Object-Oriented Programming (OOP) in Python — a powerful way to organize and reuse code.

What is OOP?

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "**objects**", which contain:

- **Attributes** (data/properties)
- **Methods** (functions that act on the data)

Simply, OOP is a way to organize code by combining data and functions into classes and objects.

Advantages of OOP

- Provides a clear structure to programs
- Makes code easier to maintain, reuse, and debug
- Helps keep your code DRY (**Don't Repeat Yourself**)
- Allows you to build reusable applications with less code

Tip: The DRY principle means you should avoid writing the same code more than once. Move repeated code into functions or classes and reuse it.

Python supports all main OOP principles:

- **Class and Object**
- **Encapsulation**
- **Inheritance**
- **Polymorphism**

1. Class and Object

Classes and objects are the two core concepts in object-oriented programming.

A class defines what an object should look like, and an object is created based on that class. For example:

Class	**Objects**
Fruit	Apple, Banana, Mango
Car	Volvo, Audi, Toyota

When you create an object from a class, it inherits all the variables and functions defined inside that class.

- **Class :** A class is a blueprint for creating objects.
- **Object :** An object is an instance of a class.
- **Constructor `__init__()` :** Automatically runs when object is created.
- **`Self` Keyword :** Refers to the current object.
- **__str__() function:** String representation of the object is returned

Example:

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

    def greet(self):      # method
        print(f"Hello, my name is {self.name} and I am {self.age} years 
old.")

# Creating object
s1 = Person("Raj", 24)
s1.greet()
```

Output:

```
Hello, my name is Raj and I am 24 years old.
```

2. Inheritance

A class can **inherit** properties and methods from another class. Allows a class to reuse code from another class.

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

d = Dog()
d.speak()  # from parent
d.bark()   # from child
```

Example 2 : Inheriting form Person Class

- **super() function** that will make the child class inherit all the methods and properties from its parent:

```
class Student(Person):  # Inheriting from Person
    def __init__(self, name, age, college):
        super().__init__(name, age)  # Calls the constructor of Person
        self.college = college       # Adds a new property to Student

    def show_college(self):
        print("College:", self.college)

s1 = Student("Sita", 21, "TU")
s1.greet()
```

3. Polymorphism

Same method name behaves differently for different classes.

```
class Cat:
    def sound(self):
        print("Meow")
class Dog:
    def sound(self):
        print("Bark")

c = Cat()
d = Dog()
c.sound()
d.sound()
```

Output:
```
Meow
Bark
```

➲ **Or you can do like this also**

```
class Dog:
    def speak(self):
        print("Woof")

class Cat:
    def speak(self):
        print("Meow")

for animal in [Dog(), Cat()]:
    animal.speak()
```

4. Encapsulation

Hiding internal data using **private** variables (prefix _ or __) and **getters/setters**.

```
class BankAccount:
    def __init__(self):
        self.__balance = 0  # private variable

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

    def get_balance(self):
        return self.__balance

acc = BankAccount()
acc.deposit(1000)
print("Balance:", acc.get_balance())
```

Or

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

    def get_balance(self):
        return self.__balance

a1 = BankAccount(1000)
print(a1.get_balance())
```

Summary Table

Concept	Description
Class	Blueprint for objects
Object	Instance of a class
Encapsulation	Hiding data using private variables (`__var`)
Inheritance	One class inherits from another
Polymorphism	Same method behaves differently in different classes

📝 Mini Assignment – OOP

Create a file `oop_assignment.py` and complete the following:

Q1. Create a `Book` Class

- Properties: `title`, `author`, `year`
- Method: `info()` to print book details

Q2. Inherit from Book → `EBook` Class

- Add: `file_size` (MB)
- Add method: `download()` to show download message

Q3. Create Multiple Objects

- Create 2 books and 1 ebook
- Call their methods

Q4. Use Encapsulation

- Add private `_rating` to Book
- Create method to safely get/set rating