

Theory Questions:- Python OOPs

Q1. What is Object-Oriented Programming(OOP)?

Definition

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of “**objects**”, which can contain **data** (attributes) and **code** (methods).

It focuses on organizing software into reusable, modular pieces instead of writing code in a linear (procedural) way.

Key Concepts of OOP

1. **Class** → Blueprint/template for creating objects.
 2. **Object** → Instance of a class (real-world entity).
 3. **Encapsulation** → Binding data and methods together in a single unit.
 4. **Abstraction** → Hiding unnecessary details and showing only the essential features.
 5. **Inheritance** → One class can acquire properties and methods of another class.
 6. **Polymorphism** → Ability to use the same interface with different forms (method overriding & overloading).
-

Example in Python

```
# Class (Blueprint)
class Car:
    def __init__(self, brand, model):
        self.brand = brand    # Attribute
        self.model = model

    def display_info(self):    # Method
        return f"Car: {self.brand} {self.model}"

# Object (Instance of Class)
car1 = Car("Tesla", "Model S")
car2 = Car("BMW", "X5")

print(car1.display_info())    # Output Car: Tesla Model S
print(car2.display_info())    # Output Car: BMW X5
```

Q2. What is a Class in OOP?

A **class** in Object-Oriented Programming (OOP) is a **blueprint or template** that defines how to create objects.

It describes the **attributes (data/properties)** and **methods (functions/behaviors)** that the objects of that class will have.

Key Points about Classes:

1. Blueprint Concept

- Think of a class as a **design/plan**.
- Example: An architect draws a blueprint of a house → but the actual house is built later.

2. Objects are Instances

- Objects are **real, usable things** created from the class.
- Example: If `Car` is a class (blueprint), then `Tesla` or `BMW` are actual objects (real cars).

3. Encapsulation of Data + Behavior

- Classes **combine attributes (variables)** and **methods (functions)** into one unit.
- Example: A `Car` has attributes like `brand`, `model` and methods like `start()`, `brake()`.

4. Reusability

- Once a class is defined, you can create **multiple objects** without rewriting code.

- Example: From one Car class, you can create car1, car2, car3 easily.

Example in Python:

```
# Defining a class
class Car:
    # Constructor (__init__) to initialize attributes
    def __init__(self, brand, model):
        self.brand = brand    # attribute (variable)
        self.model = model    # attribute (variable)

    # Method (function inside class)
    def display_info(self):
        return f"{self.brand} {self.model}"

# Creating objects (instances of the class)
car1 = Car("Tesla", "Model S")
car2 = Car("BMW", "X5")

# Using object methods
print(car1.display_info()) # Output: Tesla Model S
print(car2.display_info()) # Output: BMW X5
```

Q3. What is an Object in OOP?

An **object** in Object-Oriented Programming (OOP) is a **real-world entity** or an **instance of a class**. It represents something that has **state (data/attributes)** and **behavior (methods/functions)**.

Key Points about Objects:

1. Instance of a Class

- A class is like a **blueprint**, and an object is the **actual product** created from that blueprint.
- Example: If Car is a class, then Tesla Model S or BMW X5 are objects.

2. Has State and Behavior

- **State (attributes/variables)**: Describes the properties of the object.
 - e.g., color = red, speed = 120.
- **Behavior (methods/functions)**: Describes what the object can do.
 - e.g., drive(), brake().

3. Multiple Objects from One Class

- You can create **many objects** from the same class, each with its own data.
- Example: car1, car2, car3 all come from the Car class but can have different brands and models.

Example in Python:

```
# Class definition
class Car:
    def __init__(self, brand, color):
        self.brand = brand    # attribute
        self.color = color    # attribute

    def drive(self):
        return f"{self.brand} car is driving!"

# Creating objects (instances of Car)
car1 = Car("Tesla", "Red")
car2 = Car("BMW", "Black")

# Accessing attributes and methods
print(car1.brand)    # Output: Tesla
print(car2.color)    # Output: Black
print(car1.drive())  # Output: Tesla car is driving!
```

Q4. What is the Difference between Abstraction and Encapsulation?

Both **Abstraction** and **Encapsulation** are key concepts of Object-Oriented Programming (OOP).

They sound similar but serve **different purposes**.

1. Abstraction

- **Definition:**
Abstraction is the process of **hiding implementation details** and **showing only the essential features** of an object.
- **Focus:** *What an object does* (not how it does it).
- **Achieved by:**
 - Abstract classes
 - Interfaces
- **Example in Real Life:**
 - When you drive a car, you only know about the **steering wheel, accelerator, brakes** (essential features).
 - You don't need to know the complex implementation of the engine or gear mechanism.

2. Encapsulation

- **Definition:**
Encapsulation is the process of **wrapping data (attributes) and methods (functions)** into a single unit (class).
It also **restricts direct access** to some data using **access modifiers** (`private`, `protected`, `public`).
- **Focus:** *How the data is hidden/protected*.
- **Achieved by:**
 - Using classes
 - Access modifiers (`_protected`, `__private`) in Python
- **Example in Real Life:**
 - A **capsule medicine** wraps several ingredients inside one capsule shell.
 - You don't see or touch the ingredients directly.

Tabular Difference:

Feature	Abstraction	Encapsulation
Definition	Hides implementation details, shows only essential features.	Bundles data and methods together, hides data from direct access.
Focus	<i>What</i> an object does.	<i>How</i> the data is hidden/protected.
Achieved using	Abstract classes, Interfaces.	Classes, Access Modifiers.
Real-Life Example	Car Driver only sees pedals & steering (not engine details).	Capsule wraps medicine inside.

