Lecture 12 · Practice Exercises

File Handling: TXT, CSV and JSON

Exercise structure and progressive support

Guided (1): every logical stage is explained and scaffolded in the starter.

Laboratory (2): implement essential functions with targeted hints in the starter.

Individual (3): exam-style specifications with minimal implementation support.

All exercises specify context, input contract, required behaviour, output and edge cases.

Reference data are tests, never values to hard-code.

The problem statement must be sufficient even when the instructor is not present.

Guided · Transaction Register

Exercise 1 / 6 · Lecture 12

Guided: problem context and inputs

PROBLEM CONTEXT

An organization records financial movements in a CSV ledger. At the end of a reporting period, it wants a text summary that can be reopened after the Python program exits.

INPUT CONTRACT

Input file `transactions.csv` with header `description,amount` and rows like `Coffee,-3.5; Salary,1200; Rent,-450`. Amounts are strings in CSV.

REQUIRED BEHAVIOUR

Read with `csv.DictReader`, convert amount values, compute **row count** and **balance**, and write `report.txt` with labeled lines. Display a short completion message.

OUTPUT / REFERENCE TEST

`report.txt: Count: 3 and Balance: 746.50.`

Opening the report in a text editor must show those values.

A sample input illustrates the format; your solution must handle every input satisfying this contract.

Guided: implementation walkthrough

FOLLOW THE STAGES

1. Identify the input source and expected types
2. Choose functions with one responsibility each
3. Implement the provided reference case
4. Inspect intermediate results and the output
5. Change the input and run a new check

Trace each stage: CSV text → dictionaries → floats → sum → text report.

DESIGN REASON

Use with and `newline=""` with `csv`; do not recreate the input file on every run

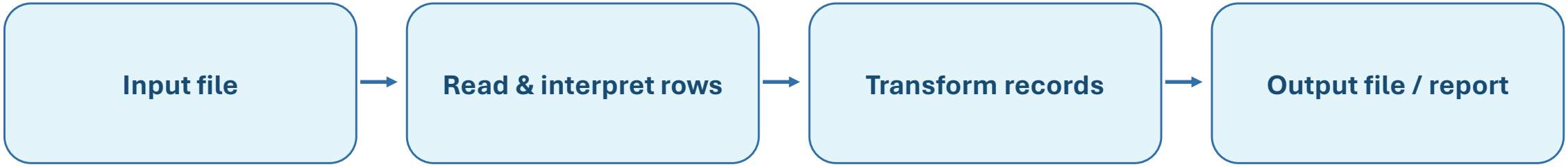
EDGE CASE TO TEST

Test an empty valid CSV file (header only).

The Guided starter contains an explicit scaffold: follow and understand it step by step.

Transaction Register: input → processing → output

Follow the data through the program before writing code.



Write the contracts before coding; one example is not the full specification.

Laboratory 1 · Expense Tracker

Exercise 2 / 6 · Lecture 12

Laboratory 1: problem context and inputs

PROBLEM CONTEXT

A student logs daily expenses in a CSV file and wants yesterday's expenses to remain when the tool is executed again.

INPUT CONTRACT

Data file `expenses.csv` with columns `category,amount`. New expense records have a nonempty category and nonnegative amount.

REQUIRED BEHAVIOUR

Implement `add_expense(path, category, amount)` and `total_expenses(path)`. Append rather than overwrite, write the header only when creating a file, and read all rows back to calculate the total.

OUTPUT / REFERENCE TEST

- Add `"Food 12"` and `"Travel 8"` → CSV with two data rows; `total 20.00`. Run a third addition and check that the first two remain.
- The reference case is one correctness check; it is not the definition of the algorithm.

A sample input illustrates the format; your solution must handle every input satisfying this contract.

Laboratory 1: targeted hints and tests

HINTS

1. Hint: choose mode a for appending; plan when the header should be written.
2. Decide which part belongs in a separate function.
3. Test normal and boundary data.
4. Verify generated messages and files.

WHY / EDGE CASE

DESIGN REASON

Preserving previous records is the central requirement.

EDGE CASE TO TEST

Test a new file and an already-existing file.

Laboratory 2 · Portfolio JSON Archive

Exercise 3 / 6 · Lecture 12

Laboratory 2: problem context and inputs

PROBLEM CONTEXT

An analyst wants to save a nested portfolio and then load the same structure in a new program run.

