Python: the practical notebook

Beginner foundations. Useful projects. A map of the possibilities.

Start with one line. Build something useful.

By Privatus Cosmas (Priver) / depriver.tech

A hands-on handbook for people learning to program: clear explanations, runnable examples, practice prompts and a small command-line project you can finish yourself.

Inside these pages

Learn variables, decisions, loops and functions. Work with files, JSON, CSV and SQLite. Test and debug your code. Then explore automation, web APIs, data analysis, AI, science, desktop apps and more.

> Type it. Predict the result. Run it. Change it. Explain it in your own words.

Edition 1.0 / 19 October 2026 Core examples use Python 3.11 or newer. Optional libraries require their own supported Python versions.

A practical foundation, not an encyclopedia

This book surveys the major uses of Python. Each specialist field has more to learn; no single beginner document covers every library or use case.

Your learning route

Read in order first; return to individual lessons later

01 / Start here - pages 3-5
Install Python, run a script and create a separate environment.

02 / Learn the language - pages 6-15
Values, collections, decisions, loops, functions, modules and errors.

03 / Work with real data - pages 16-21
Files, CSV, JSON, classes, testing, SQLite and command-line tools.

04 / Build a project - pages 22-24
A persistent expense notebook with validation and tests.

05 / Explore the uses - pages 25-32
Automation, APIs, analytics, AI, science, creative tools and operations.

06 / Keep improving - pages 33-36
Professional habits, exercises with answers, a study plan and sources.

Everything through the core project uses the standard library. Optional web and data examples are clearly labelled.

01 / Think like a programmer

A program turns a clear instruction into a repeatable action

A program is a sequence of instructions. Input is information it receives; processing is what it does with that information; output is the result. Before writing code, explain the task in everyday language.

```
prices = [1200, 800, 500]
total = sum(prices)
print("Total:", total, "TZS")
```

This list holds three prices. sum adds them; print displays Total: 2500 TZS. The equals sign assigns a value to a name. Python executes these statements from top to bottom. A name such as total makes the meaning easier to understand than repeating the calculation everywhere.

Practice: add one more price. Predict the new total before running the script.

Break down a larger task

For a spending notebook: accept a category and amount, check that they are valid, save the entry, then show a total. Each step can become a small function. Begin with the smallest useful version; add complexity after it works.

02 / Install and run

Terminal commands and Python code belong in different places

Install a supported Python 3 release from python.org/downloads or your operating system package manager. Open a terminal. On Windows, try py --version; on macOS or Linux, try python3 --version. Do not use Python 2. Create a folder called python-notebook.

```
# Save this Python code as hello.py
name = "Priver"
print(f"Hello, {name}!")
```

In the terminal, change into the folder with cd and run py hello.py on Windows or python3 hello.py on macOS/Linux. Expected output: Hello, Priver! A .py file is a script. The interactive prompt is useful for quick experiments, but save repeatable work in files.

If Python cannot find hello.py, check your current folder and the exact filename. Avoid saving it as hello.py.txt.

Read the notation

Text after # is a comment. Quoted text is a string. Parentheses call a function. Examples in blue boxes are Python unless explicitly labelled as terminal commands. Never copy the >>> prompt from a tutorial into a script.

03 / Keep projects separate

A virtual environment holds one project's installed packages

Create an environment inside the project folder. Use python3 on macOS/Linux or py on Windows for the creation command. After activation, this book uses python to mean the environment interpreter.

```
# Terminal: macOS / Linux
python3 -m venv .venv
source .venv/bin/activate
python --version

# Terminal: Windows Command Prompt
py -m venv .venv
.venv\Scripts\activate.bat
python --version
```

Use the commands for your operating system, not both sets. In PowerShell, activation is .venv\Scripts\Activate.ps1. If policy blocks activation, invoke .venv\Scripts\python.exe directly. On some Linux systems you need the distribution's python3-venv package first.

