VIDHI ROHIRA

S.Y B.TECH

SEM III

COMPUTER ENGINEERING

PD LAB 10

231071052

BATCH – C

LABORATORY 10

AIM:- To study Classes and objects in Python.

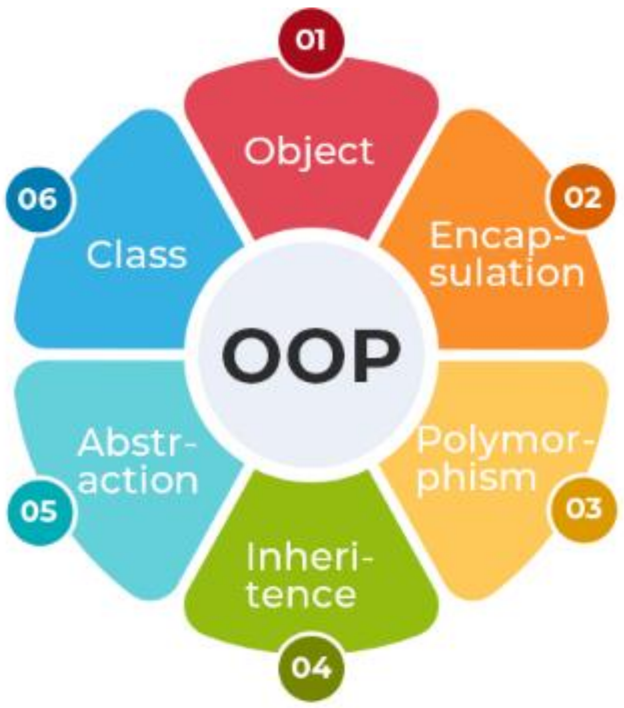
THEORY:-

Q] WHAT ARE OOPs?

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around data, or objects, rather than functions and logic. In OOP, objects are instances of classes, which serve as blueprints that define the properties (attributes) and behaviors (methods) the object should have. This approach helps manage the complexity of large software systems by breaking them down into smaller, reusable components that interact in structured ways.

OOP is built around four main principles: encapsulation, abstraction, inheritance, and polymorphism. Encapsulation involves bundling data and methods that work on that data within a single unit, or class, while hiding internal details to prevent unintended interference. Abstraction simplifies complex systems by showing only essential information to the user while hiding unnecessary details. Inheritance allows a class to inherit properties and behaviors from another class, promoting code reuse and making it easier to extend existing code. Polymorphism enables objects to be treated as instances of their parent class, allowing the same operation to work on different types of objects in an interface-driven way.

These principles allow developers to build modular, maintainable, and flexible code, as well as improve readability and collaboration across teams. OOP is widely used in languages like Python, Java, C++, and many others, and is especially helpful in building applications where different parts need to interact closely while remaining independent and interchangeable.



Q] WHAT ARE CLASSES?

In programming, classes are fundamental building blocks of object-oriented programming (OOP) that allow developers to create complex, organized, and reusable code. A class acts as a blueprint for objects, defining the structure and behaviors they should have. It typically contains attributes (variables) to store the object's state and methods (functions) to define its behavior. When a class is defined, it doesn’t actually hold any data but serves as a template from which individual instances, or objects, can be created. Each object instantiated from a class has its own copy of the attributes, allowing multiple objects to exist independently but with the same structure and capabilities. Classes also enable the use of concepts like inheritance, where a new class can inherit properties and behaviors from an existing one, making it easier to extend and maintain code. This approach helps to manage complexity, promotes modularity, and encourages code reuse across different parts of a program or even in different projects.

Q] WHAT ARE OBJECTS?

In programming, objects are instances of classes that encapsulate data and behavior. Think of an object as a specific entity created based on a class, which serves as its blueprint. Each object has its own state, stored in attributes (also known as properties or variables), and behaviors, defined by methods (functions) within the class. For example, in a program with a Car class, each Car object might have its own attributes like color, model, and speed, and methods like start, stop, or accelerate. Objects are essential in Object-Oriented Programming (OOP) because they allow developers to model real-world entities in a structured way, creating code that mirrors the behavior and characteristics of actual things. This approach helps manage complex systems by organizing data and functionality together, making it easier to work with and extend the code. Through objects, developers can manipulate data and perform tasks in an organized, modular manner, while encapsulating specific details and allowing interaction only through defined methods. This encapsulation enhances security, reduces dependencies, and promotes reusable, scalable code across applications.

