Python Project Studio

20 Apps, Automations & AI Agents



From first programs to tested local tools, desktop windows, web services, data reports, retrieval, and bounded agent workflows.

Source code, fixtures, tests, code-reading guides, debugging tasks, and practice space. This is a written tutorial workbook, not a video course.

How the 1,000 pages work

The opening 40 pages establish setup, Python foundations, verification, and deployment boundaries. Each of the 20 project chapters then has 12 focused lessons. Every lesson uses four pages: concept, source walkthrough, test investigation, and a guided extension with a model reasoning answer.

The format intentionally gives concepts room to breathe. It includes substantial guided lab and writing space, repeated source context where a concept benefits from revisiting it, and worked test analysis. It is not 1,000 pages of continuous prose or 1,000 unrelated programs.

The full runnable implementation is app.py in each project folder. Numbered source excerpts are for reading. A > marker in the line-number gutter means a long line wraps visually. Do not copy gutter numbers or wrap markers into Python.

The twelve source-walkthrough pages in each chapter divide the complete application into consecutive slices. Focused excerpts elsewhere may revisit a function. Full test files, fixture examples, and exact installation instructions remain in the source download.

Progress in order for the first four projects. Later chapters can be selected by interest after the relevant prerequisites. Run the smallest test before introducing a new dependency or service.

What the AI tests mean

The offline agent fixtures verify orchestration and output boundaries. They do not demonstrate general model intelligence, provider availability, or paid-provider answer quality.

Choose your next build

01 Task Board CLI

Build a terminal task board that adds, completes, and lists tasks while preserving them between runs. Begin with values and input, then move behavior into functions and make persistence explicit.

Starts on page 41 / folder 01_task_board

02 Expense Ledger

Create a small expense ledger that stores exact decimal amounts in CSV and groups totals by category. The emphasis is a clear record contract and validation before persistence.

Starts on page 89 / folder 02_expense_ledger

03 Unit Converter

Build a reusable conversion module and a command-line interface. Model multiplicative units separately from temperature conversions so a simple formula does not hide an offset error.

Starts on page 137 / folder 03_unit_converter

04 Quiz Trainer

Create a deterministic multiple-choice trainer that separates question data, answer validation, scoring, and presentation. The first version uses a saved question set rather than a remote service.

Starts on page 185 / folder 04_quiz_trainer

05 File Inventory and Rename Planner

Inspect files under a chosen root and produce a deterministic inventory plus a rename proposal. The baseline deliberately previews names without changing files.

Starts on page 233 / folder 05_file_inventory

Your first Python run

A source file is plain text containing instructions. The interpreter reads the file and executes its top-level statements. A terminal command starts a process, while an editor is the place where you change the file. Save before running. When a program fails, read the final exception line and then locate the first frame in your own source. A successful import does not prove every branch has been executed.

Try the smallest example

Read the example before executing it. Shell commands, HTTP messages, and architecture sketches are identified by their context; they are not all Python source.

```
print("Hello, project studio")
name = "Ada"
print(f"Welcome, {name}")
```

Guided check

Save hello.py, run python hello.py, and change the name. The second line of output should change while the greeting remains fixed.

Record the prediction first, then run the example in a disposable project folder. If the result differs, compare values, types, working directory, and interpreter before changing the program.

Your observation and explanation

Windows setup and folders

Install Python from its official distribution and VS Code from its official site. The Python extension helps editing and debugging but is separate from the interpreter. Open a dedicated project folder. Use python --version to identify the selected interpreter. If Windows uses the py launcher, py -3 can select an installed Python. The source was verified with Python 3.12; use a compatible version with the tested framework requirements.

Try the smallest example

Read the example before executing it. Shell commands, HTTP messages, and architecture sketches are identified by their context; they are not all Python source.

```
python --version
python -m venv .venv
.venv\Scripts\python.exe -m pip --version
```

Guided check

Use the environment executable directly when activation is unavailable. This avoids changing PowerShell execution policy just to run the project.

Record the prediction first, then run the example in a disposable project folder. If the result differs, compare values, types, working directory, and interpreter before changing the program.

Your observation and explanation

Functions and return values

A function names behavior and can accept parameters. return sends a value to the caller. print sends a representation to an output stream. A function that prints a total but returns None is difficult to reuse in a report. Keep calculations separate from input prompts and file access when possible. Default mutable arguments can accidentally preserve state between calls.

