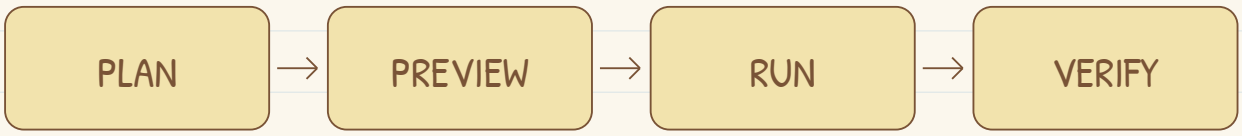
THE PRACTICAL NOTEBOOK

Python for Automation

Turn repetitive work into dependable small tools.

64 pages / 20 guided lessons / 40 code files



Understand it. Try it. Explain it.

Build automation that is understandable, repeatable and recoverable. Start with file inventories and text processing, then work through CSV, JSON, command-line interfaces, retries, reports, backups and scheduling. Examples use temporary folders and local fixtures so practice does not modify your real files.

PRIVATUS COSMAS / PRIVER

depriver.tech · First edition / October 2026

Your 20-lesson route

Work in order, or use this clickable map to revisit a topic. Each lesson has a learn page, a practice page and an explained solution.

01	Choose the right task	4
02	Paths and working directories	7
03	Inventory before changing files	10
04	Text cleanup	13
05	CSV reports	16
06	JSON configuration	19
07	Command-line interfaces	22
08	Plan before applying	25
09	Atomic output files	28
10	Backups and restore checks	31
11	Duplicates and hashes	34
12	Logs and useful summaries	37
13	HTTP integrations and fixtures	40
14	Retries and backoff	43
15	Subprocesses safely	46
16	Scheduling and repeatability	49
17	Email and notification previews	52
18	Browser automation choices	55
19	Testing failure paths	58
20	Project: a folder reporter	61

Setup and study method · 3 References and next steps · 64

Make this book yours

BEFORE YOU START

Know Python variables, loops, functions and exceptions. No paid services or third-party packages are required for the runnable lessons. External email, browser and API integrations are discussed as extensions with explicit access and failure-handling requirements.

YOUR WORKSPACE

Use Python 3.11 or newer. Create a separate practice folder and run one example at a time. Most filesystem exercises create TemporaryDirectory fixtures that are removed automatically. Copy a script into real use only after replacing sample paths deliberately, reviewing its preview and testing recovery.

TERMINAL / FOLLOW THE COMMANDS FOR THIS BOOK

```
python --version
python lesson_01_example.py
python lesson_01_solution.py
```

A SIMPLE STUDY RHYTHM

1. Read: predict what the example will do before running it.
2. Practise: write your answer using the acceptance cases.
3. Compare: read the solution, explain the difference, then extend it.

The companion ZIP contains lesson_01_example.py through lesson_20_example.py and matching solution files. Run them individually from the extracted folder. Most successful checks are silent; a traceback means a check failed.

READABLE NOTES, PRECISE CODE

Handwritten headings keep the notebook feel; body text and code use clear fonts. Copy runnable code from the ZIP to preserve indentation. The PDF is searchable and its contents links are clickable.

Choose the right task

THE GOAL

Automate a clear rule before automating a complicated workflow.

A good first automation has repeatable inputs, explicit rules and a checkable output. Write down the current manual steps and what counts as success. Avoid beginning with a task whose rules change every time. A small script that saves ten reliable minutes is more useful than a large fragile workflow.

lesson_01_example.py

```
workflow = {"input": "folder of text files",
            "rule": "list name and byte size",
            "output": "CSV inventory",
            "success": "one row per regular text file"}
assert workflow["input"] and workflow["success"]
```

The workflow describes a read-only inventory, making it easy to compare the script's output with the folder by hand.

WATCH FOR THIS

A filename extension is a naming convention, not proof of the file's actual content.

Your turn: Choose the right task

THE CHALLENGE

Write a pure function that chooses whether a filename belongs in a text-file inventory. Match .txt case-insensitively and exclude hidden filenames.

ACCEPTANCE CHECKS

[ ] notes.txt -> True

[ ] NOTES.TXT -> True

[ ] .secret.txt -> False

[ ] photo.jpg -> False

A NUDGE, IF YOU NEED ONE

Use startswith and a case-normalised suffix check.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

CHOOSE THE RIGHT TASK

lesson_01_solution.py

```
def include_name(name):
    return not name.startswith(".") and name.lower().endswith(".txt")
assert include_name("notes.txt")
assert include_name("NOTES.TXT")
assert not include_name(".secret.txt")
assert not include_name("photo.jpg")
```

WHY IT WORKS

A small decision function is easy to test before it becomes part of a larger file-processing script. Filesystem type and permission checks are separate.

TAKE IT FURTHER

Define how your inventory should handle unreadable files, symbolic links and nested folders.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Paths and working directories

THE GOAL

Make every file location intentional.

pathlib separates path operations from string manipulation. Relative paths resolve from the current working directory, which may differ from the script location. Use explicit roots for automation and avoid assuming that a scheduler starts in the same folder as your terminal.

lesson_02_example.py

```
from pathlib import Path
from tempfile import TemporaryDirectory
with TemporaryDirectory() as folder:
    root = Path(folder)
    path = root / "notes.txt"
    path.write_text("Learn automation", encoding="utf-8")
    assert path.name == "notes.txt"
    assert path.suffix == ".txt"
    assert path.read_text(encoding="utf-8") == "Learn automation"
```

The path is built using /, which joins path components. The temporary directory isolates the experiment and is cleaned up on exit.

