PYTHON OOPS

1. What is Object-Oriented Programming (OOP)

📌 **Answer:**

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of **“objects”**, which are instances of **classes**. These objects can contain **data**, in the form of **attributes** (also known as properties or fields), and **code**, in the

---

2. What is a class in OOP

📌 **Answer:**

A **class** in Object-Oriented Programming (OOP) is a **blueprint** or **template** for creating objects. It defines a set of **attributes (variables)** and **methods (functions)** that the created objects (instances) will have.

✅ **Key Points:**

- A class defines **what** an object will be.
- An object is an **instance** of a class.
- Classes support **encapsulation**, **inheritance**, and **polymorphism**.

🧠 **Example in Python:**

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

    def start_engine(self):
        print(f"{self.brand} engine started!")
```

Here, Car is a class. You can create objects (instances) from it:

```
my_car = Car("Toyota", "Red")
my_car.start_engine()  # Output: Toyota engine started!
```

🧩 **Summary:**

A class is like a **template**, and an object is the **real-world version** of that template.

---

3. What is an object in OOP

📌 **Answer:**

An **object** in Object-Oriented Programming (OOP) is an **instance of a class**. It is a self-contained component that contains both **data (attributes)** and **behaviors (methods)** that define its functionality.

When a class is defined, no memory is allocated until an object is created from it. Each object can have different values for its attributes while sharing the same methods.

✅ **Example:**

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

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

# Creating objects of the class Dog
dog1 = Dog("Buddy", "Golden Retriever")
dog2 = Dog("Charlie", "Beagle")

# Accessing attributes and methods
print(dog1.name)      # Output: Buddy
dog1.bark()           # Output: Buddy is barking!
```

In the above example:

- Dog is the class.
- dog1 and dog2 are **objects** (instances) of the Dog class.
- Each object has its own name and breed values.

---

4. What is the difference between abstraction and encapsulation

📌 **Answer:**

Abstraction and **Encapsulation** are two fundamental concepts in Object-Oriented Programming (OOP), but they serve different purposes:

🔷 **Abstraction:**

- **Definition**: Abstraction is the process of **hiding complex implementation details** and showing only the **essential features** of an object.
- **Purpose**: Focuses on **what an object does**, not **how** it does it.
- **How**: Achieved using **abstract classes** or **interfaces**.
- **Real-life Example**: A car's steering wheel allows you to steer, but you don’t need to know the mechanics inside.

✅ **Code Example:**

```
from abc import ABC, abstractmethod

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

class Car(Vehicle):
    def start_engine(self):
        print("Car engine started.")
```

```
c = Car()
c.start_engine()  # Output: Car engine started.
```

🔷 Encapsulation:

- **Definition**: Encapsulation is the practice of **wrapping data and methods** that operate on the data into a **single unit** (class), and **restricting direct access** to some of the object's components.
- **Purpose**: Focuses on **data hiding** and **protecting the object's state**.
- **How**: Achieved using **private/protected variables** and **getter/setter methods**.
- **Real-life Example**: You cannot directly access the engine of a car while driving.

✅ Code Example:

```
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(500)
print(acc.get_balance())  # Output: 500
```

🔁 Key Differences:

Feature	Abstraction	Encapsulation
Focus	Hiding **implementation complexity**	Hiding **data**
Achieved By	Abstract classes, Interfaces	Access modifiers (private, protected)
Purpose	Show only relevant features	Protect data from unauthorized access
Example	Driving a car without knowing engine logic	Can't access account balance directly

✅ In summary:

- **Abstraction** hides **complexity**.
- **Encapsulation** hides **data**.

5. What are dunder methods in Python

📌 **Answer:**

Dunder methods in Python (short for **“double underscore” methods**) are **special predefined methods** that start and end with **double underscores** (e.g., __init__, __str__, __len__, etc.). They are also known as **magic methods** or **special methods**.

🔷 Why are dunder methods used?