Example in Python

```
from abc import ABC, abstractmethod

# Abstraction Example
class Vehicle(ABC):
    @abstractmethod
    def start(self):
        pass

class Car(Vehicle):
    def start(self):
        return "Car engine starts with a key!"

# Encapsulation Example
class Account:
    def __init__(self, balance):
        self.__balance = balance    # private variable

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

    def get_balance(self):
        return self.__balance
```

```
# Using Abstraction
my_car = Car()
print(my_car.start()) # Output: Car engine starts with a key!

# Using Encapsulation
acc = Account(1000)
acc.deposit(500)
print(acc.get_balance()) # Output: 1500
```

Q5. What are Dunder Methods in Python?

Definition:

- **Dunder methods** (short for “*Double UNDERSCORE*”) are special methods in Python.
- They always start and end with **double underscores** like `__init__`, `__str__`, `__len__`.
- Also called **magic methods** because they add *extra functionality* to classes and objects.

Purpose of Dunder Methods:

- To make **classes behave like built-in types**.
- To **customize operators** (like `+`, `-`, `<`, etc.) for user-defined classes.
- To define **object creation, representation, comparison, arithmetic, etc.**

Commonly Used Dunder Methods:

Dunder Method	Purpose
<code>__init__</code>	Constructor → Initializes an object.
<code>__str__</code>	Returns a string representation of the object (used with <code>print()</code>).
<code>__repr__</code>	Official representation of the object (useful for debugging).
<code>__len__</code>	Defines behavior for <code>len(obj)</code> .
<code>__add__</code>	Defines behavior for <code>+</code> operator.
<code>__eq__</code>	Defines behavior for <code>==</code> operator.
<code>__getitem__</code>	Defines behavior for indexing <code>obj[i]</code> .
<code>__call__</code>	Makes an object callable like a function.

Example in Python

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __str__(self): # For print()
        return f"Book: {self.title}, Pages: {self.pages}"

    def __len__(self): # For len()
        return self.pages

    def __add__(self, other): # For +
        return self.pages + other.pages

# Creating objects
book1 = Book("Python Basics", 250)
book2 = Book("Advanced Python", 350)

print(book1) # Uses __str__ → Book: Python Basics, Pages: 250
print(len(book1)) # Uses __len__ → 250
print(book1 + book2) # Uses __add__ → 600
```

Q6. Explain the Concept of Inheritance in OOP

Definition:

- **Inheritance** is an Object-Oriented Programming (OOP) concept where a **class (child/derived class)** can **reuse the properties and methods** of another class (parent/base class).
 - It allows **code reusability** and represents an **"is-a" relationship** between classes.
-

Key Points:

1. The **child class** inherits data members (variables) and methods (functions) from the **parent class**.
 2. The child class can:
 - **Use** parent class features as they are.
 - **Override** parent methods to change behavior.
 - **Add new** methods or attributes of its own.
 3. Helps in writing **clean, reusable, and maintainable code**.
-

Types of Inheritance in Python:

1. **Single Inheritance** → One child inherits from one parent.
 2. **Multiple Inheritance** → A child inherits from more than one parent.
 3. **Multilevel Inheritance** → A child inherits from a parent, which in turn inherits from another class.
 4. **Hierarchical Inheritance** → Multiple child classes inherit from the same parent.
 5. **Hybrid Inheritance** → Combination of multiple inheritance types.
-

Example in Python

```
# Parent Class
class Animal:
    def speak(self):
        return "This animal makes a sound."

# Child Class
class Dog(Animal):
    def speak(self): # Overriding parent method
        return "Woof! Woof!"

# Another Child Class
class Cat(Animal):
    def speak(self):
        return "Meow!"

# Using the classes
dog = Dog()
cat = Cat()

print(dog.speak()) # Output: Woof! Woof!
print(cat.speak()) # Output: Meow!
```

Q7. What is Polymorphism in OOP?

Definition:

- The word **Polymorphism** comes from Greek:
 - *Poly* = many
 - *Morph* = forms
 - In OOP, **Polymorphism** means the ability of **different classes to respond to the same method name in different ways**.
 - It allows a single function, operator, or method to have **multiple implementations**.
-

Key Idea:

- **One interface, multiple implementations.**
 - The exact behavior depends on the **object** that is calling the method.
-

Types of Polymorphism in Python:

1. **Compile-time Polymorphism (Overloading)**

- Having multiple methods with the same name but different arguments.
- *Note:* Python doesn't support traditional overloading, but we can achieve similar behavior using *default arguments* or **args*.

2. Run-time Polymorphism (Overriding)

- When a child class provides its **own implementation** of a method already defined in the parent class.

Example 1: Method Overriding (Run-time Polymorphism)

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

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

# Same method name → different outputs
animals = [Dog(), Cat(), Animal()]
for a in animals:
    print(a.speak())

# Output
Woof!
Meow!
Some sound
```

Example 2: Polymorphism with Built-in Functions

```
print(len("Ritesh"))    # len() works with string → 6
print(len([1, 2, 3, 4])) # len() works with list → 4
```

Q8. How is Encapsulation achieved in Python?

Definition Recap:

- **Encapsulation** = wrapping up **data (variables)** and **methods (functions)** into a single unit (class).
- It also helps in **restricting direct access** to data by using *access modifiers*.

Ways to Achieve Encapsulation in Python:

1. Using Classes

- Class binds data (attributes) and methods together.
- Example: A Car class containing both speed and drive().

2. Access Modifiers in Python

Python does not have strict private/protected keywords like Java or C++, but it achieves encapsulation through **naming conventions**:

- **Public** (default): Accessible everywhere.

```
class Car:
    def __init__(self):
        self.color = "Red" # Public attribute

car = Car()
print(car.color) # Accessible
```

- **Protected** (`_var`): Should not be accessed directly outside class (convention).

```
class Car:
    def __init__(self):
        self._speed = 120 # Protected
car = Car()
print(car._speed) # Possible, but discouraged
```

- **Private (__var)**: Name mangling makes it harder to access from outside.

```
class Car:
    def __init__(self):
        self.__engine_number = "ENG123" # Private
car = Car()
# print(car.__engine_number) # Error
print(car._Car__engine_number) # Access through name mangling
```

3. Getter and Setter Methods

- Used to safely access or modify private variables.

