Lecture 18 | OOP – Part I

From data and functions to classes and objects

Lisbon School of Economics & Management
Universidade de Lisboa

Before objects: dictionaries and functions

- We already know how to represent an entity: a dictionary stores related values such as an account owner and balance
- A function performs an operation: it receives the dictionary as an argument and changes one of its entries
- **The limitation is organizational**: as a program grows, we must keep track of which functions operate on which data

```
account = {'owner': 'Anna', 'balance': 100}

def deposit(account, amount):
    account['balance'] += amount

deposit(account, 40)
print(account['balance'])
```

Output

```
140
```

One idea: keep data and behavior together

- An ***object*** represents one particular ***entity*** in the program, for example one bank account
- **Attributes** describe its ***current state***, such as owner and balance
- **Methods** define ***operations*** the object can perform, such as depositing an amount
- A ***class*** is the definition used to create objects of the same kind (its “type”); an object created from that class is an ***instance***

A class describes the shared structure

`BankAccount` **(class)**
attributes & methods:
`owner, balance, .deposit()`

↓ creates instances

`alice`
`Alice / 100`

`bob`
`Bob / 250`

A single class can create many objects; every object has its own attribute values

What is the organizational problem?

With data and functions kept separately

- An account is stored in a dictionary
- A deposit is a function receiving that dictionary
- As a program grows, we must remember which functions belong with which kinds of data

A new organization: an object

- Represent one account as a single entity in the program
- Keep its values together with the operations available for it
- The underlying calculations are the same; their organization changes

Class and object: two different ideas

- A **class** is a definition used to create objects of the same kind: think of it as the description of what a bank account should be
- An **object** (also called an ***instance***) is one specific account created from that class
- One class can be used to create many independent objects, just as one recipe can produce many separate dishes
- In Python, our own classes become new types that we can use in a program

Attributes: the data stored in an object

- An ***attribute*** is a **named value associated with an object**
- Example: for a bank account, useful attributes are `owner` (a string) and `balance` (a number)
- Attributes can hold familiar Python values: numbers, strings, Booleans, lists and even other objects; you can think of it as a variable that is linked with an object
- **Dot notation** reads an attribute belonging to a particular object: `object.attribute`

- Two separate objects; Conceptual example:
- “alice” account
  - owner → "Alice"
  - balance → 100
- “bob” account
  - owner → “Bob"
  - balance → 250

First syntax: defining a class

- Use the keyword `class` followed by a **class name** and a colon
- Names of classes conventionally use `CapitalizedWords`, such as `BankAccount`
- The indented block contains the class definition
- For now, `pass` means “no instructions yet”; we will replace it with useful definitions shortly

```
class BankAccount:
    pass

# This defines the class,
# but does not create any account
```

Creating objects from the class

- Writing `BankAccount()` asks Python to create a new object of this class
- Store the resulting object in a variable to use it later
- Calling `BankAccount()` a second time creates a different object
- At this stage, these objects have no account-specific attributes yet

```
class BankAccount:
    pass

alice = BankAccount()
bob   = BankAccount()

print(type(alice))
# <class '__main__.BankAccount'>
```

Adding instance attributes directly

- We can **attach a value** to a particular object with `object.attribute = value`
- Reading the attribute uses the same dot notation without an assignment
- **These attributes belong to that instance**, not to the variable names of all accounts
- This is possible, but having to set every attribute manually is inconvenient

```
alice = BankAccount()
alice.owner = 'Alice'
alice.balance = 100

bob = BankAccount()
bob.owner = 'Bob'
bob.balance = 250

print(alice.balance)  # 100
print(bob.balance)  # 250
```

Why do we need an initializer?