They are used to:

- Customize the behavior of **built-in operations**
- Implement **operator overloading**
- Define **object lifecycle** (like creation and deletion)
- Provide **string representations** and more

🔷 Common Examples of Dunder Methods:

Dunder Method	Purpose
__init__(self)	Constructor: Initializes a new object
__str__(self)	Returns a human-readable string for print()
__repr__(self)	Returns a developer-friendly string
__len__(self)	Returns the length using len(obj)
__add__(self, other)	Defines behavior for + operator
__eq__(self, other)	Defines behavior for ==
__getitem__(self, key)	Allows indexing like obj[key]
__iter__(self)	Returns an iterator for loops
__next__(self)	Returns the next value in iteration

✅ Example:

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

    def __str__(self):
        return f"Book: {self.title}"

    def __len__(self):
        return self.pages

book = Book("Ramayan", 350)
print(book)          # Output: Book: Ramayan
print(len(book))     # Output: 350
```

🔄 **Summary:**

- **Dunder = Double UNDERSCORE**, like __init__
- They let you **interact with Python's built-in syntax**
- Used for **custom behavior** and **operator overloading**

🧠 Pro Tip: You don't **call** dunder methods directly. Instead, Python calls them **automatically** in most cases.

---

6. Explain the concept of inheritance in OOP

📌 **Answer:**

Inheritance is a core concept in Object-Oriented Programming (OOP) that allows a class (called a **child class** or **subclass**) to **inherit attributes and methods** from another class (called a **parent class** or **superclass**).

🔷 **Purpose of Inheritance:**

- **Code Reusability**: Common code is written once in the parent class.
- **Extensibility**: Child classes can **extend or override** the behavior of the parent.
- **Hierarchy Representation**: Models real-world relationships (e.g., *Dog* is a *Animal*).

🔷 **Basic Syntax:**

```
class Parent:
    def greet(self):
        print("Hello from Parent")

class Child(Parent):
    def welcome(self):
        print("Welcome from Child")

obj = Child()
obj.greet()    # Inherited from Parent
obj.welcome()  # Defined in Child
```

🔷 **Types of Inheritance in Python:**

Type	Description
Single Inheritance	One child class inherits from one parent class
Multiple Inheritance	One class inherits from multiple parent classes
Multilevel Inheritance	A class inherits from a class that is already a child of another
Hierarchical Inheritance	Multiple classes inherit from the same parent class
Hybrid Inheritance	Combination of multiple types of inheritance

---

🔷 **Example:**

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

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

d = Dog()
d.speak()  # From Animal
d.bark()   # From Dog
```

🔁 **Summary:**

- **Inheritance** = Reusing code by deriving a new class from an existing one.
- Promotes **DRY (Don't Repeat Yourself)** principle.
- Supports **polymorphism** and **encapsulation** by allowing structured class hierarchies.

---

7. What is polymorphism in OOP

📌 **Answer:**

Polymorphism in Object-Oriented Programming (OOP) means **"many forms."** It allows objects of **different classes** to be treated as **instances of the same class through a common interface**, typically using methods with the **same name but different behavior** based on the object type.

🔷 **Key Idea:**

Polymorphism allows **methods or functions to behave differently** depending on the object that is invoking them.

🔷 **Types of Polymorphism in Python:**

Type	Description
Compile-time (Static)	Achieved via method overloading (not directly supported in Python)
Run-time (Dynamic)	Achieved via method overriding, supported in Python

🔷 **Example 1: Polymorphism with Methods**

```
class Dog:
    def sound(self):
        print("Barks")

class Cat:
    def sound(self):
        print("Meows")

# Polymorphic function
def make_sound(animal):
    animal.sound()

# Different behaviors
d = Dog()
c = Cat()
make_sound(d)  # Barks
make_sound(c)  # Meows
```

🔷 Example 2: Polymorphism with Inheritance

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

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

class Cat(Animal):
    def speak(self):
        print("Cat meows")

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

Output:

```
Dog barks
Cat meows
```

🔁 Summary:

- **Polymorphism** allows functions/methods to work with different types of objects using the same interface.
- Enhances **flexibility** and **code readability** in large programs.
- Python supports **dynamic polymorphism** naturally through method overriding.

---

8. **How is encapsulation achieved in Python**

📌 Answer:

Encapsulation in Python is the **OOP principle** of **restricting direct access** to an object's internal data and methods. It **bundles** the data (attributes) and the methods (functions) that operate on the data into a **single unit — a class**, and controls access using **access specifiers**.

🔷 Why Encapsulation?

- Protects data from unauthorized access and modification.
- Helps in **data hiding**.
- Makes the code more **modular**, **secure**, and **manageable**.

