):
        return "Bark"

dog = Dog()
print(dog.move())          # inherited concrete method
print(dog.make_sound())    # Dog's implementation

```

Listing 339 — Share Concrete Behavior

Output

```
The animal moves.
Bark
```

Dog inherits `move` and implements `make_sound`.

Require a Property

Place `@property` above `@abstractmethod`, so the abstract marker applies to the getter.

The italic class name marks the abstract base; */species* is a derived, read-only UML attribute. In Python the abstract requirement is implemented by a property getter, supplied by `Dog`.

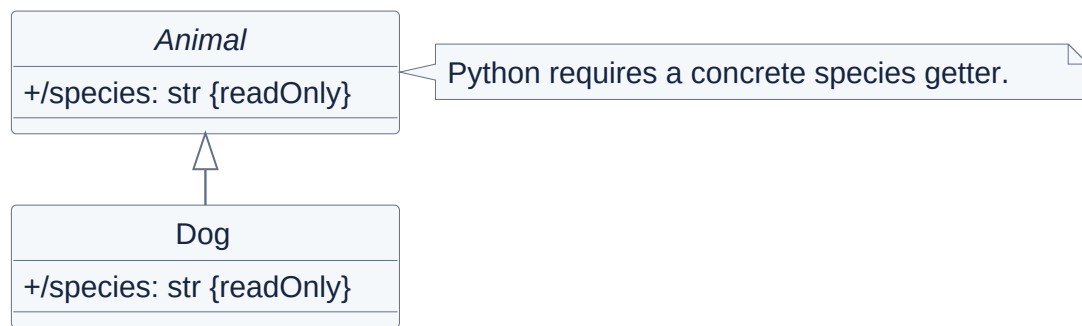


Figure 34 — An abstract property declares an attribute-style operation that a concrete class supplies.

```

from abc import ABC, abstractmethod

class Animal(ABC):
    @property
    @abstractmethod
    def species(self):
        pass
  
```

Listing 340 — Require a Property

Defines `Animal`; no visible output.

```

class Dog(Animal):
    @property
    def species(self):
        return "Canine"

dog = Dog()
print(dog.species)
  
```

Listing 341 — Require a Property

Output

```
Canine
```

The property is read as `dog.species`, without parentheses.

Check Whether a Class Is Concrete

```

from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

try:
    animal = Animal()
except TypeError:
    print("TypeError: Animal still has an abstract method")
  
```

Listing 342 — Check Whether a Class Is Concrete

Output

```
TypeError: Animal still has an abstract method
```

An ABC with no unresolved abstract members can be instantiated. Abstract checks enforce required names, not the correctness of their behavior.

Coding Challenge: Abstract Class for a Notification System

Define `NotificationChannel` with a required `send(target, message)` method, a required read-only `name` property and a shared logging method. Implement `Email` and `SMS` subclasses. Each `send` must print its simulated action, then call the shared logger. Send the same message through both.

Target output for your solution

```
[Email] Sending to user@example.com: System update at 18:00
[LOG] Email delivered: System update at 18:00

[SMS] Sending to +4912345678: System update at 18:00
[LOG] SMS delivered: System update at 18:00
```

These are console simulations, not actual email or SMS delivery. Confirm that the abstract base cannot be instantiated.

The examples so far use one base class. Multiple inheritance introduces a class with more than one base and explains how Python resolves its methods.

Multiple Inheritance

A class can inherit from more than one base. Python uses a **method resolution order (MRO)** to decide which implementation a lookup finds.

The UML hollow triangles show the two base classes. Python's lookup order is stated in the note; it is not encoded by the generalization lines themselves.

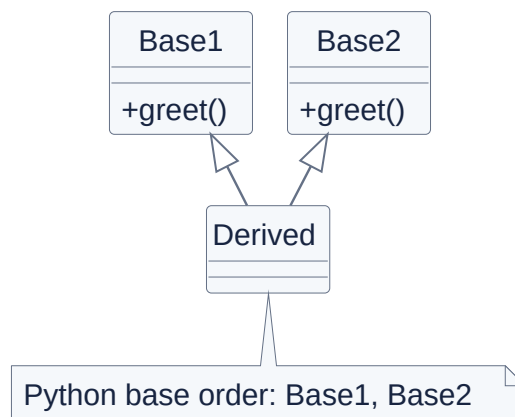


Figure 35 — For `Derived(Base1, Base2)`, `greet` is found on `Base1` first.

```

class Base1:
    def greet(self):
        return "Hello from Base1"

class Base2:
    def greet(self):
        return "Hello from Base2"

class Derived(Base1, Base2):
    pass

d = Derived()
print(d.greet())
  
```

Listing 343 — Multiple Inheritance

Output

```
Hello from Base1
```

`Base1` precedes `Base2`, so its `greet` wins. Python calculates a C3 ordering, not a simple depth-first traversal; inspect `Derived.__mro__` when needed.

Example: Teaching Assistant

The teaching assistant combines Student behavior with Employee information. Read the definitions, then run the example below.

The UML generalization triangles point to base classes. In this implementation, `Student` inherits from `User`, while `Employee` is a separate base.

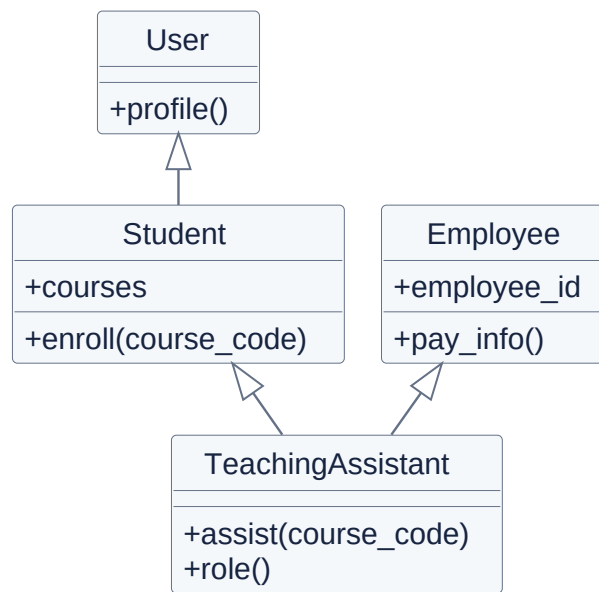


Figure 36 — The teaching assistant combines Student and Employee roles in this specific hierarchy.

```

class User:
    def __init__(self, user_id, name, email):
        self.user_id = str(user_id)
        self.name = name
        self.email = email

    def profile(self):
        return f"{self.name} <{self.email}>"

    def role(self):
        return "user"    # default label

    def describe(self):
        return f"{self.role().title()}: {self.profile()}"

class Student(User):
    def __init__(self, user_id, name, email):
        super().__init__(user_id, name, email)
        self.courses = []    # student-specific attribute

    def role(self):
        return "student"    # override the parent's version

    def enroll(self, course_code):
        if course_code not in self.courses:
            self.courses.append(course_code)

    def describe(self):
        base = super().describe()    # reuse User's describe() output
        return f"{base} – enrolled in {len(self.courses)} course(s)"

class Professor(User):
    def __init__(self, user_id, name, email):
        super().__init__(user_id, name, email)
        self.teaches = []
  
```

```

def role(self):
    return "professor"    # override parent's version

def assign_course(self, course_code):
    if course_code not in self.teaches:
        self.teaches.append(course_code)

def describe(self):
    base = super().describe()
    return f"{base} – teaches {len(self.teaches)} course(s)"

class Employee:
    def __init__(self, employee_id):
        self.employee_id = employee_id

    def pay_info(self):
        return f"Employee #{self.employee_id}"

class TeachingAssistant(Student, Employee):
    def __init__(self, user_id, name, email, employee_id):
        Student.__init__(self, user_id, name, email)    # initialize Student part
        Employee.__init__(self, employee_id)            # initialize Employee part
        self.assists = []

    def role(self):
        return "teaching assistant"

    def assist(self, course_code):
        if course_code not in self.assists:
            self.assists.append(course_code)

```

Listing 344 — Example: Teaching Assistant

Defines User, Student, Professor, Employee, TeachingAssistant; no visible output.

Student initializes User; TeachingAssistant separately initializes Employee. This works for this specific hierarchy. Do not generalize explicit base calls to cooperative diamond inheritance, where a shared base might otherwise be initialized twice.

```

ta = TeachingAssistant("S-010", "Alan Turing", "alan@example.org", "E-500")
ta.enroll("CS101")
ta.assist("CS101")

print(ta.describe())    # from Student/User
print(ta.pay_info())    # from Employee

```

Listing 345 — Example: Teaching Assistant

Output

```

Teaching Assistant: Alan Turing <alan@example.org> – enrolled in 1 course(s)
Employee #E-500

```

The inherited description uses the overridden role; pay_info reads the employee identifier.

Try it: reverse Base1 and Base2 in the first example. Predict the greeting, run it, and inspect the MRO. Prefer a simpler composition design when separate roles do not need shared inheritance.

With these relationships established in code, we can use diagrams to make an object-oriented design easier to inspect.

7.5 UML Modeling

Introduction to UML

The Unified Modeling Language (UML) describes a system's structure and behavior through diagrams [1]. Use a diagram to answer a question before adding code—not to decorate the page.

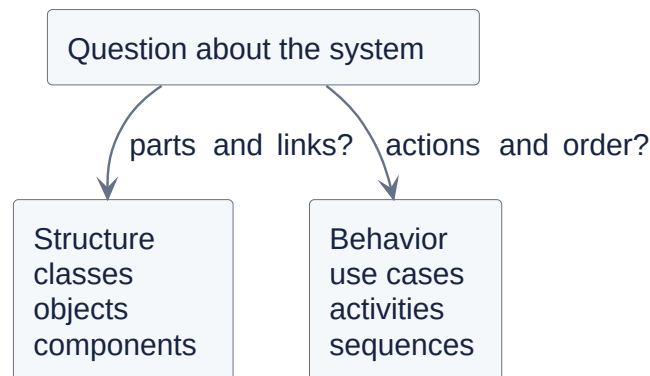


Figure 37 — Choose a diagram according to the question it needs to answer.

In this chapter, model the data, design classes, inspect an object snapshot and implement that model. Application Design then shows user goals, workflows, messages and software components. Each view describes the same University Course System from a different angle.

The information model uses crow's-foot ER notation; it is a data-modeling view, not a UML diagram type.

Start by identifying Student, Course and the Enrollment linking them.

Reference

- Object Management Group (2017). [Unified Modeling Language, version 2.5.1](#).

Information or Data Model

An information model identifies the data and relationships a system needs. An entity–relationship (ER) diagram is one way to express it.

Here, Student has an ID and name; Course has a code, title and credits; Enrollment connects one student to one course and may hold a grade.

An **entity** is a kind of thing for which we keep records; its box lists the fields in those records. This ER diagram uses **crow’s-foot notation** for relationship counts: the branching foot means many, a small circle permits zero, and two bars mean exactly one. Thus o{ means zero or many and || means exactly one. Read the symbol at the far end of a line to answer “how many of those records can relate to one of these?”

For example, start at Student and follow the line to Enrollment: a student can have zero or many enrollment records. Starting at Enrollment and following it back to Student gives exactly one student. The Course link follows the same rule. The field label **PK** means **primary key**: the value, or combination of values, that identifies a record uniquely. **FK** means **foreign key**: a value referring to an identifying field in another entity. The pair (student_id, course_code) is Enrollment’s composite primary key; the two fields together identify one enrollment, although neither field alone is unique across all enrollments. Each also refers to its Student or Course record. These are the model’s identification rules; the application must enforce them. str denotes text, int an integer and None an absent grade. These field types describe the data; they do not by themselves implement validation in Python.

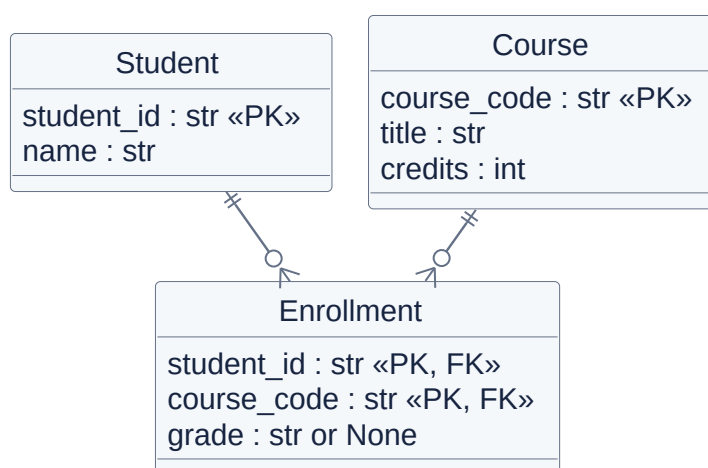


Figure 38 — Student and Course are connected through Enrollment records.

One student can have many enrollments; one course can also have many. Each enrollment refers to exactly one of each. The grade is absent until assigned.

Try it: add a second student to CS101 on paper. Which entity needs a new record? Next, add operations to turn the data model into a class model.

For the relationship symbols, see [PlantUML’s Information Engineering notation](#). This is an ER data model, using the same visual style as our UML diagrams.

Class Diagram

A class diagram shows types, attributes, operations and relationships. Its rectangles have compartments for the class name, data and operations.

+ means public, – private and # protected in UML. Our Python `_enrollments` and `_grade` use the internal-use underscore convention; Python does not enforce those UML access levels.

{readOnly} marks a public value that callers read without assigning to it directly.

1 means exactly one; 0..* means zero or more. Plain lines show associations. We use them here because students and courses have independent lifetimes.

University Course Model

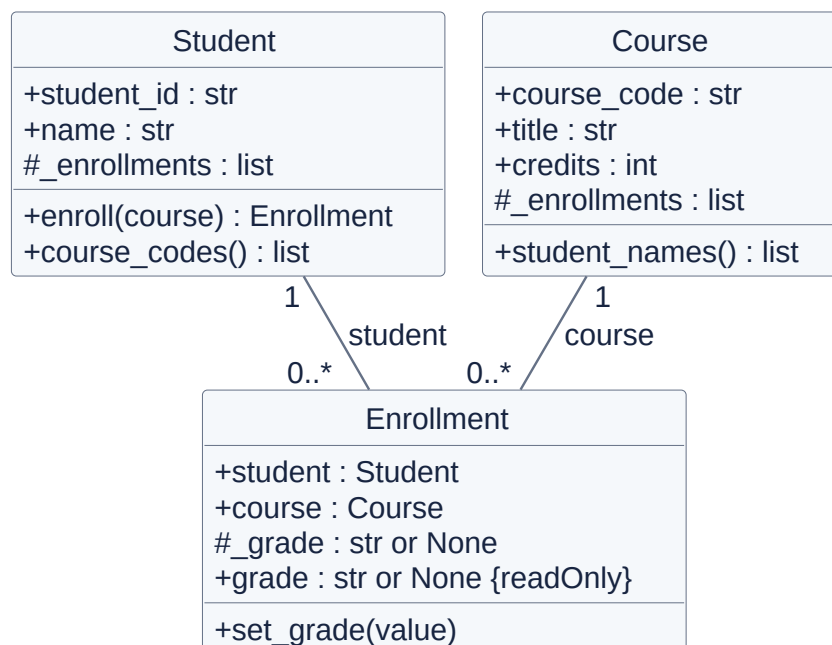


Figure 39 — Each Enrollment links exactly one Student and one Course.

Read one box from top to bottom: its class name, attributes, then operations. An attribute such as `name : str` describes stored text; an operation such as `enroll(course) : Enrollment` accepts a course and returns an Enrollment. Parentheses identify an operation, while a colon introduces a stated type.

Read a relationship from both directions. The 0..* near Enrollment allows one Student to have zero or many enrollments; the 1 near Student says each Enrollment refers to exactly one Student. The Course relationship works the same way.

Student's `enroll(course)` creates an Enrollment and records the same link on both sides. Enrollment holds the grade for that student in that course. Its public grade property is read-only; `set_grade` controls updates.

Check the model: why would a single grade stored on Course lose information? Trace a second student joining the same course, then inspect the object snapshot.

Object Diagram

An object diagram shows selected instances and values at one moment. An underlined label such as `ada : Student` names the object and its class. Lines connect those particular instances.

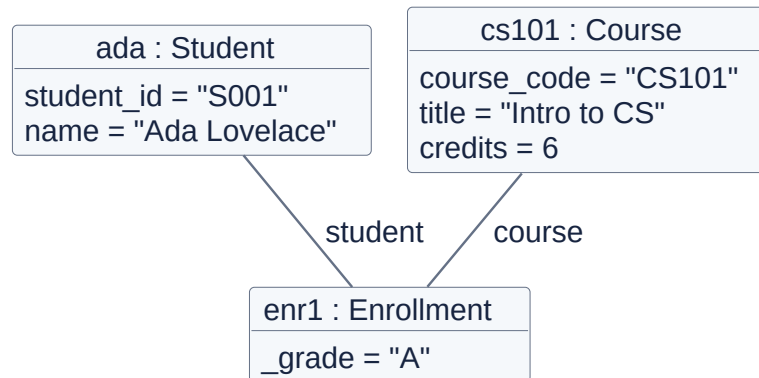


Figure 40 — Ada's CS101 enrollment after grade A is assigned.

Unlike a class model, this is a snapshot of particular objects at one moment. `student_id = "S001"` is a current value, not a type declaration. The line labelled `student` connects the `Enrollment` object to `Ada`; the line labelled `course` connects it to `CS101`. These lines represent the references stored by that enrollment, not a sequence of function calls.

The snapshot follows `ada.enroll(cs101)` and `enr1.set_grade("A")`. `enr1.student` refers to `Ada`, and `enr1.course` refers to `CS101`.

Predict: what value would `_grade` show before grading? The next example creates this snapshot in Python.

From Diagram to Code

Translate the model one responsibility at a time: fields become instance attributes; operations become methods; associations become references to other objects.

The UML activity diagram shows the translation workflow; rounded boxes are actions read from the start circle to the end bullseye.

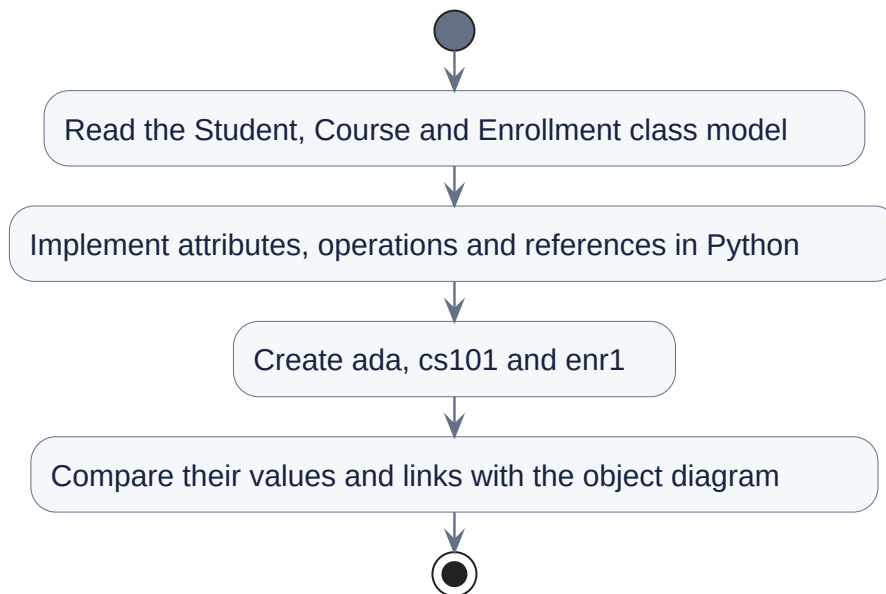


Figure 41 — Translate a class model into definitions, then verify a concrete object snapshot.

Define the Model

None means no grade has been assigned. A list comprehension builds a new list by reading the enrollment links. `raise ValueError(...)` rejects an input that breaks a model rule.

