Object Oriented Programming

Procedure Orient

- In the previous classes we worked on ***small programs***

  - Coded ***list of instructions in the sequence*** for task execution
  - Efficient task execution requires the division of any task into subtasks
  - Subsequently used **blocks of sequential instructions** defined under functions to carry out subtasks
  - This type of programming is known as ***Procedure Oriented Programming (POP)***
  - In POP, the program is divided into ***variables, data-structures and routines/functions*** to execute different tasks

Object Orient Programming

- Object Oriented Programming (OOP) is more suited for ***large size program***
  - ➢ In OOP, the programming task is divided into:
    - ❖ Objects: Known as data/attributes
    - ❖ Behaviour/Functions: Known as Methods
  - ➢ Broadly there are two components of OOP:
    - ❖ Class
    - ❖ Object

Class

- A class in Python is a user-defined template for creating objects.

- It bundles data and functions together, making it easier to manage and use them.

- Once we create a new class, we can define/create multiple instances of this object type.

- Classes are created using **class keyword**. Attributes are variables defined inside the class and represent the properties of the class. Attributes can be accessed using the dot **. operator** (e.g., MyClass.my_attribute).

Class

- Blueprint to define ***logical grouping*** of data and functions

- Provides way to create data-structures that ***model real-world entities***
- Example:

In [1]:

```
class People():
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print("Greetings, " + self.name)

person1 = People(name = "Anil", age = 35)
person1.greet()
print(person1.name)
print(person1.age)
```

```
Greetings, Anil
Anil
35
```

Using __init()__ function

- In Python, class has __init__() function. It automatically initializes object attributes when an object is created.

```
In [1]: class People():
            def __init__(self, name, age):
                self.name = name
                self.age = age
```

- __init__ method: Initializes the “name” and “age” attributes when a new object is created.

Class

- Another object’s instantiation on the “People” class

```
In [3]: person2 = People(name = "Prerna", age = 37)
        person2.greet()
        print(person2.name)
        print(person2.age)

        Greetings, Prerna
        Prerna
        37
```

- In the above class “People”:
  - Data/Instance Attributes: *name* and *age*
  - Method(s): *greet()*
  - Instantiated objects: person1 and person2
- Hence, class defines the ***whole (data as well as behaviour) structure*** of the entity, while object is just an ***instance*** of the class with actual values

Class Syntax

```
class ClassName(superclass):

    def __init__(self, arguments):
        # define or assign object attributes

    def other_methods(self, arguments):
        # body of the method
```

- Standard convention for class name: “***CapWords***”
- Parameter “***superclass***” is optional
  - ➢ Superclass name “superclass” is required when the defined class “ClassName” wants to ***inherit*** from already defined “superclass” attributes and methods
  - ➢ **__init__** method:
    - ❖ **Executes** as soon as any object of the corresponding class is instantiated
    - ❖ Used to assign ***initial values*** to object before the object is used
    - ❖ ***Two underscores*** “__” before and at the end of init indicates that it is a special method reserved for special use in the language

Class Syntax

- "***other_methods***" are used to define the instance methods that will operate on the attributes

- "***self***" parameter is an extra parameter which ***must be the first parameter*** of every instance method
  - It ***refers*** to the object itself
  - Instance methods can freely access attributes and other methods of the same object by using this "***self***" parameter as self.attribute and self.method()

- ***<u>Example:</u>*** Define a class named Employee, with the attributes ***empid (employee id), name, gender, type*** in the init method, and a method called ***say_name*** to print out the employee's name.

Class: Employee

In [5]:
```
class Employee():
    def __init__(self, emp_id, name, gender):
        self.emp_id = emp_id
        self.name = name
        self.gender = gender
        self.type = "learning"

    def say_name(self):
        print("My name is " + self.name)

    def report(self, score):
        self.say_name()
        print("My id is: ", self.emp_id)
        print("My appraisal score is: ", str(score))

emp_1 = Employee(1201, "Abhishek", "Male")
emp_1.say_name()
emp_1.report(99)
```

```
My name is Abhishek
My name is Abhishek
My id is:  1201
My appraisal score is:  99
```

Class Attributes

- Example of “class attributes”: Modify the Employee class to add a class attribute n, which will record how many objects we are creating. Also, add a method $num\_instances$ to print out the number.

- Class attributes are defined ***“outside”*** of all the method’s definitions without using “self”

- Hence it will be **shared** by all the instances/objects of that class

Class Attributes

```
In [7]: class Employee():
            n = 0;
            def __init__(self, emp_id, name, gender):
                self.emp_id = emp_id
                self.name = name
                self.gender = gender
                self.type = "learning"
                Employee.n  = Employee.n + 1

            def say_name(self):
                print("My name is " + self.name)

            def report(self, score):
                self.say_name()
                print("My id is: ", self.emp_id)
                print("My appraisal score is: ", str(score))

            def num_instances(self):
                print(f"We have {Employee.n}-employees in total")

        emp_1 = Employee("001", "Susan", "F")
        emp_1.num_instances()
        emp_2 = Employee("002", "Mike", "M")
        emp_1.num_instances()
        emp_2.num_instances()
```

```
We have 1-employees in total
We have 2-employees in total
We have 2-employees in total
```