🔷 Access Specifiers in Python:

Modifier	**Syntax Prefix**	**Meaning**
Public	No prefix	Accessible from anywhere
Protected	_variable	Suggests limited access (within class and subclasses)
Private	__variable	Name mangling is applied; hard to access directly

🔷 Example of Encapsulation:

```
class Student:
    def __init__(self, name, marks):
        self.name = name          # public
        self._marks = marks       # protected
        self.__grade = None       # private

    def set_grade(self):
        if self._marks >= 90:
            self.__grade = 'A'
        elif self._marks >= 75:
            self.__grade = 'B'
        else:
            self.__grade = 'C'

    def get_grade(self):
        return self.__grade

# Creating object
s = Student("Yash", 88)
s.set_grade()

print(s.name)        # ✅ Public - Accessible
print(s._marks)      # ⚠️ Protected - Accessible, but not recommended
print(s.get_grade()) # ✅ Accessing private via public method

# print(s.__grade)   # ❌ Error: 'Student' object has no attribute '__grade'
```

🔁 Summary:

- **Encapsulation** = Data + Functions in one unit (class).
- Use **getter/setter methods** to access private data.
- Achieved in Python using **naming conventions** like _ and __.
- Promotes **security**, **modularity**, and **data hiding**.

---

9. What is a constructor in Python

📌 **Answer:**

A **constructor** in Python is a **special method** used to **initialize** a newly created object of a class. It is automatically called **when an object is created**.

🔷 Constructor in Python = __init__() method

- Defined using the special **dunder method** __init__(self)
- Used to assign default or initial values to object attributes.
- Called **once per object** when the object is instantiated.

🔷 Syntax:

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

🔷 Example:

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

    def show(self):
        print(f"Name: {self.name}, Roll: {self.roll}")

# Creating objects (constructor gets called automatically)
s1 = Student("Yash", 101)
s2 = Student("Riya", 102)

s1.show()
s2.show()
```

✅ Output:

```
Name: Yash, Roll: 101
Name: Riya, Roll: 102
```

🔁 **Summary:**

Feature	Description
Method Name	__init__()
Called When?	Automatically during object creation
Purpose	Initialize object attributes
Self Parameter	Refers to the current object itself

📌 **Note:**

Python does **not** support **constructor overloading** (multiple __init__ methods). You can use **default arguments** instead.

---

10. What are class and static methods in Python

📌 **Answer:**

In Python, **class methods** and **static methods** are two special types of methods that are **not the same as regular instance methods**. They are used for different purposes and are defined using decorators.

🔷 **1. Class Method (@classmethod)**

✅ **Key Points:**

- Works with **class itself**, not instances.
- The first parameter is **cls** (refers to the class).
- Can **access or modify class variables** shared among all instances.
- Defined using the @classmethod decorator.

✅ **Syntax:**

```
class MyClass:
    class_var = 0

    @classmethod
    def show_class_var(cls):
        print(cls.class_var)
```

✅ **Example:**

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

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

Student.change_school("XYZ School")
print(Student.school)  # Output: XYZ School
```

🔷 2. Static Method (@staticmethod)

✅ **Key Points:**

- Does **not take self or cls** as the first parameter.
- Cannot access or modify class/instance variables.
- Used for utility/helper functions related to the class.
- Defined using the @staticmethod decorator.

✅ **Syntax:**

```
class MyClass:
    @staticmethod
    def utility():
        print("I do not need class or object.")
```

✅ **Example:**

```
class Calculator:
    @staticmethod
    def add(a, b):
        return a + b

print(Calculator.add(5, 3))  # Output: 8
```

🔁 Comparison Table:

Feature	Class Method	Static Method
Decorator	@classmethod	@staticmethod
First Arg	cls (class reference)	No default argument
Access to Class?	✅ Yes	❌ No
Access to Object?	❌ No	❌ No
Use Case	Modify class state	Utility functions

11. What is method overloading in Python

📌 **Answer:**

✅ **What is Method Overloading in Python?**

Method Overloading is the ability to define **multiple methods with the same name** but with **different parameters (number or type)**.

🔶 In many programming languages like Java or C++, method overloading is supported **natively**.

🔶 But in **Python**, **method overloading is ***not directly supported*** because Python allows **dynamic typing** and flexible function arguments.

🔷 Python's Way of Achieving Method Overloading

In Python, if you define multiple methods with the same name, **only the last one is kept** — previous ones are overridden.

🧪 **Example — Overriding Happens:**

```
class Greet:
    def hello(self):
        print("Hello!")

    def hello(self, name):
        print(f"Hello, {name}!")

obj = Greet()
obj.hello("Yash")  # Output: Hello, Yash!
```

⚠️ hello(self) is **overwritten** by hello(self, name).