```

class Student:
    def __init__(self, student_id, name):
        self.student_id = student_id
        self.name = name
        self._enrollments = []

    def enroll(self, course):
        for enrollment in self._enrollments:
            if enrollment.course is course:
                raise ValueError("Already enrolled")
        enrollment = Enrollment(self, course)
        self._enrollments.append(enrollment)
        course._enrollments.append(enrollment)
        return enrollment

    def course_codes(self):
        return [enrollment.course.course_code for enrollment in self._enrollments]

class Course:
    def __init__(self, course_code, title, credits):

```

```

        if not isinstance(credits, int) or isinstance(credits, bool) or credits <= 0:
            raise ValueError("Credits must be a positive integer")
        self.course_code = course_code
        self.title = title
        self.credits = credits
        self._enrollments = []

    def student_names(self):
        return [enrollment.student.name for enrollment in self._enrollments]

class Enrollment:
    def __init__(self, student, course):
        self.student = student
        self.course = course
        self._grade = None

    @property
    def grade(self):
        return self._grade

    def set_grade(self, value):
        if value not in {"A", "B", "C", "D", "F"}:
            raise ValueError("Grade must be A, B, C, D or F")
        self._grade = value

```

Listing 346 — Define the Model

The definitions produce no visible output. Use `Student.enroll` to create and register a link on both sides; directly constructing `Enrollment` would not register it.

Create the Object Snapshot

Continue with all three class definitions above:

```

ada = Student("S001", "Ada Lovelace")
cs101 = Course("CS101", "Intro to CS", 6)
enr1 = ada.enroll(cs101)
enr1.set_grade("A")
print(ada.course_codes())
print(cs101.student_names())
print(enr1.grade)

```

Listing 347 — Create the Object Snapshot

Output

```

['CS101']
['Ada Lovelace']
A

```

The returned enrollment links Ada and CS101; its grade becomes A. Each query builds a new list from those links.

Keep one `Course` object per catalog code: duplicate detection here uses object identity. Data is held only in memory.

Try it: enroll Grace in CS101, grade her enrollment B, and check that Ada's grade remains A. Then reject grade Z and confirm that the stored grade is unchanged.

The model now supports enrollment and grading. Application design begins by asking who uses these operations and what they want to achieve.

7.6 Application Design

Use Case Diagram

A use case describes a goal an actor has when using a system. Stick figures represent external roles; ovals represent goals; the rectangle marks the system boundary. Lines show which roles take part in each use case.

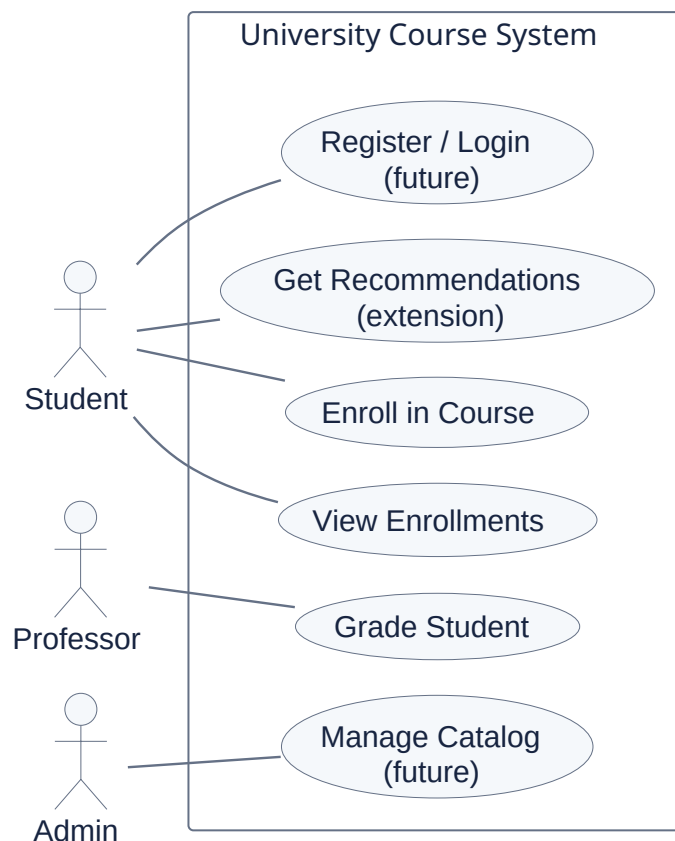


Figure 42 — User goals, with future and extension features marked.

The core project provides enrollment and grade storage. The roles describe intended users; they are not implemented access controls. Login, catalog administration and recommendations remain explicitly marked future features or extensions.

Use cases describe **what** users need. An activity diagram opens one operation to show its decisions and actions.

Activity Diagram

An activity diagram shows a workflow. Actions use rounded rectangles; a diamond branches on a condition. The filled circle starts the workflow and the bullseye ends it.

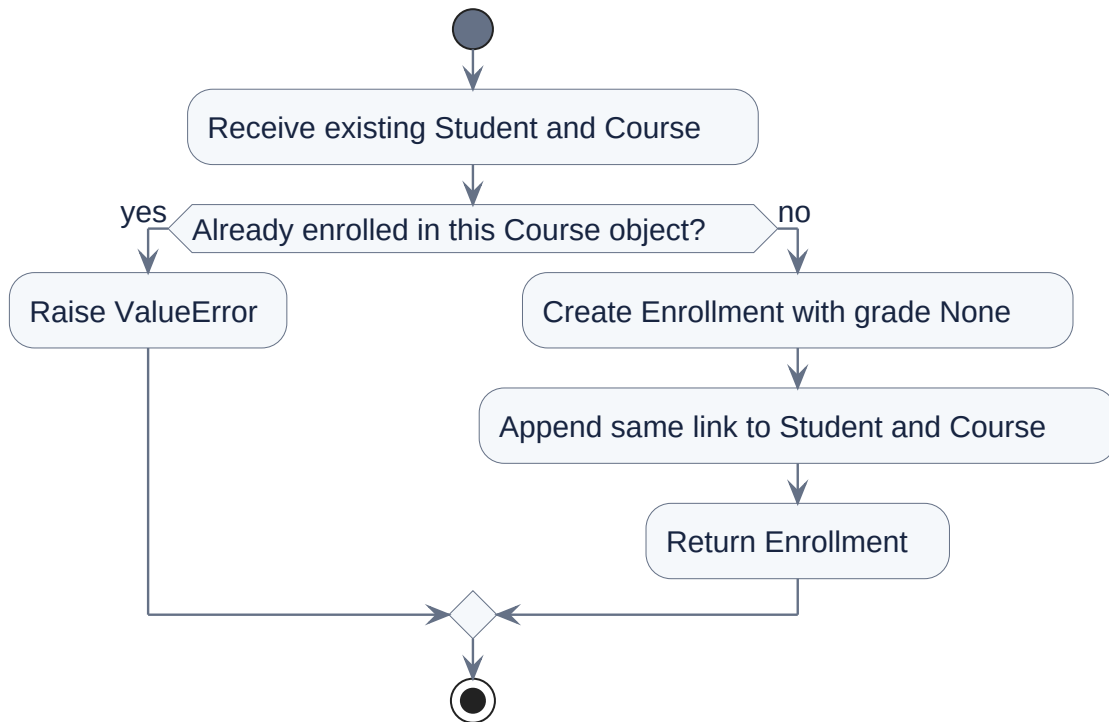


Figure 43 — Successful and duplicate enrollment paths.

This is the implemented `Student.enroll` operation. It rejects a duplicate before changing either collection. Otherwise it creates one `Enrollment`, records it on both sides and returns it.

Trace it: follow the diagram for a first enrollment and a repeated request. Capacity and prerequisite checks would need additional decisions and tests.

The activity view explains the decisions within enrollment; a sequence diagram will show the objects that carry out those steps.

Sequence Diagram

A sequence diagram shows who calls whom and in what order. Participants have vertical lifelines. Time advances downward; solid arrows show calls and dashed arrows show returns. An `alt` frame separates alternative paths.

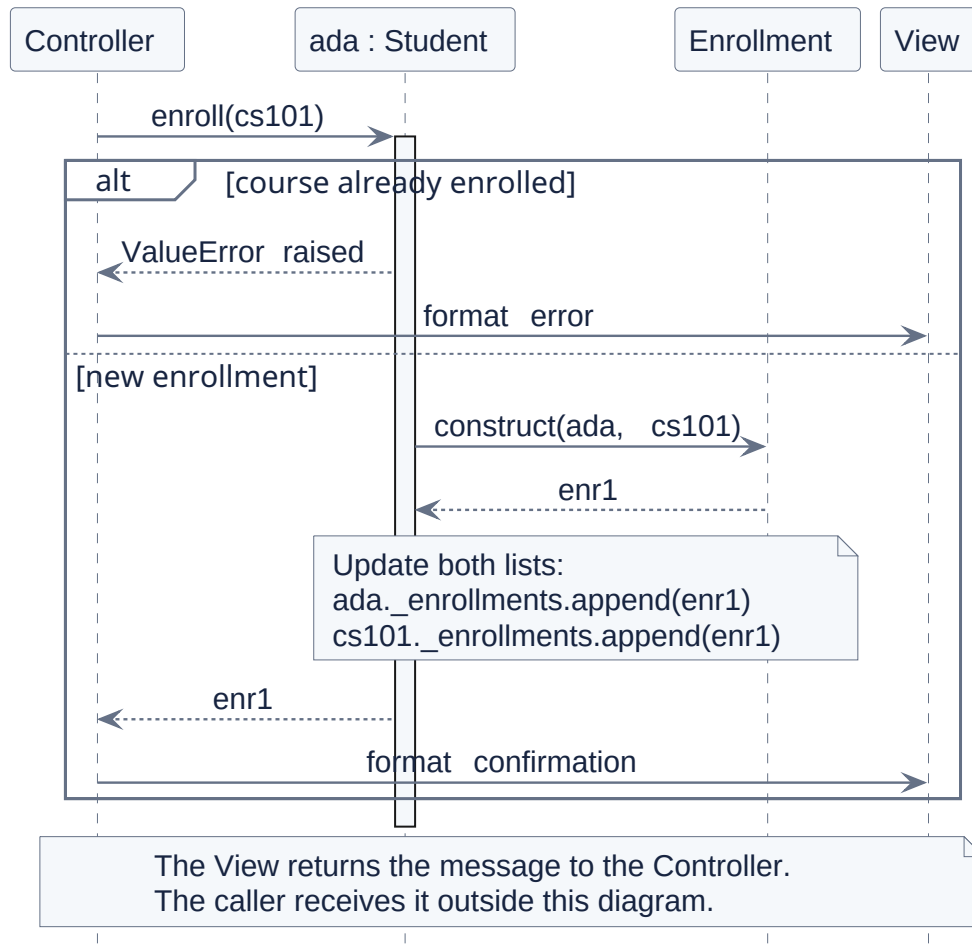


Figure 44 — The calls and list updates made during enrollment.

The Controller asks the Student to enroll. The model either raises `ValueError` or creates a link and appends it to both enrollment lists. The Controller passes the result to the View for formatting.

The note in the success branch records the two actual list updates: `ada._enrollments.append(enr1)` and `cs101._enrollments.append(enr1)`. We show the lists in this note to keep the diagram narrow; these are operations on the stored lists, not additional Student or Course methods. The duplicate branch performs neither update. The returned Enrollment is not a second newly created enrollment.

Controller and View name responsibilities in this diagram; the MVC lesson implements them as functions rather than classes. Its View returns a formatted message to the Controller, which returns it to the calling program outside this diagram. Trace either the duplicate branch or the success branch, not both in one call.

Check it: why must the duplicate path avoid both append calls?

These interactions also suggest how to divide the code into modules with clear responsibilities.

Structuring Code in OOP

A Python module is a `.py` file. Group code by responsibility so changing presentation does not require changing enrollment rules.

Here an **artifact** is a file, such as `model.py`. The tabbed **package** box groups related files in the diagram; it does not create those files or automatically make the folder an importable Python package. A dashed dependency arrow points from the file doing the import toward the file it uses. Follow those arrows to identify which definitions must be available when a module loads.

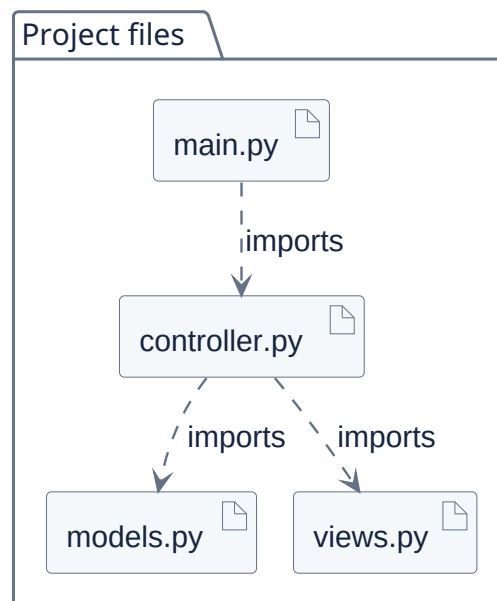


Figure 45 — The module layout gives each file a clear responsibility.

Start in one file, then separate:

- `models.py`: Student, Course, Enrollment and their rules.
- `controller.py`: coordinate a request and handle expected model errors.
- `views.py`: format the result.
- `main.py`: create objects and run the application.

For this simple layout, put the files in one project folder. `from models import Student` imports the class into another module. Keep the model independent of the view to avoid circular dependencies.

Apply it: identify which file should reject duplicate enrollment, and which should change the wording of its message.

A component diagram makes these software parts and their dependencies visible in one view.

Component Diagram

A component diagram shows software parts and their dependencies. A component is drawn as a labeled box with a component symbol. A dashed arrow points from the component using a service to the component providing it.

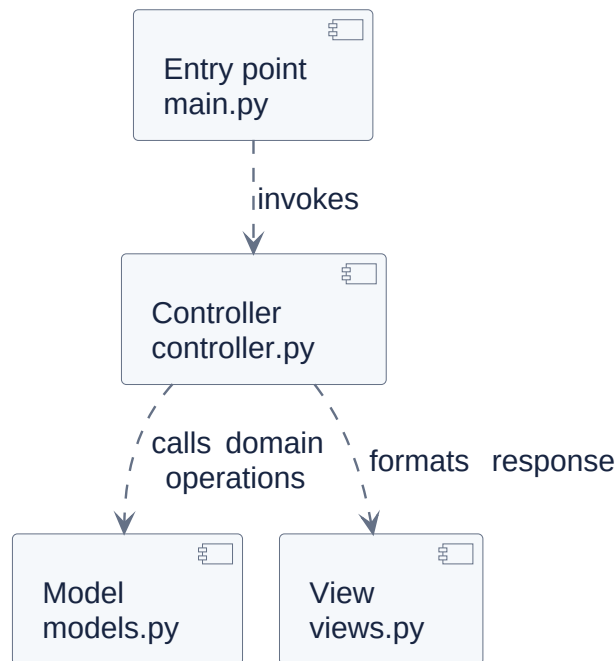


Figure 46 — Dependencies between the application’s components.

The entry point invokes the Controller. The Controller uses the Model for domain operations and the View for formatting. These responsibilities can begin in one file before you split them into modules.

A dependency arrow does not implement authentication or guarantee independence. Add storage, authentication or recommendation components only when you implement and test those features.

The next section brings these responsibilities together in a small Model–View–Controller example.

Model–View–Controller (MVC) Architecture

MVC separates three responsibilities: the **Model** holds data and domain rules; the **View** presents results; the **Controller** coordinates a user request.

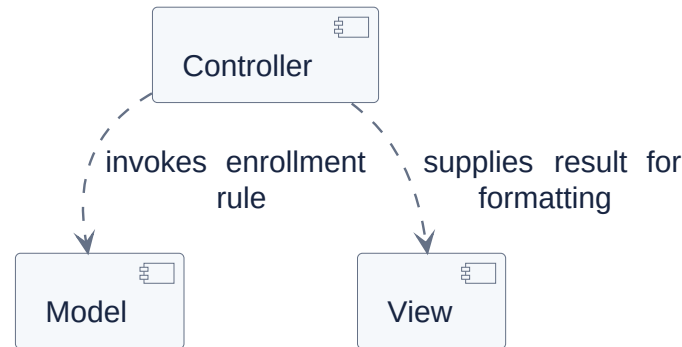


Figure 47 — The Controller uses the Model and View.

A Small Controller and View

First run the model and demonstration from From Diagram to Code. Ada is already enrolled in CS101, so her next request should take the duplicate path.

