Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (child or derived class) to inherit properties and behaviors (attributes and methods) from another class (parent or base class). This promotes code reuse and organization.

1. Single Inheritance

In single inheritance, a child class inherits from a single parent class.

Example:

```
#Parent class (Base class)
class Animal:
    def sound(self):
        return "Animal sound"
#Child class (Derived class)
class Dog(Animal):
    def bark(self):
        return "Dog barking sound" #Creating an object of Dog class
dog = Dog()
print(dog.sound())  #Inherited method from Animal class
print(dog.bark())   #Method of Dog class
```

Explanation:

- The `Animal` class is the base (parent) class, which has a method `sound()`.

- The `Dog` class is the derived (child) class, which inherits the `sound()` method from `Animal` and defines its own method `bark()`.

2. Multilevel Inheritance

In multilevel inheritance, a class inherits from a derived class, creating a chain of inheritance.

Example:

```
#Parent class (Base class)
class Animal:
    def sound(self):
        return "Animal sound"
#Child class (Derived class)
class Dog(Animal):
    def bark(self):
        return "Dog barking sound"
#Another derived class (inherits from Dog class)
class Puppy(Dog):
    def play(self):
        return "Puppy plays"
#Creating an object of Puppy class
```

```
puppy = Puppy()
print(puppy.sound())  # Inherited from Animal class
print(puppy.bark())   #Inherited from Dog class
print(puppy.play())   #Method of Puppy class
```

Explanation:

- `Animal` is the base class.
- `Dog` is derived from `Animal`.
- `Puppy` is derived from `Dog`. It inherits methods from both the `Dog` and `Animal` classes.

3. Hierarchical Inheritance

In hierarchical inheritance, multiple child classes inherit from a single parent class.

Example:

```
#Base class
class Animal:
    def sound(self):
        return "Animal sound"
#Child class (Derived class)
class Dog(Animal):
    def bark(self):
        return "Dog barking sound"
```

#Derived class 2 (inherits from Animal)

```
class Cat(Animal):
    def shout(self):
        return "Cat meows"
```

#Creating objects of Dog and Cat classes

```
dog = Dog()
cat = Cat()
print(dog.sound())  #Inherited from Animal
print(dog.bark())   #Method of Dog
print(cat.sound())  #Inherited from Animal
print(cat.shout())   #Method of Cat
```

Explanation:

- `Animal` is the base class.

- Both `Dog` and `Cat` are derived from `Animal`, so they inherit the `sound()` method.

4. Multiple Inheritance

In multiple inheritance, a class inherits from more than one parent class.

Example:

```
#Base class 1
class Animal:
    def sound(self):
        return "Animal sound"
#Base class 2
class Pet:
    def owner(self):
        return "Has an owner"
#Derived class (inherits from both Animal and Pet)
class Dog(Animal, Pet):
    def bark(self):
        return "Dog barks"
#Creating an object of Dog class
dog = Dog()
print(dog.sound())  #Inherited from Animal class1
print(dog.owner())  #Inherited from Pet class2
print(dog.bark())   #Method of Dog class
```

Explanation:

- The `Dog` class inherits from both `Animal` and `Pet` classes, so it can access methods from both parent classes.

5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance. It's generally a mix of hierarchical and multiple inheritance.

Example:

```
#Base class
class Animal:
    def sound(self):
        return "Animal sound"
#Derived class 1 (inherits from Animal)
class Dog(Animal):
    def bark(self):
        return "Dog barks"
#Derived class 2 (inherits from Animal)
class Cat(Animal):
    def shout(self):
        return "Cat meows"
#Derived class (inherits from Dog and Cat)
class PetStore(Dog, Cat):
    def show(self):
```

```
    return "Pet store has many animals"


#Creating an object of PetStore class
store = PetStore()
print(store.sound())  #Inherited from Animal class(Base class)
print(store.bark())   #Inherited from Dog class(Derived class 1)
print(store.shout())   #Inherited from Cat(Derived class 2)
print(store.show())   #Method of PetStore(Derived class)
```

Explanation:

- `PetStore` inherits from both `Dog` and `Cat`, which in turn inherit from `Animal`, demonstrating a hybrid combination of multiple and hierarchical inheritance.

Key Points:

- **Single Inheritance:** One parent, one child.
- **Multilevel Inheritance:** Inheritance chain,

    grandparent → parent → child.

- **Hierarchical Inheritance:** One parent, multiple children.
-**Multiple Inheritance:** One child, multiple parents.
- **Hybrid Inheritance:** A mix of different Inheritance types.

Encapsulation in Python:

Encapsulation is one of the key principles of object-oriented programming (OOP).

It refers to the bundling of data (variables) and methods (functions) that operate on the data into a single unit, or class.

Encapsulation also helps in restricting access to some of the object's attributes and methods from outside the class, which is known as **Data Hiding.**

In Python, encapsulation can be achieved using:

- Public members (accessible everywhere).

- Protected members (accessible only within the class and its subclasses).

- Private members (accessible only within the class).

Step-by-Step Explanation of Encapsulation with Examples

1. Public Members

Public members are accessible from outside the class. By default, all class members (attributes and methods) are public in Python.

