

OOP NOTES

Class : A class is a **blueprint or template** for creating objects. It defines the properties (attributes) and behaviors (methods) that the objects created from it will have. Think of a class as a recipe and objects as the items made from that recipe .

Real-Life Analogy

Imagine a **car factory**. The factory doesn't produce a specific car; instead, it has a **blueprint or template** (a class) for making cars. This blueprint defines what each car will have: wheels, an engine, seats, and so on. Each car made from this blueprint (an object) can be slightly different (e.g., color, brand) but shares the same general structure and behavior (drive, honk).

- **Class:** Blueprint for cars.
- **Object:** Specific cars like Toyota, BMW, or Honda.

Example with python :

```
# Defining the class
class Car:
    def __init__(self, brand, color):
        # Attributes of the class (properties of the car)
        self.brand = brand
        self.color = color

    # Method (behavior of the car)
    def drive(self):
        return f"The {self.color} {self.brand} is driving."

    def honk(self):
        return f"The {self.brand} is honking."

# Creating objects (instances of the class)
car1 = Car("Toyota", "red")
car2 = Car("BMW", "blue")

# Accessing their behaviors
print(car1.drive()) # Output: The red Toyota is driving.
print(car2.honk())  # Output: The BMW is honking.
```

- **Class Car:** Defines what attributes a car has (brand, color) and what it can do (drive, honk).
- **Objects car1, car2:** Specific instances of the class, with their own unique attributes but following the same blueprint.

Answer: OOP stands for Object-Oriented Programming. It's a programming paradigm based on the concept of objects.

Object : an object is an instance of a class, In object oriented programming .

Example with python :

```
# Defining the class
class Dog:
    def __init__(self, name, breed):
        # Attributes of the object
        self.name = name
        self.breed = breed

    # Method (behavior of the object)
    def bark(self):
        return f"{self.name} is barking!"

# Creating objects (instances of the class)
dog1 = Dog("Buddy", "Golden Retriever")
dog2 = Dog("Max", "Bulldog")

# Accessing their behaviors and attributes
print(dog1.bark())    # Output: Buddy is barking!
print(dog2.bark())    # Output: Max is barking!
print(dog1.name)      # Output: Buddy
print(dog2.breed)     # Output: Bulldog
```

- **Class Dog:** Defines what attributes a dog will have (name, breed) and what behavior it will have (e.g., bark() method).
- **Objects dog1, dog2:** These are specific instances of the Dog class, each representing a real dog with its own unique name and breed, but both can bark because they are created using the same class blueprint.

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

Answer: Object-Oriented Programming is a programming **paradigm** that organizes code into **objects**, which are instances of classes. It emphasizes concepts like **inheritance**, **encapsulation**, **polymorphism**, and **abstraction** to improve **code reusability** and **maintainability**. **four principles of OOP :**

Inheritance, Encapsulation, Polymorphism, and Abstraction (I, E, P, A):

"I Enjoy Playing Around."

2. What is inheritance?

Answer: Inheritance is a **mechanism** where one class (subclass or derived class) acquires **properties** and **behaviors** from another class (superclass or base class). It promotes **code reusability** and establishes an **"is-a"** relationship between classes.

3. Explain the different types of inheritance?

Single inheritance : A subclass acquires **properties** and **behaviors** / inherits (উত্তরাধিকারসূত্রে) from a single superclass. **Inheritance-এর বাংলা অর্থ হলো উত্তরাধিকার বা বংশগতি।**

Real-Life Analogy for Single Inheritance

Imagine a **parent-child relationship** in a family. The child inherits certain traits (like eye color, height) from the parent. In programming, this is similar to a class (child) inheriting properties and behaviors (methods) from another class (parent).

```
# Parent class
class Parent:
    def show_trait(self):
        print("I have brown eyes.")

# Child class
class Child(Parent): # Single Inheritance happens here
    def show_new_trait(self):
        print("I have inherited my parent's eyes and I am tall.")

# Creating an instance of Child class
child = Child()
child.show_trait()      # Inherited method from Parent class
child.show_new_trait()  # Method in Child class
```

This is **single inheritance** because the Child class inherits from only one parent class.

Summary of Where Single Inheritance Happens:

Python: In the class Child(Parent) line.

Multiple Inheritance: A subclass inherits from multiple superclasses .

Real-Life Analogy for Multiple Inheritance

Imagine a **person** who learns both from their **mother** and **father**. For example, they may inherit creativity from their mother and logical thinking from their father. In programming, this is similar to a class inheriting properties and behaviors from **multiple parent classes**.

Multiple Inheritance in Java

Java **does not support multiple inheritance** directly with classes, but it supports multiple inheritance using **interfaces**. This means that a class can implement multiple interfaces.

```
// First Interface
interface Father {
    void logicalThinking(); // abstract method
}

// Second Interface
interface Mother {
    void creativity(); // abstract method
}

// Child class that implements both interfaces
class Child implements Father, Mother { // Multiple Inheritance happens here
    public void logicalThinking() {
        System.out.println("I inherited logical thinking from my father.");
    }

    public void creativity() {
        System.out.println("I inherited creativity from my mother.");
    }
}

public class Main {
    public static void main(String[] args) {
        Child child = new Child();
        child.logicalThinking(); // From Father
        child.creativity();      // From Mother
    }
}
```

Explanation:

- **Father interface** defines the method `logicalThinking()`.
- **Mother interface** defines the method `creativity()`.
- **Child class** implements both interfaces using the `implements` keyword, thus achieving **multiple inheritance**.
- **Multiple Inheritance in Python :**

```

class Father:
    def logical_thinking(self):
        print("I inherited logical thinking from my father.")

# Mother class
class Mother:
    def creativity(self):
        print("I inherited creativity from my mother.")

# Child class that inherits from both Father and Mother
class Child(Father, Mother): # Multiple Inheritance happens here
    pass # No need to redefine methods, they are inherited

# Creating an instance of Child class
child = Child()
child.logical_thinking() # From Father class
child.creativity()      # From Mother class

```

Explanation:

- In Python, the Father class defines the method `logical_thinking()`, and the Mother class defines the method `creativity()`.
- The Child class inherits from both Father and Mother by listing them in parentheses, which is where multiple inheritance occurs.
- The child object can access methods from both the Father and Mother classes.

Python: In the class `Child(Father, Mother)` line (directly with classes).

Java uses interfaces to achieve this, while Python supports multiple inheritance directly with classes.

Java does **not support multiple inheritance** with classes to avoid **ambiguity and complexity**. The primary reason is to prevent the **"Diamond Problem"**.

Example of the Diamond Problem (if multiple inheritance were allowed in Java)

java

Copy code

```
class A {
    void show() {
        System.out.println("Class A show");
    }
}

class B extends A {
    void show() {
        System.out.println("Class B show");
    }
}

class C extends A {
    void show() {
        System.out.println("Class C show");
    }
}

// Class D inherits from both B and C (This is NOT allowed in Java)
class D extends B, C {
    // Ambiguity: which 'show()' method to call, from B or C?
}

public class Main {
    public static void main(String[] args) {
        D obj = new D();
        obj.show(); // Which show() should be called? B's or C's?
    }
}
```

- **Ambiguity:** In case of multiple inheritance, the compiler would face ambiguity about which method to inherit if two parent classes define the same method.
- **Complexity:** Multiple inheritance makes it difficult to manage and maintain code, especially as the class hierarchy grows.

Multilevel Inheritance: A class is derived from another class, which, in turn, is derived from a base class.

Real-Life Analogy for Multilevel Inheritance

Think of a **family tree** with three generations. For example:


1. The **Grandparent** has a certain set of traits.
2. The **Parent** inherits traits from the Grandparent and adds their own traits.
3. The **Child** inherits traits from both the Parent and the Grandparent.

This is like **multilevel inheritance** in programming, where a class inherits from another class, which itself is derived from yet another class.

Python Code Example for Multilevel Inheritance

In Python, multilevel inheritance works similarly to Java, but the syntax is simpler:

python

 Copy code

```
# Grandparent class
class Grandparent:
    def show_generational_trait(self):
        print("I am the grandparent with wisdom.")

# Parent class that inherits from Grandparent
class Parent(Grandparent): # Single Inheritance happens here
    def show_parent_trait(self):
        print("I am the parent, inheriting wisdom and strength.")

# Child class that inherits from Parent
class Child(Parent): # Single Inheritance happens here again
    def show_child_trait(self):
        print("I am the child, inheriting wisdom, strength, and energy.")

# Creating an instance of Child class
child = Child()
child.show_generational_trait() # Inherited from Grandparent
child.show_parent_trait()      # Inherited from Parent
child.show_child_trait()       # Method from Child
```

Explanation:

- The `Grandparent` class defines the method `show_generational_trait()`.
- The `Parent` class inherits from `Grandparent` and adds its own method `show_parent_trait()`. This is **single inheritance** from `Grandparent`.
- The `Child` class inherits from `Parent` and adds its own method `show_child_trait()`. This is **single inheritance** from `Parent`.
- The `child` object can call methods from the `Grandparent`, `Parent`, and `Child` classes.



Hierarchical Inheritance: Multiple subclasses inherit from a **single superclass**.

Real-Life Analogy for Hierarchical Inheritance

Imagine a **teacher** who teaches multiple students. The teacher has certain knowledge (like subject expertise), and each student inherits that knowledge but can have their own unique characteristics.

1. The **Teacher** is the base (parent) class, holding common knowledge.
2. **Student1** inherits the knowledge from the Teacher and adds their own traits.

3. **Student2** also inherits the same knowledge but has different traits.

This is **hierarchical inheritance** where multiple child classes inherit from the same parent class.

Python Code Example for Hierarchical Inheritance

In Python, hierarchical inheritance works similarly to Java, but the syntax is more concise.

python

Copy code

```
# Parent class
class Teacher:
    def teach(self):
        print("I am the teacher, I teach.")

# Child class 1 inherits from Teacher
class Student1(Teacher): # Single inheritance happens here
    def student1_skills(self):
        print("I am Student 1, I am good at mathematics.")

# Child class 2 inherits from Teacher
class Student2(Teacher): # Single inheritance happens here
    def student2_skills(self):
        print("I am Student 2, I am good at science.")

# Creating objects
student1 = Student1()
student1.teach()          # Inherited from Teacher
student1.student1_skills() # Method specific to Student1

student2 = Student2()
student2.teach()          # Inherited from Teacher
student2.student2_skills() # Method specific to Student2
```

Explanation:

- The **Teacher** class defines the method **teach()**.
- **Student1** and **Student2** inherit from **Teacher**. Each adds their own unique method (**student1_skills()** for **Student1** and **student2_skills()** for **Student2**). Both also inherit the **teach()** method from **Teacher**.

5 . What is encapsulation?

Encapsulation is the bundling of **data** (attributes) and **methods** (functions) within **single unit** or a **class**, restricting direct access to some of the class's attributes. This ensures that the **internal state of the object** is protected from unintended modification, and the **user interacts** with the object only through well-defined interfaces (**public methods**). **Encapsulation bangla meaning সংবরণ বা আবরণীকরণ | Encapsulation is the bundling of data and methods that manipulate the data, ensuring that the object's internal state remains hidden and protected.**

Real-Life Analogy for Encapsulation (Bank Account Example)

- Answer: A constructor initializes an object when it's created, while a destructor is called when an object is destroyed to perform cleanup tasks.

Imagine a bank account:

- The balance and account details are private and hidden from outsiders.
- You can only access or modify your balance through specific actions like depositing or withdrawing money. These actions are like public methods.

In this way, the bank controls how the account data (balance) is accessed and modified, ensuring security and preventing misuse, which mirrors how encapsulation works in object-oriented programming.

