PYTHON: FROM NOOB TO PRO

PRACTICAL AUTOMATION EDITION



500-PAGE GUIDE + CODE EXAMPLES

A PRACTICAL TUTORIAL AND LAB BOOK

60 chapters / 10 projects / runnable source and checks

NOVEMBER 2026 EDITION

How to use this tutorial and lab book

Start at the beginning if programming is new to you. Read a lesson, type or inspect the example, predict its output, and run it in a dedicated practice directory. Attempt the exercise before opening the reference solution.

The main PDF contains exactly 500 pages. Twelve opening pages introduce setup and navigation; sixty eight-page chapters occupy pages 13-492; eight reference pages finish the book. Each chapter includes explanation, a runnable example, a trace, debugging guidance, a practice brief, a reference solution, verification cases, and an application worksheet. The final ten chapters are practical projects.

The code bundle contains sixty examples and sixty reference solutions, plus a verification runner and fixtures. Core programs use the standard library. No paid API, account, or third-party package is needed to run the supplied examples.

This edition's programs were executed on CPython 3.11.9 and 3.12.14 on Windows. The syntax baseline is Python 3.11; other interpreter versions and operating systems were not executed in this verification. Use a currently maintained patch release appropriate to your environment.

“Pro” describes the progression toward maintainable habits, not a certification, job promise, or claim that one book covers every specialty. Professional skill grows through reviewed projects and continued practice.

Install Python and identify the interpreter

Obtain Python through python.org or an appropriate trusted operating-system package channel. Follow the current instructions for your platform. Installer interfaces change; this guide does not ask you to bypass security warnings or weaken system protections.

After installation, open a terminal and run the version command for the interpreter you intend to use. Depending on your system, that command may be python, python3, or an available Windows launcher. A command name is not proof that it selects the installation you expected.

To inspect an interpreter, run it with -c and a small fixed command: `import sys; print(sys.executable); print(sys.version)`. Keep the executable path in your project notes. Use that interpreter consistently for scripts, virtual environments, and package operations.

An editor saves source text; a terminal launches commands; the Python interpreter executes Python code. A terminal command is not necessarily valid inside a .py file. Keep those layers separate while learning.

The supplied core examples do not require package installation. If a later personal project needs dependencies, create a project environment and use the packaging guide linked in the resources.

Windows: run the project interpreter directly

Create a normal practice folder that you own and can write to. Extract the code bundle there. Open a terminal in that folder or navigate to it using the terminal's ordinary directory commands.

Check the installed Python command with --version. If an appropriate launcher is available, it may help select an installed version. Do not assume every Windows system has the same launcher or PATH configuration.

A project environment can be created with the chosen interpreter followed by `-m venv .venv`. You can run its interpreter directly as `.venv\Scripts\python.exe`. Activation is a convenience, not a requirement; direct invocation avoids changing execution-policy settings merely to run a practice script.

Run a supplied example by passing its path to that interpreter. Use quotation marks around a path containing spaces. Keep the command outside the source file.

If a file cannot be opened, check your working directory, exact filename, and hidden extension. If imports behave unexpectedly, check for local files named json.py, csv.py, or other standard-library names.

Write down the interpreter command and project directory that worked. A scheduled job will later need equally explicit paths.

macOS and Linux: keep setup project-local

Use the appropriate trusted Python installation method for your system and check its current vendor instructions. The command is often python3, but verify the actual executable and version rather than relying on a naming convention.

Create a practice directory and run the selected interpreter with `-m venv .venv` if you need an isolated environment. Its interpreter is typically `.venv/bin/python`. Directly invoking it works without activating the environment.

Do not replace or modify a system-managed Python installation to follow this book. Keep personal project dependencies separate from operating-system tools. Some distributions package environment support separately; consult their official guidance if venv is unavailable.

Run a script by passing its filename to the project interpreter. Paths are normally case-sensitive on many systems, so Report.py and report.py may be different files. Permissions and filesystem behavior also vary.

The supplied examples were executed on Windows with the two recorded CPython versions. They use portable standard-library techniques where practical, but you should run the bundle's verification on your target system before relying on it.

Your practice workspace and code bundle

The code bundle has examples, solutions, fixtures, checks.json, and verify_bundle.py. Each example and solution filename begins with its chapter number. The PDF identifies the matching file on the relevant code page.

Keep an untouched copy of the downloaded bundle, then make a working copy for experiments. Use fictional fixtures and temporary directories. Do not start by pointing a new script at your home directory, workplace records, or a production database.

