Lab 10 · Part 4 Abstraction and Organization AI Level 1

Classes II: Inheritance, Special Methods and Modules

Extend Rover with a new role, give shapes, money and cards useful interfaces, then build a real three-file shape report.

Due: Homework 4 · Tue 8 Dec, 23:59

From the classroom to the lab

The expedition now uses inheritance for related roles and special methods for Python operations. Four short tasks practise those mechanisms. Finally, reuse the shape task to build a **standalone area-report program**.

Download starters and public tests ZIP · Open HW4 submission

Use Python 3.13 and the standard library. In each of the first four task folders, edit `submission.py` and run `python -m pytest -q test_public.py`. Install pytest with `python -m pip install pytest` if needed. The module project has its own three-file folder and packing script.

Part	Tasks	Automatic	Staff
A	Shapes, Rover subclass, Money, Deck	16	8
B	Three-file shape report	8	4
C	Optional environment observation	0	0

Week 10 totals **24 automatic + 12 staff points**. HW4 covers weeks 10–12; use the submission page and course calendar for its deadline.

AI use: Level 1

You may use chat to discuss an approach, explain errors, debug or revise a portion of code. Complete the overall implementation yourself; do not request a complete task solution or use a coding agent to edit and run the project automatically. **Complete the AI-use declaration once, at the end of HW4.** It appears in your evaluation report; individual tasks and written cards do not require duplicate declarations. Declare nonuse honestly too. Submit only the evaluation report as before.

Each code entry is standalone. Retain the supplied helpers/base class and do not depend on code previously run in class. Annotate completed method interfaces; `self` needs no annotation. Tests call your interfaces; do not use `input()`.

Part A · Classes and special methods

10-1 Shape family

Goal: let the same `total_area()` work with different shapes.

```
class Shape:
    def area(self) -> float: ...
    def name(self) -> str: ...
class Circle(Shape):
    def __init__(self, r: float) -> None: ...
    def area(self) -> float: ...
class Rectangle(Shape):
    def __init__(self, w: float, h: float) -> None: ...
    def area(self) -> float: ...
class Triangle(Shape):
    def __init__(self, a: float, b: float, c: float) -> None: ...
    def area(self) -> float: ...
def total_area(shapes: list[Shape]) -> float: ...
```

The starter supplies `positive_dimensions(*values)` for finite positive dimensions and `triangle_area(a, b, c)` for triangle validation and area. Both raise `ValueError` for

invalid dimensions. Use these helpers directly; no formula derivation is required. Use `math.pi` for circles.

- `Shape.area()` raises `NotImplementedError`; `name()` returns the actual class name, such as `Circle`. The three subclasses store their own dimensions and override `area()`.
- Call the supplied helpers during construction to reject invalid dimensions, including degenerate triangles.
- `total_area()` calls each object's `area()`. Return `0` for an empty list and preserve the input list and objects. New Shape subclasses must work without branching on class names.

```
items = [Circle(1), Rectangle(2, 3), Triangle(3, 4, 5)]
# [item.name() for item in items] -> ["Circle", "Rectangle", "Triangle"]
# total_area(items) -> math.pi + 12
```

Entry: `w10-shapes`, 4 automatic points. Numerical tests use moderate real values; no conversions for unrelated object types are required.

10-2 A new Rover role

Goal: reuse initialization and movement while adding survey records.

The starter fully supplies `Rover(name, x=0, y=0)`. Each instance has `name`, `x`, `y` and an independent `bag` list. `move(dx, dy)` updates coordinates and `collect(item)` appends to the bag; both return `None`. `report()` returns a string such as `"Ada @ (3, 4)"`.

Keep the base class and complete three methods:

```
class SurveyRover(Rover):
    def __init__(self, name: str, x: int = 0, y: int = 0) -> None: ...
    def record(self, sample: str) -> None: ...
    def report(self) -> str: ...
```

1. Initialize the inherited state through `super().__init__()`, then create an independent `samples` list for each instance.
2. `record(sample)` appends a string, allows duplicates and returns `None`. Samples and bag items are separate lists.
3. Extend `super().report()` with `" | samples: N"`, where `N` is the sample count. Inherit `move()` and `collect()` without overriding them.

```
r = SurveyRover("Ada", 3, 4)
r.move(2, -1)
r.collect("apple")
r.record("rock")
# r.report() -> "Ada @ (5, 3) | samples: 1"
# r.bag -> ["apple"]; r.samples -> ["rock"]
```

Names, items and samples are strings; coordinates and offsets are integers. No map boundaries or additional game rules are required. Public tests temporarily replace base methods to check that the subclass reuses base initialization/reporting rather than copying their implementation.

Entry: `w10-rover`, 4 automatic points.

10-3 Money

Goal: support representation, arithmetic and comparison for money objects.

The supplied `to_cents(yuan)` validates and converts yuan into integer cents. For example, `1.005` becomes `101` and `-1.005` becomes `-101`. It accepts finite `int/float` values with magnitude at most `10**9`; otherwise it raises `ValueError`. Use this helper directly; implementing Decimal rounding is not part of this task.

```
class Money:
    def __init__(self, yuan: float) -> None: ...
    def __repr__(self) -> str: ...
    def __add__(self, other: "Money") -> "Money": ...
    def __sub__(self, other: "Money") -> "Money": ...
    def __eq__(self, other: object) -> bool: ...
    def __lt__(self, other: "Money") -> bool: ...
```

- Store `to_cents(yuan)` in `self._cents`. Repr uses exactly two decimal places: `Money(12.50)`, `Money(-1.01)`, `Money(0.00)`.
- Addition/subtraction return **new Money objects**, preserving operands. Keep integer cents even for results beyond the constructor's input bound. You can create `Money(0)` and set the result's `_cents` without a floating-point conversion.
- Equality and ordering compare cents. All four binary special methods return `NotImplemented` for non-Money operands so Python can handle the remaining protocol.

```
# Money(0.1) + Money(0.2) == Money(0.3) -> True
# repr(Money(10.5) - Money(0.25)) -> "Money(10.25)"
# Money(1) == object() -> False
# Money(1) + 1 -> TypeError
```

Entry: `w10-money`, 4 automatic points.