Python Code Example for Encapsulation (Bank Account) :

```
python Copy code

# BankAccount class demonstrating encapsulation
class BankAccount:
    # Constructor to initialize account holder name and balance (private attributes)
    def __init__(self, account_holder_name, initial_balance):
        self.__account_holder_name = account_holder_name
        self.__balance = initial_balance

    # Public method to get the account holder's name (encapsulation in action)
    def get_account_holder_name(self):
        return self.__account_holder_name

    # Public method to check the balance (encapsulation in action)
    def get_balance(self):
        return self.__balance

    # Public method to deposit money into the account
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"{amount} deposited. Current balance: {self.__balance}")
        else:
            print("Invalid deposit amount.")

    # Public method to withdraw money from the account
    def withdraw(self, amount):
        if amount > 0 and amount <= self.__balance:
            self.__balance -= amount
            print(f"{amount} withdrawn. Current balance: {self.__balance}")
        else:
            print("Invalid or insufficient funds for withdrawal.")
```

```
# Creating a BankAccount object
my_account = BankAccount("John Doe", 1000)

# Accessing account details through public methods (encapsulation in action)
print("Account Holder:", my_account.get_account_holder_name())
print("Initial Balance:", my_account.get_balance())

# Performing operations on the account
my_account.deposit(500)      # Depositing money
my_account.withdraw(200)    # Withdrawing money
my_account.withdraw(1500)   # Attempt to withdraw more than the balance
```

Explanation:

- The `__account_holder_name` and `__balance` are private fields due to the double underscores. They cannot be accessed directly from outside the class.
- Methods like `get_account_holder_name()`, `get_balance()`, `deposit()`, and `withdraw()` control access to these private fields, which ensures that the data is accessed in a controlled manner.
- Encapsulation is achieved by hiding the sensitive data and providing access through public methods.

- The fields `self.__account_holder_name` and `self.__balance` are private.
- Public methods like `get_account_holder_name()`, `get_balance()`, `deposit()`, and `withdraw()` provide controlled access.

Summary

In the **Bank Account** example, **encapsulation** ensures that the balance and account holder's information is not directly accessible or modifiable from outside the class. This control is important to prevent unwanted changes and protect sensitive data. In both Java and Python, encapsulation is achieved by making fields private and exposing only specific methods to interact with the data.

6. What is polymorphism?

‘Poly’ means **many**, and ‘morphism’ means **forms**. **Polymorphism is the ability of an object to take on many forms. Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.** Polymorphism is achieved **through method overloading (compile-time polymorphism) and method overriding (runtime polymorphism).** It allows better **readability and code maintenance**.

here are two types of polymorphism:

1. Compile-time polymorphism (Method Overloading): Multiple methods with the same name but different parameters.
2. Run-time polymorphism (Method Overriding): When a subclass overrides a method from its superclass to provide a specific implementation.

Example of Polymorphism in OOPs

We can understand polymorphism in OOPs with the help of many real-world examples. Let us check out some of these examples below.

- A woman is a mother to her son. She is a teacher for the other and the wife of his husband.
- A man is a labourer at work, a father at home and a shop buyer.
- A media player can play different video, audio, etc. With the help of polymorphism, we can create a superclass for all media files and use it to perform various functions.

Real-Life Analogy for Polymorphism

Think of **polymorphism** like a **shape**. A shape can be a circle, square, or triangle, but they all share some common behavior (e.g., calculating area), even though the method to calculate area differs for each shape.

For example:

- **Circle** calculates area using $\pi * r^2$
- **Square** calculates area using $\text{side} * \text{side}$
- **Triangle** calculates area using $\frac{1}{2} * \text{base} * \text{height}$

Even though the process of calculating area differs, they all have a common action, "calculate area," which represents **polymorphism** in action.

Python Code Example for Polymorphism (Shape Example)

Here's how the same example works in Python:

```
python Copy code

# Base class Shape
class Shape:
    # Method to calculate area (to be overridden by subclasses)
    def calculate_area(self):
        print("Calculating area for Shape")

# Subclass Circle overriding the calculate_area method
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    # Overriding calculate_area method
    def calculate_area(self):
        area = 3.14159 * self.radius * self.radius
        print(f"Area of Circle: {area}")

# Subclass Square overriding the calculate_area method
class Square(Shape):
    def __init__(self, side):
        self.side = side
```

```
# Subclass Square overriding the calculate_area method

class Square(Shape):
    def __init__(self, side):
        self.side = side

    # Overriding calculate_area method
    def calculate_area(self):
        area = self.side * self.side
        print(f"Area of Square: {area}")

# Polymorphism in action
# Shape can refer to both Circle and Square
my_shape = Circle(5.0)
my_shape.calculate_area() # Calls Circle's calculate_area method

my_shape = Square(4.0)
my_shape.calculate_area() # Calls Square's calculate_area method
```

Explanation:

- The `Shape` class has a method `calculate_area()` which is overridden in the `Circle` and `Square` subclasses.
- Polymorphism happens when the object `my_shape` can refer to either a `Circle` or a `Square` and invoke the same `calculate_area()` method, but the result is based on the actual object type.

- **Polymorphism** happens when the variable `my_shape` can refer to both `Circle` and `Square` objects.
- The method `calculate_area()` is invoked, and based on the object type, the corresponding method in `Circle` or `Square` is executed.

Summary:

- **Polymorphism** allows objects of different types to be treated as objects of a common superclass, letting you use the same method names with different implementations in different subclasses.
- In both the Java and Python examples, polymorphism is demonstrated by having the same method name (`calculateArea()` or `calculate_area()`) behave differently based on whether the object is a `Circle` or a `Square`.

Polymorphism in OOPs	
Compile time polymorphism in OOPs	Runtime polymorphism in OOPs
Compile time polymorphism is also known as static binding or early binding.	It is also known as Dynamic polymorphism or late binding.
Static polymorphism can be attained when the compiler resolves method calls.	In runtime polymorphism, the method calls are not resolved by the compiler.
Compile time polymorphism can be attained by method overloading.	Runtime polymorphism can be attained during runtime.
Method overloading is a type of polymorphism in which one method can have the same name with different parameters or return type.	Method overriding is the runtime polymorphism that implements a method already defined by its parent class or super class
It is known as early binding because it takes less time to execute.	It takes more time to execute the method, hence known as late binding.
No inheritance is present.	Inheritance is present.

Disadvantages : 1. Increased Complexity

- **Disadvantage:** Polymorphism can make the code more complex to understand, especially for beginners. When different objects behave differently based on the same method, it can be hard to follow the logic.
- **Example:** If multiple subclasses override the same method in different ways, it may confuse the developer when trying to understand which version of the method is being executed at runtime.

. Limited Flexibility in Static Polymorphism

- **Disadvantage:** Static polymorphism (method overloading) is determined at compile-time and lacks the flexibility of runtime decisions. This limits its adaptability to different situations during program execution.
- **Example:** In method overloading, the exact method signature must be known at compile-time, and the program cannot change behavior dynamically based on the actual data passed to the function.

1. Compile time Polymorphism

Compile time polymorphism is achieved with the help of **method overloading** and **operator overloading**.

In method overloading, we use the **same methods** with **different parameters** to achieve polymorphism. We do not need to keep different names for the same function. This improves the **effectiveness** of the code. It is also known as **Static binding or early binding**.

Let us suppose we need to multiply two different numbers, and we can write a function name multiply to achieve this. However, if we need to multiply three numbers, then with the help of polymorphism in oops, we do not need to write the complete code again. **Method overloading allows a class to have multiple methods with the same name but different parameters. Method overriding occurs when a subclass provides a specific implementation for a method already defined in its superclass.**

```
class Product:
    # Method to multiply 2 numbers
    def multiply(self, a, b):
        return a * b

    # Method to multiply 3 numbers
    def multiply(self, a, b, c):
        return a * b * c

# Main driver method
if __name__ == "__main__":
    ob = Product()

    # Multiply 2 numbers
    prod1 = ob.multiply(1, 2)
    print("Product of the two integer values:", prod1)

    # Multiply 3 numbers
    prod2 = ob.multiply(1, 2, 3)
    print("Product of the three integer values:", prod2)
```

Here, we are using multiply(int a, int b) for two-digit multiplication. For three-digit multiplication, we call the method with the same name but a different parameter.

2. Run time Polymorphism

Run time polymorphism is a type of polymorphism in OOPS which is **resolved** during runtime. Method overriding is used to implement a method that is already defined in its **parent class** or super class. Runtime polymorphism in oops is also known as **dynamic binding** or **late binding**.

What is the difference between final, finally, and finalize?

final is a keyword used to restrict class inheritance, method overriding, or variable reassignment. finally is a block that is executed regardless of exceptions. finalize is a method called by the garbage collector before an object is destroyed.

```
class Animal:
    # Base class method
    def sound(self):
        print("Animal makes a sound")

class Dog(Animal):
    # Overridden method in derived class
    def sound(self):
        print("Dog barks")

# Main driver code
if __name__ == "__main__":
    # Create an instance of Animal
    animal = Animal()
    animal.sound() # Output: Animal makes a sound

    # Create an instance of Dog
    dog = Dog()
    dog.sound() # Output: Dog barks
```

In this example, the sound method, which is already defined in the Animal class, is overridden and used in the Dog class. When this sound method is called for a dog object, it will print “Dog barks”.

What is abstraction?

Answer: Abstraction is the process of hiding the **implementation details** of **an object** and exposing only the **relevant features or behavior**. It allows developers to work with high-level concepts and ignore **low-level implementation** complexities.

REAL LIFE ANALOGY :

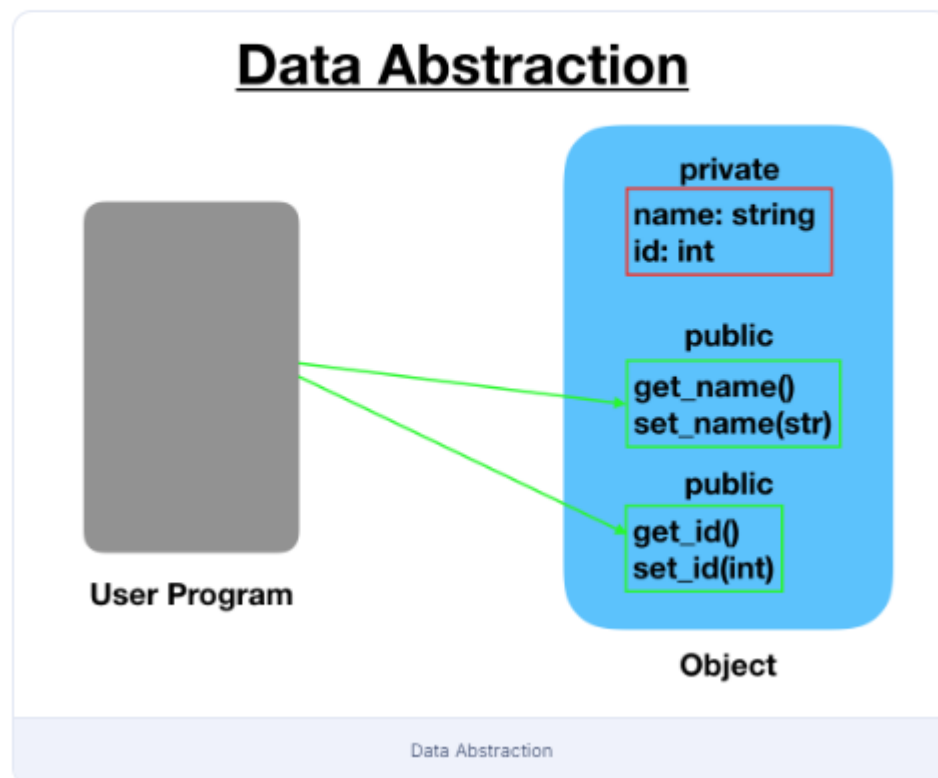
- Your car is a great example of abstraction. You can start a car by turning the key or pressing the start button. You don't need to know how the engine is getting started, what all components your car has. The car internal implementation and complex logic is completely hidden from the user.

Abstraction in OOPS

Objects are the building blocks of Object-Oriented Programming. An object contains some properties and methods. We can hide them from the outer world through access modifiers. We can provide access only for required functions and properties to the other programs. This is the general procedure to implement abstraction in OOPS.