Add .venv/ and __pycache__/ to .gitignore. Recreate environments from dependency records instead of committing them.

https://docs.python.org/3/library/venv.html

04 / Values and names

Choose the right kind of value for the job

An int holds a whole number, a float represents an approximate real number, a str holds text and a bool is True or False. None represents the absence of a value. Python names refer to objects; assignment does not declare a permanently fixed type.

```
quantity = 3
price = 1500
item = "Notebook"
available = True
total = quantity * price
print(item, total)
print(type(quantity).__name__)
print(7 / 2, 7 // 2, 7 % 2)
print(2 ** 3)
```

Output: Notebook 4500, then int, then 3.5 3 1, then 8. / divides; // uses floor division; % gives a remainder; ** raises to a power. Use descriptive snake_case names such as total_cost. Avoid naming a variable list or str because that hides a built-in name.

For these money exercises use whole TZS integers. For exact fractional money, use Decimal with a documented rounding rule.

05 / Text and user input

Input arrives as text until you convert it

String methods return useful transformed values. strip removes surrounding whitespace; lower normalises letter case. An f-string inserts values into text. Indexing starts at zero; slices stop before their end position.

```
name = "  Asha  "
clean = name.strip()
print(f"Hello, {clean}!")
print(clean[0], clean[:2])
print("python".upper())
print("web,data,ai".split(","))
```

Expected: Hello, Asha!; A As; PYTHON; then a list of three words. A string is immutable: clean.upper() does not change clean unless you assign the returned value. input("Your age: ") returns a string; int(…) converts suitable text to a whole number.

Practice: clean " MBEYA " into "Mbeya" with strip and title. Predict what int("two") does; we handle that error on page 15.

06 / Make decisions

Use conditions to choose the next action

An if block runs when its condition is true. elif checks another possibility; else handles what remains. Indent the body consistently with four spaces. Comparisons include ==, !=, <, <=, > and >=. = assigns; == compares.

```
budget = 5000
cost = 4200
if cost < 0:
    print("Invalid cost")
elif cost <= budget:
    print("Within budget")
else:
    print("Over budget")

is_member = True
print(is_member and cost <= budget)
```

This prints Within budget and True. and requires both conditions; or needs at least one; not reverses a truth value. Empty lists, empty strings, zero and None are false-like in conditions. Prefer explicit checks when zero is a valid value.

Practice: try cost = 6000, cost = -1 and cost = 5000. Explain why the order of the conditions matters.

07 / Lists and tuples

Keep related values in an ordered collection

A list is an ordered, mutable sequence. append adds one item. A tuple is an ordered sequence whose elements cannot be reassigned. Use a list for a growing collection and a tuple for a fixed grouping such as a coordinate.

```
items = ["pen", "book"]
items.append("bag")
print(items[0], items[-1])
print(items[:2])
print(len(items))
copy = items.copy()
copy.append("ruler")
print(len(items), len(copy))
location = (-8.9, 33.4)
latitude, longitude = location
```

Expected: pen bag, then ['pen', 'book'], then 3 and finally 3 4. items.copy() makes a shallow copy; nested objects are still shared. Writing copy = items would make both names refer to the same list. An out-of-range index raises IndexError.

Practice: replace "pen" with "pencil", then remove "bag". Check the list after each operation.

08 / Dictionaries and sets

Look up values by meaning rather than position

A dictionary maps unique keys to values. A set stores distinct elements and supports membership checks. Dictionary keys and set members must be hashable; strings and integers are common choices.

```
student = {"name": "Asha", "score": 82}
student["city"] = "Mbeya"
print(student["name"])
print(student.get("phone", "Not supplied"))
for key, value in student.items():
    print(key, value)

skills = {"Python", "SQL", "Python"}
print(len(skills))
print("SQL" in skills)
```

The dictionary contains three fields. get returns a fallback when a key is absent, while square-bracket lookup raises KeyError. The set has two elements because Python appears only once. Do not rely on a set's display order.

Practice: model one shop item with name, quantity and price keys. Compute quantity times price.