WATCH FOR THIS

Changing the working directory globally can surprise helper functions. Pass paths instead.

Your turn: Paths and working directories

THE CHALLENGE

Create an output subdirectory inside a temporary root, write a report file and verify that its parent is the intended output folder.

ACCEPTANCE CHECKS

[ ] Output directory exists

[ ] Report contains expected text

[ ] File lives under the chosen root

A NUDGE, IF YOU NEED ONE

Call mkdir with parents=True and exist_ok=True before writing.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

PATHS AND WORKING DIRECTORIES

lesson_02_solution.py

```
from pathlib import Path
from tempfile import TemporaryDirectory
with TemporaryDirectory() as folder:
    root = Path(folder)
    output = root / "output"
    output.mkdir(parents=True, exist_ok=True)
    report = output / "report.txt"
    report.write_text("Ready", encoding="utf-8")
    assert report.parent == output
    assert report.read_text(encoding="utf-8") == "Ready"
```

WHY IT WORKS

Creating directories explicitly makes the script repeatable. It does not automatically make writing over an existing report safe; that needs a separate policy.

TAKE IT FURTHER

Add an input argument for the root directory and print its resolved location before processing.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Inventory before changing files

THE GOAL

Inspect a folder without moving or deleting anything.

An inventory is a useful foundation for cleanup, migration and backup tasks. Gather file metadata first, sort results for reproducibility and report skipped items. A preview lets the user catch wrong paths or overly broad selection rules before any mutation happens.

lesson_03_example.py

```
from pathlib import Path
from tempfile import TemporaryDirectory
with TemporaryDirectory() as folder:
    root = Path(folder)
    (root / "b.txt").write_bytes(b"hello")
    (root / "a.txt").write_bytes(b"hi")
    rows = [(p.name, p.stat().st_size)
            for p in sorted(root.glob("*.txt")) if p.is_file()]
    assert rows == [("a.txt", 2), ("b.txt", 5)]
```

The reported size is bytes, not characters. UTF-8 text can use multiple bytes per character.

WATCH FOR THIS

A preview can become stale if another process changes files before the apply step.

Your turn: Inventory before changing files

THE CHALLENGE

Inventory only regular non-symlink files larger than a threshold. Return relative names and sizes in sorted order.

ACCEPTANCE CHECKS

[ ] 2-byte file below threshold 3 -> excluded

[ ] 5-byte file -> included

[ ] Directory -> excluded

A NUDGE, IF YOU NEED ONE

Check is_symlink before is_file, because is_file follows a symlink.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

INVENTORY BEFORE CHANGING FILES

lesson_03_solution.py

```
from pathlib import Path
from tempfile import TemporaryDirectory

def inventory(root, minimum):
    return [(p.name, p.stat().st_size) for p in sorted(root.iterdir())
            if not p.is_symlink() and p.is_file()
            and p.stat().st_size >= minimum]
with TemporaryDirectory() as folder:
    root = Path(folder)
    (root/"small").write_bytes(b"hi")
    (root/"large").write_bytes(b"hello")
    (root/"subdir").mkdir()
    assert inventory(root, 3) == [("large", 5)]
```

WHY IT WORKS

This is a top-level inventory. Recursion, permission failures and files changing during the scan require additional decisions in a production tool.

TAKE IT FURTHER

Include modification time in the plan and recheck it before applying a later action.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Text cleanup

THE GOAL

Define transformations that preserve important information.

Text cleanup should follow a precise rule. Stripping surrounding whitespace is different from collapsing internal spaces, changing case or deleting empty lines. Choose encoding explicitly and keep the original when the transformation is not obviously reversible.

lesson_04_example.py

```
def clean_lines(text):
    return [line.strip() for line in text.splitlines() if line.strip()]
assert clean_lines(" Asha \n\n Juma\n") == ["Asha", "Juma"]
```

This function removes empty lines and trims each retained line. It does not collapse spaces inside a person's name.

WATCH FOR THIS

Do not erase the original values if you need to audit why two records were merged.

Your turn: Text cleanup

THE CHALLENGE

Normalise a list of email-like identifiers for deduplication by trimming and case-folding. Treat this as a project-specific identifier rule, not universal email validation.

ACCEPTANCE CHECKS

[ ] " A@EXAMPLE.TEST " -> "a@example.test"

[ ] Repeated normalised values appear once

[ ] Original order of first occurrence is preserved

A NUDGE, IF YOU NEED ONE

Use a set for membership and a list for ordered output.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

TEXT CLEANUP

lesson_04_solution.py

```
def unique_ids(values):
    seen, result = set(), []
    for value in values:
        clean = value.strip().casefold()
        if clean and clean not in seen:
            seen.add(clean)
            result.append(clean)
    return result
assert unique_ids([" A@EXAMPLE.TEST ", "a@example.test", "B@x.test"])
assert unique_ids([" A@EXAMPLE.TEST ", "a@example.test"]) == [
    "a@example.test"]
```

WHY IT WORKS

The normalisation policy should be written beside the transformation. Some identifiers are case-sensitive, so this rule is not safe for every system.

TAKE IT FURTHER

Return a mapping from each normalised value to its original inputs for review.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

CSV reports

THE GOAL

Read columns by name and validate numbers explicitly.