Inheritance

- Enable to build ***relationships*** among classes
  - Make the code more **modular** and easy to **reuse**

- Inheritance allows to define a class that **can inherit** all the methods and attributes from another class

  - This class which inherits is known as "**child class**" while the one from which the attributes and methods are inherited is known as "**parent class/ super class**"
  - As previously shown in the class syntax the structure for basic inheritance is :
    class ClassName(SuperClass):
    - class ***ClassName*** can access all the attributes and methods of ***SuperClass***
  - Usually, parent class is of ***general type*** while child class is of ***specific type***

Inheritance Example

- Define a class named ***Sensor*** with attributes ***name, location, and record_date*** that pass from the creation of an object and an attribute ***data*** as an empty dictionary to store data.

- Create one method ***add_data*** with ***t*** and ***data*** as input parameters to take in timestamp and data arrays. Within this method, assign t and data to the data attribute with “time” and “data” as the keys.

- In addition, create one ***clear_data*** method to delete the data.

Sensor Class

```
In [1]: class Sensor():
            def __init__(self, name, location, record_date):
                self.name = name
                self.location = location
                self.record_date = record_date
                self.data = {}

            def add_data(self, t, data):
                self.data["time"] = t
                self.data["data"] = data
                print(f"We have {len(data)} points saved")

            def clear_data(self):
                self.data = {}
                print("Data cleared!")
```

- Object Creation/Instantiation:

```
In [2]: import numpy as np
        sensor1 = Sensor("sensor1", "Berkeley", "2019-01-01")
        data = np.random.randint(-10, 10, 10)
        sensor1.add_data(np.arange(10), data)
        sensor1.data
```

Child Class: Accelerometer

```
We have 10 points saved

Out[2]: {'time': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
         'data': array([ 4,  1,  1,  4,  0,  6, -9, -9, -8,  4])}
```

- Now, we have a different type of sensor known as "***AcceleroMeter***" which in addition to the methods and attributes of the "***Sensor***" requires some specific attributes and methods
    - Here we are not required to create a different class from ***scratch***
    - We can inherit from the "***Sensor***" class and extend the child class as:

Child Class: Accelerometer

```
In [4]: class Accelerometer(Sensor):
            def show_type(self):
                print("I am an accelerometer!")

        acc = Accelerometer("acc1", "Oakland", "2019-02-01")
        acc.show_type()
        data = np.random.randint(-10, 10, 10)
        acc.add_data(np.arange(10), data)
        acc.data
```

- Output

```
I am an accelerometer!
We have 10 points saved

Out[4]: {'time': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
         'data': array([  9,  -8,   7,   8,   3, -10,   1,  -1,   6,  -7])}
```

Method Overriding

- One can also ***modify the implementation*** of the method provided by the parent-class in the child-class as:

```
In [5]: class UCBAcc(Accelerometer):
            def show_type(self):
                print(f"I am {self.name}, created at Berkeley!")
        acc2 = UCBAcc("UCBAcc", "Berkeley", "2019-03-01")
        acc2.show_type()

        I am UCBAcc, created at Berkeley!
```

- This is known as "***Method Overriding***"

Super Method

- Used to refer to the parent class
- Example:

```
In [6]: class NewSensor(Sensor):
            def __init__(self,name,location,record_date,brand):
                self.name = name
                self.location = location
                self.record_date = record_date
                self.brand = brand
                self.data = {}
        new_sensor = NewSensor("OK", "SF", "2019-03-01", "XYZ")
        new_sensor.brand

Out[6]: 'XYZ'
```

Super Method

- Super():

```
In [18]: class NewSensor(Sensor):
             def __init__(self,name,location,record_date,brand):
                 super().__init__(name,location, record_date)
                 self.brand = brand

         new_sensor = NewSensor("OK", "SF", "2019-03-01", "XYZ")
         new_sensor.brand

Out[18]: 'XYZ'
```

Data Encapsulation

- (Complex) Information is not required to be revealed
- Hence encapsulate the information so that it can not be modified accidently

In [2]:
```
class Sensor():
    def __init__(self,name,location):
        self.name=name
        self.location = location
        self.__version = "1.0"

    #a getter function
    def get_version(self):
        print(f"Thesensorversionis{self.__version}")

    #asetterfunction
    def set_version(self,version):
        self.__version= version
```

Data Encapsulation

In [3]:
```
sensor1=Sensor("Acc","Berkeley")
print(sensor1.name)
print(sensor1.location)
print(sensor1.__version)
```

```
Acc
Berkeley

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[3], line 4
      2 print(sensor1.name)
      3 print(sensor1.location)
----> 4 print(sensor1.__version)

AttributeError: 'Sensor' object has no attribute '__version'
```

Data Encapsulation

- Access the private variable (name starting with '__') using getter function and modify it using setter function

```
In [4]: sensor1.get_version()
        sensor1.set_version("2.0")
        sensor1.get_version()
```

```
Thesensorversionis1.0
Thesensorversionis2.0
```