Example:

```
class Person:
    def __init__(self, name, age):
        # Public attributes
        self.name = name
        self.age = age

    # Public method
    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Creating an object of the Person class
person1 = Person("Alice", 30)

# Accessing public attributes
print(person1.name)  # Accessing public attribute directly
print(person1.age)

# Calling public method
person1.display_info()
```

Explanation:

- Here, `name` and `age` are public attributes, and `display_info()` is a public method. They can be accessed directly from outside the class.

Output:

Alice

30

Name: Alice, Age: 30

2. Protected Members

Protected members are denoted by a **single underscore** (`_`) before the variable or method name. These are meant to be accessed only within the class and its subclasses. However, in Python, this is just a convention and not strictly enforced. It means you "shouldn't" access them from outside the class, but you **can**.

Example:

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

    # Public method to access protected members
    def display_info(self):
        print(f"Name: {self._name}, Age: {self._age}")

# Creating an object of the Person class
person1 = Person("Bob", 25)

# Accessing protected attributes (not recommended, but possible)
print(person1._name)
print(person1._age)

# Calling public method
person1.display_info()
```

Explanation:

- The `_name` and `_age` are protected attributes. The leading underscore suggests that these attributes are not meant to be accessed directly outside the class.

- We can still access them from outside the class, but this violates the encapsulation principle.

Output:

Bob

25

Name: Bob, Age: 25

3. Private Members

Private members are denoted by a **double underscore** (`__`) before the variable or method name. These members are accessible only within the class and cannot be accessed directly from outside.

Example:

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

    # Public method to access private members
    def display_info(self):
        print(f"Name: {self.__name}, Age: {self.__age}")

    # Method to set private attributes
    def set_name(self, new_name):
        self.__name = new_name

# Creating an object of the Person class
person1 = Person("Charlie", 28)

# Trying to access private attributes directly (this will raise an
AttributeError)
```

```
# print(person1.__name)  # Uncommenting this line will cause an error

# Accessing private attributes using a public method
person1.display_info()

# Modifying private attribute via a public method
person1.set_name("David")
person1.display_info()
```

Explanation:

- The `__name` and `__age` are private attributes. These cannot be accessed directly from outside the class. This enforces the encapsulation concept.
- To access or modify these private attributes, we provide **public methods** like `display_info()` and `set_name()`.

Output:

```
Name: Charlie, Age: 28
Name: David, Age: 28
```

- If we try to access `person1.__name` directly, it will raise an `AttributeError`, as private attributes are name-mangled by Python to prevent direct access.

Accessing Private Attributes: Name Mangling

Though private members are hidden, they can still be accessed using a special syntax: `_ClassName__attribute`. This is called **name mangling** and is used internally by Python to protect private members.

Example (Accessing Private Members via Name Mangling):

```
# Trying to access private attributes via name mangling
print(person1._Person__name)  # This will work, but not recommended
```

Explanation:

- `_Person__name` allows access to the `__name` attribute, but using this is generally discouraged as it breaks the encapsulation.

Output:

```
David
```

Encapsulation can be implemented by making variables private (by convention using a leading underscore `_` or double underscore `__`), and providing public methods to access and modify these variables, which are known as **getters** and **setters**.

Example of Encapsulation with Getters and Setters

Let's walk through an example step by step.

Step 1: Create a Simple Class without Encapsulation

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

# Usage
emp = Employee("John", 50000)
print(emp.name)  # Accessing public attribute
print(emp.salary)  # Accessing public attribute
```

Explanation:

- In this example, `name` and `salary` are public attributes, meaning they can be accessed directly from outside the class.

- This allows anyone to change these attributes directly, which might lead to unintended issues.

Step 2: Implement Encapsulation with Private Attributes

To encapsulate the `salary` attribute and restrict its direct modification, we make it private by prefixing it with a double underscore (`__`).

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.__salary = salary  # Private attribute

# Usage
emp = Employee("John", 50000)
print(emp.name)  # Works fine
print(emp.__salary)  # This will raise an AttributeError
```

Explanation:

- Now, `__salary` is a private attribute. You cannot access it directly using `emp.__salary`, which will raise an `AttributeError`.

- This prevents users from modifying the `salary` directly, adding a layer of protection.

Step 3: Use Getters and Setters to Access and Modify Private Attributes

We can provide controlled access to private attributes using **gette**r and **setter** methods.

Getters

Getters allow read access to private attributes.

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.__salary = salary  # Private attribute
    #Getter method for salary
    def get_salary(self):
        return self.__salary
```

```
# Usage
emp = Employee("John", 50000)
print(emp.get_salary())  # Accessing private attribute via getter
```

Explanation:

- The `get_salary` method is a getter that provides read-only access to the private attribute `__salary`.

Setters

Setters allow controlled modification of private attributes. You can add logic to validate the new value before changing the attribute.

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.__salary = salary  # Private attribute
    #Getter method for salary
    def get_salary(self):
        return self.__salary

    #Setter method for salary
```

```
    def set_salary(self, new_salary):
        if new_salary > 0:
            self.__salary = new_salary
            return f”Employee Updated Salary: {new_salary}”
        else:
            print("Salary must be positive")


# Usage
emp = Employee("John", 50000)
print(emp.get_salary())
emp.set_salary(60000)  # Modifying private attribute via setter
print(emp.get_salary())
emp.set_salary(-1000)  # Trying to set an invalid salary
```

Explanation:

- The `set_salary` method is a setter that ensures the new salary is positive before updating the `__salary` attribute. If an invalid value is provided, it rejects the change with an error message.