CSV files exchange rows and columns, but fields are text until converted. Use csv.DictReader for named columns and open real files with newline="". Validate required columns, numeric formats and units. A file that parses successfully can still contain invalid business data.

lesson_05_example.py

```
import csv, io
source = "category,amount\nfood,3000\ntravel,1500\n"
rows = list(csv.DictReader(io.StringIO(source)))
total = sum(int(row["amount"]) for row in rows)
assert total == 4500
```

The explicit int conversion rejects non-integer amounts. This exercise uses whole TZS amounts, so there is no hidden floating-point currency rule.

WATCH FOR THIS

Spreadsheet applications may interpret cells beginning with certain characters as formulas. Treat untrusted spreadsheet exports according to the destination's policy.

Your turn: CSV reports

THE CHALLENGE

Group CSV amounts by category and return a dictionary of totals. Reject negative amounts and missing category names.

ACCEPTANCE CHECKS

[ ] food 3000 + food 2000 -> food 5000

[ ] Negative amount -> ValueError

[ ] Blank category -> ValueError

A NUDGE, IF YOU NEED ONE

Trim category, convert amount, validate, then update the running total.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

CSV REPORTS

lesson_05_solution.py

```
import csv, io

def totals(source):
    result = {}
    for row in csv.DictReader(io.StringIO(source)):
        category = row["category"].strip()
        amount = int(row["amount"])
        if not category or amount < 0:
            raise ValueError("Invalid row")
        result[category] = result.get(category, 0) + amount
    return result
assert totals("category,amount\nfood,3000\nfood,2000\n") == {
    "food": 5000}
```

WHY IT WORKS

The function fails instead of silently dropping invalid records. A different workflow may collect errors, but it must make omissions visible.

TAKE IT FURTHER

Include row numbers in validation errors and produce a separate rejected-rows report.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

JSON configuration

THE GOAL

Validate configuration before starting work.

JSON represents structured data, but parsing only checks syntax. Your program must validate required fields, types and supported values. Keep secrets separate from ordinary configuration files. A clear startup error is better than processing half a folder with an invalid setting.

lesson_06_example.py

```
import json
config = json.loads('{"minimum_bytes": 10, "recursive": false}')
assert isinstance(config["minimum_bytes"], int)
assert config["recursive"] is False
```

The JSON boolean becomes a Python bool. In Python, bool is a subclass of int, which matters when checking exact integer settings.

WATCH FOR THIS

Never commit API tokens alongside a reusable example configuration.

Your turn: JSON configuration

THE CHALLENGE

Validate minimum_bytes as a non-negative integer, rejecting booleans, and recursive as a boolean. Reject unknown keys.

ACCEPTANCE CHECKS

[ ] minimum_bytes=0 accepted

[ ] minimum_bytes=True rejected

[ ] Unknown option rejected

A NUDGE, IF YOU NEED ONE

Use type(value) is int when booleans must not count as integers.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

JSON CONFIGURATION

lesson_06_solution.py

```
def validate_config(data):
    if set(data) - {"minimum_bytes", "recursive"}:
        raise ValueError("Unknown option")
    size = data.get("minimum_bytes", 0)
    recursive = data.get("recursive", False)
    if type(size) is not int or size < 0:
        raise ValueError("Invalid minimum_bytes")
    if type(recursive) is not bool:
        raise ValueError("Invalid recursive")
    return {"minimum_bytes": size, "recursive": recursive}
assert validate_config({}) == {"minimum_bytes": 0, "recursive": False}
try:
    validate_config({"minimum_bytes": True})
except ValueError:
    pass
else:
    raise AssertionError("Boolean accepted as byte count")
```

WHY IT WORKS

Explicit defaults make configuration predictable. Unknown-key rejection catches misspellings that would otherwise look like ignored settings.

TAKE IT FURTHER

Add a configuration version so future format changes can be migrated deliberately.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Command-line interfaces

THE GOAL

Make scripts usable without editing source code.

argparse provides named options, help text and parsing errors. A good command-line interface states what it will process and where outputs go. Keep defaults conservative. Use exit codes to distinguish success from failure so scheduled jobs can react correctly.

lesson_07_example.py

```
import argparse
parser = argparse.ArgumentParser(description="Preview an inventory")
parser.add_argument("folder")
parser.add_argument("--minimum", type=int, default=0)
args = parser.parse_args(["sample", "--minimum", "10"])
assert args.folder == "sample" and args.minimum == 10
```

Passing a list to parse_args makes the example testable. In a real entry point call parse_args() with no list to use the terminal arguments.

WATCH FOR THIS

A script that mutates files by default can be dangerous when copied into a scheduler or run in the wrong directory.

Your turn: Command-line interfaces

THE CHALLENGE

Add an --apply flag that defaults to False. Verify that a normal invocation stays in preview mode.