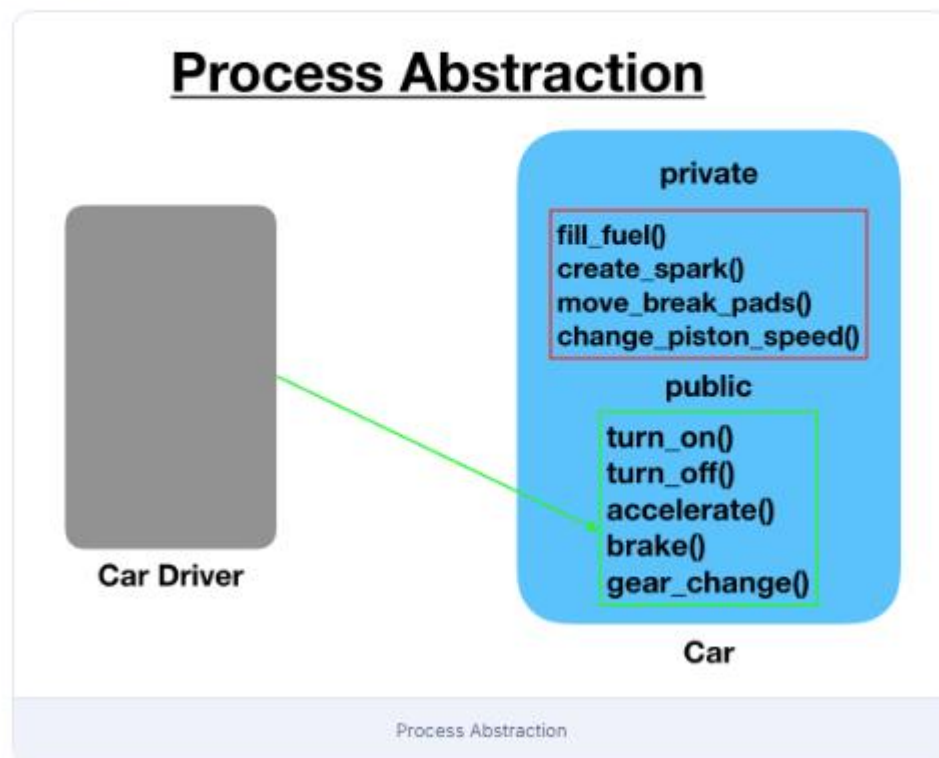
4.1) Data Abstraction

When the object data is not visible to the outer world, it creates data abstraction. If needed, access to the Objects' data is provided through some methods.



4.2) Process Abstraction


We don't need to provide details about all the functions of an object. When we hide the internal implementation of the different functions involved in a user operation, it creates process abstraction.



Example of Abstraction in Python (Vehicle Example)

Now, let's convert the above example into Python:

python

 Copy code

```
from abc import ABC, abstractmethod

# Abstract class Vehicle
class Vehicle(ABC):
    # Abstract method (no implementation in the abstract class)
    @abstractmethod
    def start_engine(self):
        pass

    # Regular method (common for all vehicles)
    def stop_engine(self):
        print("Engine stopped.")

# Concrete class Car extending Vehicle
class Car(Vehicle):
    # Implementing the abstract method
    def start_engine(self):
        print("Car engine started.")

# Concrete class Bike extending Vehicle
class Bike(Vehicle):
    # Implementing the abstract method
    def start_engine(self):
        print("Bike engine started.")
```



```
# Using abstraction
my_car = Car()
my_car.start_engine() # Calls Car's start_engine method
my_car.stop_engine() # Calls common stop_engine method

my_bike = Bike()
my_bike.start_engine() # Calls Bike's start_engine method
my_bike.stop_engine() # Calls common stop_engine method
```

Explanation:

- In Python, the `Vehicle` class is abstract, meaning it defines the high-level behavior (`start_engine()` method) but leaves the implementation to the subclasses (`Car` and `Bike`).
- Abstraction is implemented using the `ABC` module (`ABC` stands for Abstract Base Class) and the `@abstractmethod` decorator, which ensures that the `start_engine()` method is implemented by subclasses.
- Like the Java example, the internal details of starting the engine are hidden, and only essential actions are exposed to the user.

In Python:

- The **Vehicle** class is abstract, and the method **start_engine()** is declared abstract using the ABC module. This hides the implementation details from the user.
- **Abstraction** happened when the **start_engine()** method was defined in the **Vehicle** class, but the specific implementations were provided in **Car** and **Bike** subclasses.

10. What are abstract classes?

Answer: An **abstract class** is a class that cannot be **instantiated/ created** directly and may contain **one or more abstract methods**. Abstract methods are declared **without implementation** and must be implemented by subclasses.

```
from abc import ABC, abstractmethod

# Abstract class Shape
class Shape(ABC):
    # Abstract method (no implementation)
    @abstractmethod
    def calculate_area(self):
        pass

    # Concrete method (implemented in the abstract class)
    def display(self):
        print("This is a shape.")

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

    # Implementing the abstract method
    def calculate_area(self):
        return 3.1416 * self.radius * self.radius

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

    # Implementing the abstract method
    def calculate_area(self):
        return self.length * self.width
```



```
# Using the abstract class and its concrete implementations
my_circle = Circle(5.0)
print("Area of Circle:", my_circle.calculate_area()) # Circle's implementation
my_circle.display() # Calls display method from Shape

my_rectangle = Rectangle(4.0, 6.0)
print("Area of Rectangle:", my_rectangle.calculate_area()) # Rectangle's implementation
my_rectangle.display() # Calls display method from Shape
```

Explanation:

- **Abstract Class:** `Shape` is an abstract class, defined using the `ABC` (Abstract Base Class) module in Python.
- **Abstract Method:** `calculate_area()` is defined as an abstract method, meaning it must be implemented by subclasses.
- **Concrete Classes:** `Circle` and `Rectangle` are subclasses that implement the `calculate_area()` method.
- The `display()` method is common across all shapes and is defined in the `Shape` abstract class.

In Python:

- The class `Shape` is an abstract class, defined using the `ABC` module, and the method `calculate_area()` is marked as abstract with the `@abstractmethod` decorator.
- The method `display()` is concrete, shared by all shapes.
- Abstract class behavior: The subclasses `Circle` and `Rectangle` provide the actual implementation for the abstract method `calculate_area()`.


11. What is an interface?

Ans : An interface is a blueprint that defines a set of methods that a class must implement. It provides a way to achieve multiple inheritance in languages that do not support it directly.

Example of an Interface in Java

Let's create an interface `Vehicle` that defines methods like `startEngine()` and `stopEngine()`. Specific classes, like `Car` and `Bike`, will implement this interface and provide their own details for these methods.

java

 Copy code

```
// Interface Vehicle
interface Vehicle {
    void startEngine(); // Abstract method
    void stopEngine();  // Abstract method
}

// Class Car implements Vehicle
class Car implements Vehicle {
    @Override
    public void startEngine() {
        System.out.println("Car engine started");
    }

    @Override
    public void stopEngine() {
        System.out.println("Car engine stopped");
    }
}

// Class Bike implements Vehicle
class Bike implements Vehicle {
    @Override
    public void startEngine() {
        System.out.println("Bike engine started");
    }

    @Override
    public void stopEngine() {
        System.out.println("Bike engine ↓ ped");
    }
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Vehicle myCar = new Car();  
        myCar.startEngine(); // Calls Car's implementation  
        myCar.stopEngine();  // Calls Car's implementation  
  
        Vehicle myBike = new Bike();  
        myBike.startEngine(); // Calls Bike's implementation  
        myBike.stopEngine();  // Calls Bike's implementation  
    }  
}
```


Explanation:

- **Interface:** `Vehicle` is an interface that defines two abstract methods: `startEngine()` and `stopEngine()`.
- **Concrete Classes:** `Car` and `Bike` implement the `Vehicle` interface and provide specific implementations for the methods.
- The interface specifies what needs to be done, but the specific classes define how it will be done.

Example of an Interface in Python

Python does not have formal interfaces like Java, but we can achieve similar functionality using abstract base classes (`ABC` module). Here's the same example in Python:

python

 Copy code

```
from abc import ABC, abstractmethod

# Interface Vehicle (using abstract class)
class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass

    @abstractmethod
    def stop_engine(self):
        pass

# Class Car implements Vehicle
class Car(Vehicle):
    def start_engine(self):
        print("Car engine started")

    def stop_engine(self):
        print("Car engine stopped")

# Class Bike implements Vehicle
class Bike(Vehicle):
    def start_engine(self):
        print("Bike engine started")

    def stop_engine(self):
        print("Bike engine stopped")
```



```
# Using the interface-like class and its concrete implementations
my_car = Car()
my_car.start_engine() # Calls Car's implementation
my_car.stop_engine()  # Calls Car's implementation

my_bike = Bike()
my_bike.start_engine() # Calls Bike's implementation
my_bike.stop_engine()  # Calls Bike's implementation
```

Explanation:

- Abstract Class as Interface:** `Vehicle` is defined using the `ABC` module in Python, simulating an interface.
- Abstract Methods:** The methods `start_engine()` and `stop_engine()` are abstract, meaning they must be implemented by any class that inherits from `Vehicle`.
- Concrete Classes:** `Car` and `Bike` are concrete classes that implement the abstract methods, providing specific functionality.

12. What is the "this" keyword?

the **this** keyword refers to the current **instance/object** of the class . It is used to access instance variables or call **instance methods** of the class. It helps distinguish between instance variables and method parameters when they have the same name.

However, in **Python**, the keyword **self** serves the same purpose as **this** in Java. It refers to the current object instance and is used to access instance variables and methods.

Example in Java using `this`:

java

Copy code

```
class Person {
    String name;
    int age;

    // Constructor with same parameter names as instance variables
    Person(String name, int age) {
        this.name = name; // 'this' is used to refer to the instance variable
        this.age = age;
    }

    void introduce() {
        System.out.println("My name is " + this.name + " and I am " + this.age + " years o
    }
}

public class Main {
    public static void main(String[] args) {
        Person person = new Person("John", 25);
        person.introduce(); // Calls the method introduce() using 'this'
    }
}
```

Explanation:

- The `this` keyword is used in the constructor to distinguish between instance variables (`this.name` , `this.age`) and the constructor parameters (`name` , `age`).
- `this.name` refers to the `name` variable of the current instance, while `name` refers to the constructor parameter.



Equivalent Example in Python using `self`:

python

Copy code

```
class Person:
    def __init__(self, name, age):
        self.name = name # 'self' refers to the instance variable
        self.age = age

    def introduce(self):
        print(f"My name is {self.name} and I am {self.age} years old.")

# Creating an instance of Person
person = Person("John", 25)
person.introduce() # Calls the method introduce() using 'self'
```

Explanation:

- In Python, `self` is used instead of `this` to refer to the instance variables (`self.name` , `self.age`) of the current object.
- `self.name` refers to the instance variable `name` of the object being created, while `name` refers to the constructor argument.

13. What is the "super" keyword?

Answer: The "super" keyword is used to call the superclass's constructor or refer to the superclass's methods or variables from a subclass.

14. What is method hiding?

Answer: Method hiding occurs when a subclass defines a method with the same name and parameters as a method in its superclass. Method hiding occurs when a subclass defines a static method with the same name and signature as a static method in the superclass. The subclass's method hides the superclass's method, and the method invoked depends on the reference type rather than the object type.

Real-Life Analogy for Method Hiding

Imagine a **library** where there is a general **book** classification system.

- **Superclass (Book):** The library has a general classification for all books that states "All books have a title."
- **Subclass (E-Book):** The library introduces a new category of books called "E-Books." The classification for E-Books states "E-Books can be downloaded."

In this case, both categories (Book and E-Book) have a method for getting the title, but the E-Book overrides the general rule by stating its own classification. When someone asks for the title of an E-Book, they receive the specific E-Book title instead of the general title of all books.