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

    # Getter
    def get_balance(self):
        return self.__balance

    # Setter
    def set_balance(self, amount):
        if amount >= 0:
            self.__balance = amount
        else:
            print("Invalid amount!")

acc = BankAccount(500)
print(acc.get_balance()) # 500
acc.set_balance(1000) # updates balance
print(acc.get_balance()) # 1000
```

Q9. What is a Constructor in Python?

Definition:

- A **constructor** in Python is a **special method** that is automatically called when an object of a class is created.
- Its main job is to **initialize the attributes (data members)** of the object.

In Python:

- The constructor method is always written as:

```
def __init__(self, arguments):
    # initialization code
```

Note :- `__init__` is a dunder method (double underscore)

`self` parameter refers to the current instance of the class

```
### Example 1: Simple Constructor
```python
class Student:
 def __init__(self, name, age): # constructor
 self.name = name
 self.age = age

Creating objects
s1 = Student("Ritesh", 21)
s2 = Student("Aditi", 20)
```

```
print(s1.name, s1.age) # Output: Ritesh 21
print(s2.name, s2.age) # Output: Aditi 20
```

## Example 2: Default Constructor

If no constructor is defined, Python provides a default constructor automatically.

e.g:

```
class Demo:
 pass

obj = Demo() # Default constructor is called
```

## Example 3: Constructor with Default Arguments

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

c1 = Car("BMW")
c2 = Car() # Uses default value

print(c1.brand) # BMW
print(c2.brand) # Toyota
```

### Key Points:

1. `__init__` method = constructor in Python.
2. It initializes object attributes at creation time.
3. Python automatically calls it when you create an object.
4. We can use default arguments to make constructors flexible.

## Q10. What are Class and Static Methods in Python?

---

### 1. Class Methods

- A **class method** is a method that is bound to the **class itself**, not the object.
- It can access and modify **class-level attributes** (shared across all objects).
- Defined using the `@classmethod` decorator.
- The first parameter is always `cls` (refers to the class).

#### Example:

```
class Student:
 school = "ABC Public School" # Class attribute

 def __init__(self, name, age):
 self.name = name
 self.age = age

 @classmethod
 def change_school(cls, new_school):
 cls.school = new_school # modifies class-level data

Using class method
Student.change_school("XYZ International School")
print(Student.school) # Output: XYZ International School
```

### 2. Static Methods

- A **static method** does not depend on the class (`cls`) or instance (`self`).
- Used when we want a utility/helper function inside a class.
- Defined using the `@staticmethod` decorator.

#### Example:



```

class MathOperations:
 @staticmethod
 def add(x, y):
 return x + y

 @staticmethod
 def multiply(x, y):
 return x * y

Using static methods
print(MathOperations.add(5, 10)) # Output: 15
print(MathOperations.multiply(3, 4)) # Output: 12

```

## Key Differences Between Class & Static Methods

Feature	Class Method ( @classmethod )	Static Method ( @staticmethod )
First Argument	cls (class reference)	No default first argument
Access cls ?	Yes	No
Access self ?	No	No
Scope	Works with class-level data	Independent utility/helper
Use Case	Modify/read class attributes	General-purpose methods

## Real-Life Analogy

- **Class Method:** Like a **school notice board** — ek baar badal diya, to poore school (sab students) ke liye effect hoga.
- **Static Method:** Like a **calculator kept in the school** — jo sab use kar sakte hain, lekin wo school ya students ke data pe depend nahi karta.

## Q11. What is Method Overloading in Python?

### Definition

- **Method Overloading** means having **multiple methods with the same name but different parameters** (like in Java/C++).
- Python **does not support true method overloading** because:
  - The **latest defined method** with the same name will overwrite the previous ones.

### How Python Handles It

- Instead of true overloading, Python uses:
  1. **Default arguments**
  2. **Variable-length arguments ( \*args , \*\*kwargs )**

This way, a single method can handle different numbers of parameters.

### Example1: Default Arguments

```

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

calc = Calculator()
print(calc.add(2)) # 2
print(calc.add(2, 3)) # 5
print(calc.add(2, 3, 4)) # 9

```

### Example2: Variable-Length Arguments

```

class Calculator:
 def add(self, *args):
 return sum(args)

calc = Calculator()
print(calc.add(5)) # 5

```

```
print(calc.add(2, 3, 4)) # 9
print(calc.add(10, 20, 30, 40)) # 100
```

## Q12. What is Method Overriding in OOP?

### Definition

- **Method Overriding** occurs when a **child class provides a specific implementation** of a method that is already defined in its **parent class**.
- The method in the child class must have:
  - The **same name**
  - The **same number of parameters** as the method in the parent class.

---

### Key Points

1. Method overriding is used to **change or extend the behavior** of a parent class method.
2. It supports **Runtime Polymorphism** (decision happens at runtime).
3. The parent class method is replaced (or extended) when called using a child class object.

---

### Example1: Basic Overriding

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

class Dog(Animal):
 def sound(self): # Overriding the parent method
 return "Bark"

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

Test
dog = Dog()
cat = Cat()
print(dog.sound()) # Bark
print(cat.sound()) # Meow
```

### Example2: Using super() in Overriding

```
class Vehicle:
 def info(self):
 return "This is a vehicle."

class Car(Vehicle):
 def info(self):
 # Extending parent method using super()
 return super().info() + " Specifically, it is a car."

Test
car = Car()
print(car.info()) # This is a vehicle. Specifically, it is a car.
```

## Q13. What is a Property Decorator in Python?

### Definition

- The **@property decorator** in Python is used to define methods in a class that can be accessed like **attributes** (without parentheses).
- It allows you to:
  1. **Encapsulate data** (hide internal representation).
  2. Provide **getter, setter, and deleter** functionality in a clean and Pythonic way.

