Repair and refactor a broken Python application

Copilot in Practice

Session M2-06 | Intermediate | 3 hours

Agenda

Segment	Time
Start from the failure output	8 min
Python failure order	10 min
Prepared repair demonstration	12 min
Recover behavior from fixtures	12 min
Lock behavior with tests	10 min
Refactor and lab handoff	8 min

The inherited application

The inventory tool should:

- read warehouse and ledger CSV files;
- normalize and aggregate SKU quantities;
- calculate stock variance;
- write stable text and JSON reports.

It currently does none of this reliably.

The broken-state contract

Stage	Named fault
Parsing	`SYN-01`, then `SYN-02`
Startup	`IMP-01`, then `IMP-02`
Behavior	`LOG-01` through `LOG-03`
Repeatability	`STATE-01`
Design	`DESIGN-01`

Each repaired fault lets execution reach the next failure.

Start at the lowest blocked layer

```
parse
  ↓
import
  ↓
start
  ↓
behavior
  ↓
structure
```

Do not refactor code that Python cannot import.

Read the traceback from the bottom

Capture four facts:

1. exception type;
2. file and line;
3. failing source text;
4. command that produced it.

That is enough to form the first hypothesis.

Use Copilot inside a boundary

```
Explain this SyntaxError using only the shown file and traceback.
Identify the smallest syntax repair.
Do not refactor or change behavior.
```

The prompt asks for a syntax-only change. Check that the proposal stays in scope.

Repair loop

```
capture failure
  → state one hypothesis
  → make one small change
  → run one narrow check
  → record the result
```

If the next failure changes, the repair probably moved execution forward.

Prepared demonstration

```
python -m inventory_reconciliation \
  --warehouse fixtures/input/warehouse.csv \
  --ledger fixtures/input/ledger.csv \
  --output-dir build
```

Expected first stop: `SYN-01` in `cli.py`.

Narrow checks beat noisy checks

Goal	Command
Parse package	`python -m compileall inventory_reconciliation`
Import entry point	`python -c "import inventory_reconciliation.cli"`
Check startup	`python -m inventory_reconciliation --help`
Check behavior	`pytest -q`

Use the smallest command that can disprove the current hypothesis.

Expected outputs are the contract

```
SKU     warehouse  ledger  variance  status  severity
AX-100         10      10         0  MATCH   none
BX-200          4       9        -5  SHORT   review
CX-300          0       8        -8  SHORT   urgent
```

Do not edit the contract to hide a defect.

Intended reconciliation rules

Rule	Expected behavior
SKU identity	Trim and uppercase
Duplicate rows	Sum after normalization
Missing side	Use quantity zero
Variance	Warehouse minus ledger
Output order	Sort by SKU
Repeated run	No prior results retained

Classification belongs in one place

Variance	Status	Severity
`0`	`MATCH`	`none`
`-1` to `-5` or `1` to `5`	`SHORT` or `OVER`	`review`
absolute value `6+`	`SHORT` or `OVER`	`urgent`

The starter's duplicated rules disagree. Keep one implementation.

Reproduce each defect with the same fixtures

The same fixtures always expose:

- overwritten duplicate quantities;
- ledger-only SKUs that disappear;
- the wrong urgent threshold;
- stale results on a second call.

Use these repeatable failures to test each diagnosis.

Add tests before changing structure

Keep the fixture-level checks. Add focused tests for:

- severity at absolute variance `6`;
- negative quantity rejection with row context.

Refactor only after the tests establish the expected behavior.

Global state creates delayed failures

```
RUN_HISTORY = []

def reconcile(...):
    RUN_HISTORY.extend(current_results)
    return RUN_HISTORY
```

The first call may look correct. The second call exposes the leak.

Tangled control flow hides policy

```
read rows
  + normalize values
  + merge quantities
  + classify variance
  + build output
  + retain state
```

Separate these jobs so each rule has one implementation.

Target design

```
cli.py
  → parsing.py
  → service.py
  → rules.py
  → reporting.py
```

The command-line layer coordinates. It does not own business rules.

Refactor in small moves

1. Remove process-wide mutable results.
2. Extract one classification function.
3. Reuse it from service and reporting.
4. Extract parsing and aggregation.
5. Run tests after each move.

Keep public behavior stable.

Use Python LSP after the tests are green

```
/lsp show
/lsp test python
```

Ask for the definition and references of `severity_for`.
Inspect the actual LSP operation and check its locations.

Then read `service.py` for inline duplicate rules: they are not necessarily references to that function.

Approve `/plan` before structural edits. The lookup uses five minutes of the existing 25-minute refactor block.

Review Copilot output like a patch

Ask:

- Did the change touch only the current fault?
- Which test proves the claimed behavior?
- Did a rule move or get copied?
- Does output order stay deterministic?
- What assumption is still unproved?

Check generated code against tests and fixtures.

Lab workflow

Part	Time
Preflight and baseline	10 min
Restore parsing and imports	30 min
Recover behavior	30 min
Add regression tests	20 min
Refactor and check	25 min
Final output comparison and tests	5 min

Final deliverable

- completed repair log;
- runnable inventory command;
- added regression tests;
- unchanged expected outputs;
- clean separation of parsing, rules, service, reporting, and CLI;
- passing `pytest -q`.

Lab handoff

Work in failure order. Record every repair.

The solution must produce the supplied text and JSON outputs, pass all tests, and return the same result on repeated runs.