High Impact Skills Development Program

AI & Data Science

Lab 06: Object Oriented Programming

Introduction:

Object-Oriented Programming (OOP) is a powerful programming paradigm that enables us to organize and structure code efficiently. Python, being an object-oriented language, provides robust support for creating and utilizing objects. In this lesson, we will explore the core concepts of OOP in Python and understand how they can be used to build modular and reusable code.

1. Objects and Classes:

In OOP, an object is a self-contained entity that combines both data (attributes) and behavior (methods). A class, on the other hand, is a blueprint or template for creating objects. Here's an example of a class representing a simple Car:

Side note :

Difference between method and function:

- A function doesn't need any object and is independent, while the method is a function, which is linked with any object.
- We can directly call the function with its name, while the method is called by the object's name.
- Function is used to pass or return the data, while the method operates the data in a class.

```
class Car:
    def __init__(self, brand, color):
        self.brand = brand
        self.color = color

    def drive(self):
        print(f"The {self.color} {self.brand} is driving.")
```

2. Instantiation:

To create an object from a class, we use a process called instantiation. It involves calling the class as if it were a function, which returns an instance of the class. Here's an example of creating a Car object:

```
my_car = Car("Toyota", "blue")
```

3. Attributes:

Attributes are the data associated with an object. In Python, they are accessed using dot notation. In the Car class example, brand and color are attributes. To access them, we use my_car.brand and my_car.color, respectively.

4. Methods:

Methods define the behavior or actions that an object can perform. They are defined inside a class and are accessed through objects. In the Car class example, drive() is a method. To call it, we use my_car.drive().

5. Inheritance:

Inheritance allows us to create a new class that inherits the properties and methods of an existing class, known as the superclass or parent class. The new class is called the subclass or child class. It helps in reusing and extending existing code. Here's an example:

```
class SportsCar(Car):
    def boost(self):
        print(f"The {self.color} {self.brand} is boosting.")
```

```
def perform_action(car):
    car.drive()  # Common method
```

```
my_car = Car("Toyota", "blue")
perform_action(my_car)

sports_car = SportsCar("Ferrari", "red")
perform_action(sports_car)
```

In the above code, the perform_action() function can accept both Car and SportsCar objects, demonstrating polymorphic behavior.

6. Abstraction and encapsulation:

```
class BankAccount:
    def __init__(self, account_number, balance):
        self._account_number = account_number  # Encapsulated attribute
        self._balance = balance                # Encapsulated attribute

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        if self._balance >= amount:
            self._balance -= amount
        else:
            print("Insufficient balance.")

    def get_balance(self):
        return self._balance

    def display_account_info(self):
        print(f"Account Number: {self._account_number}")
        print(f"Balance: {self._balance}")
```

In the above example, we have a BankAccount class representing a bank account. Let's see how abstraction and encapsulation are applied:

Abstraction:

1. Abstraction involves hiding unnecessary details and exposing only relevant information. In the BankAccount class, we provide methods such as deposit(), withdraw(), get_balance(), and display_account_info() to interact with the account. These methods abstract away the internal workings of the account, allowing users to perform operations without worrying about the underlying implementation.

Encapsulation:

2. Encapsulation refers to the bundling of data and methods together within a class, where the data (attributes) are kept private or protected from direct external access. In the BankAccount class, the account number and balance attributes (_account_number and _balance) are encapsulated by using the underscore prefix convention (e.g., _account_number). By doing so, we indicate to other developers that these attributes should be treated as internal and accessed only through the defined methods (e.g., get_balance() and display_account_info()). ***Encapsulation helps maintain data integrity and prevents unauthorized modifications***.

Usage example:

```
# Create a bank account object
account = BankAccount("123456789", 1000)

# Deposit money
account.deposit(500)

# Withdraw money
account.withdraw(200)

# Display account information
account.display_account_info()
```

Output:

```
Account Number: 123456789
Balance: 1300
```

Polymorphism

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It enables different objects to respond to the same method or function call in different ways based on their specific implementation.

```
class Animal:
      def make_sound(self):
      print("Animal is making a sound.")

class Dog(Animal):
      def make_sound(self):
      print("Dog is barking.")

class Cat(Animal):
      def make_sound(self):
      print("Cat is meowing.")

# Creating instances of the classes
animal = Animal()
dog = Dog()
cat = Cat()

# Calling the make_sound method
animal.make_sound()  # Output: Animal is making a sound.
dog.make_sound()      # Output: Dog is barking.
```

```
cat.make_sound()      # Output: Cat is meowing.
```

Conclusion:

Object-Oriented Programming in Python provides a structured approach to programming, allowing us to model real-world entities as objects and define their behavior through methods. By utilizing classes, objects, attributes, methods, inheritance, and polymorphism, we can build modular, reusable, and maintainable code. Understanding these core concepts is essential for mastering OOP in Python.

Lab Tasks

1. Write a Python class called Rectangle that has attributes length and width. Implement methods to calculate the area and perimeter of the rectangle.

2. Create a class hierarchy in Python consisting of a base class called Animal, and derived classes Dog, Cat, and Bird. Each derived class should have its unique method. Instantiate objects of each class and demonstrate polymorphism by calling a common method on each object.

3. Write a Python class called Shape with a method area() that returns the area of the shape. Create two subclasses, Rectangle and Circle, each overriding the area() method to 7calculate the area of the specific shape. Instantiate objects of each class and display their areas.