WEEK 4 ASSIGNMENT 1

Q1. Explain Class and Object with respect to Object-Oriented Programming. Give a suitable example.

Class:

- A class is a blueprint or template for creating objects. It defines a set of attributes (data members) and methods (functions) that characterize the objects of that class.

- Think of a class as a blueprint for a specific type of object. It specifies what data can be stored in an object and what actions (methods) can be performed on the object. Classes are used to encapsulate and group related data and functions together, promoting code reusability and maintainability.

```
[98]: class Car:
          def __init__(self, make, model, year):
              self.make = make
              self.model = model
              self.year = year
              self.speed = 0

          def accelerate(self):
              self.speed += 5

          def brake(self):
              if self.speed >= 5:
                  self.speed -= 5

          def get_speed(self):
              return self.speed

[99]: a = Car("toyota", "swift", 2020)

[103]: a.accelerate()

[105]: print("Current speed:", a.get_speed())
       Current speed: 5

[106]: a.brake()

[107]: print("Current speed:", a.get_speed())
       Current speed: 0
```

Object:

- An object is an instance of a class. It represents a specific instance or occurrence of the class, with its own set of attribute values.

- Objects are created based on the blueprint defined by a class. They can have unique data while sharing the same methods defined in the class.Objects allow you to interact with and manipulate data in a structured way, as defined by the class.

```
#object

# Creating two Car objects based on the Car class
car1 = Car("Toyota", "Camry", 2022)
car2 = Car("Honda", "Civic", 2023)

# Using methods on the objects
car1.accelerate()
car2.accelerate()

# Accessing object attributes
print(f"Car 1: {car1.make} {car1.model} ({car1.year}), Speed: {car1.get_speed()} mph")
print(f"Car 2: {car2.make} {car2.model} ({car2.year}), Speed: {car2.get_speed()} mph")
```

```
Car 1: Toyota Camry (2022), Speed: 5 mph
Car 2: Honda Civic (2023), Speed: 5 mph
```

Q2. Name the four pillars of OOPs.

The four pillars of object-oriented programming (OOP) are:

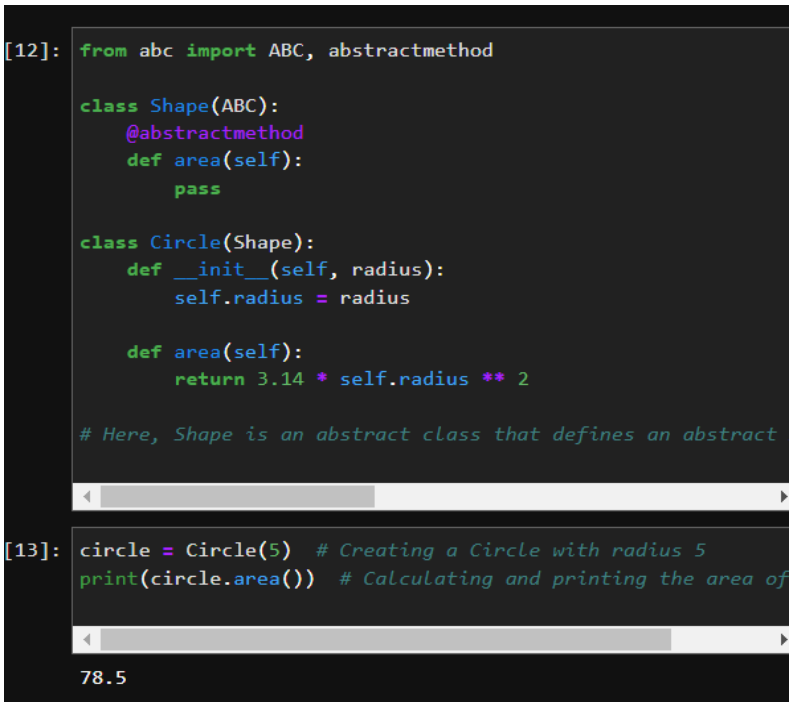
1. Encapsulation:

- Encapsulation is the concept of bundling data (attributes) and the methods (functions) that operate on that data into a single unit called a class. It restricts direct access to some of the object's components and prevents unintended interference and misuse of data.

```
[8]: class BankAccount:
         def __init__(self, account_number, balance):
             self.account_number = account_number
             self.balance = balance

         def deposit(self, amount):
             if amount > 0:
                 self.balance += amount

         def withdraw(self, amount):
             if 0 < amount <= self.balance:
                 self.balance -= amount

     # Here, the data (account_number and balance) and methods (deposit and

[9]: bank = BankAccount(123344,6000)

[12]: bank.deposit(5000)

[13]: bank.withdraw(3000)

[14]: print(bank.balance)
      8000
```

2. Abstraction:

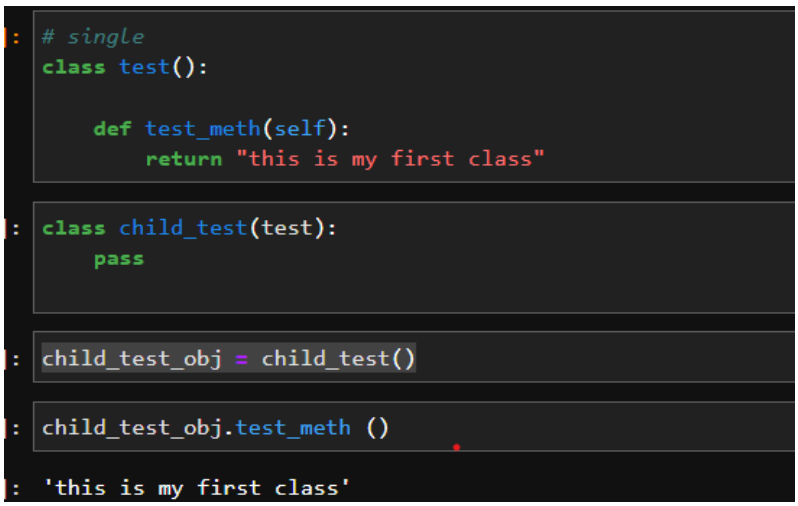
- Abstraction is the process of simplifying complex reality by modelling classes based on the essential properties and behaviours an object should have. It hides the unnecessary details of an object and only exposes the necessary features.



```
[12]: from abc import ABC, abstractmethod

      class Shape(ABC):
          @abstractmethod
          def area(self):
              pass

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

          def area(self):
              return 3.14 * self.radius ** 2

      # Here, Shape is an abstract class that defines an abstract

[13]: circle = Circle(5)  # Creating a Circle with radius 5
      print(circle.area())  # Calculating and printing the area of

      78.5
```