- This makes your class more **readable** and **maintainable**.

## Why Use @property?

- Sometimes you want to protect access to a variable and perform extra logic when **getting** or **setting** it.
- Instead of calling explicit methods (`get_name()`, `set_name()`), you can use `@property` to access them as if they were **simple attributes**.

## Example: Using @property

```
class Student:
 def __init__(self, name):
 self._name = name # private variable convention

 @property
 def name(self):
 return self._name # getter

 @name.setter
 def name(self, value):
 if not value.strip(): # validation
 raise ValueError("Name cannot be empty!")
 self._name = value # setter

 @name.deleter
 def name(self):
 print("Deleting name...")
 del self._name

Test
s = Student("Ritesh")
print(s.name) # Access like attribute → Ritesh

s.name = "Aditi" # Calls setter internally
print(s.name) # Aditi

del s.name # Calls deleter
```

## Key Points

1. `@property` → creates a getter method.
2. `@name.setter` → creates a setter method.
3. `@name.deleter` → creates a deleter method.
4. You access it like a normal attribute, but behind the scenes, methods are called.

## Q14. Why is Polymorphism Important in OOP?

### Definition

- **Polymorphism** means "**many forms**".
- In OOP, polymorphism allows the **same interface (method/function)** to work with **different types of objects**.
- Example: A method `draw()` can be defined in multiple classes (`Circle`, `Square`, `Triangle`) but each class provides its **own implementation**.

### Importance of Polymorphism

1. **Code Reusability**
  - Same function name can be used for different data types or classes.
  - Reduces duplication in code.
2. **Flexibility & Maintainability**
  - Makes code more **extensible**.
  - New classes can be added without changing the existing code structure.
3. **Readability**

- Clearer and cleaner code because method names stay consistent.

#### 4. Supports Dynamic Behavior

- Python resolves which method to call at **runtime** (dynamic polymorphism).
- This enables writing **generic and reusable code**.

### Example: Polymorphism in Action

```
class Dog:
 def sound(self):
 return "Woof!"

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

class Cow:
 def sound(self):
 return "Moo!"

Polymorphism: Same method name `sound()` behaves differently
animals = [Dog(), Cat(), Cow()]

for animal in animals:
 print(animal.sound())

Output
Woof!
Meow!
Moo!
```

## Q15. What is an Abstract Class in Python?

### Definition

- An **abstract class** is a class that **cannot be instantiated directly**.
- It serves as a **blueprint for other classes**.
- Abstract classes can contain:
  - **Abstract methods** → Methods declared but not implemented.
  - **Concrete methods** → Normal methods with implementation.

In Python, abstract classes are created using the **abc (Abstract Base Class)** module.

### Why Use Abstract Classes?

1. To **define a common interface** for all subclasses.
2. To **enforce implementation** of certain methods in subclasses.
3. To achieve **abstraction** in OOP (hiding implementation details).

### Example: Abstract Class in Python

```
from abc import ABC, abstractmethod

Abstract Class
class Vehicle(ABC):

 @abstractmethod
 def start_engine(self):
 pass # abstract method (must be implemented by subclass)

 def fuel_type(self):
 return "Petrol or Diesel" # concrete method

Subclass 1
class Car(Vehicle):
```

```
def start_engine(self):
 return "Car engine started with key"

Subclass 2
class Bike(Vehicle):
 def start_engine(self):
 return "Bike engine started with self-start"

Objects
car = Car()
bike = Bike()

print(car.start_engine()) # Car engine started with key
print(bike.start_engine()) # Bike engine started with self-start
```

## Q16. What are the Advantages of Object-Oriented Programming (OOP)?

Object-Oriented Programming (OOP) provides a structured way of writing code by organizing it around **objects** and **classes**. It offers several advantages over traditional procedural programming.

---

### Advantages of OOP

#### 1. Modularity (Code Reusability)

- Code is divided into classes and objects, making it **modular and reusable**.
- Once a class is created, it can be reused in different programs.

#### 2. Encapsulation (Data Hiding)

- Sensitive data can be hidden inside a class and accessed only through defined methods.
- This improves **security** and prevents misuse.

#### 3. Inheritance

- Classes can **reuse and extend** functionality from other classes.
- Promotes **code reusability** and avoids duplication.

#### 4. Polymorphism

- The same function or method can work in **different ways** depending on the object.
- Makes the code more **flexible and maintainable**.

#### 5. Abstraction

- Only essential details are shown, while complex implementation is hidden.
- Helps in reducing complexity.

#### 6. Maintainability

- OOP code is easier to **update, modify, and maintain** because each class is independent.

#### 7. Scalability

- Large projects can be managed efficiently as OOP makes it easier to divide work into **modules (classes)**.
- 

### Example (Real-Life Analogy)

- Think of a **Car** class:
    - Properties: `color`, `model`, `engine_type`
    - Methods: `start()`, `stop()`
  - You can create multiple car objects (e.g., Honda, BMW) without rewriting the code.
  - If you need a **SportsCar**, you can inherit from Car and just add extra features.
- 

### Summary

- OOP makes programs **modular, reusable, secure, and easier to maintain**.
- Key benefits: **Encapsulation, Inheritance, Polymorphism, Abstraction**.
- Widely used in real-world applications like **banking systems, game development, and enterprise software**.

## Q17. What is the difference between a Class Variable and an Instance Variable?

In Python (and OOP in general), variables inside a class can be categorized into **class variables** and **instance variables**. They differ in terms of **scope, behavior, and how they are shared**.

## 1. Class Variable

- Defined **inside a class**, but **outside any instance methods**.
- Shared by **all objects (instances)** of the class.
- Changing a class variable affects **all objects** unless specifically overridden by an instance.

## 2. Instance Variable

- Defined **inside a constructor** (`__init__`) or methods using `self`.
- Each object has its **own copy** of instance variables.
- Changing an instance variable affects **only that specific object**.