```
def enrollment_view(message):
    return f"Enrollment: {message}"

def enroll_controller(student, course):
    try:
        enrollment = student.enroll(course)
    except ValueError as error:
        return enrollment_view(str(error))
    return enrollment_view(f"{enrollment.student.name} -> {course.course_code}")

print(enroll_controller(ada, cs101))
grace = Student("S002", "Grace Hopper")
print(enroll_controller(grace, cs101))
```

Listing 348 — A Small Controller and View

Output

```
Enrollment: Already enrolled
Enrollment: Grace Hopper -> CS101
```

The Controller handles the model's `ValueError`. The View formats either the error or a success message; the Model still owns the enrollment rule.

Try it: change only the prefix in `enrollment_view`. Check that enrollment counts and grades remain unchanged.

MVC is a design pattern, not a requirement imposed by every web framework. This example needs neither a web framework nor a database.

The University Course System project now combines the model, diagrams and separation of responsibilities in one working application.

7.7 University Course Project

Project: University Course System

Build a small University Course System that enrolls students, stores grades and reports results. Use the diagrams and model from the preceding chapters.

The UML activity diagram outlines the baseline verification scenario. Read its actions downward from the start circle to the end bullseye.

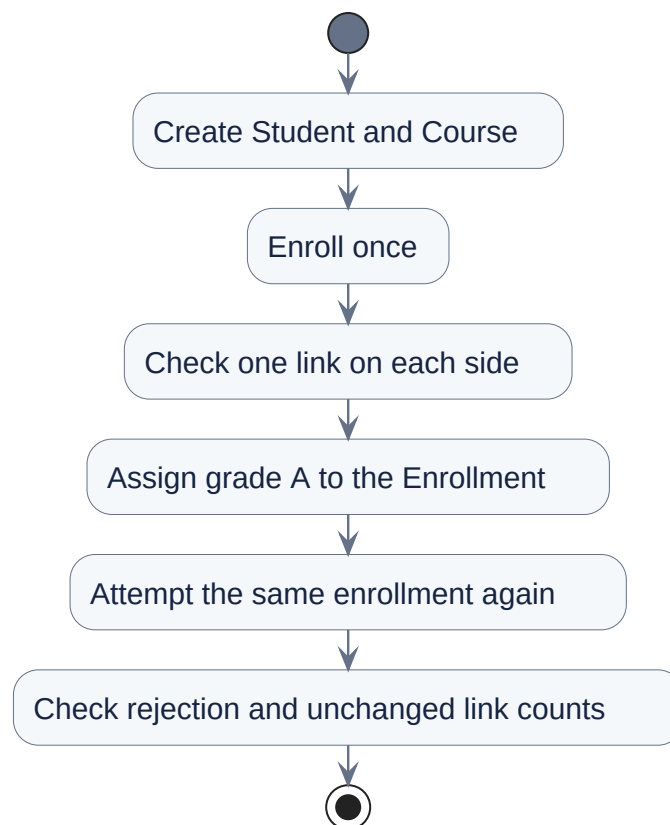


Figure 48 — The project baseline tests a successful enrollment and a duplicate rejection.

1. Define the Rules

Keep one Student per student ID and one Course per course code. Each Enrollment links exactly one of each. Reject repeated enrollment in the same Course object; accept grades A, B, C, D and F, with None before grading.

The core is in memory. Login, persistent storage, capacity limits, prerequisites and recommendations are extensions.

2. Check the Model

Use the [class diagram](#) and [object snapshot](#). Confirm that names and associations match [the Python definitions](#). A grade belongs to an Enrollment, not to Course alone.

3. Predict and Run

Run the complete definitions from From Diagram to Code, then this continuation. The [standalone project source](#) combines both.

```
ada = Student("S001", "Ada Lovelace")
cs101 = Course("CS101", "Intro to CS", 6)
enrollment = ada.enroll(cs101)
enrollment.set_grade("A")
print(ada.name, ada.course_codes())
print(cs101.course_code, cs101.student_names())
print("Grade:", enrollment.grade)
try:
    ada.enroll(cs101)
except ValueError as error:
    print("Rejected:", error)
print("Enrollment links:", len(ada._enrollments), len(cs101._enrollments))
```

Listing 349 — 3. Predict and Run

Output

```
Ada Lovelace ['CS101']
CS101 ['Ada Lovelace']
Grade: A
Rejected: Already enrolled
Enrollment links: 1 1
```

The duplicate request is rejected before changing either list, so the final counts remain one and one.

4. Change and Explain

Add Grace, enroll her in CS101 and assign her grade B. Confirm that Ada's grade remains A. Reject an invalid grade Z and verify that the earlier value remains unchanged.

For a catalog extension, store courses in a dictionary keyed by code. Reject duplicate codes before creating another course object; the current model detects duplicate enrollment by object identity.

5. Separate Responsibilities

Use the [MVC functions](#) and [module layout](#). Keep enrollment rules in the Model, request coordination in the Controller and message formatting in the View.

6. Test and Reconcile

Test empty collections, normal enrollment, duplicate rejection, two students in one course, one student in two courses, grade updates and invalid credits. Rejected operations must preserve state.

Trace the [activity](#) and [sequence](#) views for success and failure. Update the editable diagrams whenever you change a rule.

7. Extend One Feature

Choose Professor grading, course capacity, prerequisite checks or a local recommendation rule. State its inputs and acceptance examples before coding. Update the affected [use cases](#), [data model](#) and [components](#).

Deliver the Python sources, editable diagrams, SVGs and test evidence. Explain one design choice and one remaining limitation.

8. Advanced Python Concepts

Introduction

The foundations now support programs with several cooperating parts. The remaining chapters address what happens when those programs must work with files, respond to failures, test their behaviour and communicate with other systems.

As programs grow, the challenges you face change. You begin working with larger datasets, interacting with files and external services, handling errors reliably, and writing code that must run efficiently or even in parallel. At this stage, Python offers a wide range of tools and abstractions that go beyond the basics.

This chapter introduces those advanced concepts. We start with **specialized data structures** that extend the built-in types and make tasks like *counting*, *grouping*, *queuing*, and *modeling data* far easier. Later sections cover **file handling**, **error management**, **concurrency**, and working with **APIs**—skills essential for building robust, real-world applications.

With these tools, your Python code becomes not only functional but scalable, expressive, and ready for practical use.

Specialized Data Structures

Python's built-in lists, dictionaries, sets, and tuples handle most everyday tasks. But some patterns appear so often—counting items, grouping data, modeling lightweight records, or managing efficient queues—that dedicated structures make the work faster, clearer, and easier to maintain.

To support these recurring needs, Python's standard library includes a set of optimized, expressive data structures. They act as focused tools: small, powerful, and designed to solve common data-processing challenges without extra boilerplate.

In this chapter, we explore a set of specialized **data-structure classes and utilities** provided by Python's standard library, including: - `Counter`, `defaultdict`, `namedtuple`, and `deque` — high-level data-structure classes and factory functions from the `collections` module - **Combinatorial and functional iterators** such as `combinations`, `permutations`, `product`, and `chain` from the `itertools` module - **Heap-based priority queue functions** like `heappush`, `heappop`, and `nsmallest` from the `heapq` module - **Categorical enumeration types** using the `Enum` class from the `enum` module

Each of these structures addresses a specific real-world problem and helps you write code that is both efficient and elegant.

Counting with Counter

Counting is a pattern that appears everywhere—from analyzing word frequencies in a dataset to counting user interactions or summarizing event logs. Python could certainly do this with a dictionary and some careful checks, but the `Counter` class in `collections` removes the boilerplate entirely by treating counting as a first-class operation.

`Counter` can be called as:

```
Counter(iterable)
Counter(mapping)
```

Where **iterable** is anything you can loop over (list, string, generator), and **mapping** is a dict-like object of `element:count` pairs.

Consider word frequency analysis:

```
from collections import Counter

text = "to be or not to be that is the question"
counts = Counter(text.split())

print(counts)
print(counts.most_common(2))
```

Listing 350 — Counting with Counter

Output

```
Counter({'to': 2, 'be': 2, 'or': 1, 'not': 1, 'that': 1, 'is': 1, 'the': 1, 'question': 1})
[('to', 2), ('be', 2)]
```

`most_common(n)` returns the top `n` items as `(element, count)` tuples which is extremely useful in text mining or exploratory analysis.

Another example from log analysis:

```
errors = ["500", "404", "404", "503", "500", "500"]
print(Counter(errors))
```

Listing 351 — Counting with Counter

Output

```
Counter({'500': 3, '404': 2, '503': 1})
```

The output shows how often each error code appeared—an immediate snapshot of system behavior.

Grouping with defaultdict

Many tasks require grouping: by attributes, by categories, by clustering criteria. A normal dictionary needs checks to ensure the key exists before appending.

`defaultdict` (a specialized dictionary class) calls its factory when a missing key is accessed with square brackets, such as `groups[3]`. A membership test or `.get()` does not create a value.

Its signature is:

```
defaultdict(default_factory)
```

Where **default_factory** is a callable—often `list`, `int`, or `set`.

Example: grouping words by length:

```
from collections import defaultdict

words = ["cat", "apple", "dog", "pear", "bee"]
groups = defaultdict(list)

for w in words:
    groups[len(w)].append(w)

print(groups)
```

Listing 352 — Grouping with defaultdict

Output

```
defaultdict(<class 'list'>, {3: ['cat', 'dog', 'bee'], 5: ['apple'], 4: ['pear']})
```

This prints a dictionary-like structure where each *key* represents a *word length* and each *value* is the *list of words of that length*.

No checks required—`defaultdict(list)` ensures each new key starts with an empty list. This is heavily used in data preprocessing, graph adjacency lists, and clustering pipelines.

Lightweight Records with namedtuple

Sometimes you need tiny, efficient record types to represent lines in a dataset—coordinates, database rows, user entries, etc. A tuple is compact, but numbered fields can be hard to read. A small class is another valid choice; `namedtuple` offers a concise record with named fields. Python's `namedtuple` solves this by acting as a **factory function that creates lightweight, attribute-based record classes**.

Its signature:

```
namedtuple(typename, field_names)
```

Where `field_names` may be a list or a space-separated string.

A geographical example illustrates the idea:

```
from collections import namedtuple

Location = namedtuple("Location", ["city", "lat", "lon"])
paris = Location("Paris", 48.8566, 2.3522)

print(paris.city, paris.lat)
```

Listing 353 — Lightweight Records with namedtuple

Output

```
Paris 48.8566
```

This prints the city name and its latitude using `attribute` access rather than tuple indices, showing how `namedtuple` makes small records both readable and convenient.

Representing rows looks even more natural:

```
Person = namedtuple("Person", ["name", "age", "city"])
records = [
    Person("Alice", 30, "Berlin"),
    Person("Bob", 25, "Munich"),
]

print("Name      | Age | City")
print("-" * 28)
for r in records:
    print(f"{{r.name:<10}} | {{r.age:<3}} | {{r.city}}")
```

Listing 354 — Lightweight Records with namedtuple

Output

```
Name      | Age | City
-----
Alice     | 30  | Berlin
Bob       | 25  | Munich
```

Here the rows are displayed in a simple table-like form, similar to how database results often look. In data science workflows, where each row holds fixed, simple attributes, `namedtuple` provides clarity and speed without the verbosity of full class definitions.

Double-Ended Queues with deque

While lists are powerful, they are inefficient when inserting or removing elements at the start. This limitation becomes visible in queue-like workflows: task schedulers, streaming buffers, undo/redo stacks, or breadth-first search algorithms in graphs.

deque is designed for precisely these scenarios, providing fast operations on both ends.

Its core methods:

```
deque(iterable)
deque.append(x)
deque.appendleft(x)
deque.pop()
deque.popleft()
```

To see this in action, consider a simple task queue:

```
from collections import deque

tasks = deque(["task1", "task2"])
tasks.append("task3")      # add to right
tasks.appendleft("urgent") # add to left
print(tasks)
```

Listing 355 — Double-Ended Queues with deque

Output

```
deque(['urgent', 'task1', 'task2', 'task3'])
```

You can also remove from either end efficiently:

```
tasks.pop()      # remove "task3"
tasks.popleft()  # remove "urgent"
print(tasks)
```

Listing 356 — Double-Ended Queues with deque

Output

```
deque(['task1', 'task2'])
```

This structure excels in any scenario where both ends must remain efficient—queue systems, sliding windows, graph traversal, or managing chronological events.

In graph processing—an essential part of recommendation engines, social networks, and pathfinding—deque plays a crucial role in **breadth-first search (BFS)**, where nodes are repeatedly removed from the front of a queue. A regular list struggles with this because removing from the front requires shifting every remaining element, whereas deque supports efficient removal from either end. The following fragment initializes a queue with a starting node:

```
from collections import deque
start_node = "A"
queue = deque([start_node])
```

Listing 357 — Double-Ended Queues with deque

A full breadth-first search also needs a graph representation and a visited set. This fragment only initializes its queue; graph traversal is an extension, not a complete algorithm shown here.

Iteration Tools with itertools

Some problems require exploring structured combinations: forming pairs, generating permutations, Cartesian products of parameter sets, or merging sequences. Doing this manually is error-prone and memory-intensive. `itertools` supplies a suite of **combinatorial and iterative generators**—functions that produce values lazily, one at a time, without materializing all output at once. Combinatorial tools may still store their finite input pools; lazy output does not make an enormous search space small.

Combinations

A combination selects items without regard to order.

Signature:

```
combinations(iterable, r)
```

Generates `r`-length subsets where order does not matter.

For example, forming project pairs in a group of students:

```
from itertools import combinations

students = ["Ada", "Grace", "Alan"]

for pair in combinations(students, 2):
    print(pair)
```

Listing 358 — Combinations

Output

```
('Ada', 'Grace')
('Ada', 'Alan')
('Grace', 'Alan')
```

This produces all unique 2-person groups—valuable in co-occurrence analysis or team formation algorithms.

Permutations

In other scenarios, the order does matter—for example, generating possible task sequences, testing input arrangements, or exploring order-sensitive models.

Permutations help enumerate every possible ordered arrangement of a collection, making them essential in optimization, simulation, and combinatorial search problems.

Signature:

```
permutations(iterable, r)
```

For instance, if we want to generate all ordered pairs from a list of letters, `permutations` produces each possible arrangement:

```
from itertools import permutations

letters = ["A", "B", "C"]

for p in permutations(letters, 2):
    print(p)
```

Listing 359 — Permutations

Output

```
('A', 'B')
('A', 'C')
('B', 'A')
('B', 'C')
('C', 'A')
('C', 'B')
```

Here each pair appears in every possible order, giving a complete view of all ordered arrangements.

Cartesian Products

Some tasks require exploring *every possible combination* across two or more sets. This appears frequently in machine learning (hyperparameter grids), simulation studies, scheduling problems, or combining dimensions like *color* × *size* × *shape*.

The product function from `itertools` generates these combinations efficiently and lazily—no nested loops or temporary lists needed.

Signature:

```
product(iter1, iter2, ...)
```

It returns an iterator producing tuples that contain one element from each iterable.

Example:

The code below generates every combination of learning rate and batch size:

```
from itertools import product

learning_rates = [0.01, 0.1]
batch_sizes = [16, 32]

for combo in product(learning_rates, batch_sizes):
    print(combo)
```

Listing 360 — Cartesian Products

Output

```
(0.01, 16)
(0.01, 32)
(0.1, 16)
(0.1, 32)
```

`product` systematically pairs each learning rate with each batch size, giving all four combinations. This pattern is widely used in machine learning for **hyperparameter tuning**, where you try all combinations of parameter values to see which combination yields the best model performance. Using `product` makes such systematic exploration much easier and more scalable.

Chaining Iterables

Sometimes datasets come from different sources—multiple files, pages, or batches—but you want to iterate over all items as if they were one continuous sequence. The `chain` function from `itertools` provides exactly this: a lazy iterator that streams several iterables back-to-back without copying them.

Signature:

```
chain(iter1, iter2, ...)
```

The following code merges three lists of names into a single stream:

```
from itertools import chain

group1 = ["Ada", "Grace"]
group2 = ["Alan"]
group3 = ["Linus", "Margaret"]

for name in chain(group1, group2, group3):
    print(name)
```

Listing 361 — Chaining Iterables

Output

```
Ada
Grace
Alan
Linus
Margaret
```

`chain` yields items from each iterable in order, behaving like one long sequence. Because items are produced lazily, it scales efficiently to large datasets.

Priority Queues with heapq

Some algorithms repeatedly need the smallest element: scheduling events by time, retrieving shortest distances in graph algorithms, or processing tasks by importance. A **heap** is a list representation of a binary tree that satisfies the min-heap rule: each parent is no greater than its children. The smallest item is at index 0, but the whole list is not sorted.

Python's `heapq` module provides the standard heap operations.

Core function signatures:

- `heapq.heapify(list)`

Converts a regular Python list into a valid *min-heap in-place*, rearranging the elements so the smallest item is always at index 0. `heapify` builds the heap in linear time. - `heapq.heappush(heap, item)`

Inserts a new item into the heap while automatically restoring the heap property. The heap property is restored after the push. - `heapq.heappop(heap)`

Removes and returns the **smallest** item from the heap. The structure is rebalanced so the next smallest element becomes the new root. - `heapq.nsmallest(k, iterable)`

Retrieves the **k smallest elements** from any iterable efficiently. Unlike sorting the entire dataset, it focuses only on the few smallest values—useful for top-k selection or thresholding in large datasets.

Example 1: Processing Tasks by Priority

Imagine a help-desk system where lower numbers mean higher priority (1 = critical, 2 = high, etc.). We can use a heap to always handle the most urgent task first:

```
import heapq

# (priority, description)
tasks = [
    (3, "Update documentation"),
    (1, "Fix production bug"),
    (2, "Reply to customer emails"),
]

heapq.heapify(tasks) # turn list into a heap

while tasks:
    priority, description = heapq.heappop(tasks)
    print(f"Handling (priority {priority}): {description}")
```

Listing 362 — Priority Queues with heapq

Output

```
Handling (priority 1): Fix production bug
Handling (priority 2): Reply to customer emails
Handling (priority 3): Update documentation
```

This prints the tasks in priority order—1, then 2, then 3—so the critical bug is fixed first, emails are handled next, and documentation updates come last. For this fixed queue, the smallest priority number is handled first. If priorities tie, Python compares the next tuple field, so descriptions

determine the tie order; use an added sequence number when arrival order must be preserved.

Example 2: Selecting the k Smallest Errors

In data science, you might want to inspect the **k smallest error values** from a large list of model losses, without sorting everything. `heapq.nsmallest()` does this efficiently:

```
import heapq

errors = [0.42, 0.15, 0.78, 0.05, 0.33]
best = heapq.nsmallest(2, errors)
print(best)
```

Listing 363 — Priority Queues with heapq

Output

```
[0.05, 0.15]
```

It returns the two lowest error values. Instead of sorting the entire list, the `nsmallest` function efficiently extracts only the values you care about—especially useful when working with large datasets or repeated model evaluations.

Categorical Data with Enum

Many programs rely on values drawn from a fixed set: user roles, workflow states, product categories, or sentiment labels. Using strings can introduce typos; using integers hides meaning. The Enum class creates **symbolic, immutable constants** that enforce valid categories.

Signature:

```
class YourEnum(Enum):  
    NAME = value
```

Here is a simple Enum representing user roles:

```
from enum import Enum  
  
class Role(Enum):  
    ADMIN = 1  
    EDITOR = 2  
    VIEWER = 3  
  
print(Role.ADMIN)
```

Listing 364 — Categorical Data with Enum

Output

```
Role.ADMIN
```

An unknown member such as `Role.ADMNI` raises `AttributeError`, and `Role(99)` raises `ValueError`. An enum does not validate every variable automatically; code can still assign a plain string unless you check the input.

In data science, enums are useful when modeling discrete labels:

```
class Sentiment(Enum):  
    POSITIVE = 1  
    NEGATIVE = 2  
    NEUTRAL = 3
```

Listing 365 — Categorical Data with Enum

This makes sentiment analysis pipelines cleaner and easier to maintain.

The following challenges combine these structures. Choose the structure from the operation you need: counting, grouping, ordered processing or a fixed set of categories.

Summary and Coding Challenges

Specialized data structures provide fast, elegant solutions to recurring programming patterns. They remove boilerplate, improve performance, and make code clearer. Whether you're counting words with `Counter`, grouping data with `defaultdict`, modeling lightweight records with `namedtuple`, managing queues with `deque`, exploring structured iterations through `itertools`, handling priorities with `heapq`, or representing categories using `enum`, these tools help you write Python that is both efficient and expressive.

To consolidate these skills, the following challenges place you in real-world scenarios where you must combine `Counter`, `defaultdict`, `namedtuple`, `deque`, `itertools`, `heapq`, and `Enum`.

Exercise 1 — Build a Newsroom Keyword Extractor (`Counter` + `chain`)

Scenario:

You work as a data engineer in a digital newsroom. Editors send you article headlines grouped by category. You must determine which words are trending across the entire newsroom.

Task: Write a program that:

- Flattens all headlines from all categories into one sequence of words.
- Lowercase the text and use `re.findall(r"[a-z]+", headline.lower())` to extract word tokens. This exercise deliberately uses ASCII words; apostrophes split words, so `team's` contributes `team` and `s`.
- Counts how often each word appears.
- Prints the **10 most frequent words**.
- Prints how often the word **"breaking"** appears.

Starter Code:

```

from collections import Counter
from itertools import chain

headlines = {
    "world": [
        "Breaking news as climate summit begins",
        "Leaders discuss global policy changes",
    ],
    "tech": [
        "Breaking discoveries in AI research",
        "New device launch draws attention",
    ],
    "sports": [
        "Local team achieves historic victory",
        "Coach praises team's discipline",
    ],
}

# TODO: Flatten and normalize words
# TODO: Count frequencies
# TODO: Print top 10 words
# TODO: Print count of "breaking"

```

Listing 366 — Exercise 1 — Build a Newsroom Keyword Extractor (Counter + chain)

Expected Output:

```

Most common words:
breaking: 2
team: 2
news: 1
as: 1
climate: 1
...
Occurrences of 'breaking': 2

```

Listing 367 — Exercise 1 — Build a Newsroom Keyword Extractor (Counter + chain)

Exercise 2 — Academic Grade Distribution

Scenario:

A university course needs an automated report summarizing student performance. You receive raw name–score pairs and must produce structured grade records and a grade distribution table.

Task:

- Create a lightweight record structure for student data.
- Use the exercise scale A: 90–100, B: 80–89, C: 60–79, D: 50–59, F: below 50; reject scores outside 0–100. These bands are sample policy, not a universal grading standard.
- Convert raw scores into grades.
- Group students by grade.
- Print a summary such as:

```
Grade A: Alice, Eva
Grade B: Bob
Grade C: Dan
```

Listing 368 — Exercise 2 — Academic Grade Distribution

Starter Code:

```
from collections import namedtuple, defaultdict
from enum import Enum

Student = namedtuple("Student", ["name", "score", "grade"])

class Grade(Enum):
    # TODO: Add enum values here
    pass

def score_to_grade(score):
    # TODO: implement grading logic
    pass

raw_scores = [
    ("Alice", 95),
    ("Bob", 78),
    ("Cara", 88),
    ("Dan", 62),
    ("Eva", 91),
]

# TODO: Build student records
# TODO: Group by grade
# TODO: Print the summary
```

Listing 369 — Exercise 2 — Academic Grade Distribution

Example Output:

```
Grade A: Alice, Eva
Grade B: Cara
Grade C: Bob, Dan
```

Listing 370 — Exercise 2 — Academic Grade Distribution

Exercise 3 — Support Center Queue Simulation

Scenario:

A customer support center receives two kinds of customers: - **VIP customers** (served first) - **Regular customers** (served in order of arrival)

You must simulate the service process.

Task: 1. Create a queue. 2. Add several regular customers and several VIP customers. 3. For this exercise, add `cust_001`, `cust_002`, `cust_003`, then add `vip_001` and `vip_002` with `appendleft`. The last VIP added is served first. If VIPs must keep arrival order, design a separate VIP queue instead. 4. Process customers in order until the queue is empty. 5. Print the exact service order.

Starter Code:

```

from collections import deque

queue = deque()

# TODO: Add customers
# queue.append(...)      # regular customer
# queue.appendleft(...)   # VIP customer

# TODO: Process all customers

```

Listing 371 — Exercise 3 — Support Center Queue Simulation

Example Output:

```

Serving: vip_002
Serving: vip_001
Serving: cust_001
Serving: cust_002
Serving: cust_003

```

Listing 372 — Exercise 3 — Support Center Queue Simulation

Exercise 4 — Market Basket Exploration Using Combinatorics

Scenario

You're working as a data analyst for an online grocery store. Your task is to explore purchasing patterns using a single dataset:

- First, enumerate the **pairs of items** in this one basket. Repeated observations across many baskets would be needed to measure frequency.
- Then, explore **all pricing strategy combinations** the business might test for those same items.

This exercise uses the same dataset, but applies two different combinatorial tools.

Dataset

You receive a list of items purchased in a single basket:

```

basket = ["milk", "bread", "butter", "eggs"]

```

Listing 373 — Dataset

You also receive two sets of possible pricing parameters the business wants to experiment with:

```

discount_rates = [0.05, 0.10]   # 5% and 10%
bundle_sizes   = [2, 3]          # bundles of 2 or 3 items

```

Listing 374 — Dataset

Part A — Co-Purchased Item Pairs (combinations)

Identify all unique **2-item pairs** from the basket. This is used later to evaluate which bundles might be meaningful for promotions.

Starter Code

```
from itertools import combinations

basket = ["milk", "bread", "butter", "eggs"]

# TODO: Generate all 2-item combinations
```

Listing 375 — Part A — Co-Purchased Item Pairs (combinations)

Example Output

```
('milk', 'bread')
('milk', 'butter')
('milk', 'eggs')
('bread', 'butter')
('bread', 'eggs')
('butter', 'eggs')
```

Listing 376 — Part A — Co-Purchased Item Pairs (combinations)

Part B — Pricing Strategy Grid (product)

Management wants to test every combination of: - discount rate - bundle size

Use `itertools.product` to generate all pricing strategies.

Starter Code

```
from itertools import product

discount_rates = [0.05, 0.10]
bundle_sizes = [2, 3]

# TODO: Generate all discount-bundle combinations
```

Listing 377 — Part B — Pricing Strategy Grid (product)

Example Output

```
discount=0.05, bundle=2
discount=0.05, bundle=3
discount=0.10, bundle=2
discount=0.10, bundle=3
```

File Handling & Serialization

Modern programs rarely operate in isolation. They read text from configuration files, load datasets from disk, store logs, write results, exchange information with web APIs, or persist objects for later use. Python supports all of this through file handling and serialization, two fundamental techniques every data scientist and developer must master.

File handling is the foundation: opening files, reading their contents, writing results, and managing where data lives on your system.

Serialization builds on this capability by converting Python objects—lists, dictionaries, or even custom classes—into formats that can be stored and restored later. Sometimes this means using a human-readable format such as **JSON**, and sometimes a Python-specific binary representation like `pickle`.

In this chapter, we explore:

- reading and writing text files
- handling structured formats like **CSV** and **JSON**
- serializing Python objects with `pickle`
- managing file paths with `pathlib` module
- understanding when to choose each technique

By the end, you will know how to move data into and out of Python safely, efficiently, and in the right format for your needs.

File Handling in Python

File handling is the practical foundation for working with data outside your program. Whether you read input files, write analysis results, or store logs and configuration settings, Python gives you straightforward tools to open, read, write, and manage files safely.

Before exploring structured formats like CSV, JSON, and binary serialization, it's important to understand how Python handles plain text files: how to open them, how to read and write their contents, and how to ensure they close correctly.

Use a new practice folder. Before running the examples, create `notes.txt` with the two lines `First note.` and `Second note.`, `story.txt` with `A short story.`, and `large_dataset.txt` with the two lines `record one` and `record two`. End each line with a newline. Leave `config.ini` absent to test the missing-file path. These small fixtures make the output reproducible.

Opening and Closing Files

Python opens files using the built-in `open()` function. Its first argument is a file name or path; an optional second argument supplies the mode:

- the file name or path
- the **mode**, which tells Python how the file should be accessed

Common modes include: - `"r"` — read (default) - `"w"` — write (overwrites existing content) - `"a"` — append - `"rb"` / `"wb"` — binary read/write

Before reading or writing, you must open the file; once the work is done, you must close it.

Example: Opening a File in Read Mode

```
f = open("notes.txt", "r", encoding="utf-8")
print(f)
f.close()
```

Listing 378 — Opening and Closing Files

Output

```
<_io.TextIOWrapper name='notes.txt' mode='r' encoding='utf-8'>
```

If `notes.txt` exists, `open()` connects to it and returns a file object. The `close()` call releases system resources. Forgetting to close a file is a common beginner's mistake, and Python provides a safer alternative.

Using Context Managers (`with`)

In real software, anything can go wrong while reading or writing: the file might be missing, corrupted, or locked. If an error occurs, a manually opened file may remain unclosed—potentially corrupting data.

To avoid this, Python offers the `with` statement. It opens the file at the start of the block and **always** closes it afterward, even if an exception is raised.

Example: Reading a File Safely

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

Listing 379 — Using Context Managers (`with`)

Output