🔧 **Workaround: Use Default Arguments or *args**

✅ **Example 1: Using Default Arguments**

```
class Greet:
    def hello(self, name=None):
        if name:
            print(f"Hello, {name}!")
        else:
            print("Hello!")

g = Greet()
g.hello()         # Output: Hello!
g.hello("Yash")    # Output: Hello, Yash!
```

✅ **Example 2: Using *args for Flexible Parameters**

```
class Multiply:
    def product(self, *args):
        result = 1
        for num in args:
            result *= num
        return result

m = Multiply()
print(m.product(2, 3))         # Output: 6
print(m.product(2, 3, 4))       # Output: 24
```

🔁 **Summary Table:**

Feature	**Python Support**
Native Method Overloading	❌ Not supported

Overloading by Redefining	❌ Not allowed
Default Arguments	✅ Yes (workaround)
***args / **kwargs**	✅ Yes (flexible)

12. What is method overriding in OOP

📌 **Answer:**

✅ **What is Method Overriding in OOP?**

Method Overriding occurs when a **subclass provides its own implementation** of a method that is already defined in its **parent class**.

It allows a child class to **customize or completely replace** the behavior of a method inherited from the parent.

🔑 **Key Points:**

- The **method name** must be the **same**.
- The **number and type of parameters** must match.
- It supports **runtime polymorphism**.
- Enables more specific behavior for subclass objects.

🔁 **Example:**

```
class Animal:
    def speak(self):
        print("The animal makes a sound")

class Dog(Animal):
    def speak(self):  # Method overriding
        print("The dog barks")

class Cat(Animal):
    def speak(self):  # Method overriding
        print("The cat meows")

# Testing
a = Animal()
a.speak()    # Output: The animal makes a sound

d = Dog()
d.speak()    # Output: The dog barks

c = Cat()
c.speak()    # Output: The cat meows
```

🧠 Why Use Method Overriding?

Purpose	Benefit
Custom behavior in subclasses	Makes code more flexible and reusable
Runtime polymorphism	Choose method based on object at runtime
Cleaner code structure	Avoids repetition and hardcoding

🧪 Use Case in Real Life:

For example, in a GUI framework:

- A base Widget class has a method draw()
- Button, TextBox, Slider subclasses override draw() to display differently.

---

13. What is a property decorator in Python

📌 Answer:

✅ What is a @property decorator in Python?

The @property decorator in Python is used to **define getter methods** in a class. It allows **accessing methods like attributes**, improving code readability and encapsulation.

🔧 Purpose:

- Encapsulate instance variables (private attributes)
- Provide **controlled access** to them
- Avoid direct attribute modification while using attribute-like syntax

📘 Syntax Example:

```
class Person:
    def __init__(self, name):
        self._name = name  # Conventionally private

    @property
    def name(self):
        print("Getting name...")
        return self._name

    @name.setter
    def name(self, value):
        print("Setting name...")
        if len(value) > 0:
            self._name = value
        else:
            raise ValueError("Name cannot be empty")
    @name.deleter
    def name(self):
        print("Deleting name...")
        del self._name
```

🧪 **Usage:**

```
p = Person("Yash")
print(p.name)     # Access like an attribute (calls getter)
p.name = "Arjun"  # Calls setter
del p.name        # Calls deleter
```

🔐 **Benefits:**

Feature	Description
Readability	Access methods like attributes (e.g., obj.name)
Encapsulation	Controls how attributes are accessed or changed
Validation	Add logic while getting/setting data

💡 **Real-World Example:**

Useful when you want to **expose class attributes safely**, such as calculated values or attributes with validation logic — e.g., employee.salary, student.percentage.

---

14. Why is polymorphism important in OOP

📌 **Answer:**

✅ **Why is Polymorphism Important in Object-Oriented Programming (OOP)?**

Polymorphism means *"many forms."* It allows **objects of different classes to be treated through a common interface**, enabling flexibility and reusability in code.