## Example in Python

```
class Student:
 # Class Variable (shared by all instances)
 school_name = "ABC Public School"

 def __init__(self, name, grade):
 # Instance Variables (unique to each object)
 self.name = name
 self.grade = grade

Creating two objects
s1 = Student("Ritesh", "10th")
s2 = Student("Aditi", "12th")

Accessing variables
print(s1.name, s1.grade, s1.school_name) # Ritesh 10th ABC Public School
print(s2.name, s2.grade, s2.school_name) # Aditi 12th ABC Public School

Changing the class variable
Student.school_name = "XYZ International School"
print(s1.school_name) # XYZ International School
print(s2.school_name) # XYZ International School

Changing instance variable
s1.grade = "11th"
print(s1.grade) # 11th (only for s1)
print(s2.grade) # 12th (unchanged for s2)
```

## Key Differences between Class Variable and Instance Variable

Feature	Class Variable	Instance Variable
<b>Where defined</b>	Inside class, but <b>outside methods</b>	Inside constructor ( <code>__init__</code> ) or methods using <code>self</code>
<b>Shared by</b>	<b>All objects</b> of the class	<b>Only one object</b> (unique per instance)
<b>Change effect</b>	Change affects <b>all instances</b>	Change affects <b>only that particular instance</b>
<b>Accessed by</b>	<code>ClassName.variable</code> or <code>self.variable</code>	Always through <code>self.variable</code>
<b>Memory usage</b>	Stored once in memory (efficient)	Each object keeps its own copy

## Q18. What is multiple inheritance in Python?

### Definition

Multiple inheritance in Python is a feature of object-oriented programming where a single child class can inherit properties and methods from **two or more parent classes simultaneously**.

It allows the child class to combine the functionality of multiple classes, promoting **code reusability, flexibility, and better modeling of real-world scenarios**.

### Syntax

```
class Parent1:
 pass
```

```
class Parent2:
 pass

class Child(Parent1, Parent2): # Child inherits from both Parent1 and Parent2
 pass
```

### Example:-

```
class Father:
 def skill(self):
 print("Gardening")

class Mother:
 def skill(self):
 print("Cooking")

class Child(Father, Mother): # Multiple Inheritance
 def skill(self):
 print("Child also knows Coding")

Object creation
c = Child()
c.skill() # Child also knows Coding
```

## 19. Explain the purpose of “**str**” and ‘**repr**’ methods in Python

### \_\_str\_\_ and \_\_repr\_\_ Methods in Python

In Python, both `__str__` and `__repr__` are **special methods** (also called dunder methods – “double underscore”) that control how objects are represented as strings. They are very useful when printing or debugging objects.

#### 1. `__str__` Method

- **Purpose:** Returns a *human-readable* string representation of the object.
- It is called when you use the `print()` function or `str()` on an object.
- Should be **informal and easy to read** for end-users.

#### Example:

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

 def __str__(self):
 return f"Student Name: {self.name}, Marks: {self.marks}"

s = Student("Ritesh", 95)
print(s) # Calls __str__
 # Output Student Name: Ritesh, Marks: 95
```

#### 2. `__repr__` Method

- **Purpose:** Returns a *developer-friendly* string representation of the object.
- Mainly used for **debugging and logging**.
- The goal is to be **unambiguous**, often showing the code to recreate the object.
- If `__str__` is not defined, `print()` will fallback to `__repr__`.

#### Example:

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

```

self.marks = marks

def __repr__(self):
 return f"Student('{self.name}', {self.marks})"

s = Student("Aditi", 33)
print(repr(s)) # Calls __repr__
 # Output Student('Aditi', 33)

```

### 3. Key Difference Between `__str__` and `__repr__`

Feature	<code>__str__</code> (User-Friendly)	<code>__repr__</code> (Developer-Friendly)
Audience	End-users / casual display	Developers / debugging
Output Style	Readable, nicely formatted	Precise, unambiguous, recreatable
Called By	<code>print(obj)</code> or <code>str(obj)</code>	<code>repr(obj)</code> or interactive shell prompt
Fallback	Falls back to <code>__repr__</code> if not defined	No fallback (always exists by default)

## 20. What is the significance of the 'super()' function in Python?

The `super()` function is a built-in function in Python that allows us to call methods from a **parent (superclass)** inside a **child (subclass)**. It is mainly used in **inheritance** to avoid rewriting code and to ensure proper method resolution when multiple classes are involved.

### 1. Why Use `super()` ?

- **Code Reusability:** Prevents duplicate code by reusing parent class methods.
- **Maintainability:** If the parent class changes, child classes automatically inherit updated behavior.
- **Supports Multiple Inheritance:** Works with Python's **MRO (Method Resolution Order)** to correctly decide which class method to call.
- **Cleaner Syntax:** Avoids hardcoding the parent class name, making the code more flexible.

### 2. Example: Without `super()`

```

class Parent:
 def __init__(self, name):
 self.name = name

class Child(Parent):
 def __init__(self, name, age):
 Parent.__init__(self, name) # explicitly calling Parent
 self.age = age

c = Child("Ritesh", 21)
print(c.name, c.age) # Output Ritesh 21

```

### 3. Example: With `super()`

```

class Parent:
 def __init__(self, name):
 self.name = name

class Child(Parent):
 def __init__(self, name, age):
 super().__init__(name) # super() automatically finds Parent
 self.age = age

c = Child("Aditi", 20)
print(c.name, c.age) # Output Aditi 20

```

### 4. Example: Multiple Inheritance

```

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

class B(A):
 def show(self):

```



```

 super().show()
 print("Class B")

class C(B):
 def show(self):
 super().show()
 print("Class C")

obj = C()
obj.show()