```
First note.
Second note.
```

`with` ensures the file closes automatically. Inside the block, the file is open; once the block ends, Python closes it—no matter what happened. This pattern is the recommended way to handle files.

Checking File Properties

Once a file is open, Python exposes useful metadata. You can inspect the file's name, the mode used to open it, and whether the file is currently open or already closed.

Example: Inspecting a File Object

```
with open("notes.txt", "r", encoding="utf-8") as f:
    print("Filename:", f.name)
    print("Mode:", f.mode)
    print("Is closed?", f.closed)

print("Is closed after block?", f.closed)
```

Listing 380 — Checking File Properties

Output

```
Filename: notes.txt
Mode: r
Is closed? False
Is closed after block? True
```

- `f.name` prints the file name.
- `f.mode` shows "r" here, because we opened the file in read mode.
- `f.closed` is `False` inside the block and `True` afterward.

This is helpful when diagnosing and debugging problems in file-heavy workflows.

Reading Text Files

The main reading methods are:

- `read()` — load the entire file as a single string
- `readline()` — read one line at a time
- `readlines()` — return all lines as a list

Example: Reading a File with `read()`

Suppose you want to display the contents of a text file to the user:

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

Listing 381 — Reading Text Files

Output

A short story.

`read()` loads the full content into memory.

For very large files, you might prefer line-by-line reading to avoid excessive memory usage.

Writing Text Files

Writing uses the "w" or "a" modes:

- "w" overwrites the file or creates it if missing
- "a" appends to the end

Example: Writing New Content

```
with open("summary.txt", "w", encoding="utf-8") as f:
    f.write("Project Summary\n")
    f.write("=====\n")
    f.write("All tasks completed successfully.\n")
```

Listing 382 — Writing Text Files

These writes produce no visible output. Because we used "w", any existing content is replaced. This is ideal for generating reports or exporting results.

Appending to a File

Appending is useful for logs or history tracking. Each run can add new entries without deleting earlier records.

Example: Adding Log Entries

```
with open("app.log", "a", encoding="utf-8") as f:
    f.write("User logged in\n")
    f.write("User opened settings\n")
```

Listing 383 — Appending to a File

These writes produce no visible output. New log entries are added at the end, preserving the previous content.

Reading Files Line by Line

Reading line by line can reduce memory use when processing large text files, such as logs or transcripts.

Example: Efficient Line Processing

```
with open("large_dataset.txt", "r", encoding="utf-8") as f:
    for line in f:
        print(line.strip())
```

Listing 384 — Reading Files Line by Line

Output

```
record one
record two
```

This approach keeps memory usage low by reading only one line at a time—ideal for files that may be several gigabytes.

Handling Exceptions When Working with Files

Sometimes files are missing or inaccessible. Rather than crashing, programs should handle such cases gracefully.

Example: Safe Reading with Error Handling

```
try:
    with open("config.ini", "r", encoding="utf-8") as f:
        content = f.read()
        print(content)
except FileNotFoundError:
    print("Error: The configuration file is missing.")
```

Listing 385 — Handling Exceptions When Working with Files

Output

```
Error: The configuration file is missing.
```

- `try` executes the file operation.
- `except` handles common issues, such as missing files.
- The `with` block closes the file if opening succeeded. If opening failed, there is no file object to close.

Use `with` for the resource and `try` for the failure you can handle; they solve different parts of the problem.

Conclusion

File handling gives your programs persistence. It allows them to load data from the outside world, save their results, maintain logs, and process text-based datasets reliably. With these foundational tools—`open()`, context managers, reading and writing methods, and basic error handling—you are now ready to explore more advanced forms of data exchange.

In the next section, we move from raw text files to **serialization**, where data is stored and exchanged in structured formats such as CSV, JSON, and Python's own binary representations.

File Serialization

Once you understand how to read and write plain text files, the next step is learning how to store and exchange structured data. Real-world applications rarely deal only with raw text—they work with tables, hierarchical records, API responses, configuration files, model outputs, and complex Python objects. **Serialization** is the process of converting Python data structures into standardized formats, while **deserialization** restores them back into Python objects.

Python offers several serialization tools, each designed for different needs:

- **CSV** for simple tabular data
- **JSON** for structured, language-independent data
- **pickle** for Python-specific binary serialization
- **pathlib** for managing file paths; it supports file operations but is not a serialization format

Together, they form the backbone of modern data workflows. Let's explore each one, starting with the most common: CSV.

Working with CSV Files

CSV (Comma-Separated Values) is the universal language of spreadsheets and datasets. It's simple, human-readable, and understood by nearly every tool—from Excel to machine learning libraries. Because of this, CSV is often the first step in importing user data or exporting analysis results.

Python's built-in `csv` module handles the details of reading and writing these files safely.

Writing a CSV File

Consider a small dataset of classic plays. Writing it to a CSV file is straightforward:

```
import csv

rows = [
    ["id", "title", "year"],
    [1, "Hamlet", 1603],
    [2, "Macbeth", 1606],
]

with open("plays.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerows(rows)
```

Listing 386 — Writing a CSV File

`csv.writer()` handles correct CSV formatting, and `writerows()` sends each list as one table row. The `newline=""` argument prevents accidental extra blank lines.

Reading a CSV File

Reading the file back follows the same pattern:

```
with open("plays.csv", "r", newline="", encoding="utf-8") as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Listing 387 — Reading a CSV File

Output

```
['id', 'title', 'year']
['1', 'Hamlet', '1603']
['2', 'Macbeth', '1606']
```

Run the writing example first. Each row read back is a list of strings, including numeric-looking fields.

When datasets become larger or column names matter, `csv.DictReader` allows you to treat each row as a dictionary—making the data much easier to manipulate.

Now that we’ve seen how Python works with flat tables, let’s move to a more flexible and expressive format: JSON.

Working with JSON

While CSV is ideal for simple tables, many applications use **nested or hierarchical data**: configuration files, web API responses, user accounts, or structured logs. JSON (JavaScript Object Notation) is designed for exactly this kind of data and has become the standard for data exchange across the internet.

Python’s `json` module makes converting between JSON text and Python objects effortless.

Writing JSON

Here is a structured record describing a book:

```
import json

book = {
    "title": "Pride and Prejudice",
    "author": "Jane Austen",
    "year": 1813,
    "characters": ["Elizabeth Bennet", "Mr. Darcy"]
}

with open("book.json", "w", encoding="utf-8") as f:
    json.dump(book, f)
```

Listing 388 — Writing JSON

`json.dump()` serializes the Python dictionary into JSON format and writes it to the file.

Reading JSON

Deserializing the data is just as simple:

```
with open("book.json", "r", encoding="utf-8") as f:
    data = json.load(f)

print(data)
```

Listing 389 — Reading JSON

Output

```
{'title': 'Pride and Prejudice', 'author': 'Jane Austen', 'year': 1813, 'characters': ['Elizabeth Bennet', 'Mr. Darcy']}
```

JSON is readable, language-independent, and widely supported—making it the preferred choice for APIs, configuration files, and interoperable data pipelines.

But there are times when neither CSV nor JSON can capture the full richness of Python objects. For those situations, Python provides a different tool: pickling.

Binary Serialization with Pickle

JSON supports a limited set of data types: for example, tuples become JSON arrays and return as lists, while sets and arbitrary custom instances need an explicit conversion. **pickle** can preserve many Python-specific types for trusted Python-only workflows.

Pickling converts supported objects into a binary stream. Custom classes must remain importable with compatible definitions when loading. Only load trusted pickle data: [unpickling can execute code](#).

Saving Objects with Pickle

```
import pickle

numbers = {"evens": [2, 4, 6], "odds": [1, 3, 5]}

with open("numbers.pkl", "wb") as f:
    pickle.dump(numbers, f)
```

Listing 390 — Saving Objects with Pickle

- "wb" opens the file in *binary write* mode.
- `pickle.dump()` serializes the object into a compact binary format.

This process is similar to “freezing” the object so it can be reused later.

Loading Objects with Pickle

To restore the saved object back into Python:

```
with open("numbers.pkl", "rb") as f:
    restored = pickle.load(f)

print(restored)
```

Listing 391 — Loading Objects with Pickle

Output

```
{'evens': [2, 4, 6], 'odds': [1, 3, 5]}
```

After unpickling, you get the same dictionary structure, with the same lists, in the same order—just as it was when saved.

Pickling is extremely convenient for Python-only workflows, such as caching machine-learning models, storing intermediate results, or saving program state. But because the format is binary and Python-specific, it is not appropriate for data meant to be shared with other environments.

The formats define how data is stored. Path management determines where the files are read and written.

Managing Paths with `pathlib`

As projects grow, files are rarely kept in a single directory. You may need folders for logs, datasets, outputs, or temporary files. Hardcoding paths like `"C:\\users\\data\\file.txt"` quickly becomes fragile and platform-dependent.

Python's `pathlib` module offers a clean, object-oriented way to construct and work with paths across different operating systems.

Example: Creating and Using a Path

```
from pathlib import Path

base = Path(".")          # current directory
data_dir = base / "data"  # build a subdirectory path
data_dir.mkdir(exist_ok=True)

file_path = data_dir / "hello.txt"
file_path.write_text("Hello from pathlib!", encoding="utf-8")

print(file_path.read_text(encoding="utf-8"))
```

Listing 392 — Managing Paths with `pathlib`

Output

```
Hello from pathlib!
```

- `/` joins paths naturally and readably.
- `mkdir(exist_ok=True)` creates the folder if needed.
- `write_text()` and `read_text()` simplify I/O operations.

`pathlib` makes your code portable, clearer, and much easier to maintain—especially in larger applications or collaborative projects.

Conclusion

Serialization builds on basic file handling to support a range of storage needs: **CSV** for tabular data, **JSON** for structured and human-readable formats, **pickle** for supported Python types in trusted workflows, and **pathlib** for portable path management.

Together, these tools let programs save results, reload state, and exchange information reliably. With data management covered, the next chapter turns to **error handling and testing** to ensure your

programs remain robust in real-world use.

Error Handling

No program runs perfectly every time—or in every environment. Users type the wrong input, files go missing, network connections break, and code occasionally encounters situations the programmer did not foresee. Without proper safeguards, a single unexpected event can crash an entire application.

Error handling gives programs a safety net. Instead of stopping abruptly, Python allows a program to detect a problem, respond to it, and—if appropriate—*continue running*. This ability makes software robust, user-friendly, and ready for the messy realities of real-world data.

In this chapter, we explore Python’s tools for handling errors gracefully. We will distinguish between errors and exceptions, learn how `try` and `except` blocks work, explore more advanced patterns such as multiple exceptions and clean-up actions, and finish with custom exceptions that make large projects maintainable.

Syntax Errors and Runtime Exceptions

A **syntax error** means Python cannot parse the source. A **runtime exception** reports a problem encountered while valid code is executing. Both are represented by exception types in Python.

Run these two examples separately. The first is intentionally invalid: its missing parenthesis prevents that file or cell from starting.

```
print("Hello world"    # intentionally missing a closing parenthesis
```

Listing 393 — Syntax Errors and Runtime Exceptions

Expected exception type

```
SyntaxError
```

The exact diagnostic wording depends on the Python version. A `try` statement in the same malformed file cannot run. A running program can, however, catch `SyntaxError` raised while parsing other supplied source.

The next example is valid syntax, but its division fails when executed:

```
n = 10
result = n / 0
```

Listing 394 — Syntax Errors and Runtime Exceptions

Expected exception

```
ZeroDivisionError: division by zero
```

A `try` block around this division can handle the runtime exception. The following examples show how.

The Structure of Exception Handling

Now that we understand the difference between errors and exceptions, we can look at how Python allows you to respond to unexpected events. Python uses a small set of keywords—`try`, `except`, `else`, and `finally`—each playing a specific role in the control flow.

These four building blocks let you isolate risky code, catch problems, run fallback logic, and clean up resources reliably.

Basic Syntax

The following is a syntax template, not a runnable program. Replace the comments with statements and `SomeException` with a real exception type:

```
try:
    # Code that might cause an exception
except SomeException:
    # What to do if that exception happens
else:
    # Runs only if no exception occurred
finally:
    # Runs always, whether there was an exception or not
```

Each part has a specific purpose:

- `try` wraps code that may fail.
- `except` captures and handles specific exceptions.
- `else` is for code that should run only when everything succeeded.
- `finally` is ideal for cleanup tasks like closing files or releasing resources.

Example: Handling Division Safely

Here's a short example showing all four blocks working together:

```
try:
    n = 0
    res = 100 / n
except ZeroDivisionError:
    print("You can't divide by zero!")
except ValueError:
    print("Invalid value provided.")
else:
    print("Result:", res)
finally:
    print("Execution complete.")
```

Listing 395 — Basic Syntax

Output

```
You can't divide by zero!
Execution complete.
```

- The `try` block attempts to divide by zero.
- `ZeroDivisionError` is caught in the first `except`, so the second one is skipped.
- The `else` block is *not* executed because an exception occurred.
- The `finally` block always runs, confirming the operation is complete.

This pattern becomes extremely helpful when reading files, processing user input, or calling external services—anywhere something might go wrong.

Catching Specific Exceptions

While it's tempting to catch all errors at once, precise exception handling is often safer. Different issues require different responses: a missing file is not the same as invalid input, and neither should be treated like a network failure.

To make your program robust and easier to debug, you can catch specific exception types.

Example: Handling Two Different Problems

Here is a simple demonstration of how Python distinguishes the underlying issues:

```
try:
    x = int("hello")      # will cause ValueError
    inv = 1 / x           # would cause ZeroDivisionError if x were 0
except ValueError:
    print("That value cannot be converted to a number.")
except ZeroDivisionError:
    print("Zero has no multiplicative inverse.")
```

Listing 396 — Catching Specific Exceptions

Output

```
That value cannot be converted to a number.
```

Converting "hello" to an integer raises a `ValueError`, so the first `except` block handles it. If the string had been "0", the conversion would succeed but `1 / x` would trigger a `ZeroDivisionError`, and the second handler would run instead.

By matching the exact exception type, your program remains clearer and more robust.

Catching Multiple Exceptions Together

Sometimes several different errors should lead to the same response. Instead of writing multiple `except` blocks with identical logic, Python allows grouping them into a single handler. This keeps the code more readable while remaining explicit about the problems you anticipate.

Example: Handling Value Errors and Type Errors Alike

The following example shows grouped exceptions during data conversion:

```
items = ["10", "twenty", 30] # mixed input

try:
    total = int(items[0]) + int(items[1])
except (ValueError, TypeError) as e:
    print("Conversion error:", e)
except IndexError:
    print("List index out of range.")
```

Listing 397 — Catching Multiple Exceptions Together

Output

```
Conversion error: invalid literal for int() with base 10: 'twenty'
```

Here, converting "twenty" into an integer triggers a `ValueError`, which is caught by the grouped handler. A `TypeError` would be caught in the same place if incompatible types were used. An `IndexError` would be handled separately if the list had fewer elements.

Grouped exceptions let you cover several related scenarios without duplicating logic.

Catch-All Handlers — and Why to Be Careful

Python also allows you to catch *any* exception using a bare `except:` block, including `KeyboardInterrupt` and `SystemExit`.

While this can prevent crashes when you absolutely cannot afford failure, it also hides the exact error type, making real problems harder to diagnose. In large systems, that can cause silent failures.

Example: Overly Broad Exception Handling

The following example shows how a catch-all handler can obscure the real error:

```
try:
    result = "100" / 20 # TypeError: dividing string by number
except ArithmeticError:
    print("Some arithmetic problem occurred.")
except:
    print("Something went wrong!")
```

Listing 398 — Catch-All Handlers — and Why to Be Careful

Output

```
Something went wrong!
```

Dividing a string by a number raises a `TypeError`, which is not a subclass of `ArithmeticError`. The bare `except` catches it, but hides valuable information. Although catch-all blocks are sometimes necessary, they should be used sparingly and placed carefully.

Raising Exceptions

Sometimes the error is not Python's fault—it's the input or state your program has encountered. When you detect an invalid condition, raising an exception yourself is the cleanest way to alert callers that something is wrong. This ensures invalid states never silently enter your system.