09 / Repeat with loops

Process each item without rewriting the same statement

A for loop visits items from an iterable. range(1, 4) produces 1, 2 and 3. enumerate supplies an index and value. A while loop repeats as long as its condition is true; ensure something changes so it can finish.

```
prices = [1000, 2500, 500]
for number, price in enumerate(prices, start=1):
    print(number, price)

remaining = 3
while remaining > 0:
    print("Attempts:", remaining)
    remaining -= 1

print(sum(prices))
```

The first loop prints three numbered prices; the second counts 3, 2, 1; the total is 4000. break leaves a loop immediately. continue skips the rest of one iteration. Prefer iterating directly over values when you do not need their positions.

Practice: print only prices above 1000. Then total those prices with a loop before trying sum.

10 / Functions that return

Name a small task and reuse it

A function groups a task behind a name. Parameters receive inputs; return sends a result to the caller. print only displays information. A function without an explicit return returns None.

```
def line_total(price: int, quantity: int = 1) -> int:
    if price < 0 or quantity < 0:
        raise ValueError("Use non-negative values")
    return price * quantity

first = line_total(1500, 3)
second = line_total(price=500)
print(first + second)
```

The result is 5000. The default quantity is 1; keyword arguments make calls clearer. Type hints explain intended types but do not enforce them at runtime. Variables assigned inside a function are normally local to it. Keep input/output at the edges and calculations in small functions.

Practice: write a function that converts minutes into seconds. Test zero, one and five minutes.

11 / Comprehensions & generators

Transform collections clearly; stream larger results

A comprehension builds a new collection from an existing iterable. Use it for short transformations; use an ordinary loop when the logic becomes hard to read. A generator yields values on demand instead of building the complete result immediately.

```
prices = [100, 200, 300, 400]
large = [p for p in prices if p >= 300]
print(large)

def squares(limit):
    for n in range(limit):
        yield n * n

values = squares(4)
print(list(values))
print(list(values))
```

The output is [300, 400], then [0, 1, 4, 9], then []. A generator is consumed as you iterate over it. Recreate it if you need another pass. A shallow copy of a collection and a generator both have different behaviour from a reusable list.

Practice: build a dictionary mapping each number from 1 to 4 to its square.

12 / Modules and packages

Split responsibilities into files you can understand

A module is importable Python code, often one .py file. A package groups modules. The standard library ships with Python; third-party packages are installed separately. Start with standard modules before adding dependencies.

```
from statistics import mean
from datetime import date

scores = [60, 75, 90]
print(mean(scores))
print(date(2026, 10, 19).isoformat())

def main():
    print("Notebook ready")

if __name__ == "__main__":
    main()
```

This prints 75, 2026-10-19 and Notebook ready when run as a script. The main guard lets another file import functions without automatically running the application. Put reusable calculations in a module such as totals.py and import them with from totals import line_total.

Avoid filenames such as json.py or statistics.py in your project: they can hide the standard-library modules.

13 / Errors are information

Read the final error line, then trace back to its cause

SyntaxError means Python could not parse the code. NameError often means an undefined name. TypeError means an operation received an unsuitable type. ValueError means a value is invalid for an otherwise appropriate operation.

```
raw = "not a number"
try:
    quantity = int(raw)
except ValueError:
    print("Enter a whole number")
else:
    print(quantity * 1500)
finally:
    print("Check complete")
```

The output is Enter a whole number and Check complete. Catch only exceptions you know how to handle. Broad except blocks can hide programming mistakes. When debugging, reduce the input, inspect values with repr, and check the first point where reality differs from your expectation.

Practice: replace raw with "3". Why does the else block run now? Do not use eval to convert user input.

14 / Files and paths

Save information so it survives the next run

pathlib represents file paths across operating systems. Use explicit UTF-8 encoding for text. Relative paths are resolved from the current working directory, which may differ from the script's directory. This example creates a local practice file and overwrites it on each run.

```
from pathlib import Path

folder = Path("practice_output")
folder.mkdir(exist_ok=True)
path = folder / "notes.txt"
path.write_text("Learn Python\nTry a project\n",
                encoding="utf-8")
with path.open(encoding="utf-8") as handle:
    for line in handle:
        print(line.strip())
```

with closes the file even when an error occurs. Use mode "a" to append and mode "w" to replace contents. Reading line by line is useful for large text files; read_text loads everything into memory. For data beside a script, start from Path(__file__).resolve().parent.