3. Inheritance:

- Inheritance is a mechanism that allows a new class (derived or child class) to inherit properties and behaviours from an existing class (base or parent class).It promotes code reuse and the creation of a hierarchy of classes.



```
: # single
  class test():

      def test_meth(self):
          return "this is my first class"

: class child_test(test):
      pass

: child_test_obj = child_test()

: child_test_obj.test_meth ()

: 'this is my first class'
```

4. Polymorphism:

- Polymorphism allows objects of different classes to be treated as objects of a common superclass.

- It enables dynamic binding and late binding, where the method to be executed is determined at runtime based on the actual type of object.

```
[6]: class data_science:

         def syllabus(self):
             print("this is my syllabus for ds")

[7]: class full_stack:

         def syllabus(self):
             print("this is my syllabus for fs")

[8]: def class_parcer (class_obj):
         for i in class_obj :
             i.syllabus()

[9]: data_science = data_science()

[10]: full_stack =full_stack()

[11]: class_obj = [data_science ,full_stack]

[12]: class_parcer(class_obj)

      this is my syllabus for ds
      this is my syllabus for fs
```

Q3. Explain why the __init__() function is used. Give a suitable example.

The `__init__()` function in Python is a special method also known as the constructor. It is used to initialize and set up the initial state of an object when an instance of a class is created. This method is automatically called when you create a new object of a class, and it allows you to specify how the object should be initialized with default values or values provided during object creation.

```
6]: class Person:
        def __init__(self, first_name, last_name, age):
            self.first_name = first_name
            self.last_name = last_name
            self.age = age

        def introduce(self):
            return f"My name is {self.first_name} {self.last_name}, and I am {self.age} years old."

    # Creating instances of the Person class
    person1 = Person("John", "Doe", 30)
    person2 = Person("Jane", "Smith", 25)

    # Accessing object attributes and invoking methods
    print(person1.introduce())  # Output: My name is John Doe, and I am 30 years old.
    print(person2.introduce())  # Output: My name is Jane Smith, and I am 25 years old.

    My name is John Doe, and I am 30 years old.
    My name is Jane Smith, and I am 25 years old.
]:
```

Q4. Why self is used in OOPs?

In object-oriented programming (OOP), `self` is a convention used in many programming languages, including Python, to refer to the current instance of a class. It is a reference to the object itself and is used within methods of the class to access and manipulate the object's attributes and methods.

```
class Person:
    def __init__(self, name, age):
        self.name = name  # 'self.name' is an instance variable
        self.age = age    # 'self.age' is an instance variable

    def introduce(self):
        return f"My name is {self.name} and I am {self.age} years old."

# Creating two instances of the Person class
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)

# Accessing attributes and invoking methods using 'self'
print(person1.introduce())
print(person2.introduce())

My name is Alice and I am 30 years old.
My name is Bob and I am 25 years old.
```

Q5. What is inheritance? Give an example for each type of inheritance.

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a new class (derived or child class) to inherit attributes and methods from an existing class (base or parent class). It promotes code reuse and the creation of a hierarchy of classes where common attributes and behaviours can be shared among related classes. Inheritance establishes an "is-a" relationship between the base class and the derived class, indicating that the derived class is a specialized version of the base class.

1. Single Inheritance:

- In single inheritance, a derived class inherits from a single base class.

- It represents a one-to-one relationship between a base class and a derived class.

```
# Base class
class Animal:
    def speak(self):
        pass

# Derived class
class Dog(Animal):
    def speak(self):
        return "Woof!"

# Creating an instance of the derived class
dog = Dog()

# Calling a method from the base class
print(dog.speak())  # Output: Woof!
```

```
Woof!
```

2. Multiple Inheritance:

- In multiple inheritance, a derived class can inherit from more than one base class. It allows a class to inherit attributes and methods from multiple parent classes.

```python
# Base class 1
class Bird:
    def fly(self):
        return "Flying"

# Base class 2
class Mammal:
    def run(self):
        return "Running"

# Derived class inheriting from both Bird and Mammal
class Bat(Bird, Mammal):
    pass

# Creating an instance of the derived class
bat = Bat()

# Using methods from both base classes
print(bat.fly())  # Output: Flying
print(bat.run())  # Output: Running
```

```
Flying
Running
```

3. Multilevel Inheritance:

- In multilevel inheritance, a derived class inherits from a base class, and then another class inherits from the derived class, forming a chain of inheritance. It represents a hierarchy of classes.

```python
# Grandparent class
class Animal:
    def speak(self):
        pass

# Parent class inheriting from Animal
class Mammal(Animal):
    pass

# Child class inheriting from Mammal
class Dog(Mammal):
    def speak(self):
        return "Woof!"

# Creating an instance of the child class
dog = Dog()

# Calling a method from the grandparent class
print(dog.speak())  # Output: Woof!
```

```
Woof!
```