- This ensures controlled modification of the private data, which is an important aspect of encapsulation.

Step 4: Using Property Decorators (Pythonic Way)

In Python, you can use the `@property` decorator to make the code cleaner. This eliminates the need to explicitly call getter and setter methods.

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.__salary = salary  # Private attribute

    # Getter using property decorator
    @property
    def salary(self):
        return self.__salary

    # Setter using property decorator
    @salary.setter
    def salary(self, new_salary):
        if new_salary > 0:
            self.__salary = new_salary
```

```
        else:
            print("Salary must be positive")


# Usage
emp = Employee("John", 50000)
print(emp.salary)  # Access salary like an attribute
emp.salary = 60000  # Modify salary like an attribute
print(emp.salary)  # Updated salary
```

Explanation:

- The `@property` decorator turns the `salary` method into a getter, so it can be accessed like an attribute (`emp.salary`).
- The `@salary.setter` decorator allows the `salary` method to act as a setter, so we can modify it like an attribute (`emp.salary = 60000`).
- This is a more Pythonic way of achieving encapsulation while keeping the syntax clean.

Summary of Steps

1. **Without Encapsulation**: Direct access to attributes.

2. **Private Attributes**: Use double underscore (`__`) to prevent direct access.

3. **Getters and Setters**: Use methods to access and modify private attributes.

4. **Property Decorators:** Simplify getters and setters using the `@property` and `@setter` decorators.

This is how encapsulation is achieved in Python by using getters and setters to control access to private attributes.

Why Encapsulation?

1. Data Hiding: Encapsulation hides the internal state of an object and only exposes a controlled interface. This prevents external interference.

2. Controlled Access: You can define public methods (getters/setters) to control how attributes are accessed or modified.

3. Improved Security: By making sensitive data private, encapsulation ensures that this data cannot be altered or accessed improperly.

Final Recap:

- **Public members:** Accessible from anywhere.

- **Protected members:** Accessible within the class and subclasses (by convention).

- **Private members**: Accessible only within the class (using name mangling to access them outside is possible but not recommended).

Encapsulation enhances security and abstraction, promoting better design in Python programs.

Polymorphism

Polymorphism is the ability of objects of different types to be treated as objects of a common supertype.

It allows functions or methods to use objects of different types through a shared interface.

In Python, polymorphism occurs by methods defined in different classes being called through a common interface.

Example of Polymorphism:

```
class Bird:
    def fly(self):
        print("Bird is flying")


class Airplane:
    def fly(self):
        print("Airplane is flying")


def make_it_fly(flying_object):
    flying_object.fly()


# Polymorphism in action
bird = Bird()
```

```
plane = Airplane()

make_it_fly(bird)   # Output: Bird is flying
make_it_fly(plane)  # Output: Airplane is flying
```

Operator Overloading

Operator overloading in Python allows the use of built-in operators (+, -, *, etc.) with user-defined classes. Python's special methods (`__add__`, `__sub__`, etc.) enable operator overloading.

Example of Operator Overloading:

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

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

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

```
# Create points
p1 = Point(2, 3)
p2 = Point(5, 7)

# Add two points using overloaded "+" operator
p3 = p1 + p2
print(p3)  # Output: Point(7, 10)
```

Method Overloading

Method overloading allows a class to have multiple methods with the same name but different numbers or types of parameters.

However, Python does not support traditional method overloading. We can achieve similar functionality using default arguments or by manually checking argument types.

Example of Method Overloading:

```
class Calculator:
    def add(self, a=None, b=None, c=None):
        if a is not None and b is not None and c is not None:
            return a + b + c
```

```
        elif a is not None and b is not None:
            return a + b
        else:
            return "Invalid arguments"

# Create an instance
calc = Calculator()

# Different ways to call overloaded add method
print(calc.add(2, 3))       # Output: 5
print(calc.add(2, 3, 4))    # Output: 9
print(calc.add(2))          # Output: Invalid arguments
```

Method Overriding

Method overriding occurs when a subclass provides a specific implementation for a method already defined in its superclass. The method in the subclass has the same name, signature, and return type as the method in the superclass.

Example of Method Overriding:

```
class Animal:
    def sound(self):
        print("This is a generic animal sound")
class Dog(Animal):
    def sound(self):
        print("Bark!")
class Cat(Animal):
    def sound(self):
        print("Meow!")
# Create objects
animal = Animal()
dog = Dog()
cat = Cat()
# Call overridden methods
animal.sound()  # Output: This is a generic animal sound
dog.sound()     # Output: Bark!
cat.sound()     # Output: Meow!
```

Difference Between Inheritance and Method Overriding

- **Inheritance** is the mechanism by which a class inherits properties and methods from a parent class, allowing code reuse.

- **Method overriding** allows a subclass to provide a specific implementation for a method already defined in the parent class, ensuring the subclass has its specialized behavior.

Inheritance	Method Overriding
Enables code reuse from the parent class.	Allows the child class to provide a different implementation of a method inherited from the parent class.
Used to model "is-a" relationships.	Used to change or extend the behavior of inherited methods.
Example: A `Dog` class inherits from `Animal`.	Example: The `Dog` class overrides the `sound` method of `Animal`.

Advantages of Polymorphism

1. Code Reusability: Polymorphism allows for the use of a common interface, reducing code duplication.

2. Flexibility and Extensibility: Functions can work with objects of different types, making code flexible and easier to extend.

3. Maintainability: Polymorphism promotes code that is more maintainable because changes to one class do not necessarily affect other parts of the system.

4. Loose Coupling: Since classes only depend on shared behavior (interface) rather than concrete implementations, polymorphism enables loose coupling.

Disadvantages of Polymorphism

1. Complexity: Polymorphism can add complexity to the design of software as it requires careful management of inheritance and interface usage.

2. Reduced Performance: In some cases, polymorphism may lead to slightly slower performance because of the overhead of dynamic dispatch (finding the correct method to call at runtime).

3. Debugging Difficulties: It may be harder to trace the exact method being called at runtime when multiple objects or types are involved, making debugging more challenging.

Polymorphism and other object-oriented concepts provide a strong foundation for writing flexible, maintainable, and reusable code, though they require careful design choices.

Abstraction in OOP (Object-Oriented Programming)

Abstraction is one of the four fundamental principles of Object-Oriented Programming (OOP), alongside Encapsulation, Inheritance, and Polymorphism.

Abstraction refers to the concept of hiding the complex implementation details and showing only the essential features of an object to the user. It allows focusing on what an object does rather than how it does it.

In other words, abstraction simplifies complex systems by modeling classes appropriate to the problem and working at a higher level of complexity. It helps in reducing the programming complexity and effort, as the internal working is abstracted and the user is exposed only to essential information.

Key Features of Abstraction:

- **Hides complexity**: Only the relevant details are provided.
- **Improves security**: Sensitive or unnecessary implementation details are hidden.
- **Simplifies interaction**: Provides a simpler interface to interact with objects.

Real-life Example:

A TV remote allows you to switch channels and adjust the volume without knowing how the internal electronics work. The remote abstracts away the complex workings and only provides the user with simple buttons to operate.

Abstraction in Python

In Python, abstraction can be implemented using abstract classes and methods from the `abc` module (Abstract Base Classes). An abstract class cannot be instantiated and must be inherited by subclasses, which are required to implement the abstract methods.

Python Example: Using Abstraction

Let’s implement abstraction using a bank example where we define a general `BankAccount` class that provides a blueprint for specific account types (e.g., `SavingsAccount`, `CheckingAccount`).

```
from abc import ABC, abstractmethod

#Abstract Class
class BankAccount():
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    #Abstract Method (forces subclasses to implement this)
    #@abstractmethod
    def withdraw(self, amount):
        pass

    #Non-abstract Method
    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited {amount}. New balance is {self.balance}")

#Subclass that implements the abstract method
class SavingsAccount(BankAccount):
```

```
    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            print(f"Withdrew {amount}. New balance is {self.balance}")
        else:
            print("Insufficient balance")

#Subclass that implements the abstract method
class CheckingAccount(BankAccount):
    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            print(f"Withdrew {amount}. New balance is {self.balance}")
        else:
            print("Insufficient balance, but you have overdraft protection")

#Example usage
savings = SavingsAccount("Alice", 1000)
savings.deposit(200)
savings.withdraw(500)

checking = CheckingAccount("Bob", 500)
checking.deposit(300)
checking.withdraw(900)
```

Explanation of the Code:

1. **Abstract Class (`BankAccount`):**

- The class `BankAccount` is an abstract class, defined using the `ABC` module.

- It has one abstract method, `withdraw()`, which must be implemented by any subclass.

- The `deposit()` method is a regular method that can be used directly by subclasses.

2. **Subclasses (`SavingsAccount` and `CheckingAccount`)**:

- Both `SavingsAccount` and `CheckingAccount` inherit from `BankAccount`.

- They provide their own implementation of the `withdraw()` method.

- For instance, `SavingsAccount` simply checks if there's enough balance, whereas `CheckingAccount` simulates an overdraft feature.

3. **Usage**:

- Objects `savings` and `checking` are created from their respective subclasses and interact with the abstracted methods, such as `deposit()` and `withdraw()`, without needing to know their internal workings.

Key Takeaways:

- **Abstraction** allows hiding the unnecessary complexities of the system.

- **Abstract classes and methods** help in enforcing a consistent interface while letting subclasses define the details.

- In Python, the `abc` module is used to achieve abstraction.

Object as an Argument

Objects (instances of classes) can also be passed to functions as arguments, and changes made to the object's attributes inside the function affect the original object.

Example:

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


def change_name(person):
    person.name = "John"


p = Person("Alice")
change_name(p)
print("Person's name after function call:", p.name)
```

Explanation:

1. The class `Person` has an `__init__` method that initializes an object with a name.

2. A function `change_name(person)` modifies the `name` attribute of the passed object.

3. The `change_name()` function is called with the object `p` (whose name is initially "Alice").

4. After the function call, `p.name` is changed to "John", showing that passing objects to functions allows for modification.