🔑 **Importance of Polymorphism:**

Benefit	Description
✅ **Code Reusability**	Allows writing generic functions or methods that work with different types
✅ **Flexibility & Extensibility**	You can add new classes with minimal changes to existing code
✅ **Simplified Code**	Reduces complexity by using the same interface for different data types
✅ **Dynamic Behavior**	Decision about which method to call is made at runtime
✅ **Improves Maintainability**	Easier to manage and update large systems

📘 Example (Method Overriding):

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

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

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

def make_animal_speak(animal):
    print(animal.speak())

make_animal_speak(Dog())  # Output: Bark
make_animal_speak(Cat())  # Output: Meow
```

🔍 Here, make_animal_speak() works for any subclass of Animal. That's polymorphism in action.

🔄 Types of Polymorphism:

Type	Description
Compile-Time	Method Overloading (limited in Python)
Run-Time	Method Overriding (most common in Python)

🧠 Real-World Analogy:

Think of a **remote control** (interface) that works for different brands of TVs (objects). The remote performs the same function (like power on), but the behavior may vary internally depending on the brand.

15. What is an abstract class in Python

📌 Answer:

🔶 What is an Abstract Class in Python?

An **abstract class** in Python is a class that **cannot be instantiated directly** and is meant to be **inherited by subclasses**. It is used to **define a common interface** for all its subclasses.
It **may contain abstract methods**—methods that **do not have any implementation** in the base class and must be implemented in the child class.

✅ Key Points:

Feature	Description
Cannot be instantiated	You **cannot create objects** of an abstract class directly.
Defines interface	Provides a **template** or contract for child classes.
Uses abc module	Abstract classes use the abc (Abstract Base Class) module.
Abstract method	Defined using @abstractmethod decorator.
Subclass must override	All abstract methods **must** be implemented in the subclass.

🧪 Syntax Example:

```
from abc import ABC, abstractmethod

class Animal(ABC):  # Inherit from ABC (Abstract Base Class)

    @abstractmethod
    def make_sound(self):  # Abstract method
        pass

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

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

# animal = Animal() ❌ This will raise an error
dog = Dog()
print(dog.make_sound())  # ✅ Output: Bark
```

💡 Why Use Abstract Classes?

Reason	Benefit
Enforce method implementation	Forces child classes to define specific methods
Improves code organization	Keeps a clear structure in large projects
Enables polymorphism	Common interface for different implementations
Encourages clean design	Promotes a blueprint-like development approach

---

📌 Use Case:

If you have a set of related classes with a **common method signature but different logic**, use an abstract class to define the structure and enforce rules.

16. What are the advantages of OOP

📌 **Answer:**

🔶 What are the Advantages of Object-Oriented Programming (OOP)?

Object-Oriented Programming (OOP) is a programming paradigm that is **centered around objects and classes**. It offers several advantages that make code **modular, reusable, and easier to maintain**.

✅ 1. Modularity

- Code is divided into **independent classes and objects**.
- Each class handles a specific responsibility.
- 💡 *Helps in breaking down complex problems.*

✅ 2. Reusability (via Inheritance)

- You can **reuse existing classes** by extending them using **inheritance**.
- Promotes **code efficiency** and reduces duplication.

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

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

✅ 3. Encapsulation

- Hides internal object details; only exposes necessary data.
- Provides **security** and **control** over data access.

```
class Person:
    def __init__(self, name):
        self.__name = name  # Private variable
```

✅ 4. Abstraction

- Focuses on **what** an object does instead of **how** it does it.
- Simplifies complex systems by showing only essential details.

✅ 5. Polymorphism

- Same method name behaves differently depending on the object.
- Promotes **flexible and extendable** code.

```
class Bird:
    def sound(self):
        print("Chirp")

class Duck(Bird):
    def sound(self):
```

```
    print("Quack")
```

✅ 6. Easy Maintenance

- OOP makes debugging and updating code easier due to modularity.
- Changes in one class **rarely affect other parts** of the system.

✅ 7. Scalability

- OOP is well-suited for **large and complex applications**.
- Helps in collaborative development by dividing work into classes/modules.

✅ 8. Real-world Mapping

- Objects in code mimic **real-life entities**, making logic more intuitive.
- e.g., Car, BankAccount, Student, etc.

🧠 Summary Table:

OOP Feature	Advantage
Modularity	Breaks code into manageable sections
Reusability	Avoids code duplication
Encapsulation	Protects and controls data
Abstraction	Simplifies interface for users
Polymorphism	Allows flexibility with method usage
Inheritance	Promotes reuse and structure
Maintainability	Makes updates and debugging easier
Scalability	Ideal for big projects

---

17. What is the difference between a class variable and an instance variable

📌 **Answer:**

🔶 Difference Between Class Variable and Instance Variable in Python (OOP)

In Python, **variables defined inside a class** can be either:

- 🔶 **Class Variables** – shared among **all instances**
- 🔷 **Instance Variables** – unique to **each object/instance**

✅ 1. Definition

Type	Description
Class Variable	Belongs to the **class**. Shared by all objects.
Instance Variable	Belongs to the **instance/object**. Unique per object.

✅ 2. Declaration

- **Class Variable:** Declared **inside class**, **outside methods**.

- **Instance Variable:** Declared **inside __init__()** or other instance methods using self.