ACCEPTANCE CHECKS

[ ] No --apply -> False

[ ] With --apply -> True

[ ] --help describes the option

A NUDGE, IF YOU NEED ONE

Use action="store_true" for an explicit opt-in flag.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

COMMAND-LINE INTERFACES

lesson_07_solution.py

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("folder")
parser.add_argument("--apply", action="store_true",
                    help="Apply the reviewed plan")
assert not parser.parse_args(["sample"]).apply
assert parser.parse_args(["sample", "--apply"]).apply
assert "--apply" in parser.format_help()
```

WHY IT WORKS

The flag alone does not guarantee safety. The apply path must use the same selection rules as the preview and recheck conflicts.

TAKE IT FURTHER

Add --output and reject an output path inside an input tree when recursive scanning would accidentally reprocess reports.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Plan before applying

THE GOAL

Represent intended changes as data you can review.

Separate planning from execution. A plan lists source, destination and action without changing the filesystem. Validate duplicate destinations and existing files before applying it. For complex renames, plan order and rollback carefully because one rename can affect another.

lesson_08_example.py

```
def rename_plan(names):
    return [(name, name.replace(" ", "_")) for name in names
            if " " in name]
assert rename_plan(["my notes.txt", "plain.txt"]) == [
    ("my notes.txt", "my_notes.txt")]
```

The plan is a list of pairs. It is easy to display, save and test before connecting it to Path.rename.

WATCH FOR THIS

Previewing a plan is not enough if the execution code independently recomputes a different plan.

Your turn: Plan before applying

THE CHALLENGE

Reject a plan that has duplicate destinations or would overwrite an existing unplanned filename. Do not perform renames in this exercise.

ACCEPTANCE CHECKS

[ ] Two sources targeting one name -> reject

[ ] Destination already exists -> reject

[ ] Distinct new destinations -> accept

A NUDGE, IF YOU NEED ONE

Compare destination count with the size of its set, then check collisions.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

PLAN BEFORE APPLYING

lesson_08_solution.py

```
def validate_plan(plan, existing):
    destinations = [dst for _, dst in plan]
    if len(destinations) != len(set(destinations)):
        raise ValueError("Duplicate destination")
    if any(dst in existing for dst in destinations):
        raise ValueError("Destination already exists")
    return True
assert validate_plan([("a b", "a_b")], {"a b"})
for plan, existing in [([("a","x"),("b","x")], set()),
                       ([("a","x")], {"x"})]:
    try:
        validate_plan(plan, existing)
    except ValueError:
        pass
    else:
        raise AssertionError("Conflict accepted")
```

WHY IT WORKS

This deliberately conservative policy rejects rename chains into existing names. A more advanced planner can support them with temporary names and a recovery log.

TAKE IT FURTHER

Serialise the reviewed plan to JSON and require the apply command to read that exact plan.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Atomic output files

THE GOAL

Avoid leaving a half-written report after failure.

For a single report, write to a temporary file in the destination directory and replace the destination only after writing succeeds. A same-filesystem replace is atomic at the name-switch level on normal supported filesystems, but durability after power loss may require additional syncing and platform-specific care.

lesson_09_example.py

```
from pathlib import Path
from tempfile import TemporaryDirectory
import os
with TemporaryDirectory() as folder:
    target = Path(folder) / "report.txt"
    temporary = target.with_suffix(".tmp")
    temporary.write_text("complete report", encoding="utf-8")
    os.replace(temporary, target)
    assert target.read_text(encoding="utf-8") == "complete report"
```

The final path appears only when the completed temporary file is moved into place. A unique temporary filename avoids collisions between concurrent runs.

WATCH FOR THIS

Temporary files on another filesystem cannot necessarily be atomically renamed into the destination.

Your turn: Atomic output files

THE CHALLENGE

Use NamedTemporaryFile in the target directory, close it, then replace the destination. Ensure the temporary file is removed if an error occurs.

ACCEPTANCE CHECKS

[ ] Final file contains complete text

[ ] Existing report is replaced only at the end

[ ] Temporary file does not remain after success

A NUDGE, IF YOU NEED ONE

Keep the temporary path and clean it in finally with missing_ok=True.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

ATOMIC OUTPUT FILES

lesson_09_solution.py

```
from pathlib import Path
from tempfile import TemporaryDirectory, NamedTemporaryFile
import os

def write_report(target, text):
    with NamedTemporaryFile("w", encoding="utf-8", delete=False,
                            dir=target.parent) as handle:
        temporary = Path(handle.name)
        handle.write(text)
    try:
        os.replace(temporary, target)
    finally:
        temporary.unlink(missing_ok=True)
with TemporaryDirectory() as folder:
    target = Path(folder)/"report.txt"
    write_report(target, "ready")
    assert target.read_text() == "ready"
```

WHY IT WORKS

The temporary file is closed before replacement, which is especially important on Windows. This example does not promise crash durability or concurrent-write coordination.

TAKE IT FURTHER

Inject a failure before os.replace and verify the previous report remains intact.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Backups and restore checks

THE GOAL

Prove that saved copies can recover the data.

A backup is useful only if you can restore and verify it. Store it outside the failure domain of the original and protect its confidentiality. A ZIP can package files for a small exercise, but it is not a complete backup strategy with retention, encryption and disaster recovery.

lesson_10_example.py

```
from pathlib import Path
from tempfile import TemporaryDirectory
from zipfile import ZipFile
with TemporaryDirectory() as folder:
    root = Path(folder)
    source = root/"notes.txt"
    source.write_text("Important notes", encoding="utf-8")
    with ZipFile(root/"backup.zip", "w") as archive:
        archive.write(source, arcname="notes.txt")
    with ZipFile(root/"backup.zip") as archive:
        assert archive.read("notes.txt") == source.read_bytes()
```

The archive name is controlled, so the ZIP does not expose the temporary absolute path. Reading the saved bytes checks more than just the ZIP's existence.

WATCH FOR THIS

Never blindly extract untrusted archives into important directories; member paths and resource limits need validation.

Your turn: Backups and restore checks

THE CHALLENGE

Create a backup manifest containing a SHA-256 digest, then verify the archived bytes match it.

ACCEPTANCE CHECKS

[ ] Digest of archived content equals original digest

[ ] Changed bytes produce another digest

[ ] Archive contains the expected member name

A NUDGE, IF YOU NEED ONE

Hash bytes before writing and after reading the ZIP member.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

BACKUPS AND RESTORE CHECKS

lesson_10_solution.py