Syntax of Class and Object:

```python
class ClassName:
    # Statement
```

```python
obj = ClassName()
print(obj.atrr)
```

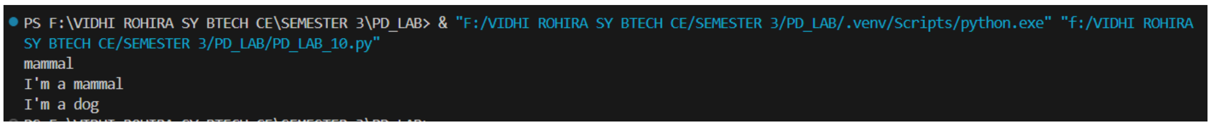
Example:

```python
class Dog:
    sound = "bark"
```

Identity	**State/Attributes**	**Behaviors**
Name of dog	*Breed*	*Bark*
	Age	*Sleep*
	Color	*Eat*

```python
class Dog:

    # A simple class
    # attribute
    attr1 = "mammal"
    attr2 = "dog"

    # A sample method
    def fun(self):
        print("I'm a", self.attr1)
        print("I'm a", self.attr2)


# Driver code
# Object instantiation
Rodger = Dog()

# Accessing class attributes
# and method through objects
print(Rodger.attr1)
Rodger.fun()
```

Output:



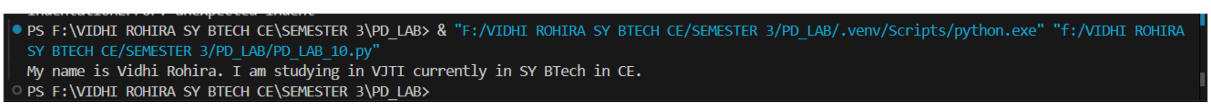
```
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\PD_LAB> & "F:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/PD_LAB/.venv/Scripts/python.exe" "f:/VIDHI ROHIRA
SY BTECH CE/SEMESTER 3/PD_LAB/PD_LAB_10.py"
mammal
I'm a mammal
I'm a dog
```

Self in Python Classes:

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.

```
class Student:
    def __init__(self, name, college, year, degree):
        self.name = name
        self.college = college
        self.year = year
        self.degree = degree

    def show(self):
        print("My name is " + self.name + ". I am studying in " + self.college
+
              " currently in " + self.year + " " + self.degree + ".")


obj = Student("Vidhi Rohira", "VJTI", "SY", "BTech in CE")
obj.show()
```

Output:



```
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\PD_LAB> & "F:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/PD_LAB/.venv/Scripts/python.exe" "f:/VIDHI ROHIRA
SY BTECH CE/SEMESTER 3/PD_LAB/PD_LAB_10.py"
My name is Vidhi Rohira. I am studying in VJTI currently in SY BTech in CE.
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\PD_LAB>
```

- The Self Parameter does not call it to be **Self**, You can use any other name instead of it. Here we change the self to the word someone and the output will be the same.

```
class Student:
    def __init__(Vidhi, name, college, year, degree):
        Vidhi.name = name
        Vidhi.college = college
        Vidhi.year = year
        Vidhi.degree = degree

    def show(Vidhi):
        print("My name is " + Vidhi.name + ". I am studying in " +
Vidhi.college +
              " currently in " + Vidhi.year + " " + Vidhi.degree + ".")


obj = Student("Vidhi Rohira", "VJTI", "SY", "BTech in CE")
obj.show()
```

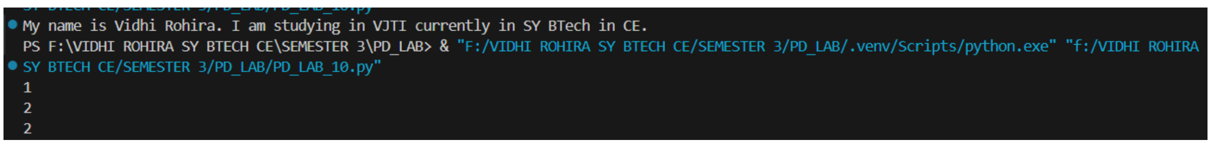
Output remains the same

Class Attributes:

Class attributes: Class attributes belong to the class itself they will be shared by all the instances. Such attributes are defined in the class body parts usually at the top, for legibility.

```python
class sampleclass:
    count = 0     # class attribute

    def increase(self):
        sampleclass.count += 1

# Calling increase() on an object
s1 = sampleclass()
s1.increase()
print(s1.count)

# Calling increase on one more
# object
s2 = sampleclass()
s2.increase()
print(s2.count)

print(sampleclass.count)
```

Output:



```
● My name is Vidhi Rohira. I am studying in VJTI currently in SY BTech in CE.
  PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\PD_LAB> & "F:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/PD_LAB/.venv/Scripts/python.exe" "f:/VIDHI ROHIRA
● SY BTECH CE/SEMESTER 3/PD_LAB/PD_LAB_10.py"
  1
  2
  2
```

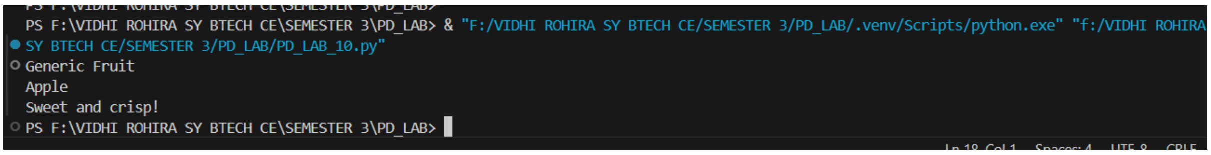
SubClasses in Python:

Q] What is a SubClass?

In Python, a subclass is a class that inherits attributes and methods from another class, known as the superclass or parent class. When you create a subclass, it can reuse and extend the functionality of the superclass. This allows you to create specialized versions of existing classes without having to rewrite common functionality. To create a subclass in Python, you define a new class and specify the superclass in parentheses after the class name.

```python
class Fruit:
    def __init__(self, name):
        self.name = name

    def taste(self):
        pass

class Apple(Fruit):
    def taste(self):
        return "Sweet and crisp!"

generic_fruit = Fruit("Generic Fruit")
apple_instance = Apple("Apple")

print(generic_fruit.name)
print(apple_instance.name)
print(apple_instance.taste())
```

Output:

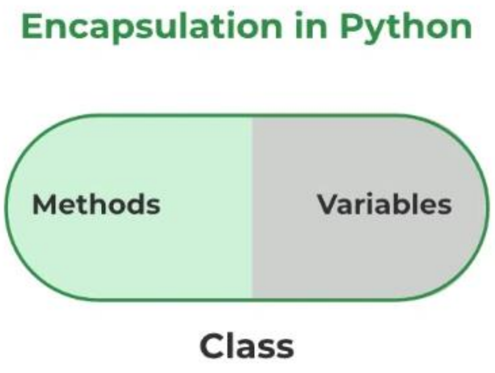


```
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\PD_LAB> & "F:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/PD_LAB/.venv/Scripts/python.exe" "f:/VIDHI ROHIRA
SY BTECH CE/SEMESTER 3/PD_LAB/PD_LAB_10.py"
Generic Fruit
Apple
Sweet and crisp!
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\PD_LAB>
```

Encapsulation:

Q] What is Encapsulation?

Encapsulation is a key concept in object-oriented programming (OOP). It refers to the practice of bundling data and the methods that operate on that data within a single unit. This approach restricts direct access to variables and methods, helping prevent accidental changes to data. To maintain control over an object's state, variables can be made accessible only through specific methods, known as private variables, which are only modifiable within the object itself. A class serves as an example of encapsulation, as it groups together data, such as member functions and variables. The objective of information hiding is to keep an object's state valid by restricting external access to its attributes, ensuring they remain hidden from the outside world.

