Lab 2 – Python Basics2 (Object Oriented Programming Concepts)

Class

The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have an employee class, then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.

Syntax

```
class ClassName:
        <statement-1>
        .
        .
        <statement-N>
```

Object

The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute __doc__, which returns the docstring defined in the function source code.

When we define a class, it needs to create an object to allocate the memory. Consider the following example.

Example:

```
class car:
   def __init__(self,modelname, year):
      self.modelname = modelname
      self.year = year
   def display(self):
      print(self.modelname,self.year)

c1 = car("Toyota", 2016)
c1.display()
```

Instantiate an Object in Python

When we define a class only the description or a blueprint of the object is created. There is no memory allocation until we create its object. The objector instance contains real data or information.

Instantiation is nothing but creating a new object/instance of a class. Let’s create the object of the above class we defined-

```
obj1 = Car()
```

And it’s done! Note that you can change the object name according to your choice.

Try printing this object-

print(obj1)

Since our class was empty, it returns the address where the object is stored i.e 0x7fc5e677b6d8

You also need to understand the class constructor before moving forward.

Class constructor

Until now we have an empty class Car, time to fill up our class with the properties of the car. The job of the class constructor is to assign the values to the data members of the class when an object of the class is created.

There can be various properties of a car such as its name, color, model, brand name, engine power, weight, price, etc. We'll choose only a few for understanding purposes.

```
class Car:

    def __init__(self, name, color):

        self.name = name

        self.color = color
```

So, the properties of the car or any other object must be inside a method that we call __init__( ). This __init__() method is also known as the constructor method. We call a constructor method whenever an object of the class is constructed.

Now let's talk about the parameter of the __init__() method. So, the first parameter of this method has to be self. Then only will the rest of the parameters come.

The two statements inside the constructor method are –

self.name = name
self.color = color:

This will create new attributes namely name and color and then assign the value of the respective parameters to them. The "self" keyword represents the instance of the class. By using the "self" keyword we can access the attributes and methods of the class. It is useful in method definitions and in variable initialization. The "self" is explicitly used every time we define a method.

Note: You can create attributes outside of this __init__() method also. But those attributes will be universal to the whole class and you will have to assign the value to them.

Suppose all the cars in your showroom are Sedan and instead of specifying it again and again you can fix the value of car_type as Sedan by creating an attribute outside the __init__().

```
class Car:
    car_type = "Sedan"                #class attribute

    def __init__(self, name, color):

        self.name = name              #instance attribute

        self.color = color            #instance attribute
```

Here, Instance attributes refer to the attributes inside the constructor method i.e self.name and self.color. And, Class attributes refer to the attributes outside the constructor method i.e car_type.

Class methods

Methods are the functions that we use to describe the behavior of the objects. They are also defined inside a class.
The methods defined inside a class other than the constructor method are known as the instance methods. Furthermore, we have two instance methods here- description() and max_speed(). Let's talk about them individually-

description()- This method is returning a string with the description of the car such as the name and its mileage. This method has no additional parameter. This method is using the instance attributes.
max_speed()- This method has one additional parameter and returning a string displaying the car name and its speed.
Notice that the additional parameter speed is not using the "self" keyword. Since speed is not an instance variable, we don't use the self keyword as its prefix. Let's create an object for the class described above.

```
obj2 = Car("Honda City",24.1)

print(obj2.description())

print(obj2.max_speed(150))
```

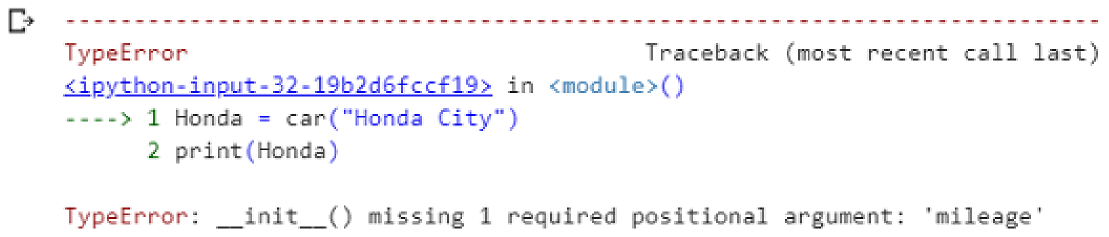
Creating more than one object of a class

```
class Car:

    def __init__(self, name, mileage):

        self.name = name

        self.mileage = mileage



    def max_speed(self, speed):

        return f"The {self.name} runs at the maximum speed of {speed}km/hr"
```

```
Honda = Car("Honda City",21.4)
print(Honda.max_speed(150))

Skoda = Car("Skoda Octavia",13)
print(Skoda.max_speed(210))
```

Passing the wrong number of arguments.

```
class Car:

    def __init__(self, name, mileage):
        self.name = name
        self.mileage = mileage

Honda = Car("Honda City")
print(Honda)
```



```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-32-19b2d6fccf19> in <module>()
----> 1 Honda = car("Honda City")
      2 print(Honda)

TypeError: __init__() missing 1 required positional argument: 'mileage'
```

Since we did not provide the second argument, we got this error.