```
from hashlib import sha256
from io import BytesIO
from zipfile import ZipFile
payload = b"Notebook content"
expected = sha256(payload).hexdigest()
buffer = BytesIO()
with ZipFile(buffer, "w") as archive:
    archive.writestr("notes.txt", payload)
buffer.seek(0)
with ZipFile(buffer) as archive:
    restored = archive.read("notes.txt")
assert sha256(restored).hexdigest() == expected
assert sha256(payload+b"!").hexdigest() != expected
```

WHY IT WORKS

A digest detects accidental changes when the expected digest is trusted. It does not by itself authenticate an untrusted archive or prove a full backup set is complete.

TAKE IT FURTHER

Write a restore checklist that includes filenames, counts, byte checks and opening representative files.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Duplicates and hashes

THE GOAL

Identify matching content without deleting it automatically.

Hashing maps bytes to a fixed-size digest. It is useful for duplicate detection and integrity checks. For large files, read chunks rather than loading everything into memory. A matching cryptographic digest is strong evidence of matching bytes, but deletion policy still needs review.

lesson_11_example.py

```
from hashlib import sha256

def digest(chunks):
    value = sha256()
    for chunk in chunks:
        value.update(chunk)
    return value.hexdigest()
assert digest([b"ab", b"cd"]) == sha256(b"abcd").hexdigest()
```

Chunk boundaries do not change the digest. This lets the same algorithm process a tiny fixture or a large file stream.

WATCH FOR THIS

Comparing only filenames or file sizes cannot prove contents are duplicates.

Your turn: Duplicates and hashes

THE CHALLENGE

Group named byte payloads by digest and return only groups with more than one filename. Produce a report, not a deletion action.

ACCEPTANCE CHECKS

[ ] Two equal contents -> one duplicate group

[ ] Different contents -> separate groups

[ ] Unique files omitted from duplicate output

A NUDGE, IF YOU NEED ONE

Use a digest-to-filenames dictionary, then filter its values by length.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

DUPLICATES AND HASHES

lesson_11_solution.py

```
from hashlib import sha256

def duplicate_groups(files):
    groups = {}
    for name, payload in files.items():
        key = sha256(payload).hexdigest()
        groups.setdefault(key, []).append(name)
    return [sorted(names) for names in groups.values() if len(names)>1]
assert duplicate_groups({"a":b"x", "b":b"x", "c":b"y"}) == [
    ["a", "b"]]
assert duplicate_groups({"a":b"x"}) == []
```

WHY IT WORKS

The report identifies candidates. The user may still need multiple copies in different workflows, so identical bytes are not automatic permission to delete.

TAKE IT FURTHER

First group by size to reduce hashing work, then confirm candidates with a streamed digest.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Logs and useful summaries

THE GOAL

Make unattended runs explain what happened.

A scheduled script needs an understandable result even when nobody watches the terminal. Log a run identifier, processed count, skipped count and failure reason. Avoid private file contents or credentials in logs. A concise summary helps distinguish no work from a broken scan.

lesson_12_example.py

```
import logging, io
stream = io.StringIO()
logger = logging.getLogger("notebook-demo")
logger.handlers.clear()
logger.setLevel(logging.INFO)
logger.addHandler(logging.StreamHandler(stream))
logger.info("processed=%s skipped=%s", 3, 1)
assert "processed=3 skipped=1" in stream.getvalue()
```

Formatting arguments keeps the log call structured and avoids unnecessary string assembly. This example captures output in memory for a deterministic test.

WATCH FOR THIS

Logging every row of sensitive input can create a second uncontrolled copy of the data.

Your turn: Logs and useful summaries

THE CHALLENGE

Create a summary from per-file results with success, skipped and failed statuses. Reject unknown statuses instead of hiding them.

ACCEPTANCE CHECKS

[ ] Two successes and one failure -> matching counts

[ ] Empty input -> zero counts

[ ] Unknown status -> ValueError

A NUDGE, IF YOU NEED ONE

Initialise every supported counter before iterating.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

LOGS AND USEFUL SUMMARIES

lesson_12_solution.py

```
def summarise(results):
    counts = {"success": 0, "skipped": 0, "failed": 0}
    for status in results:
        if status not in counts:
            raise ValueError("Unknown status")
        counts[status] += 1
    return counts
assert summarise(["success","failed","success"]) == {
    "success": 2, "skipped": 0, "failed": 1}
assert sum(summarise([]).values()) == 0
```

WHY IT WORKS

A run with partial failures should not silently report complete success. Decide the exit code and notification policy for partial completion.

TAKE IT FURTHER

Add a machine-readable JSON summary beside a short human-readable terminal summary.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

HTTP integrations and fixtures

THE GOAL

Define the response contract before connecting to a service.

An HTTP integration needs timeouts, status checks, parsing and validation. Separate network retrieval from response transformation so offline fixtures can test your logic. Use a service's documented authentication and rate limits. Do not assume every response body is valid JSON or contains the expected fields.

lesson_13_example.py