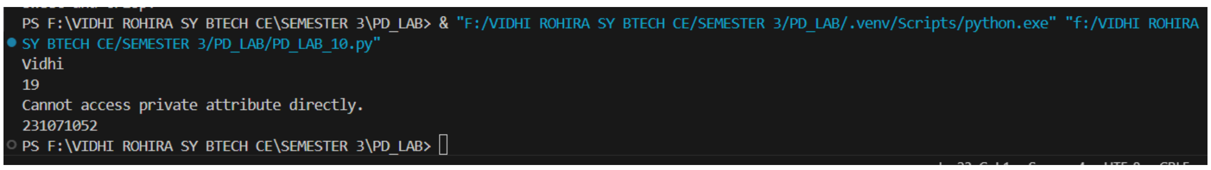


In Python, encapsulation is implemented using access specifiers that control how class members are accessed:

- **Public Members**: By default, all attributes and methods in Python are public, meaning they can be accessed from outside the class without restrictions. For example, self.name can be accessed freely as object.name.
- **Protected Members**: Using a single underscore prefix (_attribute) suggests that an attribute or method is for internal use within the class and its subclasses. Although it can still be accessed from outside, the underscore indicates that it's intended for internal use.
- **Private Members**: A double underscore prefix (__attribute) makes an attribute or method private by triggering name mangling, where the interpreter changes the attribute's name internally. This makes it more difficult to access from outside the class, although it can still be done indirectly if necessary. Private members are primarily used to safeguard sensitive data or internal logic.

```python
class Person:
    def __init__(self, name, age):
        self.name = name          # Public attribute
        self._age = age            # Protected attribute
        self.__ssn = "231071052"   # Private attribute

    def display_info(self):
        print("Name:", self.name)
        print("Age:", self._age)
        print("SSN:", self.__ssn)

person = Person("Vidhi", 19)

print(person.name)        # Output: Vidhi
print(person._age)        # Output: 19

try:
    print(person.__ssn)
except AttributeError:
    print("Cannot access private attribute directly.")

print(person._Person__ssn)  # Output: 231071052
```

Output:



```
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\PD_LAB> & "F:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/PD_LAB/.venv/Scripts/python.exe" "f:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/PD_LAB/PD_LAB_10.py"
Vidhi
19
Cannot access private attribute directly.
231071052
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\PD_LAB>
```

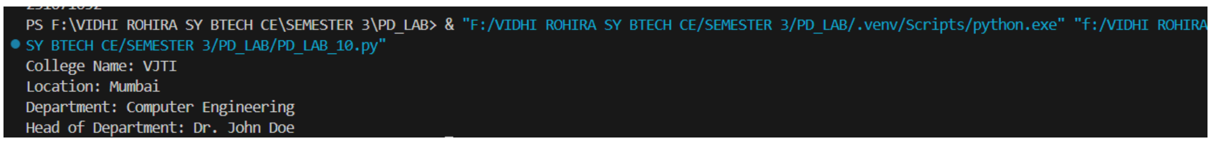
Inheritance:

Q] What is Inheritance?

Inheritance in Python is a mechanism where a new class (child or subclass) can inherit attributes and methods from an existing class (parent or superclass). This allows the child class to reuse code from the parent class and add or modify its own functionality without affecting the parent class.
In Python, inheritance is implemented by defining a subclass that specifies the parent class in its declaration. The child class inherits all public and protected members of the parent class, but private members are not directly accessible unless through name mangling.

```python
class College:
    def __init__(self, name, location):
        self.name = name
        self.location = location

    def display_college_info(self):
        print(f"College Name: {self.name}")
        print(f"Location: {self.location}")

class Department(College):
    def __init__(self, name, location, department_name, head_of_department):
        super().__init__(name, location)
        self.department_name = department_name
        self.head_of_department = head_of_department

    def display_department_info(self):
        print(f"Department: {self.department_name}")
        print(f"Head of Department: {self.head_of_department}")

# Creating an instance of Department (child class)
dept = Department("VJTI", "Mumbai", "Computer Engineering", "Dr. John Doe")

dept.display_college_info()  # Inherited from College
dept.display_department_info()  # Specific to Department
```

Output:



```
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\PD_LAB> & "F:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/PD_LAB/.venv/Scripts/python.exe" "f:/VIDHI ROHIRA
● SY BTECH CE/SEMESTER 3/PD_LAB/PD_LAB_10.py"
College Name: VJTI
Location: Mumbai
Department: Computer Engineering
Head of Department: Dr. John Doe
```

Polymorphism:

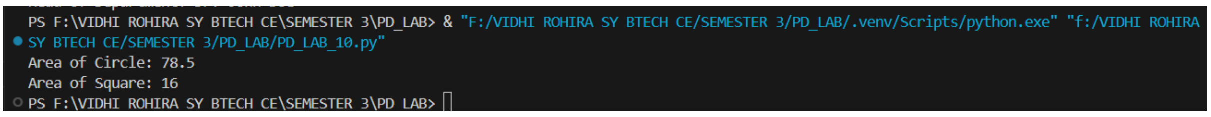
Q] What is Polymorphism?