Example of Method Hiding in Java

Here's how method hiding works in Java:

java

Copy code

```
// Superclass
class Book {
    static void getType() {
        System.out.println("This is a general book.");
    }
}

// Subclass
class EBook extends Book {
    static void getType() {
        System.out.println("This is an E-Book.");
    }
}

public class Main {
    public static void main(String[] args) {
        Book.getType(); // Calls the static method from Book
        EBook.getType(); // Calls the static method from EBook

        // Method hiding happens here
        Book book = new EBook();
        book.getType(); // Calls the static method from Book, not EBook
    }
}
```

Explanation:

- The method **getType()** is defined as static in both the superclass (Book) and the subclass (EBook).
- When calling `EBook.getType()`, it correctly accesses the method specific to E-Books.
- However, when calling `book.getType()`, it refers to the method in Book, demonstrating method hiding because `book` is a reference of type Book pointing to an EBook object.

Example of Method Hiding in Python

In Python, method hiding can be simulated using class methods or static methods. Here's the equivalent example:

python

Copy code

```
class Book:
    @classmethod
    def get_type(cls):
        print("This is a general book.")

class EBook(Book):
    @classmethod
    def get_type(cls):
        print("This is an E-Book.")

# Calling the methods
Book.get_type() # Calls the method from Book
EBook.get_type() # Calls the method from EBook

# Method hiding happens here
book = EBook() # Create an EBook instance
book.get_type() # Calls the method from EBook because it's an instance of EBook

# Demonstrating method hiding with class reference
Book.get_type() # Calls the method from Book
```

Explanation:


- In this Python example, `get_type()` is defined as a class method in both `Book` and `EBook`.
- When calling `EBook.get_type()`, it accesses the specific E-Book method.
- When creating an instance of `EBook` and calling `book.get_type()`, it correctly calls the method from `EBook` because the instance is of type `EBook`.

A method decorated with `@staticmethod` does not take `self` or `cls` as its first argument.

Sure! The Java example you provided demonstrates method hiding using static methods in a superclass and subclass. Below is the equivalent implementation in Python using static methods:

Python Example

python

 Copy code

```
# Superclass
class Book:
    @staticmethod
    def get_type():
        print("This is a general book.")

# Subclass
class EBook(Book):
    @staticmethod
    def get_type():
        print("This is an E-Book.")

# Main function to demonstrate method hiding
def main():
    # Calling the static methods
    Book.get_type() # Calls the static method from Book
    EBook.get_type() # Calls the static method from EBook

    # Method hiding happens here
    book = EBook() # Create an instance of EBook
    book.get_type() # Calls the static method from EBook

    # Method hiding when called through the superclass reference
    Book.get_type() # Still calls the static method from Book

# Run the main function
if __name__ == "__main__":
    main()
```



Explanation

- Superclass:**
 - The Book class has a static method `get_type()` that prints "This is a general book."
- Subclass:**
 - The EBook class inherits from Book and overrides the `get_type()` static method to print "This is an E-Book."
- Main Function:**
 - Calls `Book.get_type()`, which outputs "This is a general book."
 - Calls `EBook.get_type()`, which outputs "This is an E-Book."
 - The line `book = EBook()` creates an instance of EBook, and `book.get_type()` calls the static method from EBook because it is called on an instance of EBook.
- Method Hiding:**
 - If you create an instance of EBook and call the static method using `book.get_type()`, it calls the method from EBook.
 - However, calling `Book.get_type()` directly will always call the method from Book.

15. What are access modifiers, and what are their purposes?

Answer: Access modifiers define the **visibility and accessibility** of class members (attributes, methods, constructors). The main access modifiers are public, private, protected, and package-private/default. They control how members can be accessed from other classes and packages.

16. Explain the "final" keyword.

Answer: In OOP, the "final" keyword can be applied to a class, method, or variable. A final class cannot be **subclassed**, a final method **cannot** be **overridden**, and a final **variable** cannot be reassigned once initialized.

Analogy

Imagine a **final destination** in a journey. Once you reach this point, you cannot change your destination anymore. Similarly, in programming, when you declare a variable, method, or class as final, it signifies that it cannot be modified, overridden, or extended.

1. **Final Variable:** Once assigned, its value cannot be changed (like the distance to your final destination).
2. **Final Method:** It cannot be overridden in subclasses (like a strict rule that cannot be changed).
3. **Final Class:** It cannot be subclassed (like a destination that cannot be changed).

```

// Final Variable Example
class Journey {
    final int distance; // This variable cannot be changed once assigned

    Journey(int distance) {
        this.distance = distance; // Assigning value to final variable
    }

    void printDistance() {
        System.out.println("Final destination distance: " + distance + " km");
    }
}

// Final Method Example
class Travel {
    final void displayMode() {
        System.out.println("Traveling by car.");
    }
}

// Final Class Example
final class Destination {
    void showDestination() {
        System.out.println("Your final destination is Paris.");
    }
}

// Trying to extend the final class will cause an error
// class NewDestination extends Destination {} // Error: cannot inherit from final class

```

17. What is a constructor?

Answer: A constructor is a special method that is automatically called when an object of a class is created. It is used to initialize the object's state and perform setup tasks.

18. What is a destructor?

Answer: A destructor is a special method that is called when an object is destroyed or goes out of scope. It is used to release resources and perform cleanup operations.

19. Explain the concept of static members

In Java, static members (variables and methods) belong to the class rather than to any specific instance (object) of the class. This means that static members can be accessed without creating an instance of the class. All instances share the same static member.

Python Equivalent

In Python, you can achieve similar functionality using class variables and class methods. Here's the equivalent code in Python:

python

Copy code

```
class Library:
    total_books = 0 # Class variable shared by all instances

    def __init__(self):
        Library.total_books += 1 # Increment total_books whenever a new library is created

    @classmethod
    def display_total_books(cls): # Class method to display total_books
        print(f"Total books in the library: {cls.total_books}")

# Main program
if __name__ == "__main__":
    lib1 = Library() # Create first library
    lib2 = Library() # Create second library

    Library.display_total_books() # Accessing class method without creating an instance
```

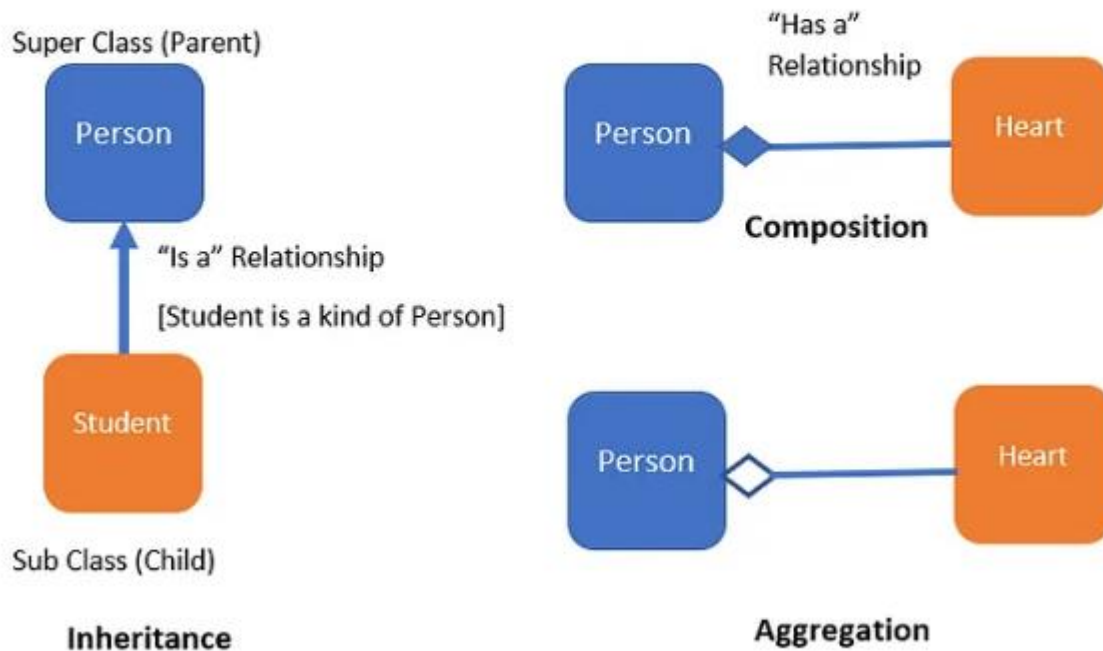
21. What is a constructor chaining?

Answer: Constructor chaining is the process of calling one constructor from another within the same class or between base and derived classes. It allows constructors to reuse code and perform common initialization tasks.

22. What is the difference between composition and inheritance?

- **Inheritance** facilitates code reuse and polymorphism by establishing an "is-a" relationship between classes. However, it can lead to tight coupling and less flexible designs.
- **Composition**, on the other hand, promotes flexibility and reusability by building classes from other classes, establishing a "has-a" relationship.

UML Diagram



Inheritance :

Let me elaborate it in terms of computer science. You will see the below code that contains two classes. One is Person and other is Student, where class Student has all data members which Person has but not the vice-versa. Moreover, Class Student has one more feature or data-member named gradYear which is not present in Person.

So, simply we can say that it establishes an "is-a" or "kind of" relationship between Student and Person class. Saying that, Student is a kind of Person.

```

1 class Person:
2     def __init__(self, fname, lname, address):
3         self.fname = fname
4         self.lname = lname
5         self.address = address
6
7     def display(self):
8         print("First Name: ", self.fname)
9         print("Last Name: ", self.lname)
10        print("Address: ", self.address)
11
12 class Student(Person):
13     def __init__(self, fname, lname, address, age, gradYear):
14         super().__init__(fname, lname, address)
15         self.age = age
16         self.gradYear = gradYear
17
18     def display(self):
19         super().display()
20         print("Age: ", self.age)
21         print("Graduation Year: ", self.gradYear)
22
23 # person object
24 per = Person("Adam", "Ho", "1234 abc blvd")
25 per.display()
26 print("=====")
27 std = Student("Peter", "kee", "9876 xyz blvd", 28, 2008)
28 std.display()

```

Inheritance in Python hosted with ❤ by GitHub

[view raw](#)

Composition : Composition established “has-a” relationship between objects. In below code, you can see that the class person is creating a heart object. So, person is the owner of the heart object. We can also say that Person and Heart objects are tightly coupled. Composition is a design principle where a class contains an object of another class, making it a part of its state .

Describe Encapsulation vs. Abstraction.

- **Answer: Encapsulation** focuses on bundling data and methods, while **abstraction** focuses on hiding implementation details and showing only essential features.


```

1 class Heart:
2     def __init__(self, heartValves):
3         self.heartValves = heartValves
4
5     def display(self):
6         return self.heartValves
7
8 class Person:
9     def __init__(self, fname, lname, address, heartValves):
10        self.fname = fname
11        self.lname = lname
12        self.address = address
13        self.heartValves = heartValves
14        self.heartObject = Heart(self.heartValves)    # Composition
15
16    def display(self):
17        print("First Name: ", self.fname)
18        print("Last Name: ", self.lname)
19        print("Address: ", self.address)
20        print("No of Heart Valves: ", self.heartObject.display())
21
22
23 p = Person("Adam", "syn", "876 Zyx Ln", 4)
24 p.display()

```

Composition in Python hosted with ❤ by GitHub

[view raw](#)

Aggregation:

Not to confuse, aggregation is a form of composition where objects are loosely coupled. There are not any objects or classes owns another object. It just creates a reference. It means if you destroy the container, the content still exists.

In below code, Person just reference to Heart. There is no tight coupling between Heart and Person object.

```

1 class Heart:
2     def __init__(self, heartValves):
3         self.heartValves = heartValves
4
5     def display(self):
6         return self.heartValves
7
8 class Person:
9     def __init__(self, fname, lname, address, heartValves):
10        self.fname = fname
11        self.lname = lname
12        self.address = address
13        self.heartValves = heartValves    # Aggregation
14
15    def display(self):
16        print("First Name: ", self.fname)
17        print("Last Name: ", self.lname)
18        print("Address: ", self.address)
19        print("No of Healthy Valves: ", hv.display())
20
21 hv = Heart(4)
22 p = Person("Adam", "Lee", "555 wso blvd", hv)
23 p.display()

```

Aggregation in Python hosted with ❤ by GitHub

[view raw](#)

23. What is method visibility, and how is it controlled?

Answer: Method visibility refers to the accessibility of methods from other classes. It is controlled by access modifiers like "public," "private," "protected," and "package-private/default."

