PRACTICAL V

DATE: 02/03/24

Class and OOPs in python

AIM: To learn and execute Class and OOPs in python.

INTRODUCTION:

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

OOPs Concepts in Python

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

Python Class

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

To understand the need for creating a class let's consider an example, let's say you wanted to track the number of dogs that may have different attributes like breed, and age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

Class Definition Syntax:

```
class ClassName:
   # Statement-1
   .
   .
   .
   # Statement-N
```

Python Objects

The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number 12 is an object, the string "Hello, world" is an object, a list is an object that can hold other objects, and so on. You've been using objects all along and may not even realize it.

An object consists of:

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

To understand the state, behavior, and identity let us take the example of the class dog (explained above).

- The identity can be considered as the name of the dog.
- State or Attributes can be considered as the breed, age, or color of the dog.
- The behavior can be considered as to whether the dog is eating or sleeping.

Creating an Object

This will create an object named obj of the class Dog defined above. Before diving deep into objects and classes let us understand some basic keywords that will we used while working with objects and classes.

```
obj = Dog()
```

The Python self

1. Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it
2. If we have a method that takes no arguments, then we still have to have one argument.
3. This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

Note: For more information, refer to self in the Python class

The Python __init__ Method

The __init__ method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object. Now let us define a class and create some objects using the self and __init__ method.

Python Inheritance

Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class. The benefits of inheritance are:

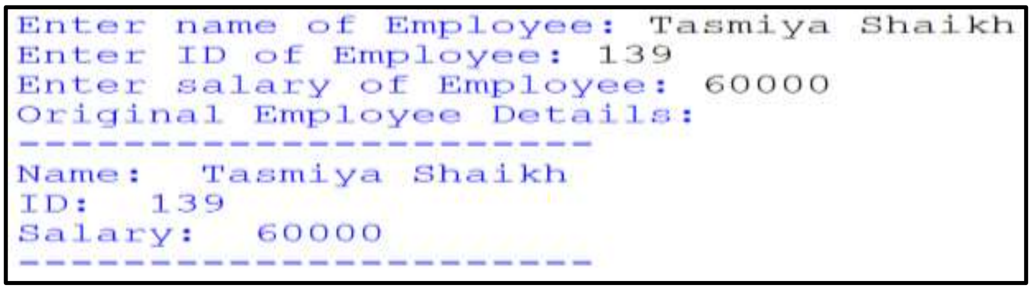
- It represents real-world relationships well.
- It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Types of Inheritance

- **Single Inheritance**: Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.
- **Multilevel Inheritance:** Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.
- **Hierarchical Inheritance:** Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.
- **Multiple Inheritance:** Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

Q1. Write a Python class named Employee with three attributes Emp_id, Emp_name and Emp_salary. display the entire attribute and their values of the class.

```
class Employee:
    def __init__(self, emp_name, emp_id, emp_salary):
        self.name = emp_name
        self.id = emp_id
        self.salary = emp_salary
    def print_employee_details(self):
        print("-----------------------------------")
        print("Name: ", self.name)
        print("ID: ", self.id)
        print("Salary: ", self.salary)
        print("-----------------------------------")
n =input("Enter name of Employee: " )
identity = input("Enter ID of Employee: " )
s = input("Enter salary of Employee: " )
employee = Employee(n, identity, s)
print("\nOriginal Employee Details:")
employee.print_employee_details()
```



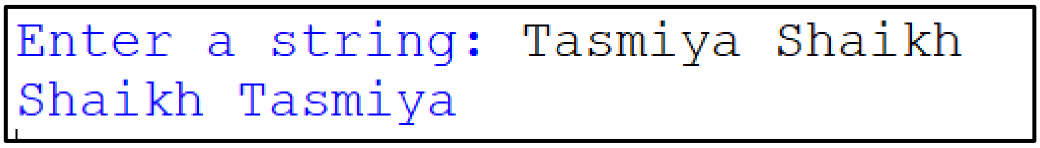
```
Enter name of Employee: Tasmiya Shaikh
Enter ID of Employee: 139
Enter salary of Employee: 60000
Original Employee Details:
----------------------
Name:  Tasmiya Shaikh
ID:  139
Salary:  60000
----------------------
```

Figure 1: Output of Q1

Q2. Write a Python class to reverse a string word by word.

```
class String:
    def reverse_words(self, s):
        return ' '.join(reversed(s.split()))

name= input("Enter a string: ")
print(String().reverse_words(name))
```

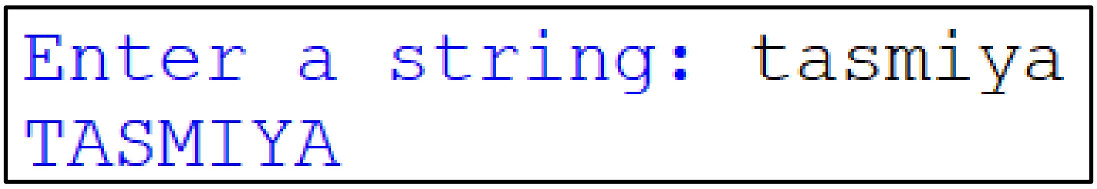


```
Enter a string: Tasmiya Shaikh
Shaikh Tasmiya
```

Figure 2: Output of Q2

Q3. Write a Python class which has two methods get_String and print_String. get_String accepts a string from the user and print String print the string in upper case.

```
class IOString():
    def __init__(self):
        self.str1 = ""
    def get_String(self):
        self.str1 = input("Enter a string: ")
    def print_String(self):
        print(self.str1.upper())
str1 = IOString()
str1.get_String()
str1.print_String()
```