```
import json

def parse_items(payload):
    data = json.loads(payload)
    if not isinstance(data, dict) or not isinstance(data.get("items"), list):
        raise ValueError("Unexpected response shape")
    return data["items"]
assert parse_items('{"items": [{"id": 1}]}') == [{"id": 1}]
```

The fixture tests the parsing contract without a network call. A live client must still handle connection failures and non-success status codes before this function.

WATCH FOR THIS

A successful HTTP status does not prove the payload matches the documented schema.

Your turn: HTTP integrations and fixtures

THE CHALLENGE

Validate that every returned item has an integer id and a string title. Reject malformed items rather than passing them downstream.

ACCEPTANCE CHECKS

[ ] Valid item accepted

[ ] Missing title rejected

[ ] Boolean id rejected under exact-integer policy

A NUDGE, IF YOU NEED ONE

Validate the list and each element before returning a normalised result.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

HTTP INTEGRATIONS AND FIXTURES

lesson_13_solution.py

```
def validate_items(items):
    clean = []
    for item in items:
        if not isinstance(item, dict):
            raise ValueError("Item must be an object")
        if type(item.get("id")) is not int:
            raise ValueError("Integer id required")
        if not isinstance(item.get("title"), str):
            raise ValueError("Title required")
        clean.append({"id": item["id"], "title": item["title"]})
    return clean
assert validate_items([{"id":1,"title":"Git"}]) == [
    {"id":1,"title":"Git"}]
```

WHY IT WORKS

The output contains only fields the automation understands. New upstream fields do not silently become part of a report or command.

TAKE IT FURTHER

Add a real read-only client with an explicit timeout and test it against a local mock server before using an external API.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Retries and backoff

THE GOAL

Retry temporary failures without repeating harmful actions.

Temporary network errors may justify another attempt, while invalid credentials or bad input usually require correction. Use a bounded retry count, a total deadline and delay between attempts. A retried write needs an idempotency policy so a lost response does not create duplicates.

lesson_14_example.py

```
def delays(attempts, base=1, cap=8):
    return [min(cap, base * 2**i) for i in range(attempts)]
assert delays(5) == [1, 2, 4, 8, 8]
```

The schedule grows exponentially and stops increasing at a cap. Production systems often add jitter to prevent many clients retrying in lockstep.

WATCH FOR THIS

Retrying authentication failures repeatedly can lock an account or waste quota without making progress.

Your turn: Retries and backoff

THE CHALLENGE

Write a retry helper with an injected sleep function so the test records delays without actually waiting.

ACCEPTANCE CHECKS

[ ] Two failures then success -> two sleeps

[ ] Success result returned

[ ] Permanent ValueError is not caught

A NUDGE, IF YOU NEED ONE

Catch TimeoutError only and call the injected sleep between attempts.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

RETRIES AND BACKOFF

lesson_14_solution.py

```
def retry(operation, sleep, attempts=3):
    for i in range(attempts):
        try:
            return operation()
        except TimeoutError:
            if i == attempts - 1:
                raise
            sleep(min(8, 2**i))
    raise ValueError("Positive attempts required")
calls, pauses = [], []
def operation():
    calls.append(1)
    if len(calls) < 3:
        raise TimeoutError()
    return "done"
assert retry(operation, pauses.append) == "done"
assert pauses == [1, 2]
```

WHY IT WORKS

The helper is deterministic in tests because waiting is supplied from outside. A real implementation also needs request-level timeouts and a total elapsed-time budget.

TAKE IT FURTHER

Add a retry-after policy for a documented rate-limit response and test it with fixed fixtures.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Subprocesses safely

THE GOAL

Call other tools without turning data into shell code.

subprocess runs external programs. Prefer an argument list with shell=False so spaces and punctuation remain data. Check return codes and set timeouts. Avoid passing secrets as command-line arguments because other local processes or logs may expose them.

lesson_15_example.py

```
import subprocess, sys
result = subprocess.run([sys.executable, "-c", "print(2 + 3)"],
                        capture_output=True, text=True,
                        timeout=5, check=True)
assert result.stdout.strip() == "5"
```

The executable is the current Python interpreter. No shell parses the argument string, and check=True raises if the command fails.

WATCH FOR THIS

Concatenating user text into a shell command can execute unintended instructions.

Your turn: Subprocesses safely

THE CHALLENGE

Pass a value containing shell punctuation to a child Python process and prove it is printed literally.

ACCEPTANCE CHECKS

[ ] Input with spaces stays one argument

[ ] Semicolon is printed, not executed

[ ] Return code is zero

A NUDGE, IF YOU NEED ONE

Pass the value as a separate list element and read sys.argv[1] in the child.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

SUBPROCESSES SAFELY

lesson_15_solution.py

```
import subprocess, sys
value = "hello; this is data"
r = subprocess.run([sys.executable, "-c",
                    "import sys; print(sys.argv[1])", value],
                   capture_output=True, text=True,
                   timeout=5, check=True)
assert r.stdout.strip() == value
assert r.returncode == 0
```

WHY IT WORKS

The example demonstrates argument boundaries. The invoked tool can still interpret its own options, so validate values according to that tool's interface.

TAKE IT FURTHER

Handle CalledProcessError with a concise message and a redacted diagnostic log.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Scheduling and repeatability

THE GOAL

Design for the second run, not only the first.

A scheduler starts a command at a time; it does not guarantee the previous run finished or that the environment matches your terminal. Use absolute paths, explicit configuration, logs and an overlap policy. A repeatable job avoids duplicate outputs when run again with the same inputs.

lesson_16_example.py