Example: Rejecting Invalid Age Values

Here is a short validation example:

```
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative.")
    print(f"Age set to {age}")

try:
    set_age(-5)
except ValueError as e:
    print(e)
```

Listing 399 — Raising Exceptions

Output

```
Age cannot be negative.
```

The function detects that the age is invalid and explicitly raises a `ValueError`. The caller handles it with a `try–except` block, keeping the program in a stable state. Raising exceptions gives your code a clear and consistent way to enforce rules.

Custom Exceptions

As your project grows, generic exception types may no longer provide enough detail. A large application benefits from having its own exception types that clearly communicate domain-specific problems: authentication failures, invalid transactions, exceeded quotas, or policy violations.

Python makes this easy—custom exceptions are created by inheriting from the built-in `Exception` class.

Example: Custom `AgeError` Class

Here is a short example demonstrating how to define and use your own exception type:

```
class AgeError(Exception):
    pass

def set_age(age):
    if age < 0:
        raise AgeError("Age cannot be negative.")
    print(f"Age set to {age}")

try:
    set_age(-5)
except AgeError as e:
    print(e)
```

Listing 400 — Custom Exceptions

Output

```
Age cannot be negative.
```

The class `AgeError` becomes a fully functional exception type unique to your application. When raised, it signals exactly what kind of problem has occurred.

Custom exceptions shine in larger systems, where different components may fail in different ways and need to be distinguished clearly.

Summary

This chapter introduced Python’s model for dealing with unexpected events: distinguishing errors from exceptions, using `try, except, else,` and `finally`, catching specific problems, raising your own exceptions, and defining custom ones for larger systems. With these tools, programs stay robust instead of crashing.

With error handling in place, we can now move on to **testing**, where we ensure code behaves correctly not just in theory—but in practice.

Exercise 1 — Building a Safe Calculator (Input Validation + Specific Exceptions)

Scenario

You are developing a tiny command-line calculator for a data-processing workflow. Because users might provide incorrect values or even attempt illegal operations such as dividing by zero, the calculator must be able to defend itself gracefully. Your task is to write a function that performs division safely, handling invalid numeric input and catching mathematical errors without stopping the program.

Task

Write a function called `safe_divide(a, b)` that tries to convert both inputs into floating-point numbers and performs the division. If the input cannot be converted, the function should catch `ValueError` or `TypeError` and print a clear message. If the division fails because the denominator is zero, the function should catch the `ZeroDivisionError` and inform the user. Whenever everything works, it should return the division result; when an error occurs, it should return `None` so the program can continue running.

The supplied test pairs cover success, zero division and invalid text. Also test `None` as an invalid input. Keep the function's error result as `None`; its caller prints the final result line.

Starter Code

```
def safe_divide(a, b):
    # TODO: convert to float and handle exceptions
    pass

tests = [("10", "2"), ("5", "0"), ("abc", "3"), ("8.5", "2")]
for x, y in tests:
    print(f"{x} / {y} = {safe_divide(x, y)}")
```

Listing 401 — Exercise 1 — Building a Safe Calculator (Input Validation + Specific Exceptions)

Example Output

```
10 / 2 = 5.0
Error: cannot divide by zero.
5 / 0 = None
Error: 'abc' is not a valid number.
abc / 3 = None
8.5 / 2 = 4.25
```

Exercise 2 — Student Registration Validator (Raising & Custom Exceptions)

Scenario

A university registration system must validate new student entries before they are stored. Invalid names, incorrect age formats, or unrealistic age values can easily slip into the database unless the system performs strict checks. Your task is to create a small validation function that raises meaningful custom exceptions whenever the input violates these rules.

Task

Define a custom exception named `RegistrationError`. Then write a function `register_student(name, age)` that checks whether the provided name is non-empty, whether the age can be interpreted as an integer, and whether the resulting age falls within the exercise's assumed range of 16 to 120 inclusive. A name containing only whitespace is empty. Accept integers or integer text, and reject fractional values rather than truncating them. These are exercise rules, not a claim about every university's admission policy. If any of these rules are broken, the function should raise `RegistrationError` with an appropriate message. When the data is valid, the function should return a confirmation message. The test loop attempts to register several students, catching and displaying any registration problems.

Starter Code

```
class RegistrationError(Exception):
    pass

def register_student(name, age):
    # TODO: raise RegistrationError in all invalid cases
    pass

students = [
    ("Alice", 20),
    ("", 19),
    ("Bob", "twenty"),
    ("Clara", 150),
]

for name, age in students:
    try:
        print(register_student(name, age))
    except RegistrationError as e:
        print("Registration failed:", e)
```

Listing 402 — Exercise 2 — Student Registration Validator (Raising & Custom Exceptions)

Example Output

```
Student Alice (20) registered.
Registration failed: Name cannot be empty.
Registration failed: Age must be an integer.
Registration failed: Age must be between 16 and 120.
```

Python Testing

Software evolves constantly, and every change—no matter how small—can introduce new bugs or revive old ones. Testing provides the safety net that keeps software stable, trustworthy, and maintainable. Tests provide evidence about the cases they exercise and help detect later regressions; they cannot prove that a program has no bugs. This chapter introduces the landscape of software testing and prepares the ground for our deep dive into **unit testing** with **unittest** and **pytest**.

Why Testing Matters

When a program fails, the result is more than an error message. A failure may break a feature customers rely on, corrupt data, or disrupt business operations. Testing helps prevent these outcomes by verifying behavior early and consistently. It also allows developers to refactor code with confidence, knowing that existing tests will catch regressions.

Testing improves the reliability of your software, supports long-term maintenance, and reduces costs by detecting problems before they reach production.

Testing Strategies in Practice

Software can be tested from several angles, each addressing a different kind of risk. Understanding these strategies helps developers choose the right tool for each situation.

Unit Testing

Unit testing examines the smallest pieces of a system—individual functions, classes, or modules. These tests run quickly, isolate logic, and provide immediate feedback when something breaks.

Integration Testing

Integration tests verify how different components behave when combined. A function may work perfectly alone yet fail when interacting with a database, API, or another module.

Functional Testing

Functional tests check a feature against its expected behavior. They may use a function, API or user interface.

Acceptance Testing

Acceptance tests validate that the system meets the requirements established by clients or stakeholders. They represent the final “yes” before delivery.

Exploratory Testing

Exploratory testing relies on human insight rather than pre-planned scripts. Testers actively explore the system, looking for unusual behavior that automated tests may miss.

Together, these strategies create a multi-layered defense against bugs and regressions.

Overview of Python Testing Tools

Python supports these testing strategies with a rich set of frameworks, each suited to a different purpose.

Unit Testing Frameworks

Python provides several ways to create automated tests:

- `unittest` — the built-in, xUnit-style framework that ships with Python.
- `pytest` — a popular, modern framework with simple syntax and powerful features.
- `Nose /Nose2` — older frameworks that extend `unittest`, still found in legacy projects.
- `doctest` — executes examples inside docstrings as tests.

In this chapter, we focus on the two most widely used unit-testing frameworks: `unittest` and `pytest`.

Mocking and Test Doubles

Many tests require replacing external dependencies—APIs, databases, or user input—with controlled substitutes. Python’s `unittest.mock` provides tools for this, while **`pytest-mock`** integrates mocking seamlessly into `pytest` workflows.

High-Level Testing Frameworks

Although not the focus of this chapter, it is useful to know that Python also supports full-system testing:

- **Selenium** automates real browsers for web UI testing.
- **Robot Framework** conducts acceptance tests using a readable, keyword-driven approach.

These tools sit higher in the testing hierarchy and are often used in large applications.

What This Chapter Focuses On

This chapter concentrates on **unit testing**, the foundation of all automated testing. We will explore two frameworks:

1. `unittest` — Python’s standard library testing framework
2. `pytest` — a more expressive and widely adopted alternative

Each framework will be presented on its own page, with narrative explanations, real-world scenarios, step-by-step examples, and clear commentary so beginners and experts alike can follow effortlessly.

What Comes Next

The next section introduces `unittest` through working examples. It will connect these terms:

- Assertions express expected outcomes.
- Fixtures prepare and clean up the environment.
- Test cases hold individual test scenarios.
- Test suites group multiple test cases.
- Test runners execute tests and provide readable reports. Together, these concepts form the foundation of Python's built-in testing ecosystem.

We will first run a complete `unittest` test file, then use `pytest` to express and run similar checks.

Unit Testing with unittest

A unit test checks a small piece of behavior against an expected result. Python's standard-library module `unittest` groups related test methods in classes and reports which checks pass or fail. We will build complete test files, then add setup and cleanup and combine tests into a suite.

Assertions and Test Cases

An **assertion** compares an actual result with an expectation. Subclass `unittest.TestCase` and write methods beginning with `test_`; each method becomes a separate test case. Useful checks include `assertEqual`, `assertTrue`, `assertFalse`, `assertIsNone`, `assertIsInstance`, `assertIn` and `assertIs`.

Save the following as `math_helpers.py` in a new practice folder:

```
# math_helpers.py
def add(a, b):
    return a + b

def safe_divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b
```

Listing 403 — Assertions and Test Cases

The module supplies functions; defining them produces no output. In the same folder, save `test_math_helpers.py`:

```
# test_math_helpers.py
import unittest
from math_helpers import add, safe_divide

class TestMathHelpers(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)
        self.assertNotEqual(add(2, 2), 5)

    def test_safe_divide(self):
        self.assertEqual(safe_divide(10, 2), 5)
        self.assertRaises(ValueError, safe_divide, 10, 0)

if __name__ == "__main__":
    unittest.main()
```

Listing 404 — Assertions and Test Cases

`assertRaises` calls `safe_divide(10, 0)` and checks that it raises `ValueError`. The `if __name__ == "__main__"` guard starts the runner when this file is executed directly, not when another file imports its classes.

Run the Tests

From the folder containing both files, run:

```
python -m unittest test_math_helpers.py
```

Listing 405 — Run the Tests

Output — elapsed time varies:

```
..
-----
Ran 2 tests in ...s

OK
```

The dots represent successful tests. OK means the assertions in those two tests passed; it does not establish correctness for untested inputs. Add `-v` for the names of individual tests.

Fixtures: Setup and Cleanup

A **fixture** prepares the data or resources a test needs. `setUp()` runs before a test method; `tearDown()` runs afterward if setup succeeded. `addCleanup()` registers cleanup that also runs if a later step in setup fails.

Save this function as `file_tools.py`:

```
# file_tools.py
def read_first_line(path):
    with open(path, "r", encoding="utf-8") as file:
        return file.readline().strip()
```

Listing 406 — Fixtures: Setup and Cleanup

Now save `test_file_tools.py`. A temporary directory keeps the test from overwriting a file in your project:

```
# test_file_tools.py
import tempfile
import unittest
from pathlib import Path
from file_tools import read_first_line

class TestFileTools(unittest.TestCase):
    def setUp(self):
        self.directory = tempfile.TemporaryDirectory()
        self.addCleanup(self.directory.cleanup)
        self.test_file = Path(self.directory.name) / "data.txt"
        self.test_file.write_text("Hello\nSecond line\n", encoding="utf-8")

    def test_read_first_line(self):
        self.assertEqual(read_first_line(self.test_file), "Hello")

if __name__ == "__main__":
    unittest.main()
```

Listing 407 — Fixtures: Setup and Cleanup

`TemporaryDirectory()` creates a separate folder. Cleanup removes it after the test, while `read_first_line` closes its file through `with`. Run the test:

```
python -m unittest test_file_tools.py
```

Listing 408 — Fixtures: Setup and Cleanup

Output — elapsed time varies:

```

.
-----
Ran 1 test in ...s

OK

```

More Test Cases

A test class can group several related behaviors. Save `test_strings.py`:

```

# test_strings.py
import unittest

class TestStringMethods(unittest.TestCase):
    def test_upper(self):
        self.assertEqual("Ada".upper(), "ADA")

    def test_lower(self):
        self.assertEqual("ADA".lower(), "ada")

    def test_split(self):
        self.assertEqual("Ada Grace".split(), ["Ada", "Grace"])

if __name__ == "__main__":
    unittest.main()

```

Listing 409 — More Test Cases

Each method checks one operation. Run `python -m unittest test_strings.py` to execute its three tests, or use discovery after saving all the example files:

```
python -m unittest discover -v
```

Listing 410 — More Test Cases

Output summary — individual test names and elapsed time vary with the selected files:

```

Ran 6 tests in ...s

OK

```

By default, discovery searches for importable files matching `test*.py`. Here it finds two math tests, one file test and three string tests. It does not automatically select every file ending in `_test.py`.

Test Suites and Runners

A **suite** groups selected tests. A **runner** executes that suite and reports results. Save `run_suite.py` alongside the files above:

```
# run_suite.py
import unittest
from test_math_helpers import TestMathHelpers
from test_strings import TestStringMethods

loader = unittest.defaultTestLoader
suite = unittest.TestSuite([
    loader.loadTestsFromTestCase(TestMathHelpers),
    loader.loadTestsFromTestCase(TestStringMethods),
])

runner = unittest.TextTestRunner()
runner.run(suite)
```

Listing 411 — Test Suites and Runners

Run it with `python run_suite.py`. This suite deliberately selects the two math tests and three string tests:

Output — elapsed time varies:

```
.....
-----
Ran 5 tests in ...s

OK
```

For a notebook, save these files first and run `python -m unittest` through that notebook's interpreter. If test classes are defined in the notebook itself, call `unittest.main(argv=[""], exit=False)` after their definitions; the adjusted arguments avoid the notebook's startup flags and `exit=False` keeps its kernel running.

Try it: change one expected string to an incorrect value, run that test and read the failure. Restore it, then add a test for a missing file using `assertRaises(FileNotFoundError, ...)`.

The same behavior can also be checked with ordinary test functions. The next section uses `pytest` to do that.

Unit Testing with pytest

pytest is a third-party framework that runs ordinary test functions using Python's `assert` statement. We will use it to check results, inspect an intentional failure and reuse test data with fixtures. Install it in your active environment with `python -m pip install pytest`, then save the examples as the named files. Run `python -m pytest` to use that same interpreter.

Writing Your First Pytest Test

Just like any well-organized project, pytest relies on a clear naming convention to find tests automatically. Test files must be named:

```
test_*.py    or    *_test.py
```

and pytest looks for functions beginning with `test_`.

This simple convention means you rarely have to register tests manually.

Let's build a small example from scratch.

Example: Testing a Simple Function

First, create a small module that we want to test. Save the following as **calculator.py**:

```
# calculator.py
def add(a, b):
    return a + b
```

Listing 412 — Writing Your First Pytest Test

This module contains one function, `add(a, b)`, which returns the sum of its two arguments.

Next, we write a test file that imports this function and checks whether it behaves correctly.

Create a second file named **test_calculator.py**:

```
# test_calculator.py
from calculator import add

def test_add_numbers():
    assert add(2, 3) == 5
```

Listing 413 — Writing Your First Pytest Test

What This Test Does

- It imports the `add` function.
- It defines a test function `test_add_numbers()`.
- It uses a plain `assert` to check that `add(2, 3)` evaluates to 5.

If the assertion passes, pytest considers the test successful; if not, pytest shows you exactly why.

Running the Test

You can now run the test suite using the terminal. From the same directory as your test files, execute:

```
python -m pytest
```

Listing 414 — Running the Test

To get a more detailed view, use:

```
python -m pytest -v
```

Listing 415 — Running the Test

Example Output

```
test_calculator.py .
```

```
1 passed in ...s
```

Pytest automatically located the file, discovered the test function, executed it, and reported the result.

Notice what you did **not** have to write:

- no test classes
- no `assertEqual()`
- no boilerplate

Pytest keeps your tests short and easy to read.

Assertions in Pytest

Unlike `unittest`, which requires calling specialized methods like `assertEqual` or `assertTrue`, `pytest` uses standard Python `assert` statements. This simplicity does not reduce clarity — in fact, when an assertion fails, `pytest` produces a rich explanation showing the values inside the expression.

Let's look at a slightly larger test module.

Example: Testing a Multiplication Function

Save this file as **test_math_ops.py**:

```
# test_math_ops.py
def multiply(a, b):
    return a * b

def test_multiply_positive():
    assert multiply(3, 4) == 12

def test_multiply_negative():
    assert multiply(-2, 5) == -10

def test_multiply_zero():
    assert multiply(0, 99) == 0
```

Listing 416 — Example: Testing a Multiplication Function

A Failure Example

After defining `multiply` above, deliberately run this incorrect assertion by itself, or place it in a separate test function to inspect pytest's failure report:

```
assert multiply(3, 4) == 11
```

Listing 417 — A Failure Example

Pytest provides a detailed explanation:

```
>      assert multiply(3, 4) == 11
E      assert 12 == 11
E      + where 12 = multiply(3, 4)
```

This introspection — showing actual values, expected values, and the expression evaluated — is one of pytest's greatest strengths.