25. What is the difference between shallow copy and deep copy?

Answer: Shallow copy creates a new object but copies only the references of the original object's members. Deep copy, on the other hand, creates a new object and copies all the content, including recursively copying nested objects, if any.

26. what is virtual function :

virtual functions are member functions in a base class that can be overridden in derived classes. They enable **runtime polymorphism**, allowing the program to decide at runtime which function to invoke based on the object type rather than the reference type.

Concept of Virtual Functions in Python

In Python, **virtual functions** are implemented using method overriding, which allows methods in derived classes to provide specific implementations for methods defined in the base class. While Python does not have a formal virtual keyword like C++, it supports the concept of polymorphism naturally.

Analogy

Using the same **musical instrument** analogy, when a musician wants to play an instrument, they reference the general term "Instrument." Each specific type of instrument (like Piano, Guitar, and Drum) can implement its own version of the play() method.

Data hiding is achieved through encapsulation, by making class fields private and providing public methods (getters and setters) to access and modify them.

```
# Base class
class Instrument:
    def play(self): # Base class method
        print("Playing a generic instrument.")

# Derived class: Piano
class Piano(Instrument):
    def play(self): # Overriding the base class method
        print("Playing the piano.")

# Derived class: Guitar
class Guitar(Instrument):
    def play(self): # Overriding the base class method
        print("Playing the guitar.")

# Derived class: Drum
class Drum(Instrument):
    def play(self): # Overriding the base class method
        print("Playing the drum.")

# Main program
if __name__ == "__main__":
    instruments = [] # List to hold instrument objects

    # Creating objects of each instrument type
    instruments.append(Piano())
    instruments.append(Guitar())
    instruments.append(Drum())

    # Looping through the instruments and calling the play method
    for instrument in instruments:
        instrument.play() # Calls the appropriate play method based on the object type
```

27. What is the Liskov Substitution Principle (LSP)?


Answer: The Liskov Substitution Principle states that objects of a superclass should be **replaceable** with objects of its subclasses without affecting the correctness of the program. Subclasses must maintain the behavior expected by the superclass.

Object Oriented Programming example

1. Create a class "BankAccount" with attributes account number and balance. Implement methods to deposit and withdraw funds, and a display method to show the account details.

Python Code with Simulated Private Variables:

python

 Copy code

```
class BankAccount:
    def __init__(self, account_number, initial_balance):
        self.__account_number = account_number # Private variable for account number
        self.__balance = initial_balance # Private variable for balance

    def deposit(self, amount):
        self.__balance += amount # Adding the deposit amount to balance

    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount # Deducting the withdrawal amount from balance
        else:
            print("Insufficient balance!") # Displaying message if insufficient funds

    def display(self):
        print(f"Account Number: {self.__account_number}") # Printing account number
        print(f"Balance: {self.__balance}") # Printing balance

# Main program
if __name__ == "__main__":
    account = BankAccount("123456789", 1000) # Creating a BankAccount object
    account.deposit(500) # Depositing money
    account.withdraw(200) # Withdrawing money
    account.display() # Displaying account details
```

Explanation:

2. Create a base class "Shape" with methods to calculate the area and perimeter (pure virtual). Implement derived classes "Rectangle" and "Circle" that inherit from "Shape" and provide their own area and perimeter calculations.

In Python, we can achieve the same functionality using **abstract base classes** (similar to abstract classes in Java) and **method overriding**. Python's `abc` module allows us to define abstract base classes and abstract methods, which is equivalent to Java's `abstract` keyword.

Here's the equivalent Python code without any major modification, just like in Java:

Python Code

python

Copy code

```
from abc import ABC, abstractmethod
import math

# Abstract base class
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

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

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

    def perimeter(self):
        return 2 * (self.length + self.width)
```

```
# Circle class inheriting from Shape
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius * self.radius

    def perimeter(self):
        return 2 * math.pi * self.radius

# Main program
if __name__ == "__main__":
    shape1 = Rectangle(5, 3) # Creating a Rectangle object
    shape2 = Circle(4)       # Creating a Circle object

    print("Rectangle Area:", shape1.area())          # Calculating and printing area of Rectangle
    print("Rectangle Perimeter:", shape1.perimeter()) # Calculating and printing perimeter of Rectangle

    print("Circle Area:", shape2.area())              # Calculating and printing area of Circle
    print("Circle Perimeter:", shape2.perimeter())    # Calculating and printing perimeter of Circle
```

3. Create a class "Person" with a static member variable "count" that keeps track of the number of instances created.

In Python, we can create a class with a static variable and static methods just like in Java. We use the `@staticmethod` decorator to define static methods, and static variables are shared across all instances of the class, just like in Java.

```
Python Code

python Copy code

class Person:
    # Static variable to keep track of the number of instances created
    count = 0

    def __init__(self, name):
        self.name = name # Instance variable to store the name
        Person.count += 1 # Increment the static count variable each time an object is created

    @staticmethod
    def get_count():
        return Person.count # Return the static count variable

    def get_name(self):
        return self.name # Return the instance's name

# Main program
if __name__ == "__main__":
    person1 = Person("Alice") # Create the first person instance
    person2 = Person("Bob") # Create the second person instance

    # Display total persons created using the static method
    print("Total Persons:", Person.get_count())


    # Display the names of the persons
    print(person1.get_name())
    print(person2.get_name())
```

4. Create a class "Employee" with attributes name and salary. Implement overloaded operators + and - to combine and compare employees based on their salaries.

In Python, we can use **operator overloading** by implementing special methods like `__add__`, `__sub__`, and comparison methods like `__lt__` (less than) and `__gt__` (greater than). These methods allow us to overload operators like +, -, <, and >.

Python Code

python

 Copy code

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

    def get_salary(self):
        return self.salary

    # Overloading the less than operator '<'
    def __lt__(self, other):
        return self.salary < other.salary

    # Overloading the greater than operator '>'
    def __gt__(self, other):
        return self.salary > other.salary

    # Overloading the addition operator '+'
    def __add__(self, other):
        return Employee("Combined", self.salary + other.salary)

    # Overloading the subtraction operator '-'
    def __sub__(self, other):
        return Employee("Difference", self.salary - other.salary)

# Main program
if __name__ == "__main__":
    emp1 = Employee("Alice", 5000)
    emp2 = Employee("Bob", 6000)
```

```
# Using the overloaded < operator
if emp1 < emp2:
    print("Bob has a higher salary than Alice.")
else:
    print("Alice has a higher salary than Bob.")

# Using the overloaded + operator
combined = emp1 + emp2
print("Combined Salary:", combined.get_salary())

# Using the overloaded - operator
difference = emp1 - emp2
print("Salary Difference:", difference.get_salary())
```

Explanation:

1. Attributes:

- The class `Employee` has two attributes: `name` and `salary`, just like in the Java version.

2. Operator Overloading:


- In Python, we use special methods for operator overloading:
 - `__lt__`: Overloads the `<` (less than) operator, which is used to compare employee salaries.
 - `__gt__`: Overloads the `>` (greater than) operator for comparing salaries.
 - `__add__`: Overloads the `+` operator to combine the salaries of two employees.
 - `__sub__`: Overloads the `-` operator to calculate the difference between salaries.

3. Comparison and Combination:

- The code compares the salaries using `emp1 < emp2` (calls `__lt__` method) and combines salaries using `emp1 + emp2` (calls `__add__` method), similar to the Java code.

Python Code Without Returning an Employee Object

python

 Copy code

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

    def get_salary(self):
        return self.salary


    # Overloading the less than operator '<'
    def __lt__(self, other):
        return self.salary < other.salary

    # Overloading the greater than operator '>'
    def __gt__(self, other):
        return self.salary > other.salary

    # Overloading the addition operator '+'
    def __add__(self, other):
        return self.salary + other.salary # Return the combined salary directly

    # Overloading the subtraction operator '-'
    def __sub__(self, other):
        return self.salary - other.salary # Return the salary difference directly
```

Overloading the subtraction operator '-'

 Copy code

```
def __sub__(self, other):
    return self.salary - other.salary # Return the salary difference directly
```

Main program

```
if __name__ == "__main__":
    emp1 = Employee("Alice", 5000)
    emp2 = Employee("Bob", 6000)

    # Using the overloaded < operator
    if emp1 < emp2:
        print("Bob has a higher salary than Alice.")
    else:
        print("Alice has a higher salary than Bob.")

    # Using the overloaded + operator
    combined_salary = emp1 + emp2
    print("Combined Salary:", combined_salary)


    # Using the overloaded - operator
    salary_difference = emp1 - emp2
    print("Salary Difference:", salary_difference)
```


5. Create a class "Time" with attributes hours and minutes. Implement the << operator to display time in the format "hh:mm".

In Python, the equivalent of the << operator for displaying formatted output is the `__str__` method, which defines how the object is represented when converted to a string. Here's how you can translate the Java `Time` class into Python while keeping the logic and structure similar.

Python Code

python

 Copy code

```
class Time:
    def __init__(self, hours, minutes):
        self.hours = hours
        self.minutes = minutes

    # Overloading the string representation (equivalent to Java's toString())
    def __str__(self):
        return f"{self.hours}:{self.minutes:02d}"

# Main program
if __name__ == "__main__":
    time = Time(14, 30)
    # This will automatically call the __str__ method when printing
    print("Time:", time)
```

20. What is the Open/Closed Principle?

- Answer: The Open/Closed Principle states that software entities (like classes, modules, functions) should be open for extension but closed for modification, promoting flexibility and minimizing potential side effects.

1. What is OOP?

- **Answer:** OOP stands for Object-Oriented Programming. It's a programming paradigm based on the concept of objects.

2. What are the four principles of OOP?

- **Answer:** Encapsulation, Inheritance, Polymorphism, and Abstraction.

3. Define Encapsulation.

- **Answer:** Encapsulation is the bundling of data and methods that manipulate the data, ensuring that the object's internal state remains hidden and protected.

4. Explain Inheritance.

- **Answer:** Inheritance allows a class (subclass) to inherit properties and methods from another class (superclass), promoting code reusability.

5. What is Polymorphism?

- **Answer:** Polymorphism allows objects of different classes to be treated as objects of a common super class, enabling methods to be called on objects without knowing their specific type.

6. Define Abstraction.

- **Answer:** Abstraction is the process of hiding complex implementation details and showing only the essential features of an object.

7. What is a Class and an Object?

- **Answer:** A class is a blueprint for creating objects, while an object is an instance of a class.

8. Explain Constructor and Destructor.

- **Answer:** A constructor initializes an object when it's created, while a destructor is called when an object is destroyed to perform cleanup tasks.

9. What is Method Overloading?

- **Answer:** Method overloading allows a class to have multiple methods with the same name but different parameters.

10. Describe Method Overriding.

- **Answer:** Method overriding occurs when a subclass provides a specific implementation for a method already defined in its superclass.

11. What are Abstract Classes?

- **Answer:** Abstract classes are classes that cannot be instantiated and are meant to be subclassed. They can have abstract methods that are implemented by subclasses.

12. What is the difference between `final`, `finally`, and `finalize`?

- **Answer:** `final` is a keyword used to restrict class inheritance, method overriding, or variable reassignment. `finally` is a block that is executed regardless of exceptions. `finalize` is a method called by the garbage collector before an object is destroyed.

13. Explain Composition.

- **Answer:** Composition is a design principle where a class contains an object of another class, making it a part of its state.

14. What are Interfaces?

Exceptions: `finalize` is a method called by the garbage collector before an object is destroyed.

13. Explain Composition.

- **Answer:** Composition is a design principle where a class contains an object of another class, making it a part of its state.

14. What are Interfaces?

- **Answer:** Interfaces define a contract for classes, specifying methods that implementing classes must provide without dictating the implementation details.