10-4 Deck

Goal: use a custom deck through Python's container operations.

Construction and `shuffle()` are supplied; retain them. Each instance owns a `self._cards` list containing 52 unique rank+suit strings. Ranks are `A/2/…/10/J/Q/K`, grouped by suits `C/D/H/S`.

```
class Deck:
    def __init__(self) -> None: ...       # supplied
    def __len__(self) -> int: ...
    def __getitem__(self, i: int) -> str: ...
    def __contains__(self, card: str) -> bool: ...
    def shuffle(self) -> None: ...        # supplied
    def deal(self, n: int) -> list[str]: ...
```

- Length is the number of remaining cards. Support positive/negative integer indices and raise `IndexError` out of bounds. Membership reports whether the card remains.
- `deal(n)` removes `n` cards from the **front** and returns a separate new list. Require a nonnegative integer no larger than the remainder; otherwise raise `ValueError` without mutation. Zero returns an empty list.
- Deck instances are independent. Raising `IndexError` for invalid indices also allows sequence-style `list(deck)`; a separate `__iter__()` is unnecessary.

```
# deck = Deck(): len(deck) -> 52; deck[0] -> "AC"; deck[-1] -> "KS"
# deck.deal(2) -> ["AC", "2C"]; len(deck) -> 50; "AC" in deck -> False
```

The provided shuffle calls `random.shuffle(self._cards)` in place and returns `None`. Tests replace randomness with a known permutation to check delegation. Random algorithms are not assessed, and a changed order on every call is not required.

Entry: `w10-deck`, 4 automatic points.

A mechanisms and verification

Entry `w10-a-notes` has two responses worth 4 staff points each:

1. Use your SurveyRover to explain which attributes `super().__init__()` creates and how `report()` extends base behavior. Give operations and actual results for two instances to check list sharing.
2. Choose an expression involving Money or Deck and explain its special method. Give one unsupported input or state-changing operation, expected/actual results and whether the original object changes.

A short explanation and a small example per response are sufficient; do not repeat every interface specification.

Part B · Three-file shape report

10-5 Separate modules and imports

Goal: reuse task 10-1 in a real multi-file report program.

The download's `module-project/` folder contains three project files and a supplied tool:

```
shapes.py             # reuse task 10-1
report.py             # format the report
main.py               # run the demo
pack_submission.py    # supplied packer
```

After completing it, `python main.py` must print:

```
Rectangle: 6.00
Triangle: 6.00
Total: 12.00
```

Interfaces:

- `shapes.py`: reuse your completed first task without depending on `submission.py`.
- `report.py`: implement `build_report(items: list[Shape]) -> list[str]`. Produce one line per object in input order using `name()` and `area()`, with two decimal places. Add a Total line using `total_area(items)` imported from shapes. Empty input returns `["Total: 0.00"]`. Return text without printing or changing inputs; new Shape subclasses must work too.
- `main.py`: implement `run_demo() -> None`. Create `Rectangle(2, 3)` and `Triangle(3, 4, 5)` using the classes from shapes, call `build_report()` from report and print each returned line. Protect entry with `if __name__ == "__main__":`.

Importing any of the three modules must be silent and must not start the demo. Both `import module` and `from module import name` are allowed. Dependencies run from

`main → report → shapes`, with main also using shapes' constructors. Public tests temporarily replace the report function to verify real integration; printing a fixed answer cannot pass.

Submission workflow:

```
python main.py
python -c "import shapes, report, main"
python pack_submission.py
python -m pytest -q test_public.py
```

The second command produces no output. The packer generates `submission.py`, whose `project_files() -> dict[str, str]` returns exactly the three filenames and their sources. Paste **the generated file's contents** into `w10-module-project`. Repack after each project edit; no manual source-string assembly is needed. Download the packer separately.

Only those three files are evaluated. Use the standard library without local-file reads, network access, `input()` or GUI windows. `project_files()` returns source without running the project.

Entry `w10-module-project`: 8 automatic points. Written entry `w10-b-notes`: 2 staff points for responsibilities and adding new shapes; 2 for actual direct-run/import commands, output and explanation.

Part C · Environment observation (optional)

Run inside the module project:

```
python -c "import sys; print(sys.executable)"
python -c "import shapes; print(shapes.__file__)"
```

Identify the interpreter and imported file. You may record observations in `w10-c-tools` or leave it blank without affecting completion or marks.

4. Solo assignment 2 launch

Solo assignment 2: five prime-counting methods opens this week and is separate from HW4. Use its submission page and the course calendar for the deadline. Read the task first; L11 continues the algorithm details.

▶ Full solo assignment 2 requirements (also shown on its submission page)

Submission checklist

- Submit four class tasks and the module project separately; complete the two short written cards with mechanisms and actual results.
- Repack after editing the project; direct execution prints a report and imports are silent.
- Complete HW4's single AI-use declaration and submit the downloaded evaluation report.