Method Encapsulation

- __call__()
  - Make the object callable
  - Simplify and improve the code, when the purpose of the object is to perform some action
    - ➢ Hence ***encapsulates*** the functionality within the object which facilitates maintaining the sate of the object
    - ➢ Useful in maintaining cleaner and more organized code

In [1]:
```
class Adder:
    def __call__(self, x, y):
        return x + y

# Creating an instance of Adder
add = Adder()

# Calling the instance as if it were a function
result = add(3, 4)  # This calls add.__call__(3, 4)
print(result)  # Output: 7
```

```
7
```

Multiple Inheritance

- Python allows more than one parent (super) classes for a child (sub) class unlike other programming languages like Java

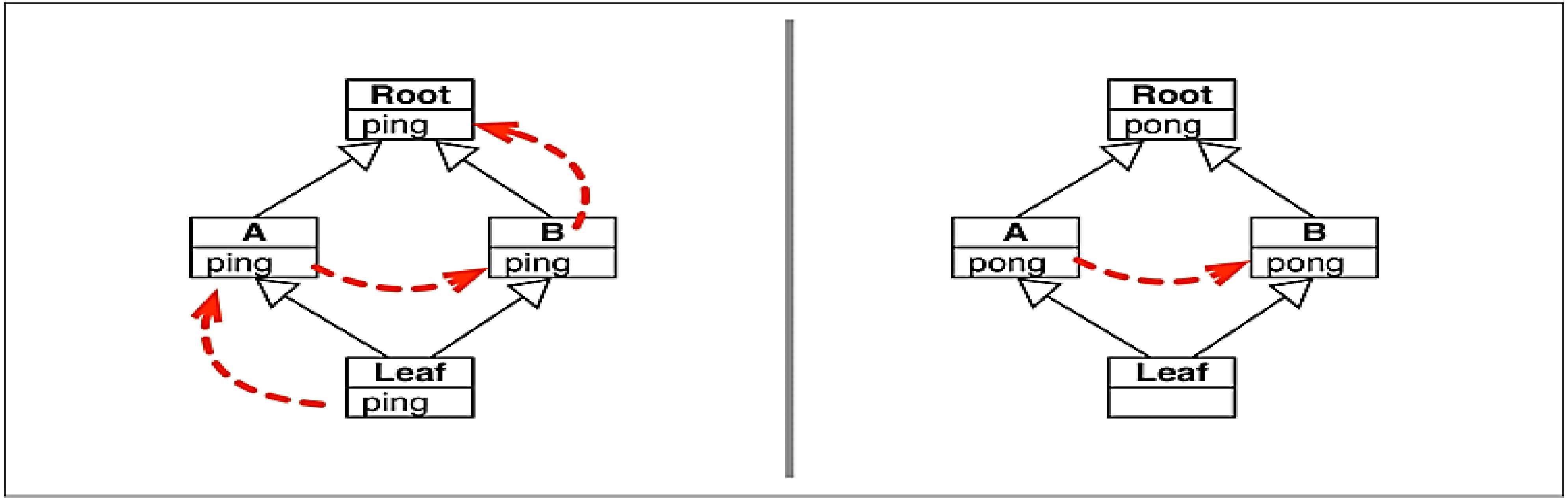

Figure 14-1. Left: Activation sequence for the `leaf1.ping()` call. Right: Activation sequence for the `leaf1.pong()` call.

```
In [1]: class Root:
            def ping(self):
                print(f'{self}.ping() in Root')

            def pong(self):
                print(f'{self}.pong() in Root')

            def __repr__(self):
                cls_name = type(self).__name__
                return f'<instance of {cls_name}>'
```

In [3]:

```python
class A(Root):
    def ping(self):
        print(f'{self}.ping() in A')
        super().ping()

    def pong(self):
        print(f'{self}.pong() in A')
        super().pong()

class B(Root):
    def ping(self):
        print(f'{self}.ping() in B')
        super().ping()

    def pong(self):
        print(f'{self}.pong() in B')
```

In [4]:
```python
class Leaf(A, B):
    def ping(self):
        print(f'{self}.ping() in Leaf')
        super().ping()
```

In [5]:
```python
leaf1 = Leaf()
leaf1.ping()
```

```
<instance of Leaf>.ping() in Leaf
<instance of Leaf>.ping() in A
<instance of Leaf>.ping() in B
<instance of Leaf>.ping() in Root
```

- Every class has an attribute called __mro__ holding a tuple of references to the super classes in method resolution order

```
In [7]: Leaf.__mro__

Out[7]: (__main__.Leaf, __main__.A, __main__.B, __main__.Root, object)
```

- The __mro__ determines the activation order only

Summary

- Benefits of OOP:
  - Combines data and operations
  - Provides a ***clear modular structure*** for programs that enhances ***code reusability***
  - Provides simple way to ***solve complex problems***
    - Helps define more ***abstract data-types***
    - Thereby***, hides implementation details*** and shows clearly defined interface