```
def pending_jobs(all_ids, completed):
    return [ident for ident in all_ids if ident not in completed]
assert pending_jobs(["a","b","c"], {"a","c"}) == ["b"]
```

The completed set represents durable progress in a real workflow. An in-memory set is lost when the process exits.

WATCH FOR THIS

A scheduler can launch overlapping runs. A local lock or durable job claim may be needed.

Your turn: Scheduling and repeatability

THE CHALLENGE

Process each identifier once within a run, including duplicates in the input. Record completion only after successful work.

ACCEPTANCE CHECKS

[ ] Input a,a,b -> process a and b once

[ ] Previously completed a -> skip it

[ ] Failed work is not marked complete

A NUDGE, IF YOU NEED ONE

Check the completed set before work and add to it only after the operation returns.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

SCHEDULING AND REPEATABILITY

lesson_16_solution.py

```
def process(ids, completed, operation):
    for ident in ids:
        if ident in completed:
            continue
        operation(ident)
        completed.add(ident)
completed, calls = set(), []
process(["a","a","b"], completed, calls.append)
assert calls == ["a","b"]
process(["a","b"], completed, calls.append)
assert calls == ["a","b"]
```

WHY IT WORKS

This does not close the crash gap between the external action and recording completion. Durable idempotency keys or transactional coordination are needed for important side effects.

TAKE IT FURTHER

Document what happens if the process crashes immediately before and immediately after recording completion.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Email and notification previews

THE GOAL

Separate preparing a message from sending it.

Notifications are useful when a job needs attention, but accidental repeated sends are disruptive. Build the message, inspect it and validate recipients before sending through an authenticated provider. Keep credentials out of source code and define deduplication and retry rules.

lesson_17_example.py

```
from email.message import EmailMessage
message = EmailMessage()
message["Subject"] = "Inventory complete"
message["From"] = "reports@example.test"
message["To"] = "owner@example.test"
message.set_content("Processed 12 files. No failures.")
assert "Processed 12 files" in message.get_content()
assert message["Subject"] == "Inventory complete"
```

This creates a local message object only. It sends nothing and uses reserved example addresses.

WATCH FOR THIS

A successfully prepared email is not proof that a provider accepted or delivered it.

Your turn: Email and notification previews

THE CHALLENGE

Choose whether to notify based on a run summary. Send a failure notification only when failed > 0; successful routine runs remain quiet.

ACCEPTANCE CHECKS

[ ] failed=0 -> no notification

[ ] failed=2 -> notification

[ ] Negative counts -> ValueError

A NUDGE, IF YOU NEED ONE

Validate counts before applying the rule.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

EMAIL AND NOTIFICATION PREVIEWS

lesson_17_solution.py

```
def should_notify(summary):
    if any(type(v) is not int or v < 0 for v in summary.values()):
        raise ValueError("Invalid counts")
    return summary.get("failed", 0) > 0
assert not should_notify({"success": 10, "failed": 0})
assert should_notify({"success": 8, "failed": 2})
```

WHY IT WORKS

Notification policy should match the user's workflow. Some jobs need a daily digest, others immediate failure alerts. Avoid treating every normal event as urgent.

TAKE IT FURTHER

Add a unique run ID to the notification record and ensure a retry does not send the same alert twice.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Browser automation choices

THE GOAL

Prefer stable interfaces and meaningful waits.

Use a documented API when it provides the needed operation. Browser automation is useful when a task exists only through a UI, but selectors, authentication and page changes can make it fragile. Wait for a meaningful element or state, not a guessed number of seconds. Keep credentials in an appropriate secret mechanism.

lesson_18_example.py

```
steps = [
    {"action": "open", "target": "local test page"},
    {"action": "fill", "target": "label: Search", "value": "Python"},
    {"action": "click", "target": "button: Search"},
    {"action": "expect", "target": "results contain Python"},
]
assert steps[-1]["action"] == "expect"
```

This is an executable plan description, not a browser driver. A real browser library needs installation and a controlled test page. The plan makes the expected state explicit.

WATCH FOR THIS

A script clicking coordinates can act on the wrong control when the layout moves.

Your turn: Browser automation choices

THE CHALLENGE

Design a decision function that selects API or browser mode from available capabilities. Reject a task when neither supported interface exists.

ACCEPTANCE CHECKS

[ ] Documented API available -> api

[ ] Only supported UI available -> browser

[ ] Neither -> ValueError

A NUDGE, IF YOU NEED ONE

Prefer the stable API path before the UI fallback.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

BROWSER AUTOMATION CHOICES

lesson_18_solution.py

```
def choose_interface(has_api, has_supported_ui):
    if has_api:
        return "api"
    if has_supported_ui:
        return "browser"
    raise ValueError("No supported interface")
assert choose_interface(True, True) == "api"
assert choose_interface(False, True) == "browser"
try:
    choose_interface(False, False)
except ValueError:
    pass
else:
    raise AssertionError("Unsupported workflow accepted")
```

WHY IT WORKS

The decision is architectural, not a claim that every UI should be automated. Respect service rules and test with accounts and data you control.

TAKE IT FURTHER

Implement the four-step plan against a local HTML fixture using semantic selectors and assert the final result.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Testing failure paths

THE GOAL

Prove recovery before trusting unattended execution.

Automation must handle missing files, malformed input, permission errors and partial completion. Use temporary folders and injected dependencies to simulate failures safely. Test what remains unchanged after a failure, not only whether an exception was raised.

lesson_19_example.py