Using Fixtures: Preparing Test Environments

As your tests grow, you will need to create repeatable setups: fake users, sample accounts, temporary files, or database connections. Pytest provides **fixtures**, reusable functions that prepare data or environments for your tests.

Fixtures are small, powerful helpers that keep tests clean and eliminate duplication.

Example: A Simple Fixture

Create a file named **test_user.py**:

```
# test_user.py
import pytest

@pytest.fixture
def sample_user():
    return {"name": "Alice", "role": "admin"}

def test_user_has_name(sample_user):
    assert sample_user["name"] == "Alice"

def test_user_role_is_admin(sample_user):
    assert sample_user["role"] == "admin"
```

Listing 418 — Example: A Simple Fixture

How This Works

- `@pytest.fixture` tells pytest that `sample_user()` is a fixture.
- Any test that needs this object simply names it as an argument.
- Pytest automatically injects the prepared dictionary.

This keeps your test files readable and reduces repetitive setup code.

Parametrized Tests

Sometimes one test scenario isn't enough. You want to run the same test logic over multiple inputs: different prices, various user roles, or multiple mathematical cases. Pytest includes a feature called parametrization, which makes this effortless.

Example: Testing Multiple Discount Scenarios

Save the following as **test_discount.py**:

```
# test_discount.py
import pytest

def apply_discount(price, percent):
    return price - (price * percent / 100)

@pytest.mark.parametrize(
    "price,percent,expected",
    [
        (100, 10, 90),
        (200, 50, 100),
        (50, 0, 50),
    ]
)
def test_apply_discount(price, percent, expected):
    assert apply_discount(price, percent) == expected
```

Listing 419 — Example: Testing Multiple Discount Scenarios

How Parametrization Works

- The decorator lists tuples of (price, percent, expected) values.
- Pytest generates three separate test cases.
- Each row appears in the test output, making debugging easier.

This is especially powerful in numerical computing, data science, and machine-learning pipelines.

Running Pytest in Different Environments

Pytest works seamlessly both locally and inside notebooks.

Running from the Terminal

Run all tests:

```
python -m pytest
```

Listing 420 — Running from the Terminal

Verbose output:

```
python -m pytest -v
```

Listing 421 — Running from the Terminal

Run a single file:

```
python -m pytest test_discount.py
```

Listing 422 — Running from the Terminal

Run one test function:

```
python -m pytest test_discount.py::test_apply_discount
```

Listing 423 — Running from the Terminal

Running Inside Jupyter Notebook

Keep application code and tests in the named `.py` files. From a notebook, use its own interpreter to run those files:

```
import subprocess
import sys

subprocess.run([sys.executable, "-m", "pytest", "test_calculator.py", "-q"], check=True)
```

Listing 424 — Running Inside Jupyter Notebook

This runs pytest in a separate process; it does not discover unsaved test functions in the current notebook cell. The expected result is one passed test after the earlier files have been saved.

A More Realistic Testing Example

Let's combine everything: application code, fixtures, assertions, and error handling.

Application Code: `bank.py`

Create a file named **`bank.py`**:

```
# bank.py
class Account:
    def __init__(self, name, balance=0):
        if balance < 0:
            raise ValueError("Initial balance cannot be negative")
        self.name = name
        self.balance = balance

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("Deposit must be positive")
        self.balance += amount

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("Withdrawal must be positive")
        if amount > self.balance:
            raise ValueError("Insufficient funds")
        self.balance -= amount
```

Listing 425 — Application Code: `bank.py`

This teaching model accepts numeric amounts, requires a nonnegative starting balance and positive deposits and withdrawals, and rejects withdrawals above the balance. We use integer amounts to

focus on those rules.

Test File: test_bank.py

Now create **test_bank.py**:

```
# test_bank.py
import pytest
from bank import Account

@pytest.fixture
def account():
    return Account("Alice", balance=100)

def test_initial_balance(account):
    assert account.balance == 100

def test_deposit(account):
    account.deposit(50)
    assert account.balance == 150

def test_withdraw(account):
    account.withdraw(30)
    assert account.balance == 70

def test_withdraw_insufficient_funds(account):
    with pytest.raises(ValueError):
        account.withdraw(200)
    assert account.balance == 100

def test_negative_initial_balance():
    with pytest.raises(ValueError):
        Account("Alice", balance=-1)

@pytest.mark.parametrize("amount", [0, -5])
def test_invalid_deposit_preserves_balance(account, amount):
    with pytest.raises(ValueError):
        account.deposit(amount)
    assert account.balance == 100

@pytest.mark.parametrize("amount", [0, -5])
def test_invalid_withdrawal_preserves_balance(account, amount):
    with pytest.raises(ValueError):
        account.withdraw(amount)
    assert account.balance == 100
```

Listing 426 — Test File: test_bank.py

What These Tests Cover - The fixture creates a fresh account for every test. - The tests check normal behavior (deposit, withdraw) and error handling. - The additional cases reject invalid amounts and check that rejected operations leave the balance unchanged.

Example Output

```
test_bank.py .....
9 passed in ...s
```

A clean, readable, and expressive test suite — exactly what pytest is designed for.

Fixtures provide fresh data, parametrization checks several inputs with one test function, and assertion reports explain a mismatch. These tools help you investigate behavior; passing tests cover only the expectations and inputs you actually checked.

With error handling and testing in place, we can examine how Python coordinates work with threads, processes and asynchronous I/O.

Concurrency & Parallelism

Modern software rarely does just one thing at a time. A data-science pipeline might download files, clean datasets, and update a database. A web server might handle hundreds of users simultaneously. A desktop application might respond to clicks while performing background computations.

To keep programs responsive and efficient, Python gives us tools for **running multiple tasks at once** — a world known as **concurrency** and **parallelism**. - **Concurrency** is about managing many tasks so they appear to run simultaneously. Only one task may run at a time, but the program switches quickly between them whenever one must wait (for example, on network or file I/O). - **Parallelism**, in contrast, is about running tasks at the same moment on different CPU cores.

Python provides three major approaches to handle this multitasking world:

- **Threads** — ideal for **I/O-bound** tasks that spend time waiting.
- **Processes** — ideal for **CPU-bound** tasks that need real parallel performance.
- **Asyncio** — ideal for running many lightweight asynchronous operations inside a single thread.

Before diving into these tools, we must understand the key architectural piece that shapes Python's multitasking behavior: the **GIL**.

The Global Interpreter Lock (GIL)

CPython is the standard Python implementation used by these examples. In a conventional GIL-enabled CPython process, the **Global Interpreter Lock** allows only one thread to execute Python bytecode at a time. This helps coordinate access to interpreter-managed objects; it does not make every operation on shared application data safe.

Waiting and Computing

A thread can release the GIL while waiting for supported I/O operations. Other threads can then make progress. That is why threads can help a program overlap network waits or file operations.

CPU-bound loops written in Python usually do not gain multi-core execution from threads in a GIL-enabled process. Separate processes can run such work in parallel, but creating workers and transferring data have costs. Some numerical libraries run native code that releases the GIL, so they need separate measurement.

The Implementation Matters

The GIL is not a rule of the Python language. CPython offers optional free-threaded builds from Python 3.13, and extensions can affect whether the GIL is enabled. The examples here assume a conventional GIL-enabled interpreter unless stated otherwise. See [Python's free-threading documentation](#).

Choose from the workload: threads can overlap waiting; processes can distribute Python computation; asynchronous I/O lets tasks cooperate through an event loop. Measure the actual program rather than assuming more workers are faster.

We will begin with a small thread-pool simulation, then run a portable process-pool example.

Threads: Best for I/O-Bound Tasks

Threads are small workers that run inside the same Python process and share memory. They're lightweight, easy to create, and ideal when work is mostly waiting.

Imagine downloading ten web pages. Suppose each download spends most of its time waiting for the server to respond. Threads can overlap this waiting time, finishing the job much earlier.

Real-World Analogy

A single cook in a kitchen needs to wait while water boils. But if you hire several assistants, they can: - set the table - fetch ingredients - prepare sauces

...all while the water still boils.

This is exactly how threads hide waiting time.

Example: Parallel “Network” Requests Using Threads

This example simulates network calls using `time.sleep()`.

If each task waits, threads help overlap that waiting.

`pool.submit` schedules a function and returns a **future**, an object representing its eventual result. `as_completed` supplies completed futures; `result()` retrieves a value or raises the task's exception.

Code Example

```
import time
from concurrent.futures import ThreadPoolExecutor, as_completed

def fake_fetch(url, delay=0.2):
    """Simulate a slow network request by sleeping briefly."""
    time.sleep(delay)
    return f"OK {url} ({delay:.1f}s)"

urls = [f"https://example.org/page/{i}" for i in range(5)]

start = time.perf_counter()

# Create a pool of threads
with ThreadPoolExecutor(max_workers=5) as pool:
    # Schedule each fetch in its own thread
    futures = [
        pool.submit(fake_fetch, u, 0.2 + (i % 3) * 0.1)
        for i, u in enumerate(urls)
    ]
    # Print results as they finish
    for fut in as_completed(futures):
        print(fut.result())

elapsed = time.perf_counter() - start
print(f"Total time ~ {elapsed:.2f}s (overlapping waits)")
```

Listing 427 — Threads: Best for I/O-Bound Tasks

What's Happening?

- Each call to `fake_fetch()` sleeps for a moment (like waiting on a slow network).

- The thread “releases” the CPU during sleep.
- Another thread immediately starts its work.
- All threads overlap their waiting time, so the whole job finishes quickly.

Change `max_workers=5` to `max_workers=1` in the executor call and run the complete program again.

Now the script must complete each request sequentially—much slower.

This is the heart of concurrency: **hiding wait time**.

Completion order and measured time can vary; these are simulated waits, not HTTP requests. The next section changes the workload from waiting to computing.

Processes: Best for CPU-Bound Tasks

In a GIL-enabled CPython interpreter, separate processes can run Python computations on different CPU cores. Each worker has its own memory, so arguments and results must be transferred. The cost can outweigh the benefit for small jobs.

`ProcessPoolExecutor` manages a pool of workers. Its `map` sends work to them and returns results in input order. Define worker functions at module level so child processes can import them.

Count Complete Words

Save the complete example as `process_word_count.py` and run `python process_word_count.py` in a terminal. The `if __name__ == "__main__"` guard keeps child processes from starting another pool when they import the file. This pattern works with spawn-based startup on Windows and macOS; do not paste this pool definition directly into a notebook and assume its functions are importable.

Split the text into words before making chunks. Splitting arbitrary character ranges could cut a word in half and produce incorrect counts.

```
from concurrent.futures import ProcessPoolExecutor
from collections import Counter
import time

def count_words(words):
    return Counter(word.lower() for word in words)

def merge(counters):
    total = Counter()
    for counter in counters:
        total.update(counter)
    return total

if __name__ == "__main__":
    words = ("To be or not to be " * 20_000).split()
    chunks = [words[i:i + 20_000] for i in range(0, len(words), 20_000)]
    start = time.perf_counter()
    with ProcessPoolExecutor(max_workers=2) as pool:
        partials = list(pool.map(count_words, chunks))
    total = merge(partials)
    print(total.most_common(3))
    print(f"Finished in {time.perf_counter() - start:.2f}s")
```

Listing 428 — Count Complete Words

Output — the elapsed time varies:

```
[('to', 40000), ('be', 40000), ('or', 20000)]
Finished in ...s
```

Each chunk contains complete words. Workers count them independently; `merge` adds their Counters. Tied counts follow first appearance in the merged data, which is deterministic here because `map` retains chunk order.

Try it: time the same chunks with a direct loop. Then import `ThreadPoolExecutor` and use it in place of `ProcessPoolExecutor`. Compare counts before comparing time. In this example, Python computation, worker startup and data transfer all affect the result; no particular speedup is

guaranteed.

With threads and processes established, asynchronous I/O offers another way to coordinate tasks that wait.

Async I/O with asyncio

When a program must handle *massive numbers of small, I/O-heavy tasks*, traditional threading begins to show its limits. Each thread carries memory overhead, and switching between thousands of them can slow a program down.

That's where `asyncio` enters the picture.

`asyncio` is Python's framework for **asynchronous, event-driven programming**. Instead of creating many threads, it runs all tasks inside **one single thread**, but lets those tasks **cooperate** by yielding control whenever they need to wait.

This model is perfect for programs that: - wait for network responses - handle thousands of client connections - coordinate APIs or microservices - manage background events or timers

The key ingredients are: - **coroutines**, defined with `async def`, which act like lightweight tasks - **`await`**, which pauses a coroutine so others can run - **an event loop** that schedules and coordinates all coroutines

Think of it as a well-organized single-lane road where every driver takes turns smoothly, rather than a chaotic multi-lane highway.

How Asyncio Works (Intuition First)

When a coroutine performs an operation that would normally block—like waiting for a network response—it does not freeze the entire program. Instead, it says: *“I’m waiting; someone else can run now.”*

`await` works with an awaitable operation such as `asyncio.sleep()`. Calling a blocking synchronous function inside a coroutine would still block that event-loop thread.

While one coroutine pauses, another continues, and so on.

The event loop orchestrates all of this, switching between coroutines extremely quickly.

The result: **true concurrency without threads**, ideal for workloads that spend most of their time waiting.

Example: Running Multiple Asynchronous Tasks

Below is a simple but representative example.

It simulates several asynchronous “fetch” operations by using `asyncio.sleep()`, which behaves like a non-blocking network wait.

Before showing the code, note the structure: - `async_fetch()` is a coroutine that simulates an asynchronous task. - `main()` creates many such tasks and runs them together. - `asyncio.gather()` waits for all of them to finish. - `asyncio.run()` starts the event loop.

Save this program as `async_demo.py` and run it from a terminal. In a Jupyter notebook, an event loop is already running; replace the final `asyncio.run(main())` with `await main()`. The list below holds

coroutine objects; `asyncio.gather` schedules them and returns results in their original order.

Code Example

```
import asyncio
import random
import time

async def async_fetch(name):
    """Simulate asynchronous I/O with asyncio.sleep()."""
    delay = 0.1 + random.random() * 0.2
    await asyncio.sleep(delay)      # Non-blocking delay
    return f"OK {name} ({delay:.2f}s)"

async def main():
    # Launch multiple coroutines at the same time
    tasks = [async_fetch(f"task-{i}") for i in range(5)]
    start = time.perf_counter()

    # Run all tasks concurrently and collect their results
    results = await asyncio.gather(*tasks)

    for r in results:
        print(r)

    elapsed = time.perf_counter() - start
    print(f"Total time ~ {elapsed:.2f}s (async I/O)")

# Start the event loop
asyncio.run(main())
```

Listing 429 — Example: Running Multiple Asynchronous Tasks

What's Happening?

- **All tasks run inside one thread**

This example runs on one event-loop thread. Tasks can still interleave around `await`, so shared mutable state may require coordination.

- **Coroutines pause themselves with `await`**

`await asyncio.sleep()` doesn't block; it yields control back to the event loop.

- **Other coroutines run while one waits**

Tasks take rapid turns, creating smooth, efficient concurrency.

- **The waits overlap**

The example usually takes roughly the longest wait plus scheduling and printing overhead. Exact timings depend on the environment.

- **Designed for thousands of I/O-bound tasks**

Coroutines are extremely lightweight, enabling large-scale systems like web servers, chat apps, and async API clients.

Project: Parallel Word Counter

Combine files, regular expressions and a process pool to count words across several documents. The pool distributes files among two workers; it does not dedicate one CPU core to every file.

Define the Counting Rule

For this project, a word is a sequence matched by `\b\w+\b` after lowercasing. This includes Unicode letters, digits and underscores, and separates punctuation. It is a simple tokenization rule, not a linguistic definition for every language. The `: Path` and `-> Counter` annotations document the argument and return types; they do not validate them automatically.