- Every bank account we create should have an owner and an initial balance
- Manually assigning those two attributes after every `BankAccount()` call is repetitive and easy to forget
- We would prefer a construction such as `BankAccount("Alice", 100)`, supplying values when the account is created
- To make this possible, we define a special method named `__init__()` that initializes an object

Introducing `__init__()`: its purpose

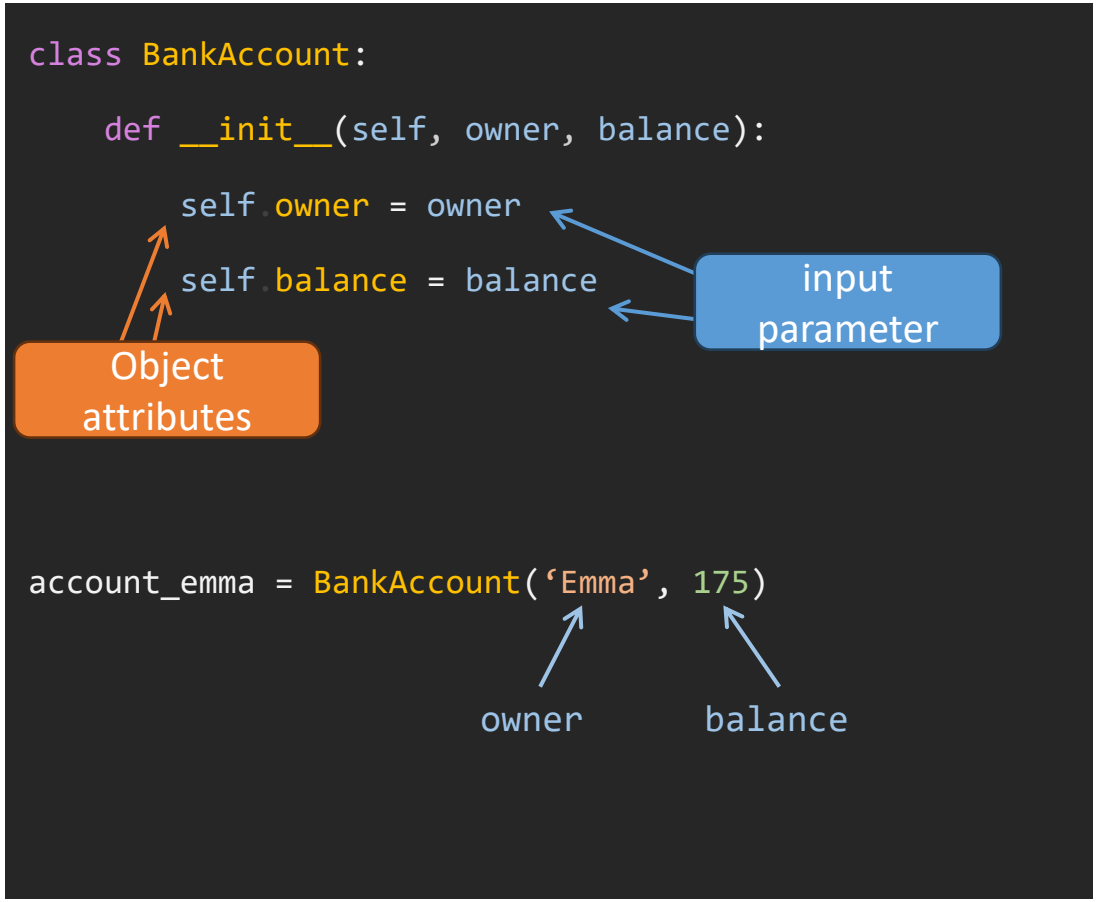
- `__init__()` is a special method: Python calls it automatically when a new instance has been created
- Its job is **to set the initial state**, for example the `owner` and starting `balance`
- Inside the method we assign the supplied information to the new object, using **parameters**
- The word `self` identifies the particular instance being initialized; we will examine it on the next slide