Output
Class A
Class B
Class C

```

## 21. What is the significance of the **del** method in Python?

In Python, the `__del__` method is a **destructor method**.

It is called **automatically** when an object is about to be destroyed (i.e., when it goes out of scope or its reference count drops to zero).

### 1. Purpose of `__del__`

- Used to **free resources** (like closing files, releasing network connections, or cleaning up memory).
- Called by Python's **garbage collector** before the object is removed from memory.
- Helps in performing **final clean-up tasks**.

### 2. Example: Basic Use of `__del__`

```

class Demo:
 def __init__(self, name):
 self.name = name
 print(f"Object {self.name} created.")

 def __del__(self):
 print(f"Destructor called, object {self.name} deleted.")

obj = Demo("Ritesh")
del obj # explicitly deleting object

Output
Object Ritesh created.
Destructor called, object Ritesh deleted.

```

### 3. Example: Automatic Call by Garbage Collector

```

class FileHandler:
 def __init__(self, filename):
 self.file = open(filename, "w")
 print("File opened.")

 def __del__(self):
 self.file.close()
 print("File closed automatically.")

handler = FileHandler("test.txt")
No explicit del required, __del__ will be called when program ends or object is destroyed

Output
File opened.
File closed automatically.

```

### 4. Important Points About `__del__`

- Not guaranteed to run immediately after an object goes out of scope; Python decides when garbage collection happens.

- If objects have circular references, **del** may not be called reliably.
- Overusing **del** can cause issues; prefer using context managers (with statement) for resource management.

## 22. What is the difference between @staticmethod and @classmethod in Python?

In Python, both @staticmethod and @classmethod are **decorators** used to define methods inside a class that are **not regular instance methods**.

They look similar, but they serve different purposes.

### 1. @staticmethod

- A **static method** does not take `self` (object reference) or `cls` (class reference) as the first argument.
- It behaves like a normal function but belongs to the class's namespace.
- Can be called using either the class name or an instance.
- Cannot access or modify **class-level data** or **instance attributes** directly.

Example:

```
class MathOperations:
 @staticmethod
 def add(x, y):
 return x + y

print(MathOperations.add(5, 3)) # Called using class
 # Output 8

obj = MathOperations()
print(obj.add(10, 7)) # Called using object
 # Output 17
```

### 2. @classmethod

- A **class method** takes `cls` (class reference) as the first argument.
- It can access and modify **class-level variables**, but not instance variables directly.
- Useful when we want to create **factory methods** (methods that return class objects).

Example:

```
class Student:
 count = 0 # Class variable

 def __init__(self, name):
 self.name = name
 Student.count += 1

 @classmethod
 def get_count(cls):
 return f"Total Students: {cls.count}"

s1 = Student("Ritesh")
s2 = Student("Aditi")

print(Student.get_count()) # Output Total Students: 2
```

### 3. Key Differences Between @staticmethod and @classmethod

Feature	@staticmethod	@classmethod
First Parameter	No <code>self</code> or <code>cls</code>	Takes <code>cls</code> as the first argument
Access to Class Data	Cannot access or modify class/instance variables	Can access and modify class-level data
Usage	Utility functions related to the class	Factory methods or operations on class-level data
Call	Can be called via class or object	Can be called via class or object

## 23. How does polymorphism work in Python with inheritance?

**Polymorphism** means "**many forms**".

In Python, it allows the same method or operation to behave differently depending on the object that calls it.

When combined with **inheritance**, polymorphism lets child classes provide their **own implementation** of methods that are already defined in the parent class.

## 1. How Polymorphism Works with Inheritance

- A **parent class** defines a method.
- **Child classes** override (redefine) that method with their own behavior.
- When we call the method on different objects, Python automatically decides which version to execute (based on the object type).

## 2. Example: Basic Polymorphism with Inheritance

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

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

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

Polymorphism in action
animals = [Dog(), Cat(), Animal()]

for a in animals:
 print(a.sound())

Output
Bark
Meow
Some generic animal sound
```

## 3. Example: Using Polymorphism in Functions

```
class Bird(Animal):
 def sound(self):
 return "Chirp"

def make_sound(animal):
 print(animal.sound())

make_sound(Dog()) # Bark
make_sound(Cat()) # Meow
make_sound(Bird()) # Chirp
```

## 4. Key Advantages of Polymorphism

- **Flexibility:** Same interface works with different types of objects.
- **Reusability:** Functions or methods can handle objects of different classes.
- **Maintainability:** Easier to extend programs by adding new subclasses.

# 24. What is method chaining in Python OOP?

**Method chaining** is a technique in object-oriented programming where **multiple methods are called sequentially on the same object in a single line**.

This is achieved by designing methods to **return self**, allowing the next method to be called directly on the same instance.

## 1. Why Use Method Chaining?

- **Concise Code:** Combine multiple operations into a single statement.

- **Fluent Interface:** Improves readability and expresses a flow of operations naturally.
  - **Convenient:** Reduces the need to repeatedly reference the object.
- 

## 2. Example: Simple Method Chaining

```
class Person:
 def __init__(self, name):
 self.name = name
 self.age = 0

 def set_age(self, age):
 self.age = age
 return self # Return self to allow chaining

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

Using method chaining
p = Person("Ritesh")
p.set_age(21).greet() # Output Hello, my name is Ritesh and I am 21 years old.
```

## 3. Example: Chaining Multiple Methods

```
class Car:
 def __init__(self, brand):
 self.brand = brand
 self.speed = 0

 def accelerate(self, value):
 self.speed += value
 return self

 def brake(self, value):
 self.speed -= value
 return self

 def display_speed(self):
 print(f"{self.brand} is moving at {self.speed} km/h")
 return self

Chaining multiple methods
my_car = Car("BMW")
my_car.accelerate(50).brake(10).display_speed() # Output BMW is moving at 40 km/h
```

## 4. Key Points About Method Chaining

- Methods must return self to enable chaining.
- Improves code readability and fluency.
- Widely used in libraries like Pandas, SQLAlchemy, and Matplotlib.