Practice: append a third line in your practice folder. Verify that the first two lines remain.

15 / CSV and JSON

Exchange structured data with other tools

CSV is useful for rows and columns. JSON represents nested dictionaries, lists, strings, numbers, booleans and null. CSV fields arrive as text and need conversion. JSON is data, not executable Python.

```
import csv
import io
import json

source = "category,amount\nfood,3000\ntravel,1500\n"
rows = list(csv.DictReader(io.StringIO(source)))
total = sum(int(row["amount"]) for row in rows)
report = {"currency": "TZS", "total": total}
encoded = json.dumps(report, indent=2)
print(encoded)
print(json.loads(encoded)["total"])
```

The JSON report contains a total of 4500; the final line prints 4500. json.dumps produces text; loads parses it. For real CSV files use open(..., newline="", encoding="utf-8"). Validate required fields, types and ranges after parsing; valid JSON is not automatically valid business data.

Practice: add a third CSV row. Then deliberately remove its amount and decide how your program should report the bad row.

16 / Classes and dataclasses

Group state with related behaviour when that helps

A class describes a kind of object. An instance is one object of that class. self refers to that instance. A dataclass generates routine methods such as initialisation, making simple records easier to express.

```
from dataclasses import dataclass

@dataclass
class Expense:
    category: str
    amount: int

    def label(self):
        return f"{self.category}: {self.amount} TZS"

entry = Expense("food", 3000)
print(entry.label())
```

The output is food: 3000 TZS. Each instance has its own field values. Use ordinary functions and dictionaries when they are simpler. Classes are helpful when a concept has state and operations that belong together. Type annotations do not validate the amount; add validation where data enters the system.

Practice: create two Expense instances. Change one amount and check that the other instance stays unchanged.

17 / Test behaviour

A good test tells you which promise stopped holding

Tests make expected behaviour explicit. Test ordinary input, boundary cases and invalid input. The standard library includes unittest. Save the example as test_total.py and run python -m unittest -v from the same folder.

```
import unittest

def total_amount(amounts):
    if any(x < 0 for x in amounts):
        raise ValueError("Negative amount")
    return sum(amounts)

class TotalTests(unittest.TestCase):
    def test_total(self):
        self.assertEqual(total_amount([2, 3]), 5)
    def test_empty(self):
        self.assertEqual(total_amount([]), 0)
    def test_invalid(self):
        with self.assertRaises(ValueError):
            total_amount([-1])
```

Expected: three tests pass. A test that only checks your code against itself proves little. Use known examples and real requirements. Automated tests complement manual checking; they cannot prove that every possible input is correct.

Practice: add a zero-value test. Change sum(amounts) to 0 and watch which test catches the mistake.

18 / A small database

SQLite stores related records in a local database

SQLite is included in Python through sqlite3. A table has rows and columns; SQL selects and updates those rows. This example uses an in-memory database, so it creates no persistent database file.

```
import sqlite3

con = sqlite3.connect(":memory:")
try:
    con.execute("CREATE TABLE expenses "
                "(category TEXT, amount INTEGER)")
    con.execute("INSERT INTO expenses VALUES (?, ?)",
                ("food", 3000))
    con.commit()
    row = con.execute(
        "SELECT SUM(amount) FROM expenses"
    ).fetchone()
    print(row[0])
finally:
    con.close()
```

The result is 3000. Replace :memory: with a filename for persistence. Use placeholders for values instead of inserting user input into SQL text. Commit writes intentionally; close connections when finished. A connection context manager manages transactions but does not itself close the connection.