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

# main code
emma = BankAccount('Emma', 175)
```

Reading the initializer line by line

- `def __init__(self, owner, balance):` defines the initializer with two input values; the first position is reserved for `self`
- **Why is `self` the first parameter?** When an instance method is called (e.g. `BankAccount('emma', 175)`), ***Python automatically calls*** `__init__()` ***and passes the instance as its first argument***. By convention, this parameter is named `self`.
- In other words, the first parameter is a reference to each new object created in the memory of that class.
- This allows the method to access or modify the attributes belonging to that particular object. The same mechanism applies to `__init__()` and all ordinary instance methods.
- `owner` and `balance` are **parameters**, exactly like parameters in ordinary functions
- `self.owner = owner` stores the supplied owner string in the object's owner attribute
- `self.balance = balance` stores the supplied number in the object's balance attribute



Understanding self without shortcuts

- `self` is the parameter referring to the **instance** currently being initialized or used
- For Alice's account, self refers to Alice's object; for Bob's, it refers to Bob's object
- We write self in a method definition, but do not supply it ourselves when calling `BankAccount(...)`
- The name `self` is a Python convention; what matters is its role as the first instance-method parameter

```
alice_acc = BankAccount('Alice', 100)
bob_acc = BankAccount('Bob', 75)
```

- During Alice's initialization:
  - `owner` receives `'Alice'`
  - `balance` receives `100`
  - `self` refers to Alice's object
  - e.g. `self.balance = balance` stores 100 in Alice's object
- During Bob's initialization:
  - `owner` receives `'Bob'`
  - `balance` receives `75`
  - `self` refers to Bob's object
  - e.g. `self.balance = balance` stores 75 in Bob's object

Creating two initialized objects

- The class is now defined; each call creates a separate account and initializes its attributes
- Python automatically calls `__init__()` for each new object
- The attributes are accessed through the variable referring to that object
- One account's starting balance does not replace the other account's balance; each object you create has his own "life"

```
accountA = BankAccount('Alice', 100)
accountB = BankAccount('Bob', 250)

print(accountA.owner, accountA.balance)
print(accountB.owner, accountB.balance)
```

Output

```
Alice 100
Bob 250
```

Parameters and attributes are not the same

Parameter: temporary input to a method

- In `__init__(self, owner, balance)`, `owner` and `balance` receive inputs when the method runs
- Their names are **local to that call**
- A parameter by itself **is not automatically stored in the object**

Attribute: value stored on an object

- `self.owner` and `self.balance` refer to values belonging to an instance
- Assigning `self.balance = balance` preserves the value after initialization
- We can later read it using `accountA.balance` or `accountB.balance`

Methods: operations associated with objects

- A ***method*** is a function defined in a class and normally called on an instance using dot notation
- **Attributes** describe an **object's state**; **methods** describe **operations that can use or modify that state**
- For a bank account, `.deposit(amount)` can add money to its current balance
- This is familiar: we already call methods such as `list.append(value)` on Python objects

Defining a simple instance method

- The method is defined with `def` inside the class, at the same indentation level as `__init__()`
- Its first parameter is `self`: the instance on which the method is called; this is used to access attributes and current state
- In the example, `amount` is an ordinary input parameter supplied by the caller
- `self.balance += amount` updates the balance stored on the selected object

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
```

Calling a method: which object changes?

- `accountA.deposit(40)` calls `.deposit()` on Alice's object
- Inside that call, `self` refers to `accountA` and amount receives 40
- The method changes Alice's balance from 100 to 140
- Bob's account is a different instance: it remains unchanged

```
accountA = BankAccount('Alice', 100)
accountB = BankAccount('Bob', 250)

accountA.deposit(40)

print(accountA.balance)  # 140
print(accountB.balance)  # 250
```

Warning: a local variable is not an attribute



- A name assigned inside `__init__()` without `self` is only **a local variable** (or parameter) of that method
- It does NOT become part of the object: in the example, after initialization, `accountA.balance` does not exist
- To store information on the instance, write `self.balance = balance`
- Reading a missing attribute raises `AttributeError`

```python
# Incorrect: balance is never stored
class BankAccount:
    def __init__(self, balance):
        balance = balance + 10

accountA = BankAccount(100)
print(accountA.balance)
# AttributeError: no attribute balance


# Correct: store the value on alice
class BankAccount:
    def __init__(self, balance):
        self.balance = balance + 10

accountA = BankAccount(100)
print(accountA.balance)  # 110
```