Run `python verify_bundle.py` from the extracted code folder using your chosen interpreter. The runner executes the standalone programs in temporary working directories and runs the supplied behavioral checks. It performs no live API calls. Timing and subprocess availability still depend on the target environment.

Code blocks labeled runnable example or reference solution are complete source files. Verification expressions in the book are review cases; their full fixture setup lives in checks.json and the runner.

The course does not automate deletion of your original files. Projects include read-only reports, collision-aware plans, exclusive output creation, and temporary test fixtures. Any future extension that changes real data needs explicit scope, authorization, and recovery planning.

Contents / Chapters 01-15

01 Run your first useful script 13

02 Names, values, and assignment 21

03 Numbers, arithmetic, and exact money 29

04 Format reports with strings 37

05 Normalize text without destroying meaning 45

06 Compare values and combine conditions 53

07 Choose a branch with if and elif 61

08 Repeat work with for loops 69

09 Use while loops with an explicit stopping rule 77

10 Use lists as ordered collections 85

11 Tuples and unpacking 93

12 Dictionaries and named records 101

13 Sets and uniqueness 109

14 Functions with a clear contract 117

15 Arguments, defaults, and keyword clarity 125

Contents / Chapters 16-30

16 Return structured results and control scope 133

17 Handle expected exceptions narrowly 141

18 Validate inputs at the boundary 149

19 Modules, imports, and the main guard 157

20 Represent paths with pathlib 165

21 Read text with explicit encoding 173

22 Write outputs without casual overwriting 181

23 Discover files deterministically 189

24 Keep paths inside an intended workspace 197

25 Plan copies before performing them 205

26 Read CSV as structured rows 213

27 Write CSV with a stable schema 221

28 Load and validate JSON 229

29 Dates, times, and explicit time zones 237

30 Use regular expressions for a narrow grammar 245

Contents / Chapters 31-45

31 Count and group records with collections 253

32 Sort records and group adjacent values 261

33 Stream work with iterators and generators 269

34 Type hints and dataclasses 277

35 Manage resources with context managers 285

36 Build a command-line interface with argparse 293

37 Configuration and environment variables 301

38 Log useful events without leaking data 309

39 Test behavior with unittest 317

40 Use fakes and mocks at external boundaries 325

41 Store records in SQLite 333

42 Keep related changes in a transaction 341

43 Separate HTTP transport from data processing 349

44 Bound pagination and retries 357

45 Run a subprocess with explicit arguments 365

Contents / Chapters 46-60

46 Use bounded threads for independent work 373

47 Understand async tasks and cancellation 381

48 Hash files for comparison and verification 389

49 Create and inspect ZIP archives 397

50 Package, measure, and maintain a small tool 405

51 Project: a read-only file inventory 413

52 Project: a CSV cleaner with rejection records 421

53 Project: an expense summary in integer cents 429

54 Project: a log summary with malformed-line reporting 437

55 Project: find duplicate-content candidates 445

56 Project: a collision-aware rename planner 453

57 Project: a ZIP backup with a manifest 461

58 Project: a small SQLite contact store 469

59 Project: an offline API reporting adapter 477

60 Capstone: a preview-first reporting pipeline 485

A practical route through the material

Phase 1 - Read and reason. Chapters 1-13 teach execution order, values, text, decisions, loops, and collections. Predict outputs and trace state changes before trying to shorten code.

Phase 2 - Build clear boundaries. Chapters 14-20 introduce functions, arguments, results, exceptions, validation, modules, and paths. The goal is a small reusable operation with an explicit contract.

Phase 3 - Process real formats. Chapters 21-30 cover text files, outputs, discovery, path scope, copies, CSV, JSON, dates, and patterns. Keep source records and rejected data traceable.

Phase 4 - Make tools testable. Chapters 31-40 add grouping, ordering, streaming, data models, resource ownership, command-line options, configuration, logs, and tests.

Phase 5 - Handle reliability. Chapters 41-50 introduce databases, transactions, HTTP boundaries, pagination, subprocesses, concurrency, hashes, archives, and releases. Read the limits as carefully as the success path.

Phase 6 - Deliver projects. Chapters 51-60 combine the techniques into practical tools. Extend one project at a time and add a meaningful test for each changed requirement.

The practice contract: inspect before applying

Before running unfamiliar automation, identify its inputs, outputs, side effects, and stopping conditions. A filename in a tutorial is not authorization to access a real system. Use only material and systems you are allowed to process.

Keep source and output locations explicit. Preview-first behavior makes a proposed change reviewable, while exclusive creation helps protect existing destinations. Neither replaces a complete recovery plan for valuable data.

