Practical No. 8

Aim: - Write a program to explain concept of Object Oriented Programming (OOP).

Theory: Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviors into individual **objects**.

Object-oriented programming in Python is a powerful and flexible way to structure and organize our code. It helps we model real-world concepts and relationships between objects, making our code more maintainable and easier to understand.

Object-oriented programming (OOP) is a programming paradigm that is based on the concept of "objects." Objects are instances of classes, and they can contain both data (attributes) and behavior (methods). Python is a versatile programming language that supports OOP principles. Here's an overview of how we can use OOP in Python:

Classes and Objects:

- In Python, we define a class using the **class** keyword. A class is a blueprint for creating objects.
- Objects are instances of classes. We create an object by calling the class as if it were a function.

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

person1 = Person("Alice", 30)
person2 = Person("Bob", 25)
```

Attributes:

- Objects can have attributes that store data.
- We can access and modify object attributes using dot notation.

```
print(person1.name)
print(person1.age)
print(person2.name)
print(person2.age)

# Output:       Alice
                30
                Bob
                25
```

Methods:

- Methods are functions defined within a class. They can operate on the object's data (attributes).

The **self** parameter is used to refer to the instance of the object inside the class methods.

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

    def say_hello(self):
        print("Hello, my name is {self.name} and I'm {self.age} years old.")

person1 = Person("Alice", 30)
person1.say_hello()
 # Output: "Hello, my name is Alice and I'm 30 years old."
```

Inheritance:

Inheritance allows we to create a new class that is a modified version of an existing class. The new class inherits attributes and methods from the base class.

```
class Student(Person):
    def __init__(self, name, age, student_id):
        super()._init__(name, age)
        self.student_id = student_id

student = Student("Eve", 20, "12345")
student.say_hello() # Student inherits the say_hello() method from the Person class.
```

1. **Encapsulation:**
   - We can use encapsulation to control the visibility of attributes and methods within a class.
   - By convention, attributes and methods that should not be accessed from outside the class are prefixed with a single underscore (e.g., **_private_var**) to indicate that they are for internal use.
2. **Polymorphism:**
   - Polymorphism allows we to use the same interface (method name) for objects of different classes.
   - It simplifies the code and allows we to work with objects in a more abstract and generic way.

```
def introduce(person):
    person.say_hello()

alice = Person("Alice", 30)
bob = Student("Bob", 25, "98765")

introduce(alice) # Output: "Hello, my name is Alice and I'm 30 years old."
introduce(bob)    # Output: "Hello, my name is Bob and I'm 25 years old."
```

Result: The practical has been performed & studied successfully.