INPUT CONTRACT

A dictionary with owner (string) and assets (list of dictionaries with name and units). Example: Bond `3 units` and `Stock 5 units`.

REQUIRED BEHAVIOUR

Write `save_portfolio(portfolio, path)` using `json.dump(..., indent=4)`, `load_portfolio(path)` with `json.load`, and `total_units(data)`

OUTPUT / REFERENCE TEST

After saving/loading: `owner Alice; total units 8.` The `portfolio.json` file must contain a readable array of assets.

A sample input illustrates the format; your solution must handle every input satisfying this contract.

Laboratory 2: targeted hints and tests

HINTS

1. Hint: JSON maps nested dictionaries and lists; loading reconstructs them.
2. Decide which part belongs in a separate function.
3. Test normal and boundary data.
4. Verify generated messages and files.

WHY / EDGE CASE

DESIGN REASON

Saving is different from printing a dictionary on screen.

EDGE CASE TO TEST

Add a third asset and verify the total changes.

Individual 1 · Student Records Exporter

Exercise 4 / 6 · Lecture 12

Individual 1: problem context and inputs

PROBLEM CONTEXT

A department exports student scores to a portable CSV file. The exported file will be used by another script to calculate the class mean.

INPUT CONTRACT

List of records: `{student_id, score}`; e.g. `S1:12, S2:16, S3:14`. Input records are provided in the script.

REQUIRED BEHAVIOUR

Write CSV using `DictWriter` with correct header, reopen with `DictReader`, convert scores to float, and compute the mean. Decide what the function returns for an empty list.

OUTPUT / REFERENCE TEST

`Mean score: 14.00;` the generated `students.csv` must have one header and three data rows.

A sample input illustrates the format; your solution must handle every input satisfying this contract.

Independent: verification and corner cases

SELF-CHECK

- Follow the specified input contract.
- Hint: one dictionary is one CSV row.
- Compare output with the reference test.
- Verify that your code works with other valid inputs.

WHY / EDGE CASE

DESIGN REASON

The mean must come from the file contents, not the original list.

EDGE CASE TO TEST

Test one student and no students.

Individual 2 · Bank Statement Summarizer

Exercise 5 / 6 · Lecture 12

Individual 2: problem context and inputs

PROBLEM CONTEXT

A bank statement contains financial movements. Positive values are credits; negative values are debits; the customer needs a persistent summary.

INPUT CONTRACT

`statement.csv` contains only an `amount` column and any number of numeric rows, e.g. 120, -30, -20.

REQUIRED BEHAVIOUR

Read the CSV, calculate **count**, **credits sum**, **debits sum** (negative), and **net sum**; save a labeled text file `statement_report.txt`.

OUTPUT / REFERENCE TEST

Reference: `count 3; credits 120.00; debits -50.00; net 70.00.`

A sample input illustrates the format; your solution must handle every input satisfying this contract.

Independent: verification and corner cases

SELF-CHECK

- Follow the specified input contract.
- Hint: accumulate positive and negative amounts separately.
- Compare output with the reference test.
- Verify that your code works with other valid inputs.

WHY / EDGE CASE

- DESIGN REASON
- The saved report is part of the required output.
- EDGE CASE TO TEST
- Test zero-valued transactions and a statement containing only debits.

Independent starters are intentionally minimal: you choose the design.

Individual 3 · Inventory Report Generator

Exercise 6 / 6 · Lecture 12

Individual 3: problem context and inputs

PROBLEM CONTEXT

A retailer stores stock in a flat CSV table but requires a nested JSON report containing per-product inventory values and an overall total.

INPUT CONTRACT

Input `stock.csv` columns name,quantity,unit_price. Reference: `A,2,10; B,3,7`. Quantities are whole nonnegative numbers; prices are nonnegative numbers.

REQUIRED BEHAVIOUR

Read CSV; compute `value=quantity×unit_price` for each product; save products list and `total_value` in `stock_report.json`.

OUTPUT / REFERENCE TEST

`Product values 20.00 and 21.00; total_value 41.00.` JSON must preserve names, quantities and prices.

Independent: verification and corner cases

SELF-CHECK

Follow the specified input contract.

Hint: the output is a dictionary containing a list of product dictionaries.

Compare output with the reference test.

Verify that your code works with other valid inputs.

WHY / EDGE CASE

DESIGN REASON

Be precise about int versus float conversion.

EDGE CASE TO TEST

Test zero quantity and an additional product.