Polymorphism in Python is a feature that allows objects of different classes to be treated as objects of a common superclass. It enables a single function, method, or operator to work in different ways depending on the object it is applied to. There are two types of polymorphism in Python:

1. **Method Overloading**: Python does not support method overloading directly, but it allows you to define methods with default arguments or handle different types of inputs in a flexible way.
2. **Method Overriding**: This occurs when a subclass provides its own implementation of a method that is already defined in its superclass.

```python
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side * self.side

# Creating instances
circle = Circle(5)
square = Square(4)

# Demonstrating polymorphism
shapes = [circle, square]
for shape in shapes:
    print(f"Area of {shape.__class__.__name__}: {shape.area()}")
```

Output:



```
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\PD_LAB> & "F:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/PD_LAB/.venv/Scripts/python.exe" "f:/VIDHI ROHIRA
SY BTECH CE/SEMESTER 3/PD_LAB/PD_LAB_10.py"
Area of Circle: 78.5
Area of Square: 16
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\PD LAB>
```

Abstraction:

Q] What is Abstraction?

Abstraction in Python is an object-oriented programming (OOP) concept that hides the implementation details of a system and exposes only the essential features or functionalities to the user. It helps to simplify complex systems by providing a clear interface while concealing the complexity of the underlying code. In Python, abstraction is typically achieved using abstract classes and methods.

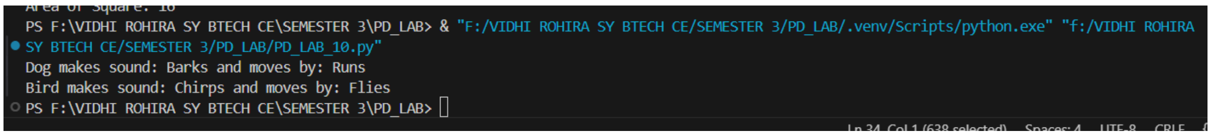
- **Abstract Class**: A class that cannot be instantiated and usually contains abstract methods.
- **Abstract Method**: A method that is declared but contains no implementation in the base class. Subclasses are required to implement this method.

Python provides the abc module (Abstract Base Class) to define abstract classes and methods.

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

    @abstractmethod
    def move(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Barks"

    def move(self):
        return "Runs"

class Bird(Animal):
    def sound(self):
        return "Chirps"

    def move(self):
        return "Flies"

# Creating instances
dog = Dog()
bird = Bird()
```

```python
# Demonstrating abstraction
animals = [dog, bird]
for animal in animals:
    print(f"{animal.__class__.__name__} makes sound: {animal.sound()} and
moves by: {animal.move()}")
```

Output:



```
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\PD_LAB> & "F:/VIDHI ROHIRA SY BTECH CE/SEMESTER 3/PD_LAB/.venv/Scripts/python.exe" "f:/VIDHI ROHIRA
SY BTECH CE/SEMESTER 3/PD_LAB/PD_LAB_10.py"
Dog makes sound: Barks and moves by: Runs
Bird makes sound: Chirps and moves by: Flies
PS F:\VIDHI ROHIRA SY BTECH CE\SEMESTER 3\PD_LAB>
```

CONCLUSION:- Hence, in this lab session, we have studied the core concepts of Object-Oriented Programming (OOP) in Python, including classes and objects, which are fundamental building blocks of OOP. We explored key OOP principles such as encapsulation, which helps in controlling access to data and protecting the internal state of an object. Additionally, we delved into polymorphism, demonstrating how methods can have different behaviors based on the object type, and abstraction, which allows us to hide the complexities of implementation while exposing only essential functionalities. Furthermore, we studied inheritance, a mechanism that allows a class to inherit attributes and methods from another, promoting code reusability and establishing hierarchical relationships between classes. Overall, these concepts enable us to write clean, modular, and maintainable code that can be easily extended and reused in various applications.