Lecture 18 | OOP: exercises

Classes, instances, attributes and first methods

Six problems, increasing autonomy

- Guided 01 — Water tanks: creating two independent objects and changing an attribute
- Lab 01 — Library stock: writing simple methods and inspecting updated state
- Lab 02 — Bicycle stations: instance state versus shared class information
- Independent 01 — Laboratory samples: defining and using a class independently
- Independent 02 — Sports scoreboard: several objects stored in a list
- Independent 03 — Parcel fees: using objects in a CSV-based reporting program

Guided 01 | Community garden water tanks

- Context: Alice and Bob each manage a separate water tank in a community garden
- Create a class `WaterTank`, and two instances of this class; both must store three instance attributes: `owner`, `liters` and `capacity`
- The `owner` is a name, `liters` is the current quantity of water and `capacity` is the tank's nominal maximum
- Define the `WaterTank` class in `water_tank.py` and `.add_water(amount)` to increase the quantity and `.take_water(amount)` to decrease it
- Use `main.py` to import `WaterTank`, construct both objects and call their methods
- The goal is to understand how methods modify the state of one particular object

Guided 01 | Building the class

- In `water_tank.py`, complete `__init__(self, owner, liters, capacity)`, store each input as an attribute using `self`
- In `.add_water(self, amount)`, increase the current quantity stored in `self.liters`
- In `.take_water(self, amount)`, decrease the current quantity stored in `self.liters`
- Use the addition and subtraction assignment operators already studied
- Do not add capacity limits or warning messages: those behaviors are studied later

`water_tank.py` **— class structure to complete**

```
class WaterTank:
    def __init__(self, owner, liters, capacity):
        # TODO: store three instance attributes
        ...

    def add_water(self, amount):
        # TODO: increase self.liters
        ...

    def take_water(self, amount):
        # TODO: decrease self.liters
        ...
```

Guided 01 | Create, modify, inspect

- In `main.py`, import `WaterTank`, then create `wt_alice` with 40 L and `wt_bob` with 70 L, both with a nominal capacity of 80 L
- Add 25 L, then another 30 L to Alice's tank: the quantity becomes 95 L
- Remove 20 L, then another 70 L from Bob's tank: the quantity becomes -20 L
- Print the owner, final water quantity and nominal capacity of each tank
- Observe that each method modifies only the object on which it is called

`main.py` — **prepared test code**

```
from water_tank import WaterTank

wt_alice = WaterTank('Alice', 40, 80)

wt_bob = WaterTank('Bob', 70, 80)

wt_alice.add_water(25)
wt_alice.add_water(30)
wt_bob.take_water(20)
wt_bob.take_water(70)

print(wt_alice.owner, wt_alice.liters)
print(wt_bob.owner, wt_bob.liters)
```

Guided 01 | Verify what changed

- After `wt_alice.add_water(25)`, Alice's tank contains 65 L
- After another addition of 30 L, it contains 95 L, although its nominal capacity is 80 L
- After `wt_bob.take_water(20)`, Bob's tank contains 50 L
- After another removal of 70 L, it contains -20 L
- The `WaterTank` class is defined in `water_tank.py` and used through an import in `main.py`; unrestricted updates do not raise a Python error
- What is problematic about these results, even though the code runs without an error?

Lab 01 | Library copy tracking

- A small library manages the number of available copies of each title
- Define a `BookStock` class with two instance attributes: title (str) and copies (int)
- Write `.add_copies(amount)` to increase the available number and `.show_stock()` to print title and copies
- Create Python Basics with 4 copies and Data Stories with 2 copies
- Receive 3 additional copies of Python Basics and 1 of Data Stories
- Use the methods to display final stock: Python Basics 7, Data Stories 3

Lab 01 | Implementation guidance

- The starter provides the class skeleton and constructs the two books
- Complete `__init__()` by storing title and copies as instance attributes
- Both methods take self as first parameter; only `.add_copies()` needs another input
- In `.add_copies()`, update the copies stored on the current object using self
- The display method reads attributes and prints a descriptive line
- Run the script to confirm the two final quantities

Prepared test code

```
stock_python = BookStock('Python Basics', 4)
stock_stories = BookStock('Data Stories', 2)


stock_python.add_copies(3)
stock_stories.add_copies(1)


stock_python.show_stock()
stock_stories.show_stock()
```

Lab 01 | Interpreting the program

- One class definition can create any number of book objects
- The methods are shared behaviors, but `self.title` and `self.copies` belong to each particular instance
- Avoid using bare variables such as `copies` inside `.add_copies()`: they are not the stored attribute
- No file reading, validation or new library is necessary: this exercise focuses on class syntax and state
- Check that changing one book has no effect on the number stored by the other

Lab 02 | Bicycle-sharing stations

- A city bicycle service operates two stations with their own available bicycle counts
- Define `BikeStation` in `bike_station.py` with instance attributes `location` and `available`
- Create a **class attribute** `city = "Lisbon"`: it describes the location shared by all stations
- Define `.receive_bikes(count)` to increase the available bicycles for one station
- In `main.py`, import `BikeStation`, create `riverside` with 8 bicycles and `central` with 12; deliver 5 bicycles only to `riverside`
- Show both quantities and read the shared `city` via `BikeStation.city`