Order of the arguments

```
class Car:

    def __init__(self, name, mileage):
        self.name = name
        self.mileage = mileage



    def description(self):

        return f"The {self.name} car gives the mileage of {self.mileage}km/l"



Honda = Car(24.1,"Honda City")

print(Honda.description())
```

Inheritance in Python Class

Inheritance is the procedure in which one class inherits the attributes and methods of another class. The class whose properties and methods are inherited is known as Parent class. And the class that inherits the properties from the parent class is the Child class.

The interesting thing is, along with the inherited properties and methods, a child class can have its own properties and methods.

How to inherit a parent class? Use the following syntax:

```
class parent_class:

body of parent class



class child_class( parent_class):

body of child class
```

Let’s see the implementation-

```
class Car:          #parent class



    def __init__(self, name, mileage):

        self.name = name

        self.mileage = mileage



    def description(self):
        return f"The {self.name} car gives the mileage of {self.mileage}km/l"



class BMW(Car):     #child class

    pass



class Audi(Car):     #child class

    def audi_desc(self):

        return "This is the description method of class Audi."
```

```
obj1 = BMW("BMW 7-series",39.53)

print(obj1.description())



obj2 = Audi("Audi A8 L",14)

print(obj2.description())

print(obj2.audi_desc())
```

We have created two child classes namely "BMW" and "Audi" who have inherited the methods and properties of the parent class "Car". We have provided no additional features and methods in the class BMW. Whereas one additional method inside the class Audi.

Notice how the instance method description() of the parent class is accessible by the objects of child classes with the help of obj1.description() and obj2.description(). And also the separate method of class Audi is also accessible using obj2.audi_desc().

Encapsulation

Encapsulation, as I mentioned in the initial part of the article, is a way to ensure security. Basically, it hides the data from the access of outsiders. Such as if an organization wants to protect an object/information from unwanted access by clients or any unauthorized person then encapsulation is the way to ensure this.

You can declare the methods or the attributes protected by using a single underscore (_) before their names. Such as- self._name or def _method( ); Both of these lines tell that the attribute and method are protected and should not be used outside the access of the class and sub-classes but can be accessed by class methods and objects.

Though Python uses ‘ _ ‘ just as a coding convention, it tells that you should use these attributes/methods within the scope of the class. But you can still access the variables and methods which are defined as protected, as usual.

Now for actually preventing the access of attributes/methods from outside the scope of a class, you can use “private members“. In order to declare the attributes/method as private members, use double underscore (**__**) in the prefix. Such as – self.**__**name or def **__**method(); Both of these lines tell that the attribute and method are private and access is not possible from outside the class.

```
class car:



    def __init__(self, name, mileage):

        self._name = name                #protected variable

        self.mileage = mileage



    def description(self):

        return f"The {self._name} car gives the mileage of {self.mileage}km/l"
```

```
obj = car("BMW 7-series",39.53)



#accessing protected variable via class method

print(obj.description())



#accessing protected variable directly from outside

print(obj._name)

print(obj.mileage)
```

Notice how we accessed the protected variable without any error. It is clear that access to the variable is still public. Let us see how encapsulation works-

```
class Car:
    def __init__(self, name, mileage):

        self.__name = name              #private variable

        self.mileage = mileage



    def description(self):

        return f"The {self.__name} car gives the mileage of {self.mileage}km/l"
```

```
obj = Car("BMW 7-series",39.53)



#accessing private variable via class method

print(obj.description())



#accessing private variable directly from outside

print(obj.mileage)

print(obj.__name)
```

Polymorphism

This is a Greek word. If we break the term Polymorphism, we get “poly”-many and “morph”-forms. So Polymorphism means having many forms. In OOP it refers to the functions having the same names but carrying different functionalities.

```
class Audi:

  def description(self):

    print("This the description function of class AUDI.")



class BMW:
  def description(self):

    print("This the description function of class BMW.")
```

```
audi = Audi()

bmw = BMW()

for car in (audi,bmw):

 car.description()
```

When the function is called using the object audi then the function of class Audi is called and when it is called using the object bmw then the function of class BMW is called.

Data abstraction

We use Abstraction for hiding the internal details or implementations of a function and showing its functionalities only. This is similar to the way you know how to drive a car without knowing the background mechanism. Or you know how to turn on or off a light using a switch but you don't know what is happening behind the socket.

