Keep Building

The Practical Python Project Extensions Pack

15 level-up challenges for the book's five projects,
10 programs to build next, an AI-code review checklist,
and the resources to keep going

A free bonus for readers of
Practical Python in the AI Era

by CodeEssentials Books

Works with Python 3.12 or newer. Each challenge builds on the finished project from the book, so keep your project folders (or the companion files) nearby.

How to Use This Pack

You finished the five projects. The fastest way to make those skills stick is to change a program you already understand, so every challenge here starts from a project exactly as you built it in the book. None of them repeats the book's own *Change it* tasks; do those first if you have not yet.

Each project has three levels. **Level 1** is an afternoon: one new option or command. **Level 2** adds a real feature with its own function and tests. **Level 3** is a stretch that changes the design a little, the kind of change real programs need after their first month of use.

There are no printed solutions, on purpose. Every challenge ends with a **Done when** line: turn it into a pytest test before you write the code, watch it fail, then make it pass. That is the habit the book built, and it is how you will know you are finished without anyone checking your work.

Keep the book's rules as you go: preview before changing files, never overwrite or drop data silently, `Decimal` for money, a timeout on every network call, and clear errors with the right exit code.

Using an AI assistant?

Good. Ask it for hints, for edge cases to test, and to review your finished code. Try each challenge yourself first, and run everything it suggests past the review checklist near the end of this pack before you keep it.

Project 1: Smart File Organizer

Your organizer previews, never overwrites, and writes a report. These challenges make it smarter about what it moves and when.

Level 1: Leave some files alone

Goal. Add an **ignore list** to the settings: exact names such as `desktop.ini` and `.DS_Store`, plus suffixes such as `.part` and `.crdownload` for downloads still in progress.

Hints. Check the ignore list inside `plan_moves`, before a file gets a category. Show ignored files in the preview under their own heading, so the user can see the tool noticed them.

Done when: an ignored file is still in place after a run, it is listed as skipped in the preview and in the report, and no other file's destination changed.

Level 2: Find duplicate files

Goal. Before moving anything, find files whose *content* is identical, whatever their names, and send the extra copies to a **Duplicates** folder instead of their category. Still nothing is deleted.

Hints. Hash each file with `hashlib.sha256`, reading it in chunks (`iter(lambda: f.read(65536), b"")`) so large files do not fill memory. Keep a dictionary from hash to the first path seen. Compare sizes first: files of different sizes can never be duplicates, so most files are never hashed.

Done when: a copied file with a new name lands in Duplicates, the original goes to its category, and the report lists each duplicate next to the file it copies.

Level 3: Run it unattended

Goal. Make the organizer safe to run from Task Scheduler or cron. Add `--yes` to skip the confirmation question and `--older-than DAYS` to organize only files that have not changed for that many days, so a file you downloaded a minute ago is never moved out from under you.

Hints. Use `argparse`; a file's age comes from `path.stat().st_mtime`. Without a person watching, the report matters more: append one summary line per run to a log file. Pass the current time into the planning function as an argument so tests can fix it.

Done when: a test with `tmp_path` proves that a new file is left in place, an old one is planned for a move, and `--yes` never asks a question.

Project 2: Text & Log Analyzer

Your analyzer counts levels, finds repeated messages, and reports bad lines. These challenges help it answer the next questions people ask.

Level 1: Show only what matters

Goal. Add `--level LEVEL`: keep entries at that level *or more serious*, so `--level WARNING` shows warnings, errors, and critical entries.

Hints. Put the levels in a list in order of severity, the same order the logging module uses: DEBUG, INFO, WARNING, ERROR, CRITICAL. Compare positions in the list, not strings. Use `choices=` in argparse so a typo is rejected at once.

Done when: `--level ERROR` on the sample log counts no INFO or WARNING entries, and `--level errror` exits with a usage error, not a traceback.

Level 2: When did it go wrong?

Goal. Print a text bar chart of errors per hour, one row per hour, and name the busiest hour. On the sample log the burst of database failures after lunch should stand out at a glance.

Hints. Group with a dictionary keyed by `entry_time.hour`. Scale the bars so the longest is 40 characters wide, and print `"#" * width`. Keep the counting function separate from the printing function, like the book's design.

Done when: the counting function returns a dictionary you can test directly, hours with no errors still print as empty rows, and the busiest hour matches the sample log.

Level 3: Read a whole folder of logs

Goal. Accept several files or a pattern, such as all the .log files in a folder, merge their entries into one timeline in time order, and count problem lines *per file*, so you know which file is damaged.

Hints. `Path(folder).glob("*.log")` lists the files; give each parsed entry its source file name. `sorted(entries, key=...)` merges the timeline. Use `pytest.mark.parametrize` to test the line parser against a table of good and bad lines.

Done when: two sample logs merge into one report with entries in time order, and each problem line is reported with its file name and line number.