Practice: insert travel = 1500, then query the total again. Expected: 4500.

https://docs.python.org/3/library/sqlite3.html

19 / Command-line tools

Turn a script into something you can run with arguments

argparse turns terminal arguments into named values and generates --help. Save this script as greet.py. Run python greet.py Asha --times 2. Terminal arguments are separate from interactive input(), which pauses to ask a question.

```
import argparse

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("name")
    parser.add_argument("--times", type=int, default=1)
    args = parser.parse_args()
    if not 1 <= args.times <= 10:
        parser.error("--times must be between 1 and 10")
    for _ in range(args.times):
        print(f"Hello, {args.name}!")

if __name__ == "__main__":
    main()
```

Expected: Hello, Asha! appears twice. Try python greet.py --help and python greet.py Asha --times 0. A useful CLI validates input, prints actionable messages and exits with a nonzero status when it cannot complete the task.

Practice: add an optional --city argument with default "Mbeya" and include it in the greeting.

20 / Project: expense notebook

A complete local application / Part 1: the data layer

Create a new folder. Save the code on this page and the next into one file named expenses.py, in that order. The companion ZIP includes the complete file. The app accepts whole positive TZS amounts and stores entries in a CSV beside the script.

```
import argparse
import csv
from pathlib import Path

DATA = Path(__file__).resolve().with_name("expenses.csv")

def add_expense(path, category, amount):
    category = category.strip()
    if not category or amount <= 0:
        raise ValueError("Category and positive amount required")
    new_file = not path.exists() or path.stat().st_size == 0
    with path.open("a", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        if new_file:
            writer.writerow(["category", "amount"])
        writer.writerow([category, amount])

def total_expenses(path):
    if not path.exists():
        return 0
    with path.open(newline="", encoding="utf-8") as f:
        return sum(int(r["amount"]) for r in csv.DictReader(f))
```

The functions receive a path, so tests can use temporary files. add_expense writes a header only for a new or empty file. total_expenses returns zero when there is no history yet.

21 / Project: make it usable

Part 2: append this below the previous page's code

```
def main():
    parser = argparse.ArgumentParser(
        description="A small TZS expense notebook")
    sub = parser.add_subparsers(dest="command", required=True)
    add = sub.add_parser("add")
    add.add_argument("category")
    add.add_argument("amount", type=int)
    sub.add_parser("total")
    args = parser.parse_args()
    try:
        if args.command == "add":
            add_expense(DATA, args.category, args.amount)
            print("Saved")
        else:
            print(f"Total: {total_expenses(DATA)} TZS")
    except (ValueError, OSError, KeyError, csv.Error) as error:
        parser.exit(1, f"Cannot complete: {error}\n")

if __name__ == "__main__":
    main()
```

Run it in a fresh project folder

```
python expenses.py total
python expenses.py add food 3000
python expenses.py add travel 1500
python expenses.py total
```

Those terminal commands print Total: 0 TZS, Saved, Saved, then Total: 4500 TZS. Running add again appends another record. Keep the generated CSV between sessions to preserve your history.

22 / Project: verify & extend

Part 3: save as test_expenses.py beside expenses.py

```
import tempfile
import unittest
from pathlib import Path
from expenses import add_expense, total_expenses

class ExpenseTests(unittest.TestCase):
    def test_persistence(self):
        with tempfile.TemporaryDirectory() as folder:
            path = Path(folder) / "expenses.csv"
            self.assertEqual(total_expenses(path), 0)
            add_expense(path, "food", 3000)
            add_expense(path, "travel", 1500)
            self.assertEqual(total_expenses(path), 4500)

    def test_invalid_amount(self):
        with tempfile.TemporaryDirectory() as folder:
            path = Path(folder) / "expenses.csv"
            with self.assertRaises(ValueError):
                add_expense(path, "food", -1)
            self.assertFalse(path.exists())
```

Run python -m unittest -v. Both tests should pass. Then try an empty category and a zero amount manually. Never test destructive file operations against your real data.

What this small app does not do yet

It has no dates, editing, currency conversion, simultaneous-writer protection or database transactions. Add a category summary first, then dates, then move storage to SQLite. Malformed existing CSV input is reported as an error, not repaired automatically.

