What is OOP?

OOP stands for **Object-Oriented Programming**. It's like organizing your code just like you organize real-life things — like cars, students, mobiles — into categories.

Encapsulation

- Wraps **data and methods** into a single unit (class).
- Uses **access modifiers** (private, public, protected).
- **Hides internal details**, only exposes what's necessary.
- Achieved using **getters and setters**.
- Ensures **data protection and security**.

---

2. Abstraction

- Shows only the **essential features** of an object.
- **Hides complex internal logic** from the user.
- Focuses on **what** an object does, not **how**.
- Achieved using **abstract classes or interfaces**.
- Increases **code simplicity** and **maintainability**.

---

3. Inheritance

- Allows a class (**child**) to inherit properties/methods of another (**parent**).
- Promotes **code reuse** and **hierarchical classification**.
- Supports **single**, **multiple**, **multilevel**, **hierarchical** inheritance.
- Helps in **extending functionality** without rewriting code.

---

4. Polymorphism

- Means "**many forms**".
- Allows **same method name** to behave differently:

- **Function overloading** (compile-time)
- **Function overriding** (run-time)

- Increases **flexibility** and **extensibility** of code.
- Helps in **dynamic behavior** based on object type.

1. Class – The Blueprint

A **class** is like a design or a plan for something.

➔Think of a **mobile phone company** that makes many phones. The design (class) is the same, but each phone (object) is different.

***Syntax*:**

```
class ClassName:
    def method_name(self):
        print("Doing something")
```

Example:

```
class Dog:
  def bark(self):
    print(f"{self.name} is barking!")
```

2. Object – Real Thing Created from Class

An **object** is the actual thing made using the class. Like a real dog from the dog design.

Syntax:

```
object_name = ClassName(value)
```

Example:

```
d1 = Dog("Tommy")
d1.bark()  # Output: Tommy is barking!
```

What is a Constructor?

A **constructor** is a special function in a class that **automatically runs** when you **create an object**.

Think of it like setting up a new phone — when you buy it, you enter your name, Wi-Fi, etc.

That setup is the constructor.

Why Use a Constructor?

- To **give values** to the object at the time of creation.
- It saves time and ensures the object is **ready to use**.

Python Constructor Syntax

```
class ClassName:
    def __init__(self, arg1, arg2):
        self.arg1 = arg1
        self.arg2 = arg2
```

- __init__is the constructor in Python.
- self refers to the current object.
- It runs **automatically** when you create the object

Point	**Explanation**
Name of constructor	__init__
Auto-run	Runs automatically when object is created
Purpose	To initialize object variables
Use self	To access object's own variables and methods

Self Parameter

Self parameter is a reference to the current instance of the class. It allows us to access the attributes and methods of the object.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def bark(self):
        print(f"{self.name} is barking!")
```

3. *Methods – Things an Object Can Do*

Methods are like actions. If an object is a mobile, the method can be "call" or "click photo".

Syntax:

```
class ClassName:
    def method_name(self):
        print("This is a method")
```

Full Example:

```
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    def show(self):
        print(f"{self.name} is in grade {self.grade}.")
s1 = Student("Aditya", 12)
s1.show()
```

Concept	Simple Meaning	Python Keyword	Syntax Example
Class	Design/Plan	class	class Car:
Object	Real thing made from class	(no keyword)	mycar = Car("BMW")
Method	Action it can do	def	def drive(self):

Example:

```
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade
```

```
    def show_info(self):
        print(f"Name: {self.name}")
        print(f"Grade: {self.grade}")
s1 = Student("Aditya", 12)
s1.show_info()
```

Part	What it does
class Student:	Creates a class called Student
__init___	Constructor – sets name and grade
show_info	Method – prints details of the student
s1 = Student(...)	Creates an object
s1.show_info()	Calls the method on the object

What is Inheritance?

Inheritance lets a **child class** use the **properties and methods** of a **parent class** — like a son inheriting money or habits from a father.

```
class Parent:
    def speak(self):
        print("Speaking...")


class Child(Parent):  # Inheriting from Parent
    def walk(self):
        print("Walking...")
c = Child()
c.speak()  # From Parent
c.walk()   # From Child
```

1. Single Inheritance

One child inherits from one parent.

```
class Parent:
    def show(self):
        print("I am the Parent")


class Child(Parent):
    def display(self):
        print("I am the Child")


c = Child()
c.show()     # From Parent
c.display()  # From Child
```

Multiple Inheritance

One child inherits from **more than one parent**.