Try the smallest example

Read the example before executing it. Shell commands, HTTP messages, and architecture sketches are identified by their context; they are not all Python source.

```
def total(cents):
    return sum(cents)

amount = total([100, 250])
print(f"Total cents: {amount}")
```

Guided check

Call total twice with different lists and verify the first result does not change. Add an empty-list case and define the expected result before running it.

Record the prediction first, then run the example in a disposable project folder. If the result differs, compare values, types, working directory, and interpreter before changing the program.

Your observation and explanation

Tests and fixtures

A test describes an observable contract. A fixture supplies controlled input and state. Use temporary directories for files and independent databases for web tests. Assert the intended output and the error path. A test that merely repeats the implementation formula can pass alongside the same mistake in both places. Choose examples whose result you can calculate independently.

Try the smallest example

Read the example before executing it. Shell commands, HTTP messages, and architecture sketches are identified by their context; they are not all Python source.

```
def double(value):
    return value * 2

def test_double():
    assert double(3) == 6
    assert double(0) == 0
```

Guided check

Run python -m pytest -q. Break the function deliberately, observe the failure, restore it, and explain which assertion detected the defect.

Record the prediction first, then run the example in a disposable project folder. If the result differs, compare values, types, working directory, and interpreter before changing the program.

Your observation and explanation

Python and local automation

Python tutorial

https://docs.python.org/3/tutorial/

Use the version selector for the interpreter you installed. Read the control-flow, data-structure, input/output, and error-handling sections alongside Projects 1-8. This book supplies beginner-oriented context before applying those interfaces.

Python standard library

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

Look up pathlib, json, csv, decimal, hashlib, re, and sqlite3 when changing a file mode, format rule, or API argument. Reference documentation is useful for resolving an exact contract rather than guessing from a remembered example.

Documentation checked 16 November 2026. Framework releases and provider interfaces can change. Use the tested versions for the first pass, then upgrade deliberately and rerun the relevant tests.

What was exercised

All 20 per-project pytest suites passed, totaling 53 pytest cases. The Django catalog also passed its framework test run in an isolated temporary database. Test logs are included in the source bundle.

Coverage includes task persistence, decimal arithmetic, invalid inputs, file metadata, hash comparison, CSV rejection, HTML extraction, scripted HTTP retry behavior, Pandas aggregation, PNG report generation, offscreen Qt widgets, Flask requests, Django views and ORM, local fixture structure, lexical retrieval, and the LangChain fixture tool loop.

The Selenium integration source and owned HTML fixture are included. The release checks HTML structure and Python syntax; it does not claim a real WebDriver browser run. Run python app.py --browser in that project after installing a compatible browser and driver.

Live provider modes were not invoked. Offline model fixtures are deterministic test doubles rather than general-purpose AI models. Public production hosting of the tutorial applications was not performed.

Tests establish the listed examples. They are not proof for every input, filesystem, browser, provider, locale, concurrent writer, or deployment. Read each chapter boundary before adapting it.

Names and values

Task Board CLI

A task needs an integer identity, a nonempty title, and a Boolean completion flag. A list preserves task order; a dictionary gives each field a meaningful name.

Choose a collection according to its identity, ordering, and duplication rules. A list preserves a sequence, a set represents membership, and a dictionary maps keys to values. These choices change observable behavior. Work through a three-record fixture containing one repeated value and write the intermediate state after each iteration.

Where it belongs in the build

Build a terminal task board that adds, completes, and lists tasks while preserving them between runs. Begin with values and input, then move behavior into functions and make persistence explicit.

Input	Validate	Transform	Observe

For this lesson, locate the exact boundary described by names and values. Name the incoming value, its allowed shape, and the next function or component that consumes it. Keep that boundary separate from the whole application.

Write a concrete input and predicted result for this boundary.

Implementation slice 1 of 12

projects/01_task_board/app.py / lines 1-6

This consecutive slice is part of the complete file. It may begin or end inside a function or string. Read it in context using the adjacent slices or the companion source; it is not a standalone snippet.

```
1  import argparse
2  import json
3  from pathlib import Path
4
5  def validate(tasks):
6      if not isinstance(tasks, list):
```

Line 5 / validate accepts tasks. Read its return paths and raised exceptions before deciding how a caller should use its result.