23 / Automation & office work

Use case: replace a repetitive manual task

Python can inventory folders, transform spreadsheets, prepare reports and batch-process documents. Start with a preview that changes nothing. A useful first project is a report of large files or a list of missing documents, followed by a reviewed action.

```
from pathlib import Path

folder = Path("practice_output")
for path in sorted(folder.glob("*.txt")):
    if path.is_file():
        print(path.name, path.stat().st_size, "bytes")
```

This prints a file inventory when the practice folder exists; it does not rename or delete anything. For CSV, start with the standard library. Spreadsheet work can use dedicated libraries; preserve formulas and formats when those matter. For scheduled jobs, record what ran, when it ran and whether it succeeded.

Build next: generate a CSV inventory with name, size and extension. Check its totals manually on a tiny folder.

Other automation uses

Document generation, email attachment processing, image resizing and business-system integrations share the same pattern: read input, validate it, transform it and produce a traceable result. Decide how repeated runs avoid duplicate outputs.

24 / Websites and APIs

Use case: expose data or actions over HTTP / Optional dependency

A web backend receives requests and sends responses. An API is a contract for that exchange. FastAPI is one framework for APIs; Django includes a larger web application toolkit, while Flask offers a smaller core. Python usually runs on the server, alongside browser HTML, CSS and JavaScript.

```
# Terminal, inside a virtual environment:
python -m pip install "fastapi[standard]"

# Save the following Python in main.py:
from fastapi import FastAPI

app = FastAPI()

@app.get("/hello/{name}")
def hello(name: str):
    return {"message": f"Hello, {name}!"}
```

Run fastapi dev main.py in the terminal. Visit http://127.0.0.1:8000/hello/Asha and /docs. The response is JSON with message: Hello, Asha! This development example has no database or authentication. Public deployment needs production configuration, validation, access control and monitoring.

Build next: a read-only catalogue API with item IDs, a missing-item response and tests.

https://fastapi.tiangolo.com/tutorial/first-steps/

25 / Data analysis & charts

Use case: turn rows into a useful business question / Optional pandas

Analysis starts with a question: which category accounts for the most spending? Check units, missing values and duplicates before calculating. pandas works with labelled tables; charts help communicate the result after the numbers are understood.

```
# Install first: python -m pip install pandas
import pandas as pd

df = pd.DataFrame({
    "category": ["food", "travel", "food"],
    "amount": [3000, 1500, 2000],
})
summary = df.groupby("category")["amount"].sum()
print(summary)
print("Total:", df["amount"].sum())
```

Expected category totals: food = 5000 and travel = 1500; the overall total is 6500. Do not sum unrelated currencies or mix a daily amount with a monthly one. Averages can hide variation: inspect distributions and sample counts too.

Build next: a monthly spending report. State the date range, currency, missing rows and the decision the report supports.

Beyond spreadsheets

Python also powers data pipelines, dashboards and analytics services. Large datasets may need database aggregation or distributed processing; loading everything into one DataFrame is not always appropriate.

https://pandas.pydata.org/docs/getting_started/

26 / AI and machine learning

Use case: learn patterns or build applications around trained models

Two different starting points

Traditional machine learning learns a relationship from examples, such as classifying support requests or estimating demand. An application around a language model sends inputs to an existing model and checks its outputs. Neither approach replaces understanding the problem or measuring quality.

A reliable first workflow

Define a target and a simple baseline. Split data into training and evaluation sets before fitting transformations. Train on the training set, evaluate on held-out examples and inspect errors. Choose metrics that match the cost of mistakes; accuracy alone can mislead when classes are imbalanced.

Tools and first project

scikit-learn provides conventional models and evaluation tools. Deep-learning frameworks support larger neural models. Start with a small public labelled dataset and compare a simple baseline with one model. Record the split, features and results so another person can reproduce the experiment.

Language-model applications

A document assistant retrieves relevant passages before drafting an answer. Keep source links, test unsupported questions and verify outputs. Treat retrieved documents as untrusted data. Do not send private data to a service without appropriate permission and controls.