2.

```
import math


# Define the Point class with x and y coordinates
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y


# Define a function to calculate the distance between two points
def calculate_distance(point1, point2):
    # Use the Euclidean distance formula
    distance = math.sqrt((point2.x - point1.x) ** 2 + (point2.y - point1.y) ** 2)
    return distance
```

```
# Create two point objects
point1 = Point(3, 4)
point2 = Point(7, 1)

# Calculate the distance between point1 and point2
distance = calculate_distance(point1, point2)

# Output the result
print(f"The distance between the points is {distance:.2f}")
```

3. **Operator Overloading**

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

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self):
```

```
        return f"Point({self.x}, {self.y})"

# Create points
p1 = Point(2, 3)
p2 = Point(5, 7)

# Add two points using overloaded "+" operator
p3 = p1 + p2
print(p3)
```

Call by value: A copy of the variable's value is passed into the function. Modifications to the parameter inside the function don't affect the original variable.

Call by reference: A reference to the actual variable is passed, meaning that changes to the parameter affect the original variable.

In Python, mutable objects (like lists, dictionaries, and instances of most classes) are passed by reference, meaning changes made to them within a function will affect the original object.

Immutable objects (like integers, strings, and tuples) behave as though passed by value.

Let's dive into two examples using Python classes and objects.

1. Call by Value-Like Behavior (Immutable objects)

In Python, integers are immutable. When you pass an integer to a function and try to modify it, a new integer is created in memory, and the original variable remains unchanged.

Example

```
class Example:
    def modify_value(self, num):
        print(f"Original value: {num}")
        num = 10  # Reassign the variable
        print(f"Modified inside function: {num}")


# Main code
obj = Example()
original_value = 5
obj.modify_value(original_value)
print(f"Value after function call: {original_value}")
```

Explanation:

1. We define a class `Example` with a method `modify_value` that takes a number `num` as an argument.

2. Inside the method, we print the original value, then change `num` to 10.

3. However, since integers are immutable, this change only affects `num` inside the method. It doesn't affect the original value.

4. Outside the function, the value of `original_value` remains unchanged.

Output:

Original value: 5

Modified inside function: 10

Value after function call: 5

- The `original_value` remains 5 after the function call, illustrating Python's behavior similar to **call by value** for immutable objects.

2. **Call by Reference-Like Behavior (Mutable objects)**

In this case, we use a list (a mutable object). When you pass a list to a function, changes made to the list inside the function will reflect on the original list.

Example:

```
class Example:
    def modify_list(self, lst):
        print(f"Original list: {lst}")
        lst.append(10)  # Modify the list
        print(f"Modified inside function: {lst}")

# Main code
obj = Example()
```

```
original_list = [1, 2, 3]
obj.modify_list(original_list)
print(f"List after function call: {original_list}")
```

Explanation:

1. We define a class `Example` with a method `modify_list` that takes a list `lst` as an argument.

2. Inside the method, we print the original list, then append the number 10 to the list.

3. Lists are mutable in Python, so this change affects the original list.

4. Outside the function, we see that `original_list` has been modified.

Output:

```
Original list: [1, 2, 3]
Modified inside function: [1, 2, 3, 10]
List after function call: [1, 2, 3, 10]
```

- The `original_list` is modified inside the function, showing **call by reference**-like behavior for **mutable objects**.

Conclusion

- For **immutable objects** like integers and strings, Python behaves similar to **call by value**. Reassigning a variable inside the function doesn’t affect the original variable.

- For **mutable objects** like **lists and dictionaries**, Python behaves like **call by reference**. Changes made to the object inside the function are reflected outside the function.

These examples show how Python's **call by object reference** works depending on the type of object passed to the function.

1Q. **A python function that accepts a list as a parameter, modifies the list within the function, and demonstrates how the changes impact the original list. Discuss whether this behavior represents call by value or call by reference.**

In Python, lists are mutable, meaning they can be modified after their creation. When a list is passed as a parameter to a function, Python doesn't create a copy of the list. Instead, it passes a reference to the original list. As a result, changes made to the list inside the function affect the original list outside the function.

This behavior can be described as "call by object reference" (or "call by sharing"). While it's not strictly "call by reference" in the traditional sense, it has a similar effect for mutable objects like lists.

```
def modify_list(my_list):
    # Modifying the list inside the function
    my_list.append(100)
    my_list[0] = 'Changed'
# Original list
original_list = [1, 2, 3, 4, 5]
# Call the function
modify_list(original_list)
# Print the list after modification
print("Modified original list:", original_list)
```

Step-by-Step Explanation:

1. Original List Creation: We create a list `original_list` with values `[1, 2, 3, 4, 5]`.

2. Function Call: The list `original_list` is passed to the function `modify_list`. Inside the function, the list is modified in two ways:

   - An element `100` is appended to the list.

   - The first element of the list (`my_list[0]`) is changed to `'Changed'`.

3. Changes to the Original List: Since lists are mutable and Python passes the reference to the list (not a copy), changes inside the function directly affect the `original_list`.

4. Output: After calling the function, printing the `original_list` shows that it has been modified:

   Modified original list: ['Changed', 2, 3, 4, 5, 100]

Call by Value vs. Call by Reference

- **Call by Value**: If Python used "call by value," the function would receive a copy of the list, and any modifications inside the function would not affect the original list.

- **Call by Reference**: If Python used "call by reference," the function would receive a direct reference to the original list, and modifications inside the function would impact the original list.

In Python, the behavior for mutable objects like lists is closer to **call by reference** because modifications inside the function are reflected in the original list. However, it's technically known as **"call by object reference"** or **"call by sharing"** because the function gets a reference to the object (list), not the variable holding the list.

2Q. A function that accepts both mutable (e.g., a list) and immutable (e.g., a string) types as parameters. Modify both within the function and explain how these changes affect the original variables outside the function. Does Python use call by value or call by reference in this scenario?

In Python, mutable and immutable types behave differently when passed as parameters to a function. Let's look at an example where a function accepts both a mutable (e.g., a list) and an immutable (e.g., a string) parameter. We'll modify both inside the function and then observe how these modifications affect the original variables outside the function.