15. What is the Diamond Problem in OOP?

- **Answer:** The Diamond Problem occurs in multiple inheritance when two superclasses of a class have a common base class, leading to ambiguity in method resolution.

16. Describe Encapsulation vs. Abstraction.

- **Answer:** Encapsulation focuses on bundling data and methods, while abstraction focuses on hiding implementation details and showing only essential features.

17. What is a Singleton Pattern?

- **Answer:** Singleton pattern ensures a class has only one instance and provides a global point of access to that instance.

18. Explain Polymorphism with an Example.

- **Answer:** Polymorphism allows a variable of a superclass type to reference a subclass object. For instance, a superclass `Animal` can have a method `makeSound()`, which can be overridden by subclasses like `Dog` or `Cat` to produce different sounds.

19. How do you achieve Data Hiding in OOP?

- **Answer:** Data hiding is achieved through encapsulation, by making class fields private and providing public methods (getters and setters) to access and modify them.

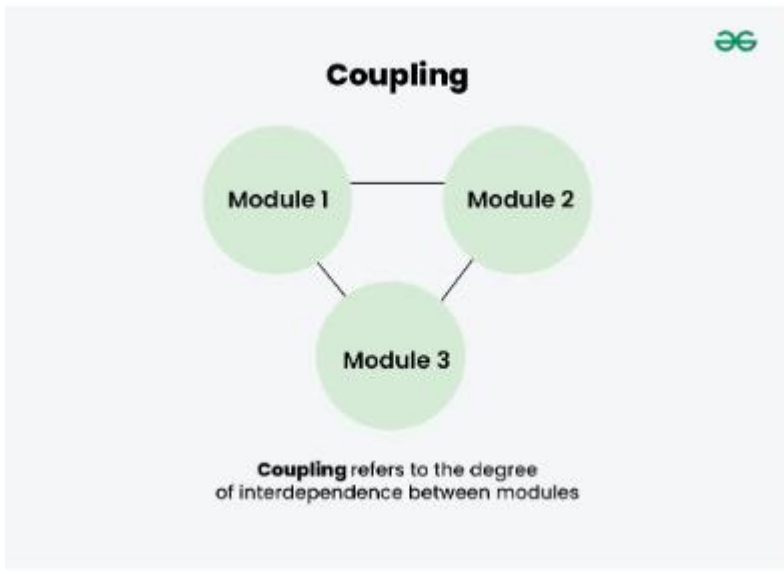
20. What is the Open/Closed Principle?

- **Answer:** The Open/Closed Principle states that software entities (like classes, modules, functions) should be open for extension but closed for modification, promoting flexibility and minimizing potential side effects.

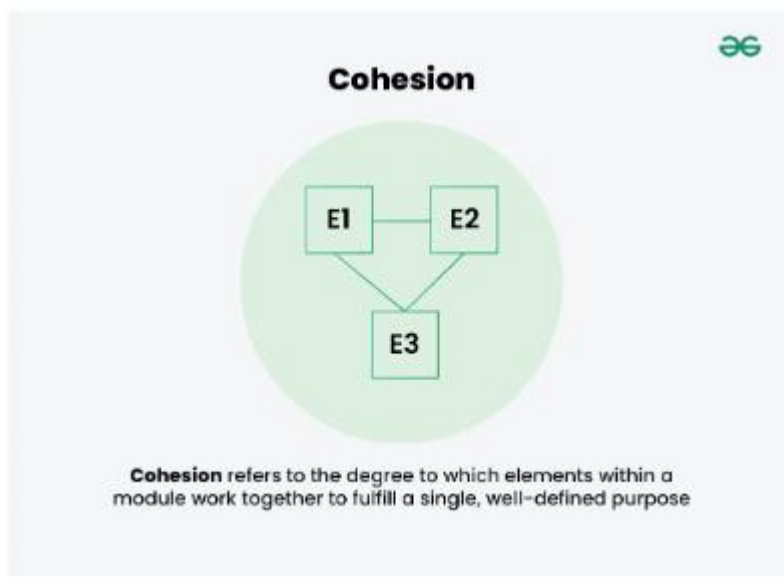
These questions and answers provide a foundational understanding of OOP concepts for interviews.

Coupling and Cohesion – Software Engineering

Coupling refers to the degree of interdependence between software modules. High coupling means that modules are closely connected and changes in one module may affect other modules. Low coupling means that modules are independent, and changes in one module have little impact on other modules.



Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose. High cohesion means that elements are closely related and focused on a single purpose, while low cohesion means that elements are loosely related and serve multiple purposes.



it's good for software to have low coupling and high cohesion. Low coupling means the different parts of the software don't rely too much on each other, which makes it safer to make changes without causing unexpected problems. High cohesion means each part of the software has a clear purpose and sticks to it, making the code easier to work with and reuse. Following these principles helps make software stronger, more adaptable, and easier to grow.

Advantage Python Feature

Python – List Comprehension :

Syntax: newList = [expression(element) for element in oldList if condition]

```
numbers = [12, 13, 14,]
```

```
doubled = [x *2 for x in numbers]
```

```
print(doubled)
```

answer : [1, 4, 9, 16, 25]

```
# Using list comprehension to iterate through loop
```

```
List = [character for character in [1, 2, 3]]
```

```
# Displaying list
```

```
print(List)
```

answer = [1, 2, 3]

Even list using List Comprehension

```
Even_list = [ i for i in range(11) if i%2==0]
```

```
Print(Even_list)
```

Output

[0, 2, 4, 6, 8, 10]

Matrix using List Comprehension

```
matrix = [[j for j in range(3)] for i in range(3)]
```

```
print(matrix)
```

Output

```
[ [0, 1, 2],
```

```
  [0, 1, 2],
```

```
  [0, 1, 2] ]
```

List Comprehensions vs For Loop

```
# Empty list
```

```
List = []
```

```
# Traditional approach of iterating
```

```
for character in 'Geeks 4 Geeks!':
```

```
    List.append(character)
```

```
# Display list
```

```
print(List)
```

```
# Using list comprehension to iterate through loop
```

```
List = [character for character in 'Geeks 4 Geeks!']
```

```
# Displaying list
```

```
print(List)
```

Output

```
['G', 'e', 'e', 'k', 's', ' ', '4', ' ', 'G', 'e', 'e', 'k', 's', '!']
```

Nested List Comprehensions in Python

Syntax: new_list = [[expression for item in list] for item in list]

```
matrix = []
```

```
for i in range(3): # Append an empty sublist inside the list
```

```
    matrix.append([])
```

```
    for j in range(5):
```

```
        matrix[i].append(j)
```

```
print(matrix)
```

Output

```
[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
```

```
# Nested list comprehension
```

```
matrix = [[j for j in range(5)] for i in range(3)]
```

```
print(matrix)
```

Flattening Nested Sub-Lists

```
# 2-D List
```

```
matrix = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]
```

```
flatten_matrix = []
```

```
for sublist in matrix:
```

```
for val in sublist:
```

```
    flatten_matrix.append(val)
```

```
print(flatten_matrix)
```

Output

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2-D List

```
matrix = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]
```

Nested List Comprehension to flatten a given 2-D matrix

```
flatten_matrix = [val for sublist in matrix for val in sublist]
```

```
print(flatten_matrix)
```

Reverse each string in a Tuple

Reverse each string in tuple

```
List = [string[::-1] for string in ('Geeks', 'for', 'Geeks')]
```

Display list

```
print(List)
```

Output

```
['skeeG', 'rof', 'skeeG']
```

Python Dictionary Comprehension

A dictionary comprehension takes the form : *{key: value for (key, value) in iterable}*

Lists to represent keys and values

```
keys = ['a','b','c','d','e']
```

```
values = [1,2,3,4,5]
```

but this line shows dict comprehension here

```
myDict = { k:v for (k,v) in zip(keys, values)}
```

We can use below too

```
# myDict = dict(zip(keys, values))
```

```
print (myDict)
```

Output :

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

Using dictionary comprehension make dictionary

creation using list comprehension

```
myDict = {x: x**2 for x in [1,2,3,4,5]}
```

```
print (myDict)
```

Output :

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

```
Dict = {x.upper(): x*3 for x in 'coding '}
```

```
print (Dict)
```

Output :

```
{'O': 'ooo', 'N': 'nnn', 'l': 'iii', 'C': 'ccc', 'D': 'ddd', 'G': 'ggg'}
```

comprehension using if.

```
newdict = {x: x**3 for x in range(10) if x**3 % 4 == 0}
```

```
print(newdict)
```

Output :

```
{0: 0, 8: 512, 2: 8, 4: 64, 6: 216}
```

Dictionaries in Python

```
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
```

```
print(Dict)
```

Output

```
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
```

Python Dictionary Syntax

```
dict_var = {key1 : value1, key2 : value2, .....}
```

How to Create a Dictionary

```
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
```

```
print("\nDictionary with the use of Integer Keys: ")
```

```
print(Dict)
```

```
Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}
```

```
print("\nDictionary with the use of Mixed Keys: ")
```

```
print(Dict)
```

Output

Dictionary with the use of Integer Keys:

```
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
```

Dictionary with the use of Mixed Keys:

```
{'Name': 'Geeks', 1: [1, 2, 3, 4]}
```

```
Dict = {} or Dict = dict()
```

```
print("Empty Dictionary: ")
```

```
print(Dict)
```

```
Dict = dict({1: 'Geeks', 2: 'For', 3: 'Geeks'})
```

```
print("\nDictionary with the use of dict(): ")
```

```
print(Dict)
```

```
Dict = dict([(1, 'Geeks'), (2, 'For')])
```

```
print("\nDictionary with each item as a pair: ")
```

```
print(Dict)
```

Output

Empty Dictionary:

```
{}
```

Dictionary with the use of dict():

```
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
```

Dictionary with each item as a pair:

```
{1: 'Geeks', 2: 'For'}
```

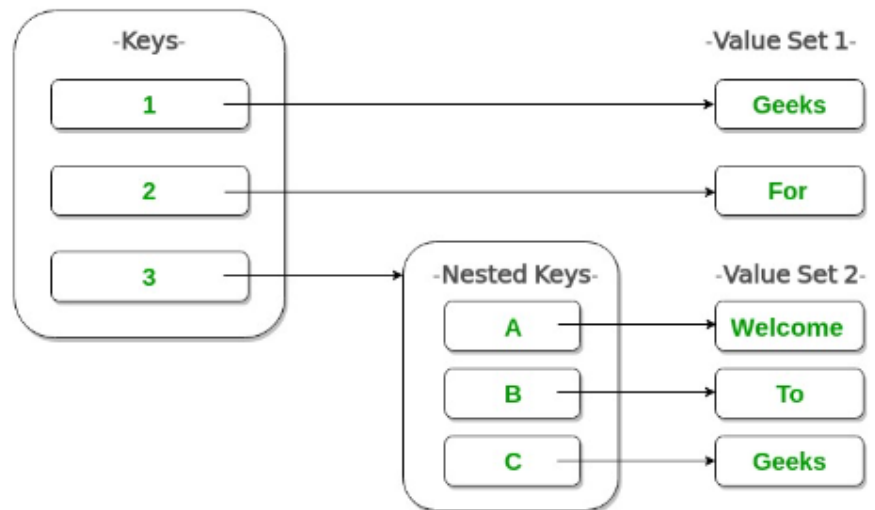


```
Dict = {1: 'Geeks', 2: 'For',
        3: {'A': 'Welcome', 'B': 'To', 'C': 'Geeks'}}
print(Dict)
```

Output

```
{1: 'Geeks', 2: 'For', 3: {'A': 'Welcome', 'B': 'To', 'C': 'Geeks'}}
```


Nested Dictionaries



Accessing an Element of a Nested Dictionary

Python

```
Dict = {'Dict1': {1: 'Geeks'},
        'Dict2': {'Name': 'For'}}


print(Dict['Dict1'])
print(Dict['Dict1'][1])
print(Dict['Dict2']['Name'])
```

Output

```
{1: 'Geeks'}
Geeks
For
```

Using a regular dictionary ({}):

python

 Copy code

```
# Using a regular dictionary
names_by_letter = {}

# Adding names to the dictionary
names = ["Alice", "Bob", "Charlie", "Anna", "Brian"]

for name in names:
    first_letter = name[0] # Get the first letter of each name


    # Check if the key exists, if not, initialize with an empty list
    if first_letter not in names_by_letter:
        names_by_letter[first_letter] = []

    # Append the name to the corresponding list
    names_by_letter[first_letter].append(name)

print(names_by_letter)
```

Output:


python

 Copy code

```
{ 'A': ['Alice', 'Anna'], 'B': ['Bob', 'Brian'], 'C': ['Charlie']}
```

Using a regular dictionary ({}):

python

 Copy code

```
# Using a regular dictionary
char_count = {}


# Adding elements and counting occurrences
for char in "hello world":
    # Check if the character is already a key in the dictionary
    if char not in char_count:
        char_count[char] = 0 # Initialize with 0 if the character is not present

    # Increment the count for the character
    char_count[char] += 1

print(char_count)
```

Output:

python

 Copy code

```
{ 'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1 }
```

Python map() function

General Rule of map() Function: it donot return list. So , use list function to convert list

• Syntax:

python

Copy code

```
map(function, iterable)
```

- **function** : A function that defines how you want to transform the items.
- **iterable** : A list, tuple, or any iterable containing the elements you want to apply the function to.