```
class Student:
    school = "DPS"       # Class variable

    def __init__(self, name):
        self.name = name   # Instance variable
```

✅ 3. Scope & Access

Feature	Class Variable	Instance Variable
Accessed by	ClassName.var or object.var	Only via object.var
Scope	Shared across all objects	Unique per object

✅ 4. Example

```
class Student:
    school = "DPS"  # Class variable

    def __init__(self, name):
        self.name = name  # Instance variable

s1 = Student("Yash")
s2 = Student("Aryan")

print(s1.name)     # Yash
print(s2.name)     # Aryan
print(s1.school)   # DPS
print(s2.school)   # DPS

Student.school = "KV"  # Change class variable

print(s1.school)   # KV (changed for all)
print(s2.school)   # KV
```

✅ 5. Key Differences Table

Feature	Class Variable	Instance Variable
Shared Across	All objects	Only one object
Defined In	Class body	__init__() or instance method
Accessed Using	ClassName.var or self.var	self.var
Memory Allocation	Once per class	Per object
Use Case	Common data for all objects	Unique data per object

18. What is multiple inheritance in Python

📌 **Answer:**

🔷 **What is Multiple Inheritance in Python?**

Multiple Inheritance is a feature in Python where a **class can inherit from more than one parent class**.

This allows the child class to access attributes and methods of **all parent classes**.

✅ **Syntax Example:**

```
class Father:
    def show_father(self):
        print("Father's traits")

class Mother:
    def show_mother(self):
        print("Mother's traits")

class Child(Father, Mother):  # Inherits from both
    def show_child(self):
        print("Child's traits")

c = Child()
c.show_father()
c.show_mother()
c.show_child()
```

🔶 **Output:**

```
Father's traits
Mother's traits
Child's traits
```

✅ **Key Points:**

Feature	Description
Inheritance Type	Multiple – inherits from more than one class
Python Support	Yes, unlike some other languages (e.g., Java)
Potential Issue	Method Resolution Order (MRO) confusion

✅ **Method Resolution Order (MRO):**

- Python uses **C3 Linearization (MRO)** to resolve the order of method calling.
- The method will be searched in **left-to-right** order in the parent classes.

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

class B:
    def show(self):
        print("B")

class C(A, B):  # A first, then B
    pass

c = C()
c.show()  # Output: A
```

✅ **Advantages:**

- Combines functionality from multiple classes
- Promotes code reusability

❗ **Disadvantages:**

- Can lead to **ambiguity** or **conflicts** (method with same name in both parents)
- Complex to manage if not handled carefully

📌 **Tip:**

Use super() wisely with MRO to avoid issues in complex multiple inheritance scenarios.

---

19. Explain the purpose of ‘’__str__’ and ‘__repr__’ ‘ methods in Python

📌 **Answer:**

🔷 __str__() vs __repr__() in Python

Both __str__() and __repr__() are **dunder methods** (double underscore) used to define how an **object is represented as a string**.

✅ __str__() – For End Users

- Called by: str(object) or print(object)
- Goal: Provide a **user-friendly** or readable string representation.
- Used for: Display to users.

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

    def __str__(self):
        return f"Book: {self.title}"

b = Book("Bhagavad Gita")
print(b)  # Output: Book: Bhagavad Gita
```

✅ __repr__() – For Developers

- Called by: repr(object)
- Goal: Provide an **unambiguous**, **developer-friendly** string.
- Used for: Debugging, logging, or recreating the object.
- If __str__() is not defined, __repr__() is used as a fallback.

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

    def __repr__(self):
        return f"Book('{self.title}')"

b = Book("Bhagavad Gita")
print(repr(b))  # Output: Book('Bhagavad Gita')
```

✅ Comparison Table:

Feature	**__str__()**	**__repr__()**
Purpose	Readable / user-facing	Precise / developer-facing
Called by	str(), print()	repr(), or object in REPL
Fallback	Falls back to __repr__() if absent	No fallback

✅ Best Practice:

Define **both** methods in your custom class:

```
def __str__(self):
    return "Readable info"

def __repr__(self):
    return "Developer info"
```

---

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

📌 **Answer:**

🔷 super() in Python – Significance & Use

The super() function is used **to call a method from the parent (or superclass)** inside a child (or subclass). It is especially useful in **inheritance**.