```
class Father:
    def skill1(self):
        print("Knows driving")
class Mother:
    def skill2(self):
        print("Knows cooking")
class Child(Father, Mother):
    pass
```

```
c = Child()
c.skill1()
c.skill2()
```

3. Multilevel Inheritance

Grandchild inherits from child who inherits from parent.

```
class Grandparent:
    def house(self):
        print("Has a big house")

class Parent(Grandparent):
    def car(self):
        print("Has a car")

class Child(Parent):
    def bike(self):
        print("Has a bike")

c = Child()
c.house()
c.car()
c.bike()
```

4. Hierarchical Inheritance

One parent, multiple children.

```
class Parent:
```

```
    def speak(self):
        print("Parent speaks")


class Child1(Parent):
    def play(self):
        print("Child1 plays")


class Child2(Parent):
    def dance(self):
        print("Child2 dances")


c1 = Child1()
c2 = Child2()
c1.speak()
c2.speak()
```

5. Hybrid Inheritance

Combination of more than one type. Python handles this using **MRO (Method Resolution Order)**.

```
class A:
    def show(self):
        print("A class")


class B(A):
    pass
```

```
class C(A):
    def show(self):
        print("C class")


class D(B, C):  # Inherits from B and C (Hybrid)
    pass


d = D()
d.show()
```

Type	Structure	Example Classes
Single	A → B	Parent → Child
Multiple	A, B → C	Father, Mother → Child
Multilevel	A → B → C	Grandparent → Parent → Child
Hierarchical	A → B, C	Teacher → Student1, Student2
Hybrid	Combo of above types	A → B, C → D

What is Encapsulation?

Encapsulation is the process of **hiding internal data** of a class and only allowing access through **methods**.

Type	Modifier	Access Level	Example
Public	No underscore	Accessible from anywhere	self.name
Protected	_single	Accessible in class & subclasses	self._name
Private	__double	Accessible only in class	self.__name

1. Public (Default)

```
class Student:
    def __init__(self):
        self.name = "Aditya"  # public
```

```
s = Student()
print(s.name)  # ✅ accessible
```

2. Protected (Single underscore _)

```
class Student:
    def __init__(self):
        self._grade = "A"  # protected
s = Student()
print(s._grade)  #  technically accessible but not recommended
```

3. Private (Double underscore __)

```
class Student:
    def __init__(self):
        self.__marks = 95  # private
    def get_marks(self):
        return self.__marks
s = Student()
print(s.get_marks())# ✅ Good
print(s.__marks)     # ❌ Error
```

Example: Employee Class with All Types

```
class Employee:
    def __init__(self, name, salary):
        self.name = name           # Public
        self._department = "IT"     # Protected
        self.__salary = salary      # Private
```

```
    def show_details(self):
        print("Name:", self.name)
        print("Department:", self._department)
        print("Salary:", self.__salary)  # Accessing private within class

    def get_salary(self):
        return self.__salary

emp = Employee("Aditya", 50000)
# Public - directly accessible
print(emp.name)  # 
# Protected - accessible but not recommended
print(emp._department)  # ⚠ Allowed but avoid direct use
# Private - cannot access directly
# print(emp.__salary)  ✖ Will give error
# Correct way to access private data
print(emp.get_salary())  #  Using method
emp.show_details()
```

What is Polymorphism?

Polymorphism means "**one name, many forms**."

In Python OOP, it allows **different classes to have methods with the same name** but **different behaviors**.

Feature	**Function Overloading**	**Function Overriding**
What it is	Same function name, different parameters	Redefining a parent method in a child class
Use case	Varying inputs for same logic	Changing inherited behavior

Supported in Python	+ Not directly (use *args, default args)	✅ Fully supported
Happens in	Same class	In parent-child class

Function Overriding in Python

Child class **overrides** a method of the parent class

```
class Parent:
    def greet(self):
        print("Hello from Parent")
class Child(Parent):
    def greet(self):  # Overrides parent method
        print("Hello from Child")
c = Child()
c.greet()  # Output: Hello from Child
```

What is Abstraction?

Abstraction means **hiding complex details** and **showing only the necessary features** to the user.

