Lecture 11 — Practice Exercises

Modular Programming and Command-Line Arguments

Exercise progression

LEARNING PATH

- GUIDED (5–10 min): two small functions in a separate module; compare sys.argv and argparse.
- LABORATORY (2): add one technique at a time — a validated CLI, then a list of inputs.
- INDIVIDUAL (3): combine modules and CLI with earlier topics (conditions, loops, lists).

COMMON RULE

- Keep calculations in reusable functions; use `main.py` for arguments and printing.
- Function contracts specify parameters, required operation and return value.
- All tasks require the Python Standard Library only.

Guided — Rectangle CLI

Two functions, two CLI approaches · Instructor demonstration: 5–10 min

Guided — A tiny reusable module

FUNCTIONS TO COMPLETE

- Create `geometry.py` with TWO functions:
- `area(width, height)`: return width * height.
- `perimeter(width, height)`: return 2 * (width + height).
- Both take positive integers and return integers; neither uses `input()` nor `print()`.

PROGRAM BEHAVIOUR

- Create two entry-point files: `main_argv.py` and `main_argparse.py`
- Each reads WIDTH and HEIGHT from the command line, imports `geometry` and prints both results
- Example: `python main_argv.py 3 4`
- Area: 12 | Perimeter: 14

Guided — Compare argv with argparse

WHAT THE STARTERS SHOW

- `main_argv.py`: `sys.argv` contains strings; the supplied starter checks argument count, digit-only values and positivity
- `main_argparse.py`: `ArgumentParser` handles missing arguments and `int` conversion; the starter checks positivity
- Both validation approaches are ALREADY implemented: read them, do not rewrite them.

LIVE DEMONSTRATION

- Complete only the two functions in `geometry.py`; use the same file with either CLI.
- Run both commands and compare the output:
- `python main_argv.py 3 4`
- `python main_argparse.py 3 4`
- Try missing arguments, text and zero. Observe the different error messages.

Laboratory

Two exercises with additional details

Lab 1 — Loan repayment

REQUIRED FUNCTION

- Write `calculations.py` with `monthly_payment(principal, months)`
- **Parameters**: positive amount principal (float) and positive number of months (int).
- **Return**: *principal / months*, without rounding or printing. Interest is NOT included.
- In `main.py`, use `argparse` for the two positional arguments and print the result.

CLI AND TEST

- Command: `python main.py 1200 12`
- Expected: Monthly payment: 100.00
- Use `type=float` for PRINCIPAL and `type=int` for MONTHS
- Check that both are > 0 in `main.py`; reject invalid values
- Hint: keep validation and printing outside `calculations.py`

Lab 2 — Three-item budget

ONE NEW IDEA: A LIST

- Write `allocation.py` with `remaining_budget(budget, items)`
- **Parameters**: a nonnegative budget (float) and a list containing exactly three nonnegative costs
- **Return**: `budget - sum(items)`, which may be negative.
- In `main.py`, get the budget and three costs with `argparse`.

CLI AND TEST

- Command: `python main.py 100 20 30 10`
- Expected: Remaining: 40.00
- Also print *Status: within budget*, or *Status: over budget* if remaining < 0
- Hint: use `nargs=3`, `type=float` for the costs. Check nonnegative values.

Individual — Integrative problems

Implement the complete specification independently

Individual 1 — Currency receipt

FUNCTION CONTRACT

- Build `rates.py`:
  - `convert(amount, rate)` returns *amount * rate*;
  - `is_large(value, limit)` returns value >= limit
- Build `main.py`: parse AMOUNT, RATE, and optional `--limit` (default 100)
- Inputs: AMOUNT >= 0, RATE > 0, LIMIT >= 0. Validate them before calling the functions.
- Display the converted value and whether it reaches the threshold.

REFERENCE BEHAVIOUR

- `python main.py 50 1.2 --limit 55`
- Converted: 60.00
- Threshold reached: True
- Implementation choice: functions return values, not formatted strings
- Test: zero amount; limit equal to converted value; invalid rate.

Individual 2 — Investment growth

CALCULATION & OPTIONAL LOOP

- Build `finance.py` with `compound_value(principal, rate, years)`
- Return principal * (1 + rate) ** years
- Build `main.py` using `argparse` for PRINCIPAL, RATE, YEARS and optional `--yearly`
- Validate: principal >= 0, rate > -1, years >= 0 integer.
- If `--yearly` is present, print values for years 0 through YEARS.

REFERENCE BEHAVIOUR

- `python main.py 1000 0.05 2`
  - Final value: 1102.50
- `python main.py 1000 0.05 2 --yearly`
  - Year 0: 1000.00
  - Year 1: 1050.00
  - Year 2: 1102.50
- Test zero years and a negative rate greater than -1.

Individual 3 — Statistics toolkit

FUNCTIONS & CLI

- Build `stats_tools.py`: `mean(values)`, `minimum(values)`, `maximum(values)`
- Each receives a nonempty list of floats and returns a numeric result.
- Build `main.py`: read one or more values as positional arguments (`nargs="+"`), plus `--operation` with choices mean/min/max/all (default all)
- Print only the requested statistic(s).

REFERENCE BEHAVIOUR

- `python main.py 10 20 30 --operation all`
- Mean: 20.00
- Min: 10.00
- Max: 30.00
- `--operation max prints only Max: 30.00`
- Test one value, negative values, nonnumeric input and `--help`.

Before submitting

CHECKLIST

- The requested filenames and function names match the specifications.
- Every reusable function returns a value; the CLI owns parsing and formatted output.
- Programs work with inputs other than the examples.

TESTING

- Guided: compare both parsing approaches, not their output formatting.
- Labs: run the reference command plus one invalid input.
- Individuals: test one boundary case and one invalid input.