✅ Why is super() important?

1. **Access Parent Class Methods or Constructors**
2. **Avoid Code Duplication**
3. **Supports Multiple Inheritance**

4. **Keeps Code Maintainable & DRY (Don't Repeat Yourself)**

✅ Basic Example – Constructor Inheritance:

```
class Animal:
    def __init__(self, name):
        self.name = name
        print(f"Animal: {self.name}")

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Calls Animal's constructor
        self.breed = breed
        print(f"Dog Breed: {self.breed}")

d = Dog("Tommy", "Labrador")
```

🟩 Output:

```
Animal: Tommy
Dog Breed: Labrador
```

✅ When to Use super()

Situation	Why Use super()
Constructor Overriding	To call parent class's __init__()
Method Overriding	To enhance or extend parent functionality
Multiple Inheritance	To follow **MRO (Method Resolution Order)**

✅ Example – Method Overriding with super():

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

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

obj = B()
obj.show()
```

🟩 Output:

```
Class A
Class B
```

🧠 Bonus: super() and MRO

In multiple inheritance, super() follows **Python's MRO (Method Resolution Order)** to decide the order of method calls.

---

21. What is the significance of the __del__ method in Python

📌 Answer:

🔷 __del__ Method in Python – Significance & Use

The __del__ method in Python is a **special (dunder) method** called a **destructor**. It is **automatically invoked when an object is about to be destroyed**, i.e., when there are **no more references** to the object.

✅ Purpose of __del__:

1. **Cleanup** resources (e.g., closing files, releasing database connections).
2. Acts like a **finalizer** before the object is garbage collected.

✅ Syntax:

```
def __del__(self):
    # cleanup code here
```

✅ Simple Example:

```
class MyClass:
    def __init__(self):
        print("Object Created")

    def __del__(self):
        print("Object Destroyed")

obj = MyClass()
del obj  # Manually deleting the object
```

🟩 Output:

```
Object Created
Object Destroyed
```

⚠️ Important Notes:

- Python automatically calls __del__() when the object is **garbage collected**.
- You can manually trigger it using del obj, but it only reduces the reference count.
- __del__() is **not guaranteed to be called** if the program ends suddenly or if there are **circular references**.

🚫 Don't Overuse __del__

Instead, use **context managers** (with statement) for resource cleanup (e.g., opening files or DB connections). They are safer and more Pythonic.

✅ **Best Practice Alternative:**

```
with open("file.txt", "r") as f:
    data = f.read()
# File is auto-closed → No need for __del__
```

---

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

📌 **Answer:**

🔷 **Difference between @staticmethod and @classmethod in Python**

Both @staticmethod and @classmethod are **decorators** used to define methods that aren't like regular instance methods. But they behave differently and serve different purposes.

✅ **1. @staticmethod – No access to self or cls**

- It **does not take self or cls** as the first argument.
- Cannot access or modify **class** or **instance state**.
- Used when the method is **logically related to the class**, but doesn't need to access class or instance.

👉 **Example:**

```
class MyClass:
    @staticmethod
    def greet(name):
        return f"Hello, {name}"

print(MyClass.greet("Yash"))  # ✅ Works without creating an object
```

✅ **2. @classmethod – Access to class (cls)**

- It **takes cls** (class itself) as the first argument.
- Can **access/modify class variables**.
- Commonly used for **factory methods** or alternate constructors.

👉 **Example:**

```
class Person:
    species = "Human"

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

    @classmethod
    def from_string(cls, string):
        name = string.split("-")[0]
        return cls(name)
```