1. Without map() (manual stamping):

Imagine you have a list of numbers and you want to double each number.

python

Copy code

```
# Without map()
numbers = [1, 2, 3, 4, 5]
doubled_numbers = []

for number in numbers:
    doubled_numbers.append(number * 2)

print(doubled_numbers) # Output: [2, 4, 6, 8, 10]
```

2. With map() (automated stamping):

Now, imagine you automate this process using the `map()` function, just like you would use an automated stamping machine in a factory.

python

Copy code

```
# Using map()
numbers = [1, 2, 3, 4, 5]

# Define a function to double a number
def double(x):
    return x * 2

# Apply the function to all elements using map()
doubled_numbers = map(double, numbers)

# Convert map object to a list
print(list(doubled_numbers)) # Output: [2, 4, 6, 8, 10]
```

Return double of n

```
def addition(n):
    return n + n
```

```
# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

Double all numbers using map and lambda

```
numbers = (1, 2, 3, 4)
result = map(lambda x: x + x, numbers)
print(list(result))
```

Output

[2, 4, 6, 8]

Output

[2, 4, 6, 8]

Add two lists using map and lambda

```
numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]
```

```
result = map(lambda x, y: x + y, numbers1, numbers2)
print(list(result))
```

Output

[5, 7, 9]

Lambda function in python

Python lambda Syntax:

```
lambda arguments : expression
```

Python lambda Example:

Python

```
calc = lambda num: "Even number" if num % 2 == 0 else "Odd number"
print(calc(20))
```

Output:

Even number

Difference between lambda and normal function call

Python

```
def cube(y):
    print(f"Finding cube of number:{y}")
    return y * y * y

lambda_cube = lambda num: num ** 3

# invoking simple function
print("invoking function defined with def keyword:")
print(cube(30))
# invoking lambda function
print("invoking lambda function:", lambda_cube(30))
```

Output:

```
invoking function defined with def keyword:
Finding cube of number:30
27000
invoking lambda function: 27000
```

Python

```
# Example list
my_list = [1, 2, 3, 4, 5]

# Use lambda to filter out even numbers from the list
new_list = list(filter(lambda x: x % 2 != 0, my_list))

# Print the new list
print(new_list)
```

Output

[1, 3, 5]

Filter in python

The filter() method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

Python filter() Syntax

The filter() method in [Python](#) has the following syntax:

Syntax: filter(function, sequence)

Parameters:

- **function:** function that tests if each element of a sequence is true or not.
- **sequence:** sequence which needs to be filtered, it can be sets, lists, tuples, or containers of any iterators.

Returns: an iterator that is already filtered.

Python Filter Function with a Custom Function

```
# function that filters vowels
def fun(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if (variable in letters):
        return True
    else:
        return False

# sequence
sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']

# using filter function
filtered = filter(fun, sequence)

print('The filtered letters are:')
for s in filtered:
    print(s)
```

The filtered letters are:
e
e

Filter Function in Python with Lambda

Python

```
# a list contains both even and odd numbers.
seq = [0, 1, 2, 3, 5, 8, 13]

# result contains odd numbers of the list
result = filter(lambda x: x % 2 != 0, seq)
print(list(result))

# result contains even numbers of the list
result = filter(lambda x: x % 2 == 0, seq)
print(list(result))
```

Output:

```
[1, 3, 5, 13]
[0, 2, 8]
```

Filter Function in Python with Lambda and Custom Function

Python

```
# Define a function to check
# if a number is a multiple of 3
def is_multiple_of_3(num):
    return num % 3 == 0

# Create a list of numbers to filter
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Use filter and a lambda function to
# filter the list of numbers and only
# keep the ones that are multiples of 3
result = list(filter(lambda x: is_multiple_of_3(x), numbers))

# Print the result
print(result)
```

Output

```
[3, 6, 9]
```

How to filter a list in Python?

To filter a list in Python, you can use a list comprehension or the `filter()` function along with a lambda function or a defined function.

Using list comprehension:

```
# Example using list comprehension
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
filtered_numbers = [num for num in numbers if num % 2 == 0]
print(filtered_numbers) # Output: [2, 4, 6, 8]
```

Using filter() with a lambda function:

```
# Example using filter() with lambda function
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
filtered_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(filtered_numbers) # Output: [2, 4, 6, 8]
```

How to filter an object in Python?

Filtering an object in Python typically involves filtering its attributes or properties based on certain conditions. Here's an example using `filter()` and a lambda function to filter a list of objects based on an attribute:

```
# Example filtering objects based on attribute
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
people = [
    Person('Alice', 25),
    Person('Bob', 30),
    Person('Charlie', 22)
]
# Filter people older than 25
filtered_people = list(filter(lambda person: person.age > 25, people))
for person in filtered_people:
    print(person.name, person.age)
```

How to filter a text in Python?

Filtering text in Python usually involves searching for specific patterns or conditions within strings. Here's an example using list comprehension to filter out words containing a specific substring:

```
# Example filtering text using list comprehension
text = "Python is awesome and versatile"
filtered_words = [word for word in text.split() if 'o' in word]
print(filtered_words) # Output: ['Python', 'awesome']
```

ZIP FUNCTION IN PYTHON

Python zip() Syntax

It is used to map the similar index of multiple containers so that they can be used just using a single entity.

Syntax : `zip(*iterators)`

Parameters : Python iterables or containers (list, string etc)

Return Value : Returns a single iterator object.

zip() in Python Examples

Python

```
name = [ "Manjeet", "Nikhil", "Shambhavi", "Asta" ]
roll_no = [ 4, 1, 3, 2 ]

# using zip() to map values
mapped = zip(name, roll_no)

print(set(mapped))
```

Output

```
{('Nikhil', 1), ('Shambhavi', 3), ('Manjeet', 4), ('Asta', 2)}
```

Python

```
names = ['Mukesh', 'Roni', 'Chari']
ages = [24, 50, 18]

for i, (name, age) in enumerate(zip(names, ages)):
    print(i, name, age)
```

Output

```
0 Mukesh 24
1 Roni 50
2 Chari 18
```

Python zip() with Dictionary

Python

```
stocks = ['GEEKS', 'For', 'geeks']
prices = [2175, 1127, 2750]

new_dict = {stocks: prices for stocks,
            prices in zip(stocks, prices)}
print(new_dict)
```

Output

```
{'GEEKS': 2175, 'For': 1127, 'geeks': 2750}
```

Python zip() with Tuple

When used with tuples, `zip()` works by pairing the elements from tuples based on their positions. The resulting iterable contains tuples where the *i*-th tuple contains the *i*-th element from each input tuple.

Python

```
tuple1 = (1, 2, 3)
tuple2 = ('a', 'b', 'c')
zipped = zip(tuple1, tuple2)
result = list(zipped)
print(result)
```

Output:

```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Python zip() with Multiple Iterables

Python's `zip()` function can also be used to combine multiple iterables and return an iterable of tuples, where each tuple contains elements from each iterable.

Zippping lists of unequal size

The `zip()` function will only iterate over the smallest list passed. If given lists of different lengths, the resulting combination will only be as long as the smallest list passed. In the following code example:

Python

```
# Define lists for 'persons', 'genders', and a tuple for 'ages'
persons = ['Chandler', 'Monica', 'Ross', 'Rachel', 'Joey', 'Phoebe', 'Joanna']
genders = ['Male', 'Female', 'Male', 'Female', 'Male', 'Female', 'Female']
ages = (35, 36, 38, 34)

# Create a zipped object combining the 'persons' and 'genders'
# Lists along with the 'ages' tuple
zipped_result = zip(persons, genders, ages)

# Print the zipped object
print("Zipped result as a list:")
for i in list(zipped_result):
    print(i)
```

Output

```
Zipped result as a list:
('Chandler', 'Male', 35)
('Monica', 'Female', 36)
('Ross', 'Male', 38)
('Rachel', 'Female', 34)
```

Output

```
[(1, 'a', 'x'), (2, 'b', 'y'), (3, 'c', 'z')]
```

How to unzip a list of tuples in Python?

To unzip a list of tuples, you can use the `zip()` function with the unpacking operator `*`. This will separate the tuples back into individual lists.

Example:

```
zipped = [(1, 'a', True), (2, 'b', False), (3, 'c', True)]

list1, list2, list3 = zip(*zipped)
print(list1)
print(list2)
print(list3)
```

Output:

```
(1, 2, 3)
('a', 'b', 'c')
(True, False, True)
```

To convert the resulting tuples into lists:

```
list1, list2, list3 = map(list, zip(*zipped))
print(list1)
print(list2)
print(list3)
```

Output:

```
[1, 2, 3]
['a', 'b', 'c']
[True, False, True]
```

Using zip() with Python Loops

There are many possible applications that can be said to be executed using zip, be it **student database** or **scorecard** or any other utility that requires mapping of groups. A small example of a scorecard is demonstrated below.

Python

```
# Python code to demonstrate the application of
# zip()

# initializing list of players.
players = ["Sachin", "Sehwag", "Gambhir", "Dravid", "Raina"]

# initializing their scores
scores = [100, 15, 17, 28, 43]

# printing players and scores.
for pl, sc in zip(players, scores):
    print("Player : %s    Score : %d" % (pl, sc))
```

Output

```
Player : Sachin    Score : 100
Player : Sehwag    Score : 15
Player : Gambhir    Score : 17
Player : Dravid    Score : 28
Player : Raina     Score : 43
```


Python enumerate function

Syntax: `enumerate(iterable, start=0)`

Parameters:

- **Iterable:** any object that supports iteration
- **Start:** the index value from which the counter is to be started, by default it is 0

Return: Returns an iterator with index and element pairs from the original iterable

```
l1 = ["eat", "sleep", "repeat"]
s1 = "geek"

# creating enumerate objects
obj1 = enumerate(l1)
obj2 = enumerate(s1)

print ("Return type:", type(obj1))
print (list(enumerate(l1)))

# changing start index to 2 from 0
print (list(enumerate(s1, 2)))
```

Output:

```
Return type: <class 'enumerate'>
[(0, 'eat'), (1, 'sleep'), (2, 'repeat')]
[(2, 'g'), (3, 'e'), (4, 'e'), (5, 'k')]
```

Using Enumerate Object in Loops :

Python

```
l1 = ["eat", "sleep", "repeat"]

# printing the tuples in object directly
for ele in enumerate(l1):
    print (ele)

# changing index and printing separately
for count, ele in enumerate(l1, 100):
    print (count, ele)

# getting desired output from tuple
for count, ele in enumerate(l1):
    print(count)
    print(ele)
```

Output:

```
(0, 'eat')
(1, 'sleep')
(2, 'repeat')
100 eat
101 sleep
102 repeat
0
eat
1
sleep
2
repeat
```

How to use enumerate in Python?