Any class with at least one abstract function is an abstract class. In order to create an abstraction class first, you need to import ABC class from abc module. This lets you create abstract methods inside it. ABC stands for Abstract Base Class.

```
from abc import ABC



class abs_class(ABC):

    Body of the class
```

Important thing is– you cannot create an object for the abstract class with the abstract method. For example-

```
from abc import ABC, abstractmethod



class Car(ABC):

    def __init__(self,name):
        self.name = name


 @abstractmethod

    def price(self,x):

        pass

obj = Car("Honda City")
```

```
from abc import ABC, abstractmethod



class Car(ABC):

    def __init__(self,name):

        self.name = name



    def description(self):

        print("This the description function of class car.")



    @abstractmethod

    def price(self,x):

        pass

class new(Car):

    def price(self,x):

        print(f"The {self.name}'s price is {x} lakhs.")

obj = new("Honda City")



obj.description()
obj.price(25)
```

Car is the abstract class that inherits from the ABC class from the abc module. Notice how I have an abstract method (price()) and a concrete method (description()) in the abstract class. This is because the abstract class can include both of these kinds of functions but a normal class cannot. The other class inheriting from this abstract class is new(). This method is giving definition to the abstract method (price()) which is how we use abstract functions.

After the user creates objects from new() class and invokes the price() method, the definitions for the price method inside the new() class comes into play. These definitions are hidden from the user. The Abstract method is just providing a declaration. The child classes need to provide the definition.

But when the description() method is called for the object of new() class i.e obj, the Car's description() method is invoked since it is not an abstract method.

Collections In Python :

What Are Collections In Python?

Collections in python are basically container data types, namely lists, sets, tuples, dictionary. They have different characteristics based on the declaration and the usage.

A list is declared in square brackets, it is mutable, stores duplicate values and elements can be accessed using indexes.

A tuple is ordered and immutable in nature, although duplicate entries can be there inside a tuple.

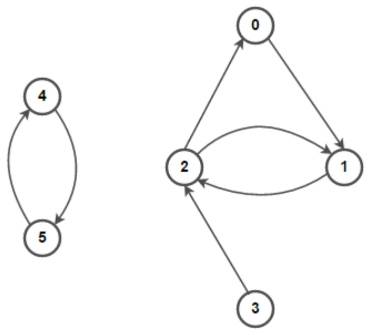
A set is unordered and declared in square brackets. It is not indexed and does not have duplicate entries as well.

A dictionary has key value pairs and is mutable in nature. We use square brackets to declare a dictionary.

These are the python's general purpose built-in container data types. But as we all know, python always has a little something extra to offer. It comes with a python module named collections which has specialized data structures.

Lab Questions:

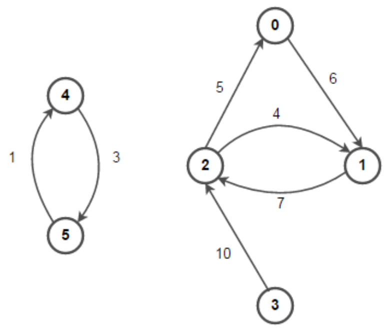
Q1. Implement the following directed unweighted graph using class, methods, and data structures of Python.



Expected output:

```
(0 —> 1)
(1 —> 2)
(2 —> 0) (2 —> 1)
(3 —> 2)
(4 —> 5)
(5 —> 4)
```

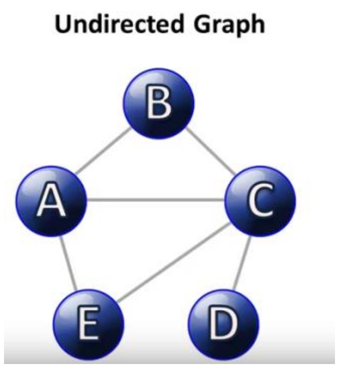
Q2. Implement the following directed weighted graph using class, methods, and data structures of Python.



Expected Output:

```
(0 —> 1, 6)
(1 —> 2, 7)
(2 —> 0, 5) (2 —> 1, 4)
(3 —> 2, 10)
(4 —> 5, 1)
(5 —> 4, 3)
```

Q3. Implement the following undirected weighted graph using class, methods, and data structures of Python. Print the adjacency list and adjacency matrix.



Expected Output:
Adjacency List:

```
["A:['B', 'C',
'E']", "C:['A',
'B', 'D', 'E']",
"B:['A', 'C',
'D']", "E:['A',
'C']", "D:['B',
'C']"]
```

Adjacency Matrix

```
[[ 0.  1.  1.  0.  1.]
 [ 1.  0.  1.  1.  0.]
 [ 1.  1.  0.  1.  1.]
 [ 0.  1.  1.  0.  0.]
 [ 1.  0.  1.  0.  0.]]
```

Additional Questions:
Consider a situation where there is a single teller in a bank who can assist customers with their transactions. When a customer arrives at the bank, they join the end of a queue to wait for the teller. When the teller is available, they assist the first customer in the queue and remove them from the queue.

Description:

Customer class is defined to store information about each customer, such as their name and transaction. The Bank class is defined with a queue attribute to store instances of the Customer class, and methods to add customers to the queue (add_customer), serve the next customer in the queue (serve_customer), and check if the queue is empty (is_queue_empty). The code simulates customers arriving at the bank and being served by the teller. The teller serves customers in the order they arrive and removes them from the queue using the pop(0) method.