Project 3: Expense Tracker

Your tracker parses a bank export with exact Decimal money and never drops a row silently. These challenges teach it to explain your spending.

Level 1: Where does the money go?

Goal. Add `--top N`: list the N largest expenses, and the places you pay most often, with how many times and how much in total.

Hints. Sort expenses by amount (they are negative, so the largest is the smallest number). `collections.Counter` counts descriptions. Every total stays a `Decimal`; format only when printing.

Done when: `--top 3` on the sample export lists Rent first, Supermarket is the most frequent place, and no amount ever passes through a float.

Level 2: Categorize by rules

Goal. Rows with no category become *Uncategorized*. Add a `rules.json` file that maps words in a description to a category, such as `"bike": "Transport"`, and apply it to those rows. Report the rows that no rule matched, so the user can add a rule.

Hints. Match case-insensitively and apply rules in file order, first match wins. Never overwrite a category the bank already provided. Validate the file when you load it: a rule that is not a string pair is a clear `ValueError`.

Done when: the sample's Bike repair row is counted under Transport, a row with a bank category keeps it, and unmatched rows are listed, not hidden.

Level 3: A second bank

Goal. Real people have more than one account. Support a second export layout with different column names, dates written `DD/MM/YYYY`, and separate Debit and Credit columns instead of one signed Amount, and combine both files into one summary.

Hints. Describe each layout in a small JSON *bank profile*: column names and the date format. Convert every row into the same transaction dictionary as early as possible, so filtering and totals do not change at all. Debit amounts become negative.

Done when: tests prove the same purchase gives the identical transaction from either layout, and an unknown profile name exits with a clear error.

Project 4: Contact Manager

Your contact manager validates every record, stores JSON safely, and is covered by tests. These challenges bring in data from outside and add a useful reminder.

Level 1: Tidy listings

Goal. Add `list --sort name|email` and a `count` command that prints how many contacts you have.

Hints. `sorted(contacts, key=lambda c: c.name.lower())` sorts without caring about capital letters. `choices=` keeps the sort field valid. Sorting belongs in the service layer, printing in the command line.

Done when: a test sorts three contacts entered in the wrong order, and count on an empty file prints 0 instead of failing.

Level 2: Import from CSV

Goal. Add `import FILE`: read contacts from a CSV file, validate each row with the same rules as add, skip rows whose email is already stored, and finish with a summary: added, skipped as duplicates, rejected with the reason.

Hints. `csv.DictReader` reads rows by header name. Build each Contact inside `try`/`except ValueError` so one bad row cannot stop the import. Save once at the end, and only if something was added.

Done when: a test with `tmp_path` imports a file with one good row, one duplicate, and one bad email, and checks all three outcomes and the saved file.

Level 3: Birthday reminders

Goal. Add an optional birthday field and an `upcoming --days 30` command that lists birthdays in the next 30 days, soonest first.

Hints. Store the date in the strict YYYY-MM-DD form from Chapter 7. For each contact, work out the next occurrence: this year's date, or next year's if it has passed. Decide what February 29 means in a year without one (February 28 is a common choice) and write that rule down in the docstring. Pass today's date in as an argument so tests do not depend on the calendar.

Done when: tests with a fixed today cover a birthday later this year, one in early January seen from December, and February 29.

Project 5: Command-Line Task Tracker

Your task tracker has strict dates, safe storage, and a regression test for a real bug. These challenges make it the tool you keep open every day.

Level 1: How am I doing?

Goal. Add a `stats` command: open tasks, done tasks, how many open tasks are overdue, and how many are due in the next seven days.

Hints. Compare due dates with today's date, passed in as an argument like every other time-dependent function. Tasks with no due date are never overdue.

Done when: a test with a fixed today and four tasks (no due date, overdue, due soon, done) gets every number right.

Level 2: Repeating tasks

Goal. Add `--every day|week|month` to add. When a repeating task is marked done, create the next one with the due date moved forward, and keep the finished one as done.

Hints. Days and weeks are a `timedelta`. Months are not: January 31 plus one month has no answer until you choose one. `calendar.monthrange(year, month)` gives the number of days in a month, so you can move to the last day when the day does not exist. A repeating task needs a due date; reject one without.

Done when: tests cover each interval, January 31 becoming February 28 (or 29 in a leap year), and the new task getting a new id.

Level 3: Put it in your calendar

Goal. Add `export-ics FILE`: write the open tasks that have a due date as an iCalendar (.ics) file that Google Calendar, Outlook, or Apple Calendar can import.

Hints. An .ics file is plain text. It starts with BEGIN:VCALENDAR, VERSION:2.0, and a PRODID line, and holds one VEVENT per task. Each event needs a UID, a DTSTAMP, a SUMMARY, and an all-day date such as `DTSTART;VALUE=DATE:20261001`. Lines end with CRLF; the format is defined in RFC 5545.