```
p = Person.from_string("Yash-25")
print(p.name)      # Yash
print(p.species)   # Human
```

🔁 Comparison Table:

Feature	@staticmethod	@classmethod
First Parameter	No self or cls	Takes cls
Access to class?	❌ No	✅ Yes
Access to instance?	❌ No	❌ No
Common Use	Utility/helper functions	Alternate constructors / class logic

✅ Summary:

- Use @staticmethod when your method **does not need access** to class or instance.
- Use @classmethod when you need to **modify class-level data** or create instances in a flexible way.

23. How does polymorphism work in Python with inheritance

📌 **Answer:**

🔷 How Polymorphism Works in Python with Inheritance

Polymorphism means **"many forms"**. In Python, **polymorphism with inheritance** allows **different classes to define methods with the same name**, and the right method is called based on the object type — even when using a **common interface**.

✅ 1. Using Inheritance for Polymorphism

When a **parent class** defines a method, and **child classes override** it with their own version, you can use **polymorphism** to call the appropriate method based on the object.

👉 **Example:**

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

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

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

👇 **Using Polymorphism:**

```
def make_sound(animal):
    print(animal.speak())

dog = Dog()
cat = Cat()

make_sound(dog)  # Output: Bark
make_sound(cat)  # Output: Meow
```

✅ Even though make_sound() expects an Animal, it works for Dog and Cat — this is **runtime polymorphism**.

✅ 2. Key Benefits:

- **Code Reusability:** You don’t need to write separate functions for each type.
- **Interface Consistency:** All classes follow the same method name/structure.
- **Flexibility and Extensibility:** New classes can be added with minimal changes to existing code.

✅ 3. Real-Life Example:

```
class Document:
    def print_doc(self):
        raise NotImplementedError

class PDF(Document):
    def print_doc(self):
        return "Printing PDF..."

class Word(Document):
    def print_doc(self):
        return "Printing Word Document..."

docs = [PDF(), Word()]

for d in docs:
    print(d.print_doc())
```

📌 Output:

```
Printing PDF...
Printing Word Document...
```

Each object responds differently to the same method call — that's polymorphism in action!

✅ Summary:

Concept	**Description**
What it is	Ability of objects to respond differently to the same method call
How it's achieved	Method overriding in child classes

When it runs	At runtime (dynamic dispatch)
Benefits	Cleaner, scalable, maintainable code

---

24. What is method chaining in Python OOP

📌 **Answer:**

🔷 What is Method Chaining in Python (OOP)?

Method chaining is a programming technique where **multiple methods are called on the same object in a single line**, one after another.

Each method in the chain **returns the object itself (self)**, allowing the next method to be called directly.

✅ **Example:**

```
class Person:
    def __init__(self, name):
        self.name = name
        self.age = None
        self.city = None

    def set_age(self, age):
        self.age = age
        return self  # Returning self for chaining

    def set_city(self, city):
        self.city = city
        return self

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}, City: {self.city}")
        return self
```

🔗 **Method Chaining in Action:**

```
person = Person("Yash")
person.set_age(24).set_city("Pune").display()
```

📤 **Output:**

```
Name: Yash, Age: 24, City: Pune
```

✅ **Key Points:**

Aspect	**Description**
Purpose	Write cleaner and fluent code
Requires	Each method must return self
Works With	Instance methods of the same object

Useful For	Builder patterns, configuration setup, etc.

✅ **When to Use Method Chaining:**

- When you're configuring or setting up an object in multiple steps.
- When you want **compact, readable** object-building code.

⚠️ **Note:**

Method chaining can reduce readability **if overused** or if methods have **side effects**. Use wisely in production code.

---

25. What is the purpose of the __call__ method in Python

📌 **Answer:**

🔷 **What is the purpose of the __call__() method in Python?**

The __call__() method in Python allows **an instance of a class to be called as if it were a function**.

✅ **Purpose:**

It makes objects **callable**, just like functions.

When you do:

```
obj()
```

...it internally executes:

```
obj.__call__()
```

✅ **Example:**

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

    def __call__(self):
        print(f"Hello, {self.name}!")
```

🔗 **Usage:**

```
greeter = Greet("Yash")
greeter()  # Calls greeter.__call__()
```

📤 **Output:**

```
Hello, Yash!
```

✅ **Why Use __call__()?**

Use Case	Benefit
Function-like object behavior	Objects behave like functions
Clean syntax	Avoid separate .run() or .execute() methods
Used in decorators, ML models	Allows flexible object invocation

✅ **Real-world Uses:**

- **Decorators** in Python
- **Machine Learning models** in libraries like TensorFlow or PyTorch (e.g., model(input))
- **Custom callable objects** in OOP designs

⚠️ **Note:**

Using __call__ is **optional** and advanced. Use it when making an object act like a function improves clarity or usability.

Let me know if you want a real-world analogy or want to implement this in a custom class of your own.