MD01w03: D1: Class and Object in OOP

GRID038

17/06 - 22/06 2025

1 Introduction to OOP

Early languages like Pascal and C were procedural programming languages. Lines of code were executed from top to bottom (series of instructions) in a certain order. Basically, a sequential execution from start to finish is still required to achieve a goal. So, code reuse is limited because it is often required to copy code.

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into objects that have attributes (data) and methods (behaviors). OOP makes code easier to reuse, extend, and maintain. So, it has become a paradigm widely used in many programming languages today.

1.1 Class and Object

A class is a blueprint or template that defines the properties and behavior of an object. An object is an instance of a class, created using the class definition.

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

    def show(self):
        print("Model is", self.model)
        print("Color is", self.color)

# Creating instances of the class
audi = Car("Audi A4", "Blue")
ferrari = Car("Ferrari 488", "Green")

# Calling instance methods
audi.show()
ferrari.show()
```

1.2 Encapsulation

Encapsulation is the concept of hiding the implementation details of an object from the outside world and only exposing the necessary information through public methods. This approach:

- Provides better control over data.
- Prevents accidental modification of data.
- Promotes modular programming.

Python achieves encapsulation through public, protected and private attributes.

1.3 Inheritance

Inheritance is a mechanism that allows a class to inherit properties and methods from another class, called the superclass or parent class.

```
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name  # Initialize the name attribute

    def speak(self):
        pass

# Child class inheriting from Animal
class Dog(Animal):
    def speak(self):
        return f"{self.name} barks!"  # Override the speak method

# Creating an instance of Dog
dog = Dog("Buddy")
print(dog.speak())  #Output: Buddy barks!
```

1.4 Polymorphism

Polymorphism is the ability of an object to take on the same method but behave differently.

```
class Animal:
    def sound(self):
        return "Some generic sound"

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

class Cat(Animal):
    def sound(self):
        return "Meow"

# Polymorphic behavior
animals = [Dog(), Cat(), Animal()]
for animal in animals:
    print(animal.sound())
```

```
#Output: Bark
#Output: Meow
#Output: Some generic sound
```

1.5 Abstraction

Abstraction is the concept of showing only the necessary information to the outside world while hiding unnecessary details.

In Python, you can achieve abstraction using abstract base classes (ABC) and abstract methods.

```
from abc import ABC, abstractmethod

# Define an abstract class
class Animal(ABC):

    @abstractmethod
    def sound(self):
        pass  # This is an abstract method, no implementation here.

# Concrete subclass of Animal
class Dog(Animal):

    def sound(self):
        return "Bark"  # Providing the implementation of the
                                          abstract method

# Create an instance of Dog
dog = Dog()
print(dog.sound())  # Output: Bark
```

2 Summary

In this article, we covered the following.

- Undertanding about OOP
- Basic about Characteristics of OOP
- Some basic Python code about OOP

In summary, Python's flexible and efficient support for Object-Oriented Programming (OOP) empowers developers to build well-structured, maintainable, and scalable applications by leveraging key principles like modularity, code reusability, and extensibility, all while naturally modeling real-world problems.