Run context

python app.py add "Write first test". Run from the project folder after installing its requirements. Preserve any source fixtures and use a fresh output destination when the application writes a file.

Investigate a passing contract

Test / test_roundtrip

The excerpt below comes from the tested companion suite. A test may address several chapter concepts, so it can be revisited from a different lesson. Its fixture and assertion remain unchanged.

```
3  def test_roundtrip(tmp_path):
4      tasks=[]; assert add_task(tasks,"  Write tests ")["id"]==1
5      complete(tasks,1); p=tmp_path/"tasks.json";save(p,tasks)
6      assert load(p)==[{"id":1,"title":"Write tests","done":True}]
```

Required result / add_task(tasks, ' Write tests ')['id'] == 1

Required result / load(p) == [{'id': 1, 'title': 'Write tests', 'done': True}]

Release result / 3 passed in 0.10s

Connect this test to names and values. Identify what the fixture controls, what the assertion observes, and which failure path is outside its scope. A test that passes does not validate an unrelated deployment or a provider that was not contacted.

Add one boundary the current test does not establish.

Build, diagnose, explain

Focused requirement

Apply the rule from names and values to one small change in Task Board CLI. First copy the passing project. Keep its baseline tests and add a new fixture that exposes the proposed change.

Check the empty collection and a one-record collection before a large fixture. The smallest cases often expose initialization, off-by-one, and silent truncation errors more clearly than a realistic-looking dataset.

A model reasoning answer

A task needs an integer identity, a nonempty title, and a Boolean completion flag. A list preserves task order; a dictionary gives each field a meaningful name. The implementation must make that rule observable at its boundary. State the value before the operation and the expected value or error after it. Use the relevant function or class name from the source rather than describing only a click or a printed sentence.

Diagnosis sequence

1. Reproduce the smallest failing fixture.
2. Locate the first value that differs from the prediction.
3. Check its type, identity, and ownership.
4. Change one condition or transformation.
5. Run the new case and the original suite.

Completion evidence / save the command, fixture, expected result, actual result, and the reason your change satisfies this lesson. If you have only an interface screenshot, add a state or persistence check where the requirement depends on it.

Your change, observation, and remaining limitation

Text input

Task Board CLI

strip removes outer whitespace. Validate the cleaned title rather than the original string, because a title made only of spaces is not meaningful.

Start from the stated input and output contract, then inspect how the supplied implementation realizes it. Names, types, ordering, error behavior, and side effects all contribute to the result. Predict the smallest useful case before running it. When the prediction and result differ, locate the first state transition that disagrees rather than rewriting the whole program.

Where it belongs in the build

Build a terminal task board that adds, completes, and lists tasks while preserving them between runs. Begin with values and input, then move behavior into functions and make persistence explicit.

Input	Validate	Transform	Observe

For this lesson, locate the exact boundary described by text input. Name the incoming value, its allowed shape, and the next function or component that consumes it. Keep that boundary separate from the whole application.

Write a concrete input and predicted result for this boundary.

Implementation slice 2 of 12

projects/01_task_board/app.py / lines 7-12

This consecutive slice is part of the complete file. It may begin or end inside a function or string. Read it in context using the adjacent slices or the companion source; it is not a standalone snippet.

```
7         raise ValueError("task file must contain a list")
8     seen = set()
9     for task in tasks:
10        if not isinstance(task, dict):
11            raise ValueError("each task must be an object")
12        if type(task.get("id")) is not int or task["id"] < 1:
```

Line 7 / This path raises ValueError('task file must contain a list'). It stops normal processing rather than returning a success-looking value.

Line 9 / The loop binds task to successive values from tasks. Trace one complete iteration before the next begins.

Line 11 / This path raises ValueError('each task must be an object'). It stops normal processing rather than returning a success-looking value.

Run context

python app.py add "Write first test". Run from the project folder after installing its requirements. Preserve any source fixtures and use a fresh output destination when the application writes a file.