Lab 02 | Class versus instance data

- The starter separates `bike_station.py` (class skeleton) from `main.py` (import and test calls)
- Place city directly inside the class body, not inside `__init__()`
- Store location and available inside `__init__()` using self
- Read `BikeStation.city` as the explicit class attribute; `station.city` also works if it is not shadowed
- Check the output: Lisbon, Riverside 13 bikes and Central 12 bikes

bike_station.py — class structure to complete

```
class BikeStation:
    city = 'Lisbon'

    def __init__(self, location, available):
        # Store independent station data
        ...

    def receive_bikes(self, count):
        # Update only this station
        ...
```

Lab 02 | Checks and common mistake

- Change only the attribute of the station receiving bicycles, not a class attribute
- Correct design: each station has its own `self.available` and `self.location`
- Incorrect design: `available = 0` in the class body when independent counts are intended
- An instance assignment such as `station.city = "Porto"` would create an instance attribute hiding the class attribute; it would not change `BikeStation.city`
- Run main.py with `bike_station.py` in the same folder; the city is shared but bicycle counts are independent

Independent 01 | Scientific sample records

- A research laboratory stores the code and mass (grams) of each physical sample
- Define a Sample class in `sample.py` with an initializer accepting `code` and `mass_g`, saved as instance attributes
- Define `.add_mass(amount)` to increase the mass of the selected sample
- Create sample “S-01” with mass 12.5 g and “S-02” with mass 8.0 g
- Add 2.5 g to S-01 only, then display the code and final mass of both samples
- Expected final masses: S-01 = 15.0 g; S-02 = 8.0 g

Independent 01 | Deliverable and verification

- Implement `sample.py` (the class) and `main.py` (import, object creation and tests)
- Show the final state by reading the attributes of each object
- Do not use a dictionary in place of the class: the aim is to define your own object type
- Use ordinary numeric values, assignments, dot notation and one instance method
- Run `main.py` with `sample.py` in the same folder; no external input files are needed

Independent 02 | Sports tournament scores

- A youth basketball tournament keeps separate scores for three teams
- Define `TeamScore` in `team_score.py` with a class attribute `sport = "Basketball"`
- Every instance must store its team name and accumulated points, initially given during construction
- Define `.add_points(points)` to increase the score of one team
- Create Falcons = 8, Tigers = 12 and Comets = 6; then award 5, 3 and 7 additional points respectively
- Store all three objects in a list and print each team name with its updated points using a loop

Independent 02 | Aggregation and checks

- Using a `for` loop over the list of objects, compute the sum of all updated scores
- Print the class attribute once, referring to `TeamScore.sport`
- Check the expected points: Falcons = 13, Tigers = 15, Comets = 13; grand total = 41
- Write `main.py` to import `TeamScore`; no pandas method or file I/O is needed; use concepts from Python basics and the current lesson
- The team name and points must be instance attributes, not shared class attributes

Independent 03 | Parcel delivery fee report

- A courier receives a CSV file describing individual deliveries. Every row has a parcel identifier, a customer, a base fee in EUR and additional delivery distance in km
- The company charges EUR 0.50 per additional kilometer, on top of each parcel's base fee
- Define a `ParcelDelivery` class in `parcel_delivery.py` whose `__init__()` stores `parcel_id`, customer and `base_fee` as instance attributes
- The object also keeps `total_fee`, initially equal to its base fee
- Define `.add_distance_charge(kilometres, rate)` to increase `total_fee` by kilometres × rate
- Read the supplied `parcels.csv` and build one instance for each row

Independent 03 | Input file and expected records

- **Input**: `parcels.csv`, with five rows and columns `ParcelID, Customer, BaseFee_EUR, ExtraKm`
- Use pandas `.read_csv()` to load the records; row positions are `0` to `len(df) - 1`
- In `main.py` import `ParcelDelivery`, then for each record create an object, add its distance charge, then place it **in a list**
- Use these records to compute an amount for every parcel; do not type the expected totals into the code

Content of `parcels.csv`

```
ParcelID,Customer,BaseFee_EUR,ExtraKm
P101,Alice,5.0,4
P102,Bob,6.0,2
P103,Alice,4.0,6
P104,Charlie,7.0,0
P105,Bob,5.0,8
```

Independent 03 | Report requirements

- Print one descriptive line per parcel: identifier, customer and total fee in EUR
- **Accumulate total fees per customer** in a **dictionary**, using a **loop** over the parcel objects
- Build a two-column `DataFrame` with `Customer` and `TotalFee_EUR`, then save `customer_totals.csv` using `index=False`
- Expected parcel fees: P101 = 7, P102 = 7, P103 = 7, P104 = 7, P105 = 9 EUR
- Expected customer totals: Alice = 14, Bob = 16, Charlie = 7 EUR (overall total = 37 EUR)

Independent 03 | Scope and final checks

- Keep the program small: `parcel_delivery.py` defines the class; `main.py` imports it, processes the CSV and writes the report
- Use class methods for behavior that belongs to an individual delivery; use a separate loop for the customer aggregation
- The final output CSV must contain one row for each of the three customers
- Do not change `parcels.csv`: write a new output file
- All information needed to run the program is included in the exercise folder