Treat external text as data. Do not use eval for JSON, concatenate untrusted shell commands, disable certificate verification, or upload private records to an unapproved service. The chapters explain safer boundaries for these tasks.

When a batch rejects records or stops early, preserve that state in the result. A polished report must not silently imply that incomplete processing was complete.

Use the worksheets to record your local choices: what the tool is allowed to do, how you will test it, what a failure means, and who reviews changes. These are practical engineering decisions rather than obstacles to automation.

Your first task is small: run a harmless example, predict its output, and explain why it did not change any personal file.

01 / Run your first useful script

Core lesson

A program is a sequence of instructions executed by an interpreter. A Python source file is plain text, usually with a .py extension. The interpreter reads that file and performs its statements in order. A script does not know what you meant; it acts on the instructions and inputs it actually receives.

Begin with a tiny report about a fictional folder. Use an editor to save the code, then run the saved file from a terminal. In this book, `python` means the Python interpreter selected for your project. On some systems you will use `python3` or the Windows launcher instead. Check the version and executable before assuming a command uses the installation you intended.

Do not paste the terminal prompt itself into a file. The code example is a complete script; the command used to run it is a separate instruction.

01 / Run your first useful script

Runnable example

CODE BUNDLE / examples/01_run_your_first_useful_script.py

```
print("Daily automation check")
print("Folder:", "practice-inbox")
print("Files reviewed:", 3)
print("Mode:", "report only")
```

OBSERVED DEMONSTRATION OUTPUT

```
Daily automation check
Folder: practice-inbox
Files reviewed: 3
Mode: report only
```

01 / Run your first useful script

Trace and inspect

Read the four statements from top to bottom. Each call to print writes one line to standard output, the normal stream for a program's results. Quotation marks delimit text; they are not printed as part of that text. Commas let print display several values separated by spaces.

Save the example as first_report.py. Run it from the directory containing the file. Change the folder label, save again, and rerun. The output comes from the saved file, not from unsaved editor contents. This distinction explains many apparent cases where Python seems to ignore a change.

INPUT / INITIAL STATE

IMPORTANT STATE CHANGE

EXPECTED RESULT

01 / Run your first useful script

Debug a failure

If the terminal reports that a file cannot be opened, first check the working directory and filename. A file named first_report.py.txt is not the same as first_report.py. If Python reports an unterminated string, compare the opening and closing quotation marks.

Read the error before changing several things. Record the command, the current directory, and the exact final error line. Troubleshooting becomes easier when you preserve what happened instead of repeatedly clicking Run without knowing which file it executes.

SMALLEST REPRODUCIBLE CASE

EXPECTED VERSUS ACTUAL

ONE CHANGE TO VERIFY

01 / Run your first useful script

Practice lab

Create a script that prints a weekly report heading, a fictional project name, a count of seven completed tasks, and a statement that no files were changed. Store the task count in a variable named completed so the supplied check can inspect it.

Predict all four output lines before running. Then change only the project name. Confirm that the count remains seven and the report-only statement remains visible.

INPUTS AND OUTPUT CONTRACT

PLAN OR PSEUDOCODE

BOUNDARY AND FAILURE CASES

01 / Run your first useful script

Reference solution

CODE BUNDLE / solutions/01_run_your_first_useful_script.py

```
completed = 7
print("Weekly automation report")
print("Project:", "Cedar Practice")
print("Completed tasks:", completed)
print("No files were changed.")
```

Run the matching verification cases before adapting this reference solution. The complete source is included in the code bundle.

01 / Run your first useful script

Verify the behavior

The bundle verifies this solution using controlled fixtures. The expressions below show the selected assertions; their full setup is in checks.json. Run verify_bundle.py rather than copying these expressions as a standalone program.

```
1. completed == 7
```

Passing these cases is evidence for the behaviors exercised. It does not establish every possible input, platform, live service, or concurrent filesystem state.

01 / Run your first useful script

Apply and extend

A useful automation starts with a visible, harmless result. Printing a plan lets you learn execution order without touching important files. Later chapters will separate report output from diagnostic messages, but this first distinction already matters: displaying a filename does not copy, rename, or delete it.

Keep all practice work in a dedicated folder. Use invented data until you understand the script's behavior. Write down the interpreter command that works on your machine and the location of your practice folder.

EXTENSION CHALLENGE

Add a fifth line naming the next review date as plain text. Explain which line you would change if the number of tasks changed. Do not add file-writing behavior yet.

YOUR ADAPTATION AND ITS LIMITS

CHECK TO ADD BEFORE RELEASE