Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming paradigm that uses **objects** and **classes** to design and structure code efficiently. Python is an **object-oriented** language, meaning everything in Python is an object.

1. Basics of OOP

1.1 What is OOP?

OOP is a way of structuring code using objects that bundle data (attributes) and behavior (methods) together. It helps in:

- **Code reusability**
- **Scalability**
- **Better organization**
- **Data encapsulation and security**

1.2 Four Pillars of OOP

1. **Encapsulation** – Hiding data to protect it from unintended modifications.
2. **Abstraction** – Hiding implementation details and exposing only necessary functionality.
3. **Inheritance** – Creating new classes from existing ones to promote code reuse.
4. **Polymorphism** – Allowing different classes to use the same interface.

2. Classes and Objects

2.1 Creating a Class and Object

In Python, a class is a blueprint for creating objects.

```
# Defining a class
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def show_details(self):
        print(f"Car: {self.brand} {self.model}")
```

```
# Creating an object
car1 = Car("Toyota", "Corolla")
car1.show_details()
```

🔹 `__init__` is the **constructor** method that initializes object properties.
🔹 `self` refers to the current instance of the class.

---

3. Encapsulation (Data Hiding)

Encapsulation is achieved using **private attributes** (denoted by __ before the attribute name).

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

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

    def get_balance(self):
        return self.__balance

# Creating an object
account = BankAccount(5000)
account.deposit(2000)
print(account.get_balance())  # Output: 7000

# Accessing private attribute directly will result in an error
# print(account.__balance)  ❌ AttributeError
```

🔹 Use **getter (`get_balance()`) and setter (`deposit()`) methods** to access private attributes safely.

---

4. Inheritance (Code Reusability)

Inheritance allows a class to inherit properties and methods from another class.

4.1 Single Inheritance

```
class Animal:
    def sound(self):
        print("Animals make different sounds.")

class Dog(Animal):  # Inheriting Animal class
    def sound(self):
        print("Dog barks.")
```

```
dog = Dog()
dog.sound()  # Output: Dog barks.
```

4.2 Multiple Inheritance

A class can inherit from multiple parent classes.

```
class Parent1:
    def feature1(self):
        print("Feature 1 from Parent1")

class Parent2:
    def feature2(self):
        print("Feature 2 from Parent2")

class Child(Parent1, Parent2):
    pass

obj = Child()
obj.feature1()  # Output: Feature 1 from Parent1
obj.feature2()  # Output: Feature 2 from Parent2
```

4.3 Multilevel Inheritance

```
class Grandparent:
    def feature1(self):
        print("Feature 1 from Grandparent")

class Parent(Grandparent):
    def feature2(self):
        print("Feature 2 from Parent")

class Child(Parent):
    def feature3(self):
        print("Feature 3 from Child")

obj = Child()
obj.feature1()  # Inherited from Grandparent
obj.feature2()  # Inherited from Parent
obj.feature3()  # Defined in Child
```

---

5. Polymorphism (Multiple Forms)

5.1 Method Overriding

A child class can override a method of the parent class.

```
class Bird:
    def sound(self):
        print("Birds chirp.")
```

```
class Sparrow(Bird):
    def sound(self):
        print("Sparrow chirps sweetly.")

obj = Sparrow()
obj.sound()  # Output: Sparrow chirps sweetly.
```

5.2 Method Overloading (Simulated)

Python does not support method overloading directly, but it can be simulated using **default arguments**.

```
class MathOperations:
    def add(self, a, b, c=0):
        return a + b + c

math = MathOperations()
print(math.add(2, 3))      # Output: 5
print(math.add(2, 3, 4))   # Output: 9
```

---

6. Abstraction (Hiding Details)

Python provides **abstract classes** using the `abc` module.

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def fuel_type(self):
        pass

class Car(Vehicle):
    def fuel_type(self):
        return "Petrol or Diesel"

class ElectricCar(Vehicle):
    def fuel_type(self):
        return "Electricity"

car = Car()
print(car.fuel_type())  # Output: Petrol or Diesel
```

◆ `ABC` stands for **Abstract Base Class**.
◆ `@abstractmethod` enforces implementation in child classes.

---

7. Special Methods (Dunder Methods)

These methods start and end with __ (double underscores).

7.1 `__str__` Method (String Representation)

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

    def __str__(self):
        return f"Person({self.name}, {self.age})"

p1 = Person("Alice", 25)
print(p1)  # Output: Person(Alice, 25)
```

7.2 Operator Overloading

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2
print(p3.x, p3.y)  # Output: 4 6
```

---

8. Advanced OOP Concepts

8.1 Class Methods and Static Methods

- **Class Method (`@classmethod`)** works with class-level attributes.
- **Static Method (`@staticmethod`)** does not depend on class attributes.

```
class Student:
    school = "ABC School"

    @classmethod
    def change_school(cls, new_school):
        cls.school = new_school

    @staticmethod
    def info():
        print("This is a Student class.")

Student.change_school("XYZ School")
print(Student.school)  # Output: XYZ School
Student.info()  # Output: This is a Student class.
```

9. OOP Best Practices

- Use **meaningful class and method names**.
- Follow **PEP 8** coding conventions.
- Keep **attributes private** when needed.
- Use **inheritance wisely** to avoid complexity.
- Implement **abstraction for maintainability**.

10. Summary

Concept	Description
Encapsulation	Hiding data using private attributes
Abstraction	Hiding implementation details using abstract classes
Inheritance	Reusing code by inheriting classes
Polymorphism	Using the same method name for different behaviors
Dunder Methods	Special methods for operator overloading and customization

☺