Build next: classify a small set of labelled messages. Write down where the model fails before adding more complexity.

27 / Science & engineering

Use case: calculate, simulate and explain physical or numerical systems

Scientific work often combines arrays, numerical methods and plots. NumPy supports array operations; SciPy provides scientific algorithms. Start with units and assumptions, then compare a computation with a case whose answer you already know.

```
from math import pi
from statistics import mean, stdev

radius_m = 2
area_m2 = pi * radius_m ** 2
readings = [20.0, 20.2, 19.8, 20.1]
print(round(area_m2, 2))
print(round(mean(readings), 3))
print(round(stdev(readings), 3))
```

Expected: 12.57, 20.025 and 0.171. stdev calculates sample standard deviation. A precise-looking number is not proof of a precise measurement: sensor uncertainty, calibration and sampling still matter.

Specialist branches

Geospatial work connects records to locations and coordinate systems. Bioinformatics processes biological sequences and experimental results. Finance uses time-series analysis and simulations. Each field needs domain knowledge as well as Python; model assumptions must be reviewed by someone qualified for the application.

Build next: plot a sensor log with units and missing readings clearly identified. Compare the summary against a hand calculation.

28 / Desktop, games & media

Use case: create an interactive or creative tool

Desktop applications

Tkinter offers a route into desktop interfaces; some operating systems require a separate Tk package. A useful first project is a unit converter with two inputs and clear validation. Keep the calculation function independent of the button handler so it can be tested without opening the window.

Games and learning tools

Start with a text quiz, a guessing game or a small 2D game. Learn the loop: receive input, update state, draw the result. Python is useful for learning and prototypes; demanding real-time graphics often rely on engines and lower-level components.

Images, audio and creative workflows

Batch-resize images, generate thumbnails, organise media or script creative software. Keep originals and write outputs into a new folder. Choose output resolution and compression deliberately, then inspect several files rather than assuming a successful command means a good result.

Mobile and browser interfaces

Python can participate in mobile and browser projects, but check the target platform and ecosystem before choosing it for the interface. A common architecture uses a Python API behind a mobile or JavaScript frontend.

Build next: a quiz that records answers and explains mistakes. Add a graphical interface only after the underlying logic works.

29 / Systems, cloud & security

Use case: make software operations observable and repeatable

Operations and cloud automation

Python scripts can inspect logs, check service health, prepare backups and call cloud APIs. Prefer narrowly scoped credentials and explicit resource names. A useful first project reads a log file and counts status codes, without changing any infrastructure.

Testing and quality assurance

Automate API checks, regression tests and browser workflows. Tests should wait for meaningful state rather than arbitrary delays. Keep sample accounts and test data separate from production. Capture enough information to reproduce a failure without logging secrets.

Defensive security

Analyse your own access logs, inventory dependencies and check configuration against a documented baseline. Learn networking and permissions alongside Python. Security automation should operate only on systems you own or have permission to assess.

Hardware and connected devices

Python can coordinate sensors and robotics on suitable computers. MicroPython targets some microcontrollers, but it is not identical to desktop Python; libraries and hardware support differ. Start with reading a sensor and logging a timestamp before controlling motors or physical equipment.

Build next: a log summary that reports the time range, total requests and error count. Verify it on a tiny log you can count by hand.

30 / Pick your next project

Use-case map: problem, first deliverable and what to learn next

Automation / Office

A file inventory or CSV report. Learn paths, validation, scheduling and repeatable runs.

Web / Business systems

An inventory API. Learn HTTP, SQL, authentication, authorisation and deployment.

Data / Dashboards

A spending analysis. Learn pandas, statistics, plotting and data-quality checks.

AI / Language / Vision

A labelled-data experiment. Learn evaluation, leakage prevention and error analysis.

Science / GIS / Research

A reproducible analysis notebook. Learn units, numerical methods and domain assumptions.