```
def convert(value):
    number = int(value)
    if number < 0:
        raise ValueError("Negative count")
    return number
assert convert("12") == 12
for bad in ["text", "-1"]:
    try:
        convert(bad)
    except ValueError:
        pass
    else:
        raise AssertionError("Invalid input accepted")
```

The tests cover invalid representation and invalid meaning. Both should produce useful messages when wrapped in a command-line tool.

WATCH FOR THIS

Testing only successful input makes an unattended tool look more reliable than it is.

Your turn: Testing failure paths

THE CHALLENGE

Validate every row before writing a report so one bad row does not leave a partial output file.

ACCEPTANCE CHECKS

[ ] Valid rows -> report created

[ ] Invalid row -> exception

[ ] Invalid input leaves no report

A NUDGE, IF YOU NEED ONE

First build validated output in memory for this small exercise, then write once.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

TESTING FAILURE PATHS

lesson_19_solution.py

```
from pathlib import Path
from tempfile import TemporaryDirectory

def report(values, target):
    parsed = [int(value) for value in values]
    if any(value < 0 for value in parsed):
        raise ValueError("Negative count")
    target.write_text(str(sum(parsed)), encoding="utf-8")
with TemporaryDirectory() as folder:
    target = Path(folder)/"report.txt"
    try:
        report(["2", "bad"], target)
    except ValueError:
        pass
    assert not target.exists()
    report(["2", "3"], target)
    assert target.read_text() == "5"
```

WHY IT WORKS

For large inputs, validating everything in memory may be unsuitable. Use a staged file or database transaction and a deliberate commit step instead.

TAKE IT FURTHER

Inject a write failure and verify the previous successful report remains available using the atomic-write technique.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Project: a folder reporter

THE GOAL

Combine scanning, validation and a reproducible output.

The final project inventories regular files in one folder and writes a CSV report outside that folder. It does not move, rename or delete input files. Start with a top-level scan, skip symlinks and use a stable name order. The lab verifies both output contents and repeatability.

lesson_20_example.py

```
from pathlib import Path
from tempfile import TemporaryDirectory
import csv

def inventory(root):
    return [{"name": p.name, "bytes": p.stat().st_size}
            for p in sorted(root.iterdir())
            if not p.is_symlink() and p.is_file()]
def write_csv(rows, target):
    with target.open("w", newline="", encoding="utf-8") as handle:
        writer = csv.DictWriter(handle, fieldnames=["name", "bytes"])
        writer.writeheader()
        writer.writerows(rows)
with TemporaryDirectory() as folder:
    root = Path(folder)/"input"
    root.mkdir()
    (root/"a.txt").write_bytes(b"abc")
    output = Path(folder)/"inventory.csv"
    write_csv(inventory(root), output)
    assert "a.txt,3" in output.read_text()
```

The output is outside input, so later runs do not inventory their own report. The script handles a controlled fixture; permissions and concurrent file changes need additional policies.

WATCH FOR THIS

Writing the report inside a recursively scanned input tree can make each run include earlier outputs.

Your turn: Project: a folder reporter

THE CHALLENGE

Create two input files and a subdirectory. Run the report twice and prove the output bytes are identical and input files unchanged.

ACCEPTANCE CHECKS

[ ] Two regular files -> two data rows

[ ] Subdirectory excluded

[ ] Second run produces identical CSV

[ ] Input contents preserved

A NUDGE, IF YOU NEED ONE

Capture both the first output bytes and the original input bytes before rerunning.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

PROJECT: A FOLDER REPORTER

lesson_20_solution.py

```
from pathlib import Path
from tempfile import TemporaryDirectory
import csv

def build(root, target):
    with target.open("w", newline="", encoding="utf-8") as handle:
        writer = csv.writer(handle)
        writer.writerow(["name", "bytes"])
        for path in sorted(root.iterdir()):
            if path.is_file() and not path.is_symlink():
                writer.writerow([path.name, path.stat().st_size])
with TemporaryDirectory() as folder:
    root = Path(folder)/"input"
    root.mkdir()
    (root/"a.txt").write_bytes(b"abc")
    (root/"b.txt").write_bytes(b"hello")
    (root/"subdir").mkdir()
    target = Path(folder)/"report.csv"
    build(root, target)
    first = target.read_bytes()
    build(root, target)
    assert target.read_bytes() == first
    assert (root/"a.txt").read_bytes() == b"abc"
    assert len(target.read_text().splitlines()) == 3
```

WHY IT WORKS

The project demonstrates a repeatable read-only workflow. Turn it into a real tool by adding argparse, an atomic report write, error summaries and a documented scheduling policy.

TAKE IT FURTHER

Create a portfolio release with --minimum, --output, a JSON run summary, failure tests, sample data and a README showing the exact expected report.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Sources & next steps

Use these official references when you need the full API contract or a deeper explanation. Documentation can evolve; check the version used by your own environment.

Paths and files

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

CSV processing

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

File operations

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

Command-line arguments

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

Subprocesses

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

Logging

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

ZIP files

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

WHAT THE CHECKS ESTABLISH

All 40 companion scripts were executed independently during preparation. Their assertions check the small fixtures shown here. They do not establish production readiness, security under attack, browser accessibility or performance at scale.

YOUR NEXT PROJECT

Choose a small problem you can explain. Write its inputs, outputs and success checks. Build a minimal version, test one failure path, and ask another learner to follow your README. Record what you would improve next.

Keep your notes. Keep experimenting.