Warning: class attributes may be shared



- An attribute assigned **in the class body belongs to the class**; instances can access that same value
- Useful: `currency = "EUR"` can describe a common default for all accounts
- Danger: a mutable class attribute such as `transactions = []` is **ONE shared list**, not one list per account
- For independent histories, initialize `self.transactions = []` inside `__init__()`
- Class attributes can be accessed using `ClassName.attribute` or `instance.attribute`. The first form is preferable when referring explicitly to shared class-level information. Assigning to `instance.attribute` normally creates or updates an **instance attribute** rather than modifying the class attribute.
- Important distinction: assigning `accountA.currency = "USD"` creates an instance attribute; it does not change `BankAccount.currency`

```python
# Unintended: the list is shared
class BankAccount:
    currency = "EUR"  # useful shared default
    transactions = []  # dangerous!

accountA = BankAccount()
accountB = BankAccount()

accountA.transactions.append(50)
print(accountB.transactions)  # [50] (!)

# ------------------------------------
# Correct: each instance has its own list
class BankAccount:
    currency = "EUR"
    def __init__(self):
        self.transactions = []

accountA = BankAccount()
accountB = BankAccount()

accountA.transactions.append(50)
print(accountB.transactions)  # []
```

A complete program: definition and use

- First define the class once, including the initializer and the deposit method
- Create any number of independent instances by calling the class with the required inputs
- Use dot notation to read attributes and to call methods
- The program combines syntax we know already: functions, assignments, arithmetic and printing

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

# Main code
a = BankAccount('Alice', 100)
a.deposit(25)
print(a.balance)  # 125
```

A class can be defined in a separate module

Modularity: separate a class from the main program

- A Python module is a .py file containing definitions that other programs can import
- We alredy used modules, both from Standard Library and custom .py files; we know that functions are usually defined in a module, and imported in the main script(s) that require those operations, without rewriting the same code
- --> We do the same when defining new classes
- Place the `WaterTank` class in a file `water_tank.py` so its definition can be reused
- Remember: the class definition contains the attributes and operations of a tank, but it does not create any tank objects by itself; it only defines the type structure
- In the simplest case, keep `water_tank.py` and `main.py` in the same folder

`water_tank.py`

```
class WaterTank:

    def __init__(self, owner, liters, capacity):
        self.owner = owner
        self.liters = liters
        self.capacity = capacity

    def add_water(self, amount):
        self.liters += amount

    def take_water(self, amount):
        self.liters -= amount
```

Importing a class and creating its instances

Using a class defined in another file

- `from water_tank import WaterTank` imports the class by name from `water_tank.py` (omit .py in the import)
- `WaterTank(...)` creates an instance, just as when the class is defined in the same file
- The method call `wt_alice.add_water(25)` acts on that particular instance
- Alternative: import `water_tank`, then create objects with `water_tank.WaterTank(...)`
- Run `main.py` with `water_tank.py` available in the same folder

`main.py`

```
from water_tank import WaterTank

wt_alice = WaterTank('Alice', 40, 80)
wt_alice.add_water(25)

print(wt_alice.liters)  # 65

# ------------------------------------
# Alternative import style
import water_tank

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

print(wt_bob.liters)  # 70
```

The key ideas to carry into Lecture 19

- A class defines the structure and behavior shared by objects of the same kind
- An instance is a particular object; its attributes store its own state
- `__init__()` initializes the new object, and self refers to the object currently being used
- A method is an operation called on an object, such as `account.deposit(25)`
- Next, we will examine what methods return, how they change the object, and how to reject invalid operations