```
def modify_parameters(my_list, my_string):
    # Modify the list (mutable)
    my_list.append(100)
    my_list[0] = 'Changed'

    # Modify the string (immutable)
    my_string = "Modified string"
```

```
# Original variables
original_list = [1, 2, 3, 4, 5]
original_string = "Original string"

# Call the function
modify_parameters(original_list, original_string)

# Print the list and string after modification
print("Modified original list:", original_list)
print("Modified original string:", original_string)
```

Step-by-Step Explanation:

1. Original Variables:
   - We start with two variables:
     - `original_list`: A list `[1, 2, 3, 4, 5]` (mutable)
     - `original_string`: A string `"Original string"` (immutable)
2. Function Call:
   - The list and string are passed to the function `modify_parameters`. Inside the function:
     - List Modification: The list is modified in two ways:

- We append `100` to the list.

- We change the first element of the list to `'Changed'`.

- **String Modification**: We attempt to change the string by assigning a new value `"Modified string"` to `my_string`.

3. Changes to the List (Mutable):

- The changes to the list (a mutable object) persist outside the function because the list is passed by reference (technically, by object reference). The list inside the function is the same as the original list, so modifications to the list inside the function affect the original list.

- After the function call, `original_list` becomes: `['Changed', 2, 3, 4, 5, 100]`.

4. Changes to the String (Immutable):

- Strings are immutable, meaning they cannot be modified after they are created. When we assign `"Modified string"` to `my_string` inside the function, this creates a new string object rather than modifying the original one. This change does **not** affect the original string `original_string`.

- After the function call, `original_string` remains: `"Original string"`.

Output:

Modified original list: ['Changed', 2, 3, 4, 5, 100]

Modified original string: Original string

Call by Value or Call by Reference

- Mutable Types (e.g., List)**: For mutable objects like lists, Python passes the reference to the object, so changes made inside the function affect the original object outside the function. This is similar to **call by reference**.

- Immutable Types (e.g., String): For immutable objects like strings, assigning a new value inside the function does not affect the original object. Instead, a new object is created, and the original object remains unchanged. This behavior is closer to **call by value**, as the original variable remains unaffected.

Key Takeaways:

- **Mutable objects (like lists)** behave like they're passed by reference because changes inside the function reflect on the original object.

- **Imutable objects (like strings)** behave more like they're passed by value because any attempt to modify them inside the function results in a new object being created, leaving the original object unchanged.

In Python, this is referred to as **call by object reference** or **call by sharing**: the reference to the object is passed to the function, and the behavior depends on whether the object is mutable or immutable.

3Q. **A python function that takes a tuple (an immutable object) as an argument and tries to alter its contents. Describe what happens and why, addressing whether Python uses call by value or call by reference in this context.**

In Python, tuples are immutable, which means once they are created, their contents cannot be altered. When a tuple is passed as an argument to a function, even if we attempt to modify its contents inside the function, Python will not allow it. This is because tuples, like strings, are immutable types.

Let's write a Python function that attempts to alter the contents of a tuple.

```
def modify_tuple(my_tuple):
    # Trying to modify the tuple
    try:
        my_tuple[0] = 'Changed'
    except TypeError as e:
        print(f"Error: {e}")
# Original tuple
original_tuple = (1, 2, 3, 4, 5)
```

Call the function

modify_tuple(original_tuple)

Print the original tuple after the function call

print("Original tuple:", original_tuple)

Step-by-Step Explanation:

1. Original Tuple: We create a tuple `original_tuple` with the values `(1, 2, 3, 4, 5)`. Since tuples are immutable, their contents cannot be changed once created.

2. Function Call: We pass the tuple `original_tuple` to the function `modify_tuple`.

3. Attempted Modification Inside the Function:

- Inside the function, we try to change the first element of the tuple using `my_tuple[0] = 'Changed'`.

- Since tuples are immutable, this will raise a `TypeError`. We catch the exception and print the error message:

Error: 'tuple' object does not support item assignment

4. Effect on the Original Tuple:

- The tuple remains unchanged after the function call. This is because tuples cannot be modified, so any attempt to do so results in an error.

Output:

Error: 'tuple' object does not support item assignment

Original tuple: (1, 2, 3, 4, 5)

Key Takeaways:

- Immutability of Tuples: Tuples are immutable, so you cannot modify their contents after they are created. Any attempt to change an element within a tuple will result in a `TypeError`.

- Call by Value or Call by Reference

- Although the function receives a reference to the tuple, Python's immutability constraint means you cannot modify the contents of the tuple. This behavior is akin to **call by value** for immutable objects, as any attempt to alter the tuple results in a new object being created (if you reassign the variable) rather than modifying the existing one.

- For immutable objects like tuples, Python behaves more like call by value because the original object remains unchanged regardless of what happens inside the function.

- However, technically, Python still uses **call by object reference**: the reference to the tuple is passed to the function. But since tuples are immutable, the original object cannot be altered.

Conclusion:

When a tuple is passed to a function, Python passes a reference to the object, but since tuples are immutable, you cannot change their contents.

If you attempt to modify them, Python raises a `TypeError`.

This behavior mimics **call by value** for immutable objects, as the original tuple remains unchanged regardless of operations inside the function.

RECURSION

Recursion occurs when a function calls itself. This is useful for solving problems that can be broken down into smaller, similar problems (e.g., factorial, Fibonacci).

1Q. Repeatedly compute the sum of all digits of the given number N until the sum itself becomes a single digit number using recursion.

Program:

```
def sum_of_digits(n):
    if n<10:
        return n
    else:
        return (n%10)+sum_of_digits(n//10)
n=int(input("enter a number: "))
m=sum_of_digits(n)
if m<10:
    print(m)
else:
    print(sum_of_digits(m))
```

2Q. A recursive function fibonacci(n) that takes a non-negative integer n and returns the nth Fibonacci number. For example, if the input is 6, the output should be 8.

Program:

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
n = int(input("Enter a non-negative integer: "))
result = fibonacci(n)
print(f"The {n}th Fibonacci number is: {result}")
```

3Q. The factorial of a given number N using recursion.

Program:

```
def factorial(n):
    if n==1 or n==0:
        return n
    else:
        return n * factorial(n-1)
n=int(input())
result=factorial(n)
print(f"the factorial of {n} is: {result}")
```

Explanation:

1. The function `factorial(n)` calculates the factorial of `n` recursively.

2. If `n` is 0 or 1 (base case), it returns 1.

3. Otherwise, it calls itself with `n - 1`, multiplying `n` by the factorial of the smaller number.

4. The output shows `120` as the factorial of 5.

Nested Classes

Nested classes in Python are classes defined within another class. They are used to logically group classes that are only used in one place, which can help with code organization and readability. Nested classes can also access the outer class's attributes and methods, making them useful for certain design patterns.

Types of Nested Classes:

1. **Single Nested Class**: A class defined within another class.

2. **Multiple Nested Classes:** Multiple classes defined within another class.

3. Multilevel Nested Class: A class defined within another nested class.

Let’s break down each type with definitions and examples.

1. Single Nested Class

A single nested class is simply a class defined within another class.

Example:

```
class OuterClass:
    class InnerClass:
        def display(self):
            return "This is the Inner Class."
outer = OuterClass()
inner = outer.InnerClass()
print(inner.display())
```

Explanation:

- We define `OuterClass` and within it, we define `InnerClass`.
- `InnerClass` has a method `display()` that returns a string.
- To use `InnerClass`, we first create an instance of `OuterClass`, then access `InnerClass` through the outer class instance.

2. Multiple Nested Classes

In multiple nested classes, we define more than one class inside another class.

Example:

```
class OuterClass:
    class InnerClass1:
        def display(self):
            return "This is Inner Class 1."

    class InnerClass2:
        def display(self):
            return "This is Inner Class 2."

outer = OuterClass()
inner1 = outer.InnerClass1()
inner2 = outer.InnerClass2()
print(inner1.display())
print(inner2.display())
```

Explanation:

- `OuterClass` contains two nested classes: `InnerClass1` and `InnerClass2`.

- Each inner class has its own `display()` method.

- We can create instances of each inner class and call their methods independently.

3. Multilevel Nested Class

In multilevel nesting, you can have a class defined within another nested class.

Example:

```
class OuterClass:
    class InnerClass:
        class NestedInnerClass:
            def display(self):
                return "This is the Nested Inner Class."

outer = OuterClass()
inner = outer.InnerClass.NestedInnerClass()
print(inner.display())
```

Explanation:

- We have `OuterClass` containing `InnerClass`, which further contains `NestedInnerClass`.

- `NestedInnerClass` has a method `display()`, which can be accessed after creating instances of the outer classes.

Summary of Key Points:

- Nested classes can improve code organization.

- Single nested classes are defined within a single outer class.

- Multiple nested classes allow the definition of more than one class within another class.

- Multilevel nested classes are classes defined within other nested classes.

1Q. Create a Python class Color with an instance attribute name set to 'Green'. Inside the Color class, define a nested class Lightgreen that has two instance attributes: name set to 'Light Green' and code set to “431cva”. Add a method show() in the Color class to display the name attribute, and a method display() in the Lightgreen class to print both the name and code attributes. Instantiate an object outer of the Color class, call the show() method to print the color name, and then access a Lightgreen object g from outer to call the display() method and print the name and code of the light green color.