```
from abc import ABC, abstractmethod
class Shape(ABC):  # Abstract Class
    @abstractmethod
    def area(self):
        pass
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):  # Must override
        return 3.14 * self.radius * self.radius
```

```
c = Circle(5)

print(c.area())  # Output: 78.5
```

What is Exception Handling?

Exception Handling means **dealing with errors** in a clean way **without breaking the program**.

Simple Example:

Without handling:

```
a = 5

b = 0

print(a / b) # + Error: ZeroDivisionError
```

With handling:

```
try:
    print(a / b)
except ZeroDivisionError:
    print("Cannot divide by zero")
```

SYNTAX

```
try:
    # Code that might cause error
except ErrorType:
    # What to do if error happens
else:
    # If no error happens
finally:
    # Always runs (used for cleanup)
```

Block	Purpose
try	Code that might crash
except	Handle error gracefully
else	Runs only if no error occurred
finally	Always runs (e.g., closing file/db)

Common Exceptions:

Exception	When it happens
ZeroDivisionError	Divide by 0
ValueError	Invalid input type (int("abc"))
TypeError	Wrong data type in operation
FileNotFoundError	File doesn't exist

Example 1: Handling ZeroDivisionError

```
try:
    a = int(input("Enter a: "))
    b = int(input("Enter b: "))
    print("Result:", a / b)
except ZeroDivisionError:
    print("+ Can't divide by zero!")
```

Example 2: Multiple Exceptions

```
try:
    x = int(input("Enter a number: "))
    print(10 / x)
except ZeroDivisionError:
    print("Cannot divide by zero.")
except ValueError:
```

```
    print("Please enter a valid number.")
```

Example 3: else and finally

```
try:
    print("No error here")
except:
    print("Error!")
else:
    print("Else: Only if try is successful")
finally:
    print("Finally: Always runs")
```

```
def divide_numbers():
    try:
        num1 = int(input("Enter the first number: "))
        num2 = int(input("Enter the second number: "))
        result = num1 / num2
    except ZeroDivisionError:
        print("+ Cannot divide by zero!")
    except ValueError:
        print("+ Please enter valid numbers only!")
    else:
        print(" Result is:", result)
    finally:
        print("This block runs no matter what.")
```

```
# Call the function
divide_numbers()
```

Sample Output 1 (Valid Input):

```
Enter the first number: 10
Enter the second number: 2
Result is: 5.0
This block runs no matter what.
```

Sample Output 2 (Divide by Zero):

```
Enter the first number: 10
Enter the second number: 0
Cannot divide by zero!
This block runs no matter what.
```

Sample Output 3 (Invalid Input):

```
Enter the first number: ten
Please enter valid numbers only!
This block runs no matter what.
```

What is File Handling?

File handling allows you to **read from or write to files** (like .txt, .csv) using Python code.

Basic Steps:

1. **Open** a file
2. **Read** or **Write**
3. **Close** the file

Syntax:

```
file = open("filename.txt", "mode")
# do something
file.close()
```

Mode	Meaning	Use Case
"r"	Read only	File must exist
"w"	Write (overwrite if exists)	Creates new or replaces
"a"	Append to file	Adds at the end
"r+"	Read + Write	File must exist

Example 1: Reading from a File

```
file = open("myfile.txt", "r")
data = file.read()
print(data)
file.close()
```

Example 2: Writing to a File

```
file = open("myfile.txt", "w")
file.write("Hello students!\nWelcome to Python file handling.")
file.close()
```

Example 3: Using with Block (Auto-close)

```
with open("myfile.txt", "r") as file:
    print(file.read())
```

Method	What it does
read()	Reads entire file
readline()	Reads one line
readlines()	Reads all lines as list

SUMMURY:

Action	Code Example

Write	open("file.txt", "w")
Read	open("file.txt", "r")
Append	open("file.txt", "a")
Best Style	with open(...) as file:

Code Writing Questions

- Write a class Person with a constructor that takes name and age. Add a method greet() to print a greeting using the person's name.
- Create a class Calculator with methods: add, subtract, multiply, and divide that take two numbers.
- Write a program that defines a class Mobile with attributes: brand, price, and a method show_details().
- Create a class Employee. When an object is created, print a welcome message using a constructor. When deleted, print a goodbye message using a destructor.
- Write a program for **single inheritance** with class Parent and Child.
- Create an example of **multiple inheritance** using two parent classes.
- Write a program to handle divide by zero using try-except.
- Handle both ValueError and ZeroDivisionError in one program.
- Write a program to create and write to a file data.txt.
- Read a file and print its content.