Done when: a test reads the file back and finds one event per dated open task, with the right dates, and your calendar app imports it.

10 Programs to Build Next

Each of these is a small, complete tool that uses skills from the book. Start every one the same way: write down what it must never do, plan its functions, and write the first test.

1. Folder size report

Show the 20 largest files and folders under a path, with sizes printed as KB, MB, or GB. *Uses: pathlib, rglob, sorting.*

2. Photo renamer

Rename photos in a folder by their modified date, such as `2026-09-30_001.jpg`, with a preview, no overwrites, and an undo report. *Uses: the plan, preview, confirm, apply pattern.*

3. Monthly CSV merger

Combine twelve monthly CSV exports into one file, drop exact duplicate rows, and report every row it could not read. *Uses: csv, validation, error reports.*

4. Link checker

Read URLs from a text file and report each one's status code. Set a timeout on every request, and try a GET when a server rejects a HEAD request. *Uses: requests, timeouts, exceptions.*

5. Weather at a glance

Fetch today's temperature for your city from a free weather API such as Open-Meteo, which needs no API key for non-commercial use (check its current terms), and cache the answer for an hour. *Uses: web APIs, JSON, dates.*

6. Timesheet

Commands `start` and `stop` record work sessions in a CSV file; `week` prints the hours per day and the total. *Uses: argparse subcommands, datetime, timedelta.*

7. Backup copier

Copy files that are new or changed since the last run into a dated backup folder, with `shutil.copy2` and a dry-run mode. *Uses: pathlib, file metadata, safe writes.*

8. Flashcard quiz

Ask questions from a JSON deck in random order, and keep a score history so the cards you miss come back more often. *Uses: JSON storage, dataclasses, random.*

9. Word frequency report

Count the most common words in a text file, ignoring a stop-word list, with `collections.Counter`. *Uses: text processing, dictionaries.*

10. Unit converter

Convert lengths, weights, and temperatures from the command line, with a pytest test table that checks every conversion both ways. *Uses: argparse, pytest.mark.parametrize.*

The AI-Code Review Checklist

Chapter 15 showed the mistakes AI-generated code makes again and again. Run this list over any code an assistant writes for you, and over your own, before you keep it.

Does it exist and does it run?

- [ ] Every function, method, and option it calls is in the official documentation
- [ ] It runs on *your* Python version, and imports only what is installed
- [ ] I ran it, and I can explain every line in my own words

Does it fail safely?

- [ ] Bad input produces a clear error message, not a traceback
- [ ] No bare `except:` and no `except Exception: pass` hiding errors
- [ ] Every network call has a timeout, and a failed request is handled
- [ ] It never overwrites or deletes a file without a preview or a backup
- [ ] No row, record, or file is dropped without being reported

Is it correct?

- [ ] Money is `Decimal` created from strings, never float
- [ ] Dates are parsed strictly; impossible dates are rejected
- [ ] Edge cases are covered: empty input, one item, duplicates, the last day of a month
- [ ] There is a test that fails when the code is wrong (I watched it fail)

Is it safe to share?

- [ ] No passwords, API keys, or personal data written into the code
- [ ] No hard-coded paths that only work on one computer

Resources to Keep Going

All free and official. When a tutorial, a video, and the documentation disagree, trust the documentation for your version.

The Python documentation

The tutorial, the Library Reference for every module in the book, and the How-To guides.
https://docs.python.org/3/

What's New in Python 3.15

Everything that changed in Python 3.15, beyond the everyday features in Chapter 14.
https://docs.python.org/3.15/whatsnew/3.15.html

pytest documentation

Fixtures, parametrize, and the how-to guides for the testing habits in this pack.
https://docs.pytest.org/

Python Packaging User Guide

When a project outgrows one folder: virtual environments, packages, and publishing.
https://packaging.python.org/

Requests documentation

Sessions, timeouts, and authentication for the web API challenges.
https://requests.readthedocs.io/

The book's page and companion files

The code, sample data, and answers for every chapter and project, kept up to date.
https://codeessentialsbooks.com/practical-python-ai-era

Keep Going

This pack is a free bonus for readers of **Practical Python in the AI Era: Build Real Programs with Automation, APIs, Testing, and AI-Assisted Coding**. If a friend is ready to move past the basics, the book's page has the companion files and everything else they need to start.

Just getting started, or know someone who is? **Python for Beginners** is the gentler first step in the CodeEssentials Python Series.

Practical Python in the AI Era · Python for Beginners · All our books on Amazon

Enjoyed the book? A short, honest review on Amazon helps other readers find it.

© CodeEssentials Books. You received this file by subscribing to our email list; please don't republish it, but do share the book's page with a friend who is learning Python. Python and the Python logos are trademarks of the Python Software Foundation; this pack is not affiliated with or endorsed by it.