Program:

```
class Color:
    def __init__(self):
        self.name = 'Green'  # Instance attribute for Color


    def show(self):
        print(f"Color: {self.name}")  # Method to display the color name


    class Lightgreen:
        def __init__(self):
            self.name = 'Light Green'  # Instance attribute for Lightgreen
```

```
        self.code = "431cva"       # Instance attribute for
Lightgreen code


    def display(self):
        print(f"Name: {self.name}, Code: {self.code}")  #
Method to display name and code


#Instantiate the Color class
outer = Color()
#Call the show() method to print the color name
outer.show()
#Access the nested Lightgreen class
g = outer.Lightgreen()
#Call the display() method to print the name and code of the
light green color
g.display()
```

2Q. Create a Python class Doctors with an instance attribute name set to 'Doctor'. Inside this class, define two nested classes:

- **Dentist, with instance attributes name set to 'Dr. Sita' and degree set to 'MBBS'.**
- **Cardiologist, with instance attributes name set to 'Dr. Ram' and degree set to 'FRCS'.**

In the Doctors class, implement a method show() to display the name attribute, and in both the Dentist and Cardiologist classes, implement methods display() to show their name and degree attributes. Instantiate an object outer of the Doctors class, call the show() method to display general doctor information, and then access the Dentist object d1 and Cardiologist object d2 from outer. Call their respective display() methods to print their names and degrees.

Program:

```
class Doctors:
    def __init__(self):
        self.name = 'Doctor'  # Instance attribute for Doctors


    def show(self):
        print(f"General Information: {self.name}")  # Method to display the general doctor name


    class Dentist:
        def __init__(self):
            self.name = 'Dr.Ajay'  # Instance attribute for Dentist
            self.degree = 'MBBS'    # Instance attribute for Dentist's degree


        def display(self):
            print(f"Dentist: {self.name}, Degree: {self.degree}")  # Method to display name and degree
```

```
    class Cardiologist:
        def __init__(self):
            self.name = 'Dr.Ram'   # Instance attribute for Cardiologist
            self.degree = 'FRCS'    # Instance attribute for Cardiologist's degree

        def display(self):
            print(f"Cardiologist: {self.name}, Degree: {self.degree}")  # Method to display name and degree

# Instantiate the Doctors class
outer = Doctors()
# Call the show() method to display general doctor information
outer.show()

# Access the nested Dentist and Cardiologist classes
d1 = outer.Dentist()
d2 = outer.Cardiologist()
```

```
# Call their respective display() methods to print names and degrees
d1.display()
d2.display()
```

3Q. Develop a Python class University with an instance attribute university_name initialized to 'KLU'. Inside this class, implement a nested class Department, which has an instance attribute department_name initialized to 'Computer Science'. Inside the Department class, create another nested class Professor with instance attributes professor_name initialized to 'Dr. Smith' and subject initialized to 'Machine Learning'. Implement methods to print the details of the university, department, and professor. Instantiate the classes and display their information.

Program:

```
class University:
    def __init__(self):
        self.university_name = 'KLU'  # Instance attribute for University

    def show_university_info(self):
        print(f"University Name: {self.university_name}")  # Method to display university information

    class Department:
```

```
    def __init__(self):
        self.department_name = 'Computer Science'  # Instance attribute for Department

    def show_department_info(self):
        print(f"Department Name: {self.department_name}")  # Method to display department information

    class Professor:
        def __init__(self):
            self.professor_name = 'Dr.Smith'  # Instance attribute for Professor
            self.subject = 'Machine Learning'   # Instance attribute for Professor's subject

        def show_professor_info(self):
            print(f"Professor Name: {self.professor_name}, Subject: {self.subject}")  # Method to display professor information

#Instantiate the University class
university = University()
#Call the method to display university information
```

```
university.show_university_info()
#Access the nested Department class
department = university.Department()

#Call the method to display department information
department.show_department_info()
#Access the nested Professor class
professor = department.Professor()
#Call the method to display professor information
professor.show_professor_info()
```

4Q. Define a Python class Library with an instance attribute library_name set to 'Central Library'. Inside this class, create a nested class Section with an instance attribute section_name set to 'Thriller'. Within the Section class, define another nested class Book, which has instance attributes title set to 'Gone Girl' and author set to 'Gillian Flynn'. Implement methods in each class to display the details of the library, section, and book. Instantiate the classes and display the information about the library, the section, and the book.

Program:

```
class Library:
    def __init__(self):
        self.library_name = 'Central Library'  # Instance attribute for Library

    def show_library_info(self):
```

```
        print(f"Library Name: {self.library_name}")  # Method to display library information


    class Section:
        def __init__(self):
            self.section_name = 'AI&DS'  # Instance attribute for Section


        def show_section_info(self):
            print(f"Section Name: {self.section_name}")  # Method to display section information


        class Book:
            def __init__(self):
                self.title = 'Core Python Programming'         # Instance attribute for Book title
                self.author = 'Guido van Rossum'     # Instance attribute for Book author


            def show_book_info(self):
                print(f"Book Title: {self.title}, Author: {self.author}")  # Method to display book information
#Instantiate the Library class
library = Library()
#Call the method to display library information
library.show_library_info()
```

```
#Access the nested Section class
section = library.Section()

# Call the method to display section information
section.show_section_info()
# Access the nested Book class
book = section.Book()
# Call the method to display book information
book.show_book_info()
```