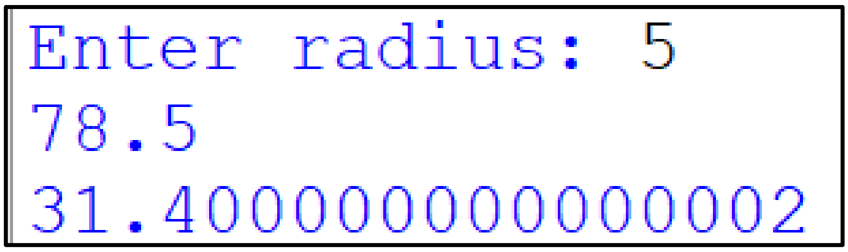


```
Enter a string: tasmiya
TASMIYA
```

Figure 3: Output of Q3

Q4. Write a Python class named Circle constructed by a radius and two methods which will compute the area and the perimeter of a circle.

```
class Circle():
    def __init__(self, r):
        self.radius = r
    def area(self):
        return self.radius*self.radius*3.14
    def perimeter(self):
        return 2*self.radius*3.14
radius1 = int(input("Enter radius: "))
NewCircle = Circle(radius1)
print(NewCircle.area())
print(NewCircle.perimeter())
```

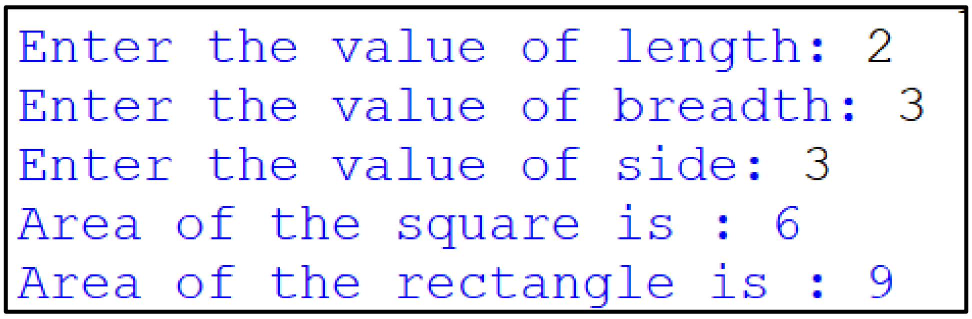


```
Enter radius: 5
78.5
31.400000000000002
```

Figure 4: Output of Q4

Q5. Create class Area for calculating area of a square and rectangle using method overloading.

```
class Area:
  def area(self, l = None, b = None):
    if l != None and b != None:
      print("Area of the square is :" ,l*b)
    elif l != None:
      print("Area of the rectangle is :" ,l*l)
length = int(input("Enter the value of length: "))
breadth = int(input("Enter the value of breadth: "))
side = int(input("Enter the value of side: "))
obj = Area()
obj.area(length, breadth)
obj.area(side)
```

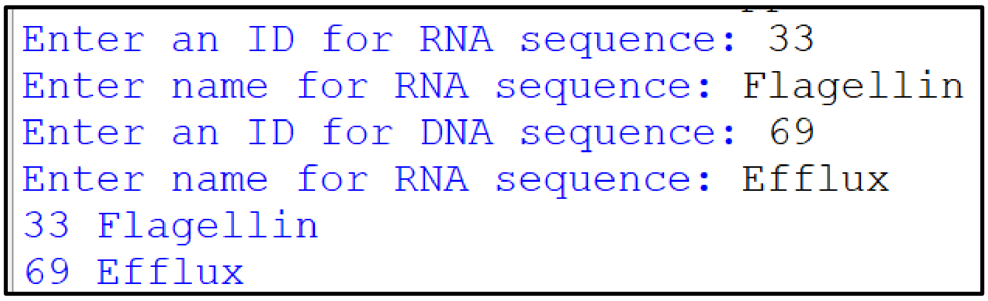


```
Enter the value of length: 2
Enter the value of breadth: 3
Enter the value of side: 3
Area of the square is : 6
Area of the rectangle is : 9
```

Figure 5: Output of Q5

Q6. Create class Sequence (seq_id, seq_name) and inherit in class RNA and DNA (use any attributes) using hierarchical inheritance.

```
class Sequence:
    def __init__(self, seq_id, seq_name):
        self.seq_id = seq_id
        self.seq_name = seq_name
class RNA(Sequence):
    def __init__(self, seq_id, seq_name):
        super().__init__(seq_id, seq_name)
class DNA(Sequence):
    def __init__(self, seq_id, seq_name):
        super().__init__(seq_id, seq_name)
rnaid = input("Enter an ID for RNA sequence: ")
rnaname = input("Enter name for RNA sequence: ")
dnaid = input("Enter an ID for DNA sequence: ")
dnaname = input("Enter name for RNA sequence: ")
rna = RNA(rnaid,rnaname)
dna = DNA(dnaid,dnaname)
print(rna.seq_id, rna.seq_name)
print(dna.seq_id, dna.seq_name)
```



```
Enter an ID for RNA sequence: 33
Enter name for RNA sequence: Flagellin
Enter an ID for DNA sequence: 69
Enter name for RNA sequence: Efflux
33 Flagellin
69 Efflux
```

Figure 6: Output of Q6