Enumerate in Python is used to loop over an iterable and automatically provide an index for each item. It returns tuples with the index and corresponding value from the iterable. Example:

```
my_list = ['apple', 'banana', 'cherry']
for index, value in enumerate(my_list):
    print(index, value)

Output:
0 apple
1 banana
2 cherry
```

This allows you to easily access both the index and the value of each item in the iterable during iteration.

How to enumerate in reverse in Python?

To enumerate in reverse in Python, you can combine enumerate with reversed. Here's an example:

```
my_list = ['apple', 'banana', 'cherry']
for index, value in enumerate(reversed(my_list)):
    print(len(my_list) - index - 1, value)

Output:
2 cherry
1 banana
0 apple
```

In this example, `reversed(my_list)` reverses the order of elements in `my_list`, and `len(my_list) - index - 1` calculates the index in reverse order.

Regular Expression (RegEx) in Python with Examples


A Regular Expression or RegEx is a special sequence of characters that uses a search pattern to find a string or set of strings.

Real-Life Example

Let's say you're building a system that validates email addresses when users sign up. You want to ensure they provide a valid email like `username@domain.com`. You can use a regex pattern to check whether the provided email matches the expected format.

Example Code

python

 Copy code

```
import re

# Define a regex pattern for validating an email address
email_pattern = r'^[a-zA-Z0-9_+-.]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-]+$'

def validate_email(email):
    if re.match(email_pattern, email):
        return "Valid email address"
    else:
        return "Invalid email address"

# Test cases
emails = ["user@example.com", "invalid-email@", "john.doe@site"]

for email in emails:
    print(f"{email}: {validate_email(email)}")
```



Real-Life Use Cases of Regular Expressions:

1. **Form validation:** Validate user input like phone numbers, emails, or postal codes.
2. **Log file parsing:** Extract specific patterns, such as error codes or timestamps, from server logs.
3. **Web scraping:** Find specific content like URLs, dates, or product prices on a webpage.
4. **Data cleaning:** Remove or replace unwanted characters from large datasets, such as extra spaces or special symbols.

PYTHON GENERATOR

Generator Function in Python

Create a Generator in Python

In Python, we can create a generator function by simply using the `def` keyword and the `yield` keyword. The generator has the following syntax in [Python](#):

```
def function_name():  
    yield statement
```

Python

[Learn More or Try Now](#)



```
# A generator function that yields 1 for first time,  
# 2 second time and 3 third time  
def simpleGeneratorFun():  
    yield 1  
    yield 2  
    yield 3  
  
# Driver code to check above generator function  
for value in simpleGeneratorFun():  
    print(value)
```

Output:

```
1  
2  
3
```

```
def count_up_to(max_value):  
    count = 1  
    while count <= max_value:  
        yield count  
        count += 1  
  
# Using the generator  
counter = count_up_to(5)  
for number in counter:  
    print(number)
```

In this example, `count_up_to` is a generator function that yields numbers from 1 up to a specified maximum value.


Advantages of Generators

- Memory Efficiency:** Generators are much more memory efficient than lists, especially when dealing with large data sets. They generate values on the fly and do not require the entire sequence to be stored in memory.
- Lazy Evaluation:** Generators compute values only when needed, which can improve performance and responsiveness in applications where the full sequence of values is not required all at once.
- Readable and Concise Code:** Generators can lead to more readable and concise code, particularly in situations where you need to create iterators or perform complex data processing in a step-by-step manner.


Python Closures

Nested functions in Python


Python3




Python program to illustrate
nested functions



def outerFunction(text):



def innerFunction():



print(text)

innerFunction()

if __name__ == '__main__':

outerFunction('Hey!')

Output:

Hey!

Example of a Closure in Python

Let's implement a simple example where a closure captures a variable from the outer function:

```
python Copy code
```

```
def coffee_shop(order):  
    def make_coffee():  
        return f"Your coffee is ready: {order}"  
    return make_coffee  
  
# Create two different coffee orders  
latte_order = coffee_shop("Latte with one sugar")  
cappuccino_order = coffee_shop("Cappuccino with no sugar")  
  
# Even after the coffee_shop function is done, the make_coffee function remembers the order  
print(latte_order()) # Output: Your coffee is ready: Latte with one sugar  
print(cappuccino_order()) # Output: Your coffee is ready: Cappuccino with no sugar
```

Explanation:

Python Closures: Closures can be used to avoid global values and provide data hiding, and can be an elegant solution for simple cases with one or few methods.

Python closure is a nested function that allows us to access variables of the outer function even after the outer function is closed.

Nested function in Python

In Python, we can create a function inside another function. This is known as a nested function. For example,

```
def greet(name):  
    # inner function  
    def display_name():  
        print("Hi", name)  
  
    # call inner function  
    display_name()  
  
# call outer function  
greet("John")  
  
# Output: Hi John
```

Run Code >>

Explain :

message = greet()

The returned function is now assigned to the message variable. At this point, the execution of the outer function is completed, so the name variable should be destroyed. However, when we call the anonymous function using

`print(message())` we are able to access the `name` variable of the outer function. It's possible because the nested function now acts as a closure that closes the outer scope variable within its [scope](#) even after the outer function is executed.

Example: Print Odd Numbers using Python Closure

```
def calculate():
    num = 1
    def inner_func():
        nonlocal num
        num += 2
        return num
    return inner_func

# call the outer function
odd = calculate()

# call the inner function
print(odd())
print(odd())
print(odd())

# call the outer function again
odd2 = calculate()
print(odd2())
```

Output

```
3
5
7
3
```

In the above example,

```
odd = calculate()
```

This code executes the outer function `calculate()` and returns a closure to the `odd` number. T

That's why we can access the `num` variable of `calculate()` even after completing the outer function.

Again, when we call the outer function using

```
odd2 = calculate()
```

a new closure is returned. Hence, we get **3** again when we call `odd2()`.

DECORATOR IN PYTHON

Python Decorators

In Python, a **decorator** is a design pattern that allows you to modify the functionality of a **function** by wrapping it in another function.

The outer function is called the decorator, which takes the **original function as an argument** and returns a **modified version** of it.

Nested Function

We can include one function inside another, known as a nested function. For example,

```
def outer(x):
    def inner(y):
        return x + y
    return inner

add_five = outer(5)
result = add_five(6)
print(result) # prints 11

# Output: 11
```

Run Code »

Here, we have created the `inner()` function inside the `outer()` function.

Pass Function as Argument

We can pass a function as an argument to another function in Python. For Example,

```
def add(x, y):
    return x + y

def calculate(func, x, y):
    return func(x, y)

result = calculate(add, 4, 6)
print(result) # prints 10
```

Run Code »

Output

10

In the above example, the `calculate()` function takes a function as its argument. While calling `calculate()`, we are passing the `add()` function as the argument.

In the `calculate()` function, arguments: `func`, `x`, `y` become `add`, `4`, and `6` respectively.

Return a Function as a Value

In Python, we can also return a function as a return value. For example,

```
def greeting(name):
    def hello():
        return "Hello, " + name + "!"
    return hello

greet = greeting("Atlantis")
print(greet()) # prints "Hello, Atlantis!"

# Output: Hello, Atlantis!
```

Run Code »

In the above example, the `return hello` statement returns the inner `hello()` function. This function is now assigned to the `greet` variable.

That's why, when we call `greet()` as a function, we get the output.

Basically, a decorator takes in a function, adds some functionality and returns it.

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner

def ordinary():
    print("I am ordinary")

# Output: I am ordinary
```

Run Code »

Here, we have created two functions:

- `ordinary()` that prints "I am ordinary"
- `make_pretty()` that takes a function as its argument and has a nested function named `inner()`, and returns the inner function.

We are calling the `ordinary()` function normally, so we get the output "I am ordinary". Now, let's call it using the decorator function.

```
def make_pretty(func):
    # define the inner function
    def inner():
        # add some additional behavior to decorated function
        print("I got decorated")

        # call original function
        func()

    # return the inner function
    return inner

# define ordinary function
def ordinary():
    print("I am ordinary")

# decorate the ordinary function
decorated_func = make_pretty(ordinary)

# call the decorated function
decorated_func()
```

Run Code »

Output

I got decorated
I am ordinary

In the example shown above, `make_pretty()` is a decorator. Notice the code,

```
decorated_func = make_pretty(ordinary)
```

- We are now passing the `ordinary()` function as the argument to the `make_pretty()`.
- The `make_pretty()` function returns the inner function, and it is now assigned to the `decorated_func` variable.

```
decorated_func()
```

Here, we are actually calling the `inner()` function, where we are printing

@ Symbol With Decorator

Instead of assigning the function call to a **variable**, Python provides a much more elegant way to achieve this functionality using the `@` symbol. For example,

```
def make_pretty(func):  
    def inner():  
        print("I got decorated")  
        func()  
    return inner  
  
@make_pretty  
def ordinary():  
    print("I am ordinary")  
  
ordinary()
```

Run Code >>

Output

```
I got decorated  
I am ordinary
```

Here, the `ordinary()` function is decorated with the `make_pretty()` decorator using the `@make_pretty` syntax, which is equivalent to calling `ordinary = make_pretty(ordinary)`.

FOR @PROPERTY IN PYTHON DETAILS : <https://www.programiz.com/python-programming/methods/built-in/property>

@staticmethod in python details : <https://www.programiz.com/python-programming/methods/built-in/staticmethod>

@classmethod in python details : <https://www.programiz.com/python-programming/methods/built-in/classmethod>

difference between @classmethod and @staticmethod with example and real life analogy sothat i can understand easily in python . important :

In Python, `@classmethod` and `@staticmethod` are two types of methods that behave differently. Let me explain both with a real-life analogy to make it clearer.

1. `@classmethod`:

- A class method takes `cls` (the class itself) as its first parameter.
- It can modify class-level attributes, but it cannot directly modify instance-level attributes.
- It is called on the class rather than an instance, but it can still access and modify class variables.

Example:

```
python Copy code

class Car:
    total_cars = 0 # Class variable

    def __init__(self, model):
        self.model = model # Instance variable
        Car.total_cars += 1

    @classmethod
    def car_count(cls):
        return f"Total cars created: {cls.total_cars}"

# Using class method
print(Car.car_count()) # Output: Total cars created: 0

car1 = Car("Tesla")
car2 = Car("BMW")
print(Car.car_count()) # Output: Total cars created: 2
```

Real-life analogy for `@classmethod`:

Imagine you're working at a car factory. The `Car` class is like the factory itself, and `car_count` is a method that counts how many cars (like Tesla or BMW) the factory has produced. The class method doesn't care about specific cars; it only cares about the total production. Similarly, `@classmethod` only interacts with the factory (class) and not individual cars (instances).

2. `@staticmethod`:

- A static method doesn't take `self` or `cls` as a parameter.
- It behaves like a regular function but belongs to the class's namespace.
- It cannot modify class-level or instance-level data.

Example:

```
python Copy code

class Car:
    @staticmethod
    def make_sound():
        return "Vroom Vroom!"

# Using static method
print(Car.make_sound()) # Output: Vroom Vroom!

car1 = Car()
print(car1.make_sound()) # Output: Vroom Vroom!
```

Real-life analogy for `@staticmethod`:

Now, imagine the `make_sound` method as the sound a car makes (e.g., "Vroom Vroom!"). This sound is universal and independent of the factory or any specific car. It doesn't care whether it's coming from a Tesla or a BMW. Any car can use this sound, and it doesn't change based on the number of cars produced. Similarly, `@staticmethod` is a method that's independent of both the class and instances.

Key Differences:

- `@classmethod` knows about the class it was called on and can modify class-level variables.
- `@staticmethod` is just like a regular function that doesn't care about class or instance data.

Simple summary analogy:

Think of a car factory. The class method is like a manager who tracks how many cars the factory has produced (class-level behavior). The static method is like a universal feature all cars have (e.g., a horn sound), which doesn't depend on any specific car or factory.