# 25. What is the purpose of the **call** method in Python?

In Python, the `__call__` method is a **special (dunder) method** that allows an instance of a class to be **called like a regular function**. This means that after defining `__call__`, you can use the object with parentheses `()` as if it were a function.

---

### 1. Why Use `__call__`?

- **Function-like behavior:** Treat objects as callable functions.
  - **Encapsulation:** Combine data and behavior in an object that can still be invoked easily.
  - **Flexibility:** Useful in designing **functors, decorators, or APIs** where objects need to be callable.
- 

### 2. Example: Basic Use of `__call__`

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

 def __call__(self, y):
 return self.x + y

add_five = Adder(5)
print(add_five(10)) # Calling the object like a function
Output 15
```

### 3. Example: **call** for Logging

```
class Logger:
 def __init__(self, prefix):
 self.prefix = prefix

 def __call__(self, message):
 print(f"{self.prefix}: {message}")

log = Logger("INFO")
log("This is a log message") # Output INFO: This is a log message
```

### 4. Key Points About `__call__`

- Makes objects callable like functions.
- Can accept any number of arguments ( `*args` and `**kwargs` ).
- Useful in functional programming patterns, decorators, and APIs.
- Enhances flexibility and readability when objects encapsulate behavior.

## Practical Questions:-

- ✓ 1. Create a parent class `Animal` with a method `speak()` that prints a generic message. Create a child class `Dog` that overrides the `speak()` method to print "Bark!".

```
Parent class
class Animal:
 def speak(self):
 print("This is a generic animal sound")

Child class
class Dog(Animal):
 def speak(self):
 print("Bark!")

animal = Animal()
animal.speak()

dog = Dog()
dog.speak()
```

↗ This is a generic animal sound  
Bark!

- ✓ 2. Write a program to create an abstract class `Shape` with a method `area()`. Derive classes `Circle` and `Rectangle` from it and implement the `area()` method in both.

```
from abc import ABC, abstractmethod

Abstract class
class Shape(ABC):
```

```

@abstractmethod
def area(self):
 pass

Circle class
class Circle(Shape):
 def __init__(self, radius):
 self.radius = radius

 def area(self):
 return 3.14 * self.radius * self.radius

Rectangle class
class Rectangle(Shape):
 def __init__(self, length, width):
 self.length = length
 self.width = width

 def area(self):
 return self.length * self.width

c = Circle(5)
r = Rectangle(4, 6)

print("Circle area:", c.area())
print("Rectangle area:", r.area())

```

```

↗ Circle area: 78.5
Rectangle area: 24

```

3. Implement a multi-level inheritance scenario where a class Vehicle has an attribute type. Derive a class Car and further derive a class ElectricCar that adds a battery attribute

```

Parent class
class Vehicle:
 def __init__(self, type):
 self.type = type

Child class
class Car(Vehicle):
 def __init__(self, type, model):
 super().__init__(type)
 self.model = model

Grandchild class
class ElectricCar(Car):
 def __init__(self, type, model, battery):
 super().__init__(type, model)
 self.battery = battery

ecar = ElectricCar("Car", "Tesla Model 3", "75 kWh")
print("Type:", ecar.type)
print("Model:", ecar.model)
print("Battery:", ecar.battery)

```

```

↗ Type: Car
Model: Tesla Model 3
Battery: 75 kWh

```

4. Demonstrate polymorphism by creating a base class Bird with a method fly(). Create two derived classes Sparrow and Penguin that override the fly() method.

```

Base class
class Bird:
 def fly(self):
 print("Some birds can fly")

Derived class Sparrow

```

```

class Sparrow(Bird):
 def fly(self):
 print("Sparrow can fly high")

Derived class Penguin
class Penguin(Bird):
 def fly(self):
 print("Penguin cannot fly")

birds = [Sparrow(), Penguin()]

for bird in birds:
 bird.fly()

```

```

→ Sparrow can fly high
 Penguin cannot fly

```

## 5. Write a program to demonstrate encapsulation by creating a class BankAccount

- ✓ with private attributes balance and methods to deposit, withdraw, and check balance.

```

Encapsulation example
class BankAccount:
 def __init__(self, balance):
 self.__balance = balance # private attribute

 def deposit(self, amount):
 self.__balance += amount
 print(f"Deposited: {amount}")

 def withdraw(self, amount):
 if amount <= self.__balance:
 self.__balance -= amount
 print(f"Withdrawn: {amount}")
 else:
 print("Insufficient balance")

 def check_balance(self):
 print(f"Balance: {self.__balance}")

account = BankAccount(1000)
account.deposit(500)
account.withdraw(300)
account.check_balance()

```

```

→ Deposited: 500
 Withdrawn: 300
 Balance: 1200

```

## 6. Demonstrate runtime polymorphism using a method play() in a base class

- ✓ Instrument. Derive classes Guitar and Piano that implement their own version of play().

```

Base class
class Instrument:
 def play(self):
 print("Playing instrument")

Derived class Guitar
class Guitar(Instrument):
 def play(self):
 print("Playing guitar")

Derived class Piano
class Piano(Instrument):
 def play(self):
 print("Playing piano")

Testing runtime polymorphism
instruments = [Guitar(), Piano()]

```

```
for instrument in instruments:
 instrument.play()
```

↗ Playing guitar  
Playing piano

7. Create a class MathOperations with a class method add\_numbers() to add two numbers and a static method subtract\_numbers() to subtract two numbers.

```
Class demonstrating classmethod and staticmethod
class MathOperations:

 @classmethod
 def add_numbers(cls, a, b):
 return a + b

 @staticmethod
 def subtract_numbers(a, b):
 return a - b

print("Addition:", MathOperations.add_numbers(10, 5))
print("Subtraction:", MathOperations.subtract_numbers(10, 5))
```

↗ Addition: 15  
Subtraction: 5

8. Implement a class Person with a class method to count the total number of persons created.