Desktop / Games / Media

A converter, quiz or image pipeline. Learn event-driven code and user feedback.

Cloud / Testing / Security

A health or log report. Learn APIs, permissions, testing and observability.

IoT / Education

A sensor logger or teaching exercise. Learn hardware constraints and clear feedback.

31 / Work like a professional

Make it understandable, reproducible and safe to change

A small project structure

```
my_project/
  README.md
  pyproject.toml
  src/
    my_app/
      __init__.py
      core.py
  tests/
    test_core.py
  .gitignore
```

Start simple; adopt this layout when packaging helps. Write a README with setup, run commands, expected input and tests. Record a supported Python version. A dependency record or lockfile should capture the environment that you actually tested.

Before a release

Run tests from a clean environment. Check invalid input and empty data. Keep secrets out of source control. Log useful events without passwords or personal data. Back up persistent data and verify restoration. Write an update note describing what changed.

When it becomes slow

Measure first. Improve the algorithm or reduce unnecessary I/O before adding concurrency. async helps coordinate waiting tasks; it does not automatically accelerate CPU-heavy calculations. Use processes or specialised numeric tools when measurements justify them.

A project is ready to share when another person can set it up, understand its limits and reproduce its result.

32 / Practice and answers

Try first. Cover the answer before reading it.

A / Sum selected values

Task: total only amounts above 1000 from [500, 1500, 2000]. Expected: 3500.

```
amounts = [500, 1500, 2000]
answer = sum(x for x in amounts if x > 1000)
print(answer)
```

B / Count words

Task: count the words in "code learn code". Expected: code appears twice and learn once.

```
counts = {}
for word in "code learn code".split():
    counts[word] = counts.get(word, 0) + 1
print(counts)
```

C / Fix the shared-list surprise

Task: a = [1]; b = a; b.append(2). Why is a now [1, 2]? Both names refer to the same list. Use b = a.copy() for an independent shallow copy.

D / Project acceptance checks

A fresh total is zero. Two entries of 3000 and 1500 total 4500. A negative amount does not create a file. Empty categories are rejected. Running total twice leaves the stored data unchanged.

33 / Your four-week route

A suggested pace; move on when you can explain what you built

Week 1 / Read and change code

Pages 3-12. Practise 30-45 minutes a day. Build a price calculator and a short quiz. Explain each variable and branch without looking at the example.

Week 2 / Organise and save

Pages 13-21. Split calculations into functions. Read and write CSV. Add tests. Build a contact list using fictional data.

Week 3 / Finish one project

Pages 22-24. Run the expense notebook, test it and add a category summary. Write a README and save changes in Git.

Week 4 / Choose one direction

Pages 25-32. Pick automation, web, data or another field. Deliver a small working version with sample input, expected output and documented limitations.

When you get stuck

Read the error carefully. Reduce the problem to a few lines. Print the values you assumed were correct. Check the relevant documentation. Ask a precise question that includes the code, expected result and actual result.

> Progress is a small project you can explain and improve, not the number of libraries you have installed.

34 / Reference shelf

Official documentation and a short glossary

Python tutorial and language reference
https://docs.python.org/3/tutorial/

Standard library: files, CSV, JSON, tests, SQL
https://docs.python.org/3/library/

Install Python
https://www.python.org/downloads/

Environments and packages
https://packaging.python.org/en/latest/tutorials/installing-packages/

Application areas
https://www.python.org/about/apps/

FastAPI: first steps
https://fastapi.tiangolo.com/tutorial/first-steps/

pandas: getting started
https://pandas.pydata.org/docs/getting_started/

Machine learning: scikit-learn
https://scikit-learn.org/stable/getting_started.html

Words to remember

Interpreter: runs Python code. Iterable: supplies successive values. Function: reusable operation. Module: importable code. Exception: a reported failure. API: an interface contract. Dependency: software your project requires. Test: an executable check of expected behaviour.

Documentation reviewed 19 October 2026. Core lessons use stable Python 3 features. Optional ecosystem examples may require updated dependencies; consult their official compatibility guidance.