Python Foundations Workshop III

Operate Python Applications



96 runnable cases across 48 operating patterns, with three projects in durable jobs, worker recovery and release rehearsals.

Companion website

Bring your Part II skills

You should be able to run a Python module, validate input, write an assertion and follow a SQLite transaction. Part II built a local ledger and a read-only service. This volume moves a synthetic reporting job into a queue, then follows it through retries and recovery.

LOCAL SETUP

```
python --version
python -m venv .venv
```

Use Python 3.12 or newer for this edition. Use the virtual environment interpreter directly if your terminal does not activate it. On Windows it is .venv\Scripts\python.exe. On macOS and Linux it is .venv/bin/python. The bundled examples and projects use the standard library.

HOW THE LABS WORK

Work through pairs of cases. They share an implementation and test different conditions. Predict the output and compare the trace. Run the assertion, reproduce a failure in a scratch copy, then make the change request.

ONE SKILL TO REFRESH BEFORE CHAPTER 1

Run the projects locally

Extract the complete ZIP, enter its projects folder, and run the following commands with the same interpreter. Use synthetic inputs and a fresh database. Ticks are simulated integers so every claim and retry can be reproduced.

```
python queue_cli.py queue.db enqueue A1 sum 125 475 175
python queue_cli.py queue.db work worker-a --now 0
python queue_cli.py queue.db audit
python -m unittest -v
python queue_cli.py rehearsal.db rehearse rehearsal-copy.db
```

The final rehearsal uses two fresh paths and exercises duplicate submission, an expired lease, a fenced result, a transient failure and a verified snapshot. This is a local teaching queue with synthetic work. External effects need their own idempotency or reconciliation contract.

Python sqlite3 reference

SQLite transactions

SQLite backup API

Python logging reference

Python concurrent futures

State machines and contracts

Represent job lifecycles with explicit states and invariants. Distinguish a valid state value from an allowed transition.

FOUR PATTERNS AND EIGHT CASES

01.01 Claim a queued task / 6

01.02 Reject a terminal replay / 15

01.03 Route a pending identifier / 24

01.04 Separate missing from held / 33

01.05 Recognize an exact retry / 42

01.06 Hold an altered payload / 51

01.07 Validate a completed record / 60

01.08 Reject an impossible completion / 69

Explicit state makes recovery easier to reason about. Every new state increases the number of transitions you must test.

Claim a queued task

Build brief

ASSIGNMENT

Claim a queued task. Predict the result and identify the input boundary that controls this case.

```
decide("queued", "claim")
```

CHAPTER CONTEXT

Represent job lifecycles with explicit states and invariants. Distinguish a valid state value from an allowed transition.

INPUTS, RETURN SHAPE, AND FIRST EXPECTED RESULT

Claim a queued task

Reference implementation

```
def decide(state, action):
    table = {("queued","claim"):"leased",("leased","finish"):"done"}
    if (state, action) not in table:
        return "invalid transition"
    return table[state, action]
result = decide("queued", "claim")
print(repr(result))
```

Run the bundled file with Python before trying a variation.

`labs/ch01_01.py`

Claim a queued task

Observed behavior and trace

RECORDED OUTPUT

```
'leased'
```

EXECUTED LINE NUMBERS AND COUNTS

```
01 x 1    02 x 1    03 x 1    05 x 1    06 x 1    07 x 1
```

Trace counts follow the current Python thread. Some callbacks or worker bodies can run in another thread. Compare the output with the contract even if some worker lines are absent from this trace.

FIND THE LINE RESPONSIBLE FOR A VISIBLE OUTPUT

Claim a queued task

Verification

Run this check after the reference implementation. The bundled script includes the check, and a successful run means the assertion passed as well as producing the recorded output.

```
expected = "leased"
assert type(result) is type(expected)
assert result == expected
```

DESIGN A BOUNDARY CHECK

A state value alone is not enough to authorize every action. Restrict transitions by state and action together.

SMALLEST INPUT, EXPECTED RESULT, AND WHAT THE ASSERTION PROVES

Claim a queued task

Diagnose a failure

DEFECT TO INVESTIGATE

A state value alone is not enough to authorize every action. Restrict transitions by state and action together.

REPRODUCE IN A SCRATCH COPY

Write a counterexample based on the defect. Run it first against the reference, then make one deliberately flawed edit that introduces the defect. Record the changed output or exception before restoring the reference.

ORIGINAL LINE AND CHANGED LINE

OBSERVED DIFFERENCE

Claim a queued task

Compare design choices

ENGINEERING CHOICE

Explicit state makes recovery easier to reason about. Every new state increases the number of transitions you must test.

PAIRED CASE

Reject a terminal replay

```
decide("done", "finish")
# Expected: "invalid transition"
```

Compare the paired case with the original. Name the condition that changed and explain which behavior should remain invariant.

ALTERNATIVE AND DECISION CRITERION

Claim a queued task

Implement the change request

NEW REQUIREMENT

Add an explicit failed terminal state and an allowed transition into it.

PRESERVE A PASSING BEHAVIOR

```
expected = "leased"
assert type(result) is type(expected)
assert result == expected
```

Decide whether this original check still describes the supported interface. Add a new check for the changed behavior and keep it alongside the original when both contracts apply.

CHANGE AND NEW VERIFICATION

Claim a queued task

Integrate it with a caller

CALLER CONTRACT

Claim a queued task. Predict the result and identify the input boundary that controls this case.

Sketch a second caller that supplies data from a terminal command, an import file, or a fixture. Keep the calculation independent from that input boundary. State which errors the caller can handle and which should stop the workflow.

REVIEW THE HANDOFF

Add an explicit failed terminal state and an allowed transition into it.

INPUTS, OUTPUT, AND A FAILURE THE CALLER MUST HANDLE

Claim a queued task

Acceptance record

Submit the original run, one counterexample, and one version that handles the change request. Keep the script and invocation together so another reader can reproduce your result.

EXPECTED BEHAVIOR

Claim a queued task. Predict the result and identify the input boundary that controls this case.

REGRESSION TO CATCH

A state value alone is not enough to authorize every action. Restrict transitions by state and action together.

COMMAND, OBSERVED RESULT, AND REMAINING LIMIT

Reject a terminal replay

Build brief

ASSIGNMENT

Reject a terminal replay. Predict the result and identify the input boundary that controls this case.

```
decide("done", "finish")
```

CHAPTER CONTEXT

Represent job lifecycles with explicit states and invariants. Distinguish a valid state value from an allowed transition.

INPUTS, RETURN SHAPE, AND FIRST EXPECTED RESULT

Reject a terminal replay

Reference implementation

```
def decide(state, action):
    table = {("queued","claim"):"leased",("leased","finish"):"done"}
    if (state, action) not in table:
        return "invalid transition"
    return table[state, action]
result = decide("done", "finish")
print(repr(result))
```

Run the bundled file with Python before trying a variation.

`labs/ch01_02.py`

Reject a terminal replay

Observed behavior and trace

RECORDED OUTPUT

```
'invalid transition'
```

EXECUTED LINE NUMBERS AND COUNTS

```
01 x 1    02 x 1    03 x 1    04 x 1    06 x 1    07 x 1
```

Trace counts follow the current Python thread. Some callbacks or worker bodies can run in another thread. Compare the output with the contract even if some worker lines are absent from this trace.

FIND THE LINE RESPONSIBLE FOR A VISIBLE OUTPUT