```
Class demonstrating counting instances using class method
class Person:
 count = 0 # Class variable to track number of persons

 def __init__(self, name):
 self.name = name
 Person.count += 1

 @classmethod
 def total_persons(cls):
 return cls.count

p1 = Person("Ritesh")
p2 = Person("Aditi")
p3 = Person("Anshu")

print("Total persons created:", Person.total_persons())
```

↗ Total persons created: 3

9. Write a class Fraction with attributes numerator and denominator. Override the str method to display the fraction as "numerator/denominator".

```
Class demonstrating __str__ method
class Fraction:
 def __init__(self, numerator, denominator):
 self.numerator = numerator
 self.denominator = denominator

 def __str__(self):
 return f"{self.numerator}/{self.denominator}"

f = Fraction(3, 4)
print(f)
```



 3/4

- ✓ 10. Demonstrate operator overloading by creating a class Vector and overriding the add method to add two vectors


```
Class demonstrating operator overloading
class Vector:
 def __init__(self, x, y):
 self.x = x
 self.y = y

 # Overloading the + operator
 def __add__(self, other):
 return Vector(self.x + other.x, self.y + other.y)

 def __str__(self):
 return f"({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1 + v2

print("Vector addition:", v3)
```


 Vector addition: (6, 8)

- ✓ 11. Create a class Person with attributes name and age. Add a method greet() that prints "Hello, my name is {name} and I am {age} years old."

```
Simple class with instance attributes and method
class Person:
 def __init__(self, name, age):
 self.name = name
 self.age = age

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

Testing
p = Person("Ritesh", 21)
p1 = Person("Aditi", 20)
p.greet()
p1.greet()
```


 Hello, my name is Ritesh and I am 21 years old.  
Hello, my name is Aditi and I am 20 years old.

- ✓ 12. Implement a class Student with attributes name and grades. Create a method average\_grade() to compute the average of the grades.

```
Class demonstrating method to calculate average
class Student:
 def __init__(self, name, grades):
 self.name = name
 self.grades = grades # List of grades

 def average_grade(self):
 return sum(self.grades) / len(self.grades) if self.grades else 0

s = Student("Ritesh", [85, 90, 78, 92])
print("Average grade:", s.average_grade())
```

 Average grade: 86.25

- ✓ 13. Create a class Rectangle with methods set\_dimensions() to set the dimensions and area() to calculate the area.

```
Rectangle class with set_dimensions and area methods
class Rectangle:
 def set_dimensions(self, length, width):
 self.length = length
 self.width = width

 def area(self):
 return self.length * self.width

Testing
rect = Rectangle()
rect.set_dimensions(5, 3)
print("Area of rectangle:", rect.area())
```

```
↗ Area of rectangle: 15
```

- ✓ 14. Create a class Employee with a method calculate\_salary() that computes the salary based on hours worked and hourly rate. Create a derived class Manager that adds a bonus to the salary

```
Base class
class Employee:
 def __init__(self, name, hours_worked, hourly_rate):
 self.name = name
 self.hours_worked = hours_worked
 self.hourly_rate = hourly_rate

 def calculate_salary(self):
 return self.hours_worked * self.hourly_rate

Derived class
class Manager(Employee):
 def __init__(self, name, hours_worked, hourly_rate, bonus):
 super().__init__(name, hours_worked, hourly_rate)
 self.bonus = bonus

 def calculate_salary(self):
 return super().calculate_salary() + self.bonus

Testing
emp = Employee("Ritesh", 40, 20)
mgr = Manager("Aditi", 40, 20, 500)

print("Employee salary:", emp.calculate_salary())
print("Manager salary:", mgr.calculate_salary())
```

```
↗ Employee salary: 800
 Manager salary: 1300
```

- ✓ 15. Create a class Product with attributes name, price, and quantity. Implement a method total\_price() that calculates the total price of the product.

```
Product class with total_price method
class Product:
 def __init__(self, name, price, quantity):
 self.name = name
 self.price = price
 self.quantity = quantity

 def total_price(self):
 return self.price * self.quantity

Testing
p = Product("Laptop", 500, 3)
```

```
print("Total price:", p.total_price())
```

```
↩ Total price: 1500
```

16. Create a class Animal with an abstract method sound(). Create two derived classes Cow and Sheep that implement the sound() method.

```
from abc import ABC, abstractmethod
```

```
Abstract class
class Animal(ABC):
 @abstractmethod
 def sound(self):
 pass
```

```
Derived class Cow
class Cow(Animal):
 def sound(self):
 print("Moo")
```

```
Derived class Sheep
class Sheep(Animal):
 def sound(self):
 print("Baa")
```

```
cow = Cow()
sheep = Sheep()
```

```
cow.sound()
sheep.sound()
```

```
↩ Moo
 Baa
```

17. Create a class Book with attributes title, author, and year\_published. Add a method get\_book\_info() that returns a formatted string with the book's details.

```
Book class with get_book_info method
class Book:
 def __init__(self, title, author, year_published):
 self.title = title
 self.author = author
 self.year_published = year_published

 def get_book_info(self):
 return f"'{self.title}' by {self.author}, published in {self.year_published}"
```

```
b = Book("1984", "George Orwell", 1949)
print(b.get_book_info())
```

```
↩ '1984' by George Orwell, published in 1949
```

18. Create a class House with attributes address and price. Create a derived class Mansion that adds an attribute number\_of\_rooms.

```
Parent class
class House:
 def __init__(self, address, price):
 self.address = address
 self.price = price

Derived class
class Mansion(House):
 def __init__(self, address, price, number_of_rooms):
 super().__init__(address, price)
 self.number_of_rooms = number_of_rooms
```

```
Testing
m = Mansion("123 Luxury St", 500000, 8)
print("Address:", m.address)
print("Price:", m.price)
print("Number of rooms:", m.number_of_rooms)
```

```
↗ Address: 123 Luxury St
Price: 500000
Number of rooms: 8
```