Save the complete program as `parallel_word_counter.py` in a new practice folder. It creates three demonstration files, overwriting those same files if run again. The explicit file list keeps unrelated files out of the results.

```
from concurrent.futures import ProcessPoolExecutor
from pathlib import Path
from collections import Counter
import re
import time

def count_file(path: Path) -> Counter:
    text = path.read_text(encoding="utf-8")
    return Counter(re.findall(r"\b\w+\b", text.lower()))

def merge(counters):
    total = Counter()
    for counter in counters:
        total.update(counter)
    return total

if __name__ == "__main__":
    demo_dir = Path("demo_corpus")
    demo_dir.mkdir(exist_ok=True)
    (demo_dir / "a.txt").write_text("To be, or not to be.\n", encoding="utf-8")
    (demo_dir / "b.txt").write_text("To be! To learn. To build.\n", encoding="utf-8")
    (demo_dir / "c.txt").write_text("Build, learn, repeat.\n", encoding="utf-8")
    files = [demo_dir / name for name in ("a.txt", "b.txt", "c.txt")]
    start = time.perf_counter()
    with ProcessPoolExecutor(max_workers=2) as pool:
        results = list(pool.map(count_file, files))
    total = merge(results)
    print("Top 5 words:", total.most_common(5))
    print(f"Completed in {time.perf_counter() - start:.2f}s")
```

Listing 430 — Define the Counting Rule

Output — timing varies:

```
Top 5 words: [('to', 5), ('be', 3), ('learn', 2), ('build', 2), ('or', 1)]
Completed in ...s
```

The words `be,`, `be.` and `be!` all become `be`. Counts for `learn` and `build` tie; their order follows first occurrence as the file results are merged. `or` is the first of the words with count 1.

Check Before Measuring

Count the same files sequentially and assert that the merged Counter is identical. Test an empty file, mixed capitalization and punctuation. A decoding error should be reported rather than silently discarding text.

Change it: add more documents explicitly to the file list. Measure sequential and process-pool versions using the same inputs. These tiny files may run faster sequentially because worker startup costs more than counting; performance is a result to investigate, not a promised benefit.

The next chapter uses similar records and text-processing ideas with data obtained through HTTP and HTML parsing.

APIs & Web Scraping

Modern research relies on data that increasingly lives online — from newspaper archives and research portals to museum catalogs and digital libraries. Python offers two complementary ways to access this information:

1. APIs

Structured, machine-friendly endpoints that return data (often JSON) in a predictable format. When available, APIs are the safest, cleanest, and most reliable way to obtain data.

2. Web Scraping

Extracting information directly from a webpage's **HTML** when no API exists. Scraping is more flexible, but requires careful parsing — and ethical responsibility.

This chapter introduces the full toolkit: - How **HTTP** works - How to fetch online content - How to parse **JSON** (API responses) - How to understand and interpret **HTML** - How to scrape webpages safely and responsibly - How to combine all of these techniques into real-world workflows

At the end, a complete mini-project will show all these concepts working together.

Understanding HTTP

HTTP, the Hypertext Transfer Protocol, defines how a client and server exchange requests and responses. Your Python program is the **client**: it requests a resource identified by a URL. The **server** responds with a status code, headers describing the response, and usually a body containing data such as HTML or JSON.

Read the Status Code

A status code describes the outcome of a request:

Code	Meaning
200 OK	The request succeeded.
404 Not Found	The server could not find the requested resource.
429 Too Many Requests	The client has exceeded a request limit.
500 Internal Server Error	The server encountered an unexpected problem.

Table 11 — Read the Status Code

These codes help your program decide whether to process the response or handle a failure. A successful response still needs to contain the data your program expects.

Make a GET Request

A **GET request** asks for a resource without requesting a change to it. The third-party `requests` package provides `get()` for this purpose. Install it in your active environment with `python -m pip install requests` if needed.

This complete example uses JSONPlaceholder, a service providing sample data for practice [1]. It requires an internet connection.

```
import requests

url = "https://jsonplaceholder.typicode.com/posts/1"
response = requests.get(url, timeout=10)
response.raise_for_status()
print("Status code:", response.status_code)
print("Content type:", response.headers.get("Content-Type"))
```

Listing 431 — Make a GET Request

Expected output — requires a successful live request

```
Status code: 200
Content type: application/json; charset=utf-8
```

`get()` returns a response object. `timeout=10` limits how long Requests waits for a connection or incoming data; it is not a fixed limit on the whole download. `raise_for_status()` raises an exception for an HTTP error, so the program does not process an error page as valid data. Connection failures and timeouts also raise exceptions. The two `print` calls then show the status and the body's declared

format; a server can change its headers.

We can now fetch a response. After looking at HTML structure, we will extract useful values from JSON and HTML rather than printing their raw contents.

Reference

- JSONPlaceholder. [Guide](#).

HTML Basics

Once your Python program can fetch information from the web, the next challenge is understanding **what** it has retrieved.

When a website doesn't offer an API — which is very common — the content arrives as **HTML**, the structural language of the web.

To a browser, HTML describes how a page should look.

To a human, it's text sprinkled with angle brackets.

To Python, it's a rich, hierarchical document that can be navigated and analyzed.

Understanding just a few core ideas will make web pages far less mysterious.

What HTML Really Is

HTML — HyperText Markup Language — is not a programming language. It's a description of a page: a hierarchy of nested elements that serve different roles.

A small example:

```
<h1>News Headlines</h1>
<p>Python 3.12 Released!</p>
```

Listing 432 — What HTML Really Is

To a human reading it in the browser, this becomes bold text at the top followed by a paragraph. To a machine, it becomes a tree of tags: - `<h1>` — a main heading - `<p>` — a paragraph

Nearly every component on a webpage — links, images, lists, tables — is represented as a combination of tags.

Learning the most common ones is enough for most scraping tasks: - `<h1>`, `<h2>` ... for headings - `<p>` for paragraphs - `<a>` for links - `<ul>` and `<li>` for lists and list items - `<table>`, `<tr>`, `<td>` for tables

Viewing HTML Behind a Real Page

Let's visit a simple site using Python and see what HTML looks like when downloaded.

```
import requests

url = "https://example.org"
resp = requests.get(url, timeout=10)
resp.raise_for_status()

print(resp.text[:300], "...")
```

Listing 433 — Viewing HTML Behind a Real Page

Running this prints the actual *raw HTML* returned by the server.

It contains everything the browser uses to build the visible page: tags, attributes, nested elements.

You don't need to memorize any of it.

You only need to recognize that HTML is a structured document Python can parse.

Next, we'll see how to extract information from this structure.

HTML Is a Tree, Not a String

Although `resp.text` looks like a long string, conceptually it represents a document tree.

For example, the following snippet

```
<ul>
  <li>Python 3.12 Released</li>
  <li>AI Research Advances</li>
</ul>
```

Listing 434 — HTML Is a Tree, Not a String

means:

- A `<ul>` (unordered list) contains two `<li>` (list items) each containing text

This tree structure is what makes scraping possible.

Your program doesn't "search the text"; it navigates the tree, picking out exactly the elements you want.

Fetching and Parsing Online Data

Fetching retrieves content; **parsing** turns that content into a structure your program can inspect. JSON commonly becomes Python dictionaries and lists. HTML becomes a tree of elements that we can search. We will separate these steps so that parsing can be tested without a network connection.

Parse JSON

The standard-library `json` module converts JSON text into Python values. This complete example uses a small local response, so its result is reproducible:

```
import json

response_text = '{"id": 1, "title": "Learning Python", "userId": 2}'
post = json.loads(response_text)
print(post["id"])
print(post["title"])
```

Listing 435 — Parse JSON

Output

```
1
Learning Python
```

`loads()` reads JSON from a string. Here it returns a dictionary, so the keys `"id"` and `"title"` select the values to print. These are sample values, not a recording of a live service.

For a live JSONPlaceholder response, Requests can parse the body directly [1]:

```
import requests

response = requests.get(
    "https://jsonplaceholder.typicode.com/posts/1", timeout=10
)
response.raise_for_status()
post = response.json()
print("Post:", post["id"])
print("Author ID:", post["userId"])
```

Listing 436 — Parse JSON

Expected output — requires a successful live request

```
Post: 1
Author ID: 1
```

`response.json()` parses the response body. It can fail if the body is not valid JSON, even when the HTTP request succeeded. The selected endpoint returns one post; endpoints returning several records commonly produce a list that you can loop over. Check an API's documentation for its actual structure and request limits.

Parse HTML

Beautiful Soup is a third-party library for searching HTML. Install it in the active environment with `python -m pip install beautifulsoup4`. The imported name is `bs4`.

This complete example uses a small fictional news page:

```
from bs4 import BeautifulSoup

html = """
<html><body>
  <h1>Latest News</h1>
  <ul>
    <li class="headline"><a href="/story1">Python Workshop Opens</a></li>
    <li class="headline"><a href="/story2">New Tools for Data Science</a></li>
    <li class="headline"><a href="/story3">Digital Humanities in Practice</a></li>
  </ul>
</body></html>
"""

soup = BeautifulSoup(html, "html.parser")
headlines = []
for item in soup.find_all("li", class_="headline"):
    link = item.find("a")
    if link is not None and link.has_attr("href"):
        headlines.append((link.get_text(strip=True), link["href"]))

for title, url in headlines:
    print(f"{title} -> {url}")
```

Listing 437 — Parse HTML

Output

```
Python Workshop Opens -> /story1
New Tools for Data Science -> /story2
Digital Humanities in Practice -> /story3
```

BeautifulSoup builds a tree using Python's `html.parser`. `find_all()` returns the matching list items; `class_` searches the HTML class attribute because `class` itself is a Python keyword. Within each item, `find()` looks for a link and can return `None`. The condition checks that a link and its `href` exist before reading them. `get_text(strip=True)` extracts text and trims whitespace; `link["href"]` reads the destination. The resulting pairs are ordinary Python data.

To parse a downloaded page, pass a successful response's text to `BeautifulSoup` instead of the local `html` string. Search rules must match that page's structure. Content loaded later by JavaScript may be absent from the downloaded HTML.

Before combining these steps into a scraper, we need to consider the site's access rules and the people whose data it contains.

References

- JSONPlaceholder. [Guide](#).
- BeautifulSoup. [Searching the tree](#).

Ethics & Responsible Web Scraping

Automated requests use a site's resources and may collect information about people. Before writing a scraper, check whether automated access and your intended use are permitted. Prefer a documented API when it provides the data you need, and read its access, licensing and rate-limit policies.

Check Access Rules

A site can publish crawler instructions at its root, for example `https://example.org/robots.txt`. The following illustrative file asks all crawlers to avoid `/private/`:

```
User-agent: *  
Disallow: /private/  
Allow: /public/
```

`User-agent: *` identifies rules for crawlers without a more specific matching group. `Disallow` and `Allow` specify paths. These are crawler instructions, not access authorization; the robots standard explicitly distinguishes them from security controls [1]. Check the site's terms and applicable permissions as well. Do not bypass authentication, paywalls or other access restrictions.

Limit Requests and Identify Your Program

Follow the service's published request limits. Cache results, fetch only what you need, and avoid repeated downloads while testing your parser. If the service responds with 429, stop and follow its retry instructions, including `Retry-After` when present. A fixed one-second pause is not a universal guarantee of acceptable traffic.

Use a meaningful `User-Agent` when the service requests one. Wikimedia, for example, asks API clients to identify their application and provide contact information [2]. Use your own real details rather than impersonating a browser or copying somebody else's contact address.

Collect and Reuse Carefully

Public visibility does not imply unrestricted reuse. Avoid collecting unnecessary personal information, check content licenses, and retain the source information needed for attribution. Share only what your permissions and the applicable license allow.

We will first test the next project's parser with local HTML. That lets us check our code repeatedly without sending repeated requests to a real website.

References

- Koster, M. et al. (2022). [RFC 9309: Robots Exclusion Protocol](#).
- Wikimedia. [API access policy](#).

Project: Fetch & Parse Real Website Data

Build a small collection of article titles and introductory paragraphs. We will test extraction with local HTML, then connect the same function to live Wikipedia pages. This separates parsing errors from network failures and keeps repeated tests offline.

The project uses `requests` and `beautifulsoup4`, introduced in the previous sections. Install them in your active environment if necessary. Before fetching live pages, review Wikimedia's current access and content-reuse policies; its documented APIs are often preferable for a larger application [1].

Extract an Article

The extraction function accepts HTML and returns a dictionary. It searches for the article heading and the first non-empty paragraph in the article content. Missing elements raise a clear error rather than causing an attribute-access error.

```
from bs4 import BeautifulSoup

def extract_article(html):
    soup = BeautifulSoup(html, "html.parser")
    heading = soup.find("h1")
    content = soup.select_one("#mw-content-text")
    if heading is None or content is None:
        raise ValueError("Article heading or content is missing")
    for paragraph in content.find_all("p"):
        intro = paragraph.get_text(" ", strip=True)
        if intro:
            return {"title": heading.get_text(" ", strip=True), "intro": intro}
    raise ValueError("Article has no non-empty paragraph")
```

Listing 438 — Extract an Article

Defining the function produces no output. `select_one("#mw-content-text")` finds the element with that HTML ID. Passing a space to `get_text()` separates text from adjacent tags. The search is a simple extraction rule, not a guarantee that every Wikipedia layout puts the introduction first; inspect and adapt it when the page structure changes.

Test with Local HTML

Run the function definition above, then this example. The paragraphs are short, original test fixtures, not copied Wikipedia introductions.

```

samples = [
    ("Python (programming language)", "Python programs can process text and numbers."),
    ("Artificial intelligence", "AI systems perform tasks using computational models."),
    ("Web scraping", "A scraper extracts selected information from web pages."),
]
articles = []
for title, intro in samples:
    html = f'<h1>{title}</h1><div id="mw-content-text"><p></p><p>{intro}</p></div>'
    articles.append(extract_article(html))

for article in articles:
    print(article["title"])
    print(article["intro"])

```

Listing 439 — Test with Local HTML

Output

```

Python (programming language)
Python programs can process text and numbers.
Artificial intelligence
AI systems perform tasks using computational models.
Web scraping
A scraper extracts selected information from web pages.

```

The empty first paragraph is deliberately skipped. Each successful extraction adds one dictionary to `articles`; the final loop reads its two fields. Change a fixture's heading or paragraph and verify that the extracted value changes. Then remove the heading and check that the function raises its stated `ValueError`.

Connect the Parser to Live Pages

Keep `extract_article` above this code in the same script. It fetches three pages sequentially. Enter a real contact email or project URL when prompted; the program uses it to identify the client. Check the site's current crawler instructions, access rules and request limits before running it.

```

import time
import requests

contact = input("Your contact email or project URL: ").strip()
if not contact:
    raise ValueError("Contact information is required")
headers = {"User-Agent": f"ArticleStudy/1.0 ({contact})"}
urls = [
    "https://en.wikipedia.org/wiki/Python_(programming_language)",
    "https://en.wikipedia.org/wiki/Artificial_intelligence",
    "https://en.wikipedia.org/wiki/Web_scraping",
]
articles = []
for url in urls:
    try:
        response = requests.get(url, headers=headers, timeout=10)
        response.raise_for_status()
        article = extract_article(response.text)
        article["url"] = url
        articles.append(article)
    except (requests.RequestException, ValueError) as error:
        print("Stopped:", error)
        break
    time.sleep(2)

for article in articles:
    print(article["title"])
    print(article["intro"][:200])
    print(article["url"])

```

Listing 440 — Connect the Parser to Live Pages

Output — depends on the live pages and connection

The program first asks for contact information. On success it prints a title, up to 200 introduction characters and a source URL for each retrieved page. On failure it prints Stopped: and the error, then displays any records already collected. No fixed live transcript is given because pages, access conditions and error messages can change.

The exception handler covers HTTP, connection and parsing failures. It stops after a failure rather than repeatedly requesting an unavailable resource. The two-second pause is an example delay, not a substitute for the service's current limits. The saved URL supports source tracking; check licensing and attribution requirements before redistributing extracted text.

Apply: save the collected dictionaries as JSON using the file-handling techniques introduced earlier. Test saving with the local fixtures first. Your program should preserve the titles and introductions and handle a missing heading without inventing data.

Reference

- Wikimedia. [API access policy](#) and [REST API documentation](#).

Learn Python by understanding what your code does.

Start with a few instructions. Build the habits to read, test and explain a complete program.

Python Programming from First Principles takes you from values, text and collections to functions, classes and practical applications. Small examples show each idea in action; purposeful changes and coding challenges help you make it your own.

Explore object-oriented design with UML and build a University Course System. Then extend your toolkit with files, tests, concurrency and web data. AI assistance remains optional: your understanding and evidence come first.

- Build a foundation in Python without prior programming experience.
- Read runnable examples with clearly separated output.
- Practise through changes, challenges and connected projects.
- Use companion files to continue experimenting on your own computer.

Prof. Philippe Tamla

From First Principles to Intelligent Solutions

kcslab.org