Investigate a passing contract

Test / test_invalid

The excerpt below comes from the tested companion suite. A test may address several chapter concepts, so it can be revisited from a different lesson. Its fixture and assertion remain unchanged.

```
def test_invalid():
    with pytest.raises(ValueError):add_task([]," ")
    with pytest.raises(ValueError):complete([],5)
    with pytest.raises(ValueError):validate([{"id":True}])
```

Release result / 3 passed in 0.10s

Connect this test to text input. Identify what the fixture controls, what the assertion observes, and which failure path is outside its scope. A test that passes does not validate an unrelated deployment or a provider that was not contacted.

Add one boundary the current test does not establish.

Build, diagnose, explain

Focused requirement

Apply the rule from text input to one small change in Task Board CLI. First copy the passing project. Keep its baseline tests and add a new fixture that exposes the proposed change.

Preserve a passing baseline and make one change at a time. A useful extension has a new requirement, a new test, an explanation of its effect, and an unchanged result for earlier supported cases.

A model reasoning answer

strip removes outer whitespace. Validate the cleaned title rather than the original string, because a title made only of spaces is not meaningful. The implementation must make that rule observable at its boundary. State the value before the operation and the expected value or error after it. Use the relevant function or class name from the source rather than describing only a click or a printed sentence.

Diagnosis sequence

1. Reproduce the smallest failing fixture.
2. Locate the first value that differs from the prediction.
3. Check its type, identity, and ownership.
4. Change one condition or transformation.
5. Run the new case and the original suite.

Completion evidence / save the command, fixture, expected result, actual result, and the reason your change satisfies this lesson. If you have only an interface screenshot, add a state or persistence check where the requirement depends on it.

Your change, observation, and remaining limitation

Collections

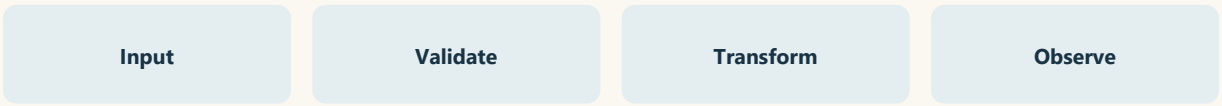
Task Board CLI

A list of task dictionaries is a useful first record model. Avoid parallel lists for titles and completion flags because their indexes can drift.

Keep the model-facing decision separate from the application permission boundary. A well-formed tool call is not permission to perform a new action. The supplied agent exercises use narrow tools and deterministic fixtures for repeatable verification. A live provider can behave differently and may incur cost. Evaluate that path separately with labeled examples and an explicit operational limit.

Where it belongs in the build

Build a terminal task board that adds, completes, and lists tasks while preserving them between runs. Begin with values and input, then move behavior into functions and make persistence explicit.



For this lesson, locate the exact boundary described by collections. Name the incoming value, its allowed shape, and the next function or component that consumes it. Keep that boundary separate from the whole application.

Write a concrete input and predicted result for this boundary.

Implementation slice 3 of 12

projects/01_task_board/app.py / lines 13-18

This consecutive slice is part of the complete file. It may begin or end inside a function or string. Read it in context using the adjacent slices or the companion source; it is not a standalone snippet.

```
            raise ValueError("invalid identity")
        if task["id"] in seen:
            raise ValueError("duplicate identity")
        seen.add(task["id"])
        if not isinstance(task.get("title"), str) or not task["title"].strip():
            raise ValueError("invalid title")
```

Line 13 / This path raises ValueError('invalid identity'). It stops normal processing rather than returning a success-looking value.

Line 15 / This path raises ValueError('duplicate identity'). It stops normal processing rather than returning a success-looking value.

Line 18 / This path raises ValueError('invalid title'). It stops normal processing rather than returning a success-looking value.

Run context

python app.py add "Write first test". Run from the project folder after installing its requirements. Preserve any source fixtures and use a fresh output destination when the application writes a file.

Investigate a passing contract

Test / test_missing

The excerpt below comes from the tested companion suite. A test may address several chapter concepts, so it can be revisited from a different lesson. Its fixture and assertion remain unchanged.

```
def test_missing(tmp_path):
    assert load(tmp_path/"missing.json")==[]
```

Required result / load(tmp_path / 'missing.json') == []

Release result / 3 passed in 0.10s

Connect this test to collections. Identify what the fixture controls, what the assertion observes, and which failure path is outside its scope. A test that passes does not validate an unrelated deployment or a provider that was not contacted.

Add one boundary the current test does not establish.