Chapter 4 - Inheritance, Types of Inheritance, and Polymorphism

1. Introduction

In **Object-Oriented Programming (OOP)**, **inheritance** allows one class (child class) to acquire properties and methods from another class (parent class). This promotes **code reuse**, reducing redundancy.

Polymorphism enables objects of different classes to be treated as objects of a common superclass. This allows functions or methods to process different object types seamlessly.

---

2. Inheritance in Python

2.1 What is Inheritance?

Inheritance allows a class (**child class**) to derive properties and behaviors from another class (**parent class**).

Example of a parent class:

Python

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

    def make_sound(self):
        return "Some sound"
```

Example of a child class inheriting from `Animal`:

Python

```python
class Dog(Animal):
    def make_sound(self):  # Overriding the parent method
        return "Woof!"
```

```python
dog = Dog("Buddy")
print(dog.name)           # Output: Buddy
print(dog.make_sound())  # Output: Woof!
```

---

3. Types of Inheritance

3.1 Single Inheritance

In single inheritance, a **child class** derives from **one parent class**.

Diagram:

```
      Vehicle
         |
         Car
```

Code:

Python

```python
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

    def show_brand(self):
        print(f"Brand: {self.brand}")

class Car(Vehicle):  # Single Inheritance
    def __init__(self, brand, model):
        super().__init__(brand)
        self.model = model

    def show_model(self):
        print(f"Model: {self.model}")

car = Car("Toyota", "Corolla")
car.show_brand()  # Output: Brand: Toyota
car.show_model()  # Output: Model: Corolla
```

3.2 Multiple Inheritance

A class can inherit from **multiple parent classes**.

Diagram:

```
    Engine    Wheels
       \      /
        Truck
```

Code:

Python

```python
class Engine:
    def engine_type(self):
        return "Diesel Engine"

class Wheels:
    def wheel_count(self):
        return "Four Wheels"

class Truck(Engine, Wheels):  # Multiple Inheritance
    pass

truck = Truck()
print(truck.engine_type())  # Output: Diesel Engine
print(truck.wheel_count())  # Output: Four Wheels
```

⚠️ **Avoid multiple inheritance unless necessary**, as it can make debugging difficult.

3.3 Multilevel Inheritance

A class can inherit from another child class, forming a **chain of inheritance**.

Diagram:

```
    Animal
       |
     Mammal
```

```
      |
      Dog
```

Code:

Python

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

class Mammal(Animal):
    def has_fur(self):
        return True

class Dog(Mammal):  # Multilevel Inheritance
    def make_sound(self):
        return "Woof!"

dog = Dog()
print(dog.make_sound())  # Output: Woof!
print(dog.has_fur())     # Output: True
```

3.4 Hierarchical Inheritance

One parent class is inherited by **multiple child classes**.

Diagram:

```
      Animal
     /      \
    Cat      Dog
```

Code:

Python

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

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

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

cat = Cat()
dog = Dog()
print(cat.make_sound())  # Output: Meow!
print(dog.make_sound())  # Output: Woof!
```

3.5 Hybrid Inheritance

A combination of multiple inheritance types.

Diagram:

```
       A
      / \
     B   C
      \ /
       D
```

Code:

Python

```
class A:
    def method_A(self):
        return "Method A"

class B(A):
    def method_B(self):
        return "Method B"

class C(A):
    def method_C(self):
        return "Method C"

class D(B, C):  # Hybrid Inheritance
    def method_D(self):
```

```python
        return "Method D"

obj = D()
print(obj.method_A())  # Output: Method A
print(obj.method_B())  # Output: Method B
print(obj.method_C())  # Output: Method C
print(obj.method_D())  # Output: Method D
```

Hybrid inheritance should be used carefully as it can make debugging complex.

---

4. Polymorphism

4.1 What is Polymorphism?

Polymorphism allows **different classes** to be treated as **a single entity** by using a common interface.

4.2 Method Overriding

A **child class** provides a different implementation for a method already defined in the **parent class**.

```python
class Bird:
    def fly(self):
        return "I can fly!"

class Penguin(Bird):
    def fly(self):  # Overriding parent method
        return "I can't fly!"

bird = Bird()
penguin = Penguin()
print(bird.fly())    # Output: I can fly!
print(penguin.fly()) # Output: I can't fly!
```

4.3 Method Overloading (Achieved via Default Arguments)

Python does **not** support method overloading directly, but it can be achieved using default parameters.

```python
class MathOperations:
    def add(self, a, b, c=0):  # Default argument for overloading behavior
        return a + b + c

math = MathOperations()
print(math.add(2, 3))       # Output: 5
print(math.add(2, 3, 4))    # Output: 9
```

4.4 Operator Overloading

We can define how operators (`+` , `-` , `*` , etc.) behave for objects.

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

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

v1 = Vector(2, 3)
v2 = Vector(1, 5)
v3 = v1 + v2  # Uses __add__()
print(v3.x, v3.y)  # Output: 3 8
```

---

5. Inheritance and Polymorphism in Django

5.1 Inheritance in Django Models

Django models support **model inheritance**, which helps in code reuse.

Abstract Base Class

Useful when you want to share fields across multiple models but don't want to create instances of the base class.

```python
from django.db import models

class BaseModel(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    class Meta:
        abstract = True  # This makes it an abstract class

class Product(BaseModel):
    name = models.CharField(max_length=100)
    price = models.FloatField()
```

Multi-table Inheritance

Each subclass gets its own table.

```
class User(models.Model):
    name = models.CharField(max_length=100)

class Admin(User):  # Multi-table inheritance
    access_level = models.IntegerField()
```

---

6. Exercises

Exercise 1: Implement Single Inheritance (Detailed)

Problem: Create a class `Person` with attributes `name` (string) and `age` (integer). Create a subclass `Student` that inherits from `Person` and adds an attribute `grade` (string). Implement methods to display all attributes for both classes.

Example:

```
person = Person("Alice", 30)
student = Student("Bob", 20, "A")
person.display() # Output: Name: Alice, Age: 30
student.display() # Output: Name: Bob, Age: 20, Grade: A
```

Exercise 2: Implement Method Overriding (Detailed)

Problem: Create a parent class `Shape` with a method `area()` that returns 0 (base area). Create subclasses `Rectangle` and `Circle` that override `area()` to calculate the respective areas.

Example:

```
rectangle = Rectangle(5, 10)
circle = Circle(7)
print(rectangle.area()) # Output: 50
print(circle.area()) # Output: 153.93804002589985 (approximately)
```

Exercise 3: Operator Overloading (Detailed)

Problem: Create a class `BankAccount` with an attribute `balance` (float). Overload the `+` operator to merge the balances of two `BankAccount` objects and return a new `BankAccount` object with the combined balance.

Example:

```
account1 = BankAccount(100)
account2 = BankAccount(200)
merged_account = account1 + account2
print(merged_account.balance) # Output: 300
```

Exercise 4: Hybrid Inheritance (Detailed)

Problem: Create classes `ElectronicDevice`, `PortableDevice`, and `GPSDevice`. `SmartPhone` should inherit from `PortableDevice` and `GPSDevice`. `Tablet` should inherit from `PortableDevice`. `SmartWatch` should inherit from `PortableDevice` and `GPSDevice`. Each class should have a unique method that prints a message indicating its functionality.

Example:

```
smartphone = SmartPhone()
tablet = Tablet()
smartwatch = SmartWatch()
smartphone.make_call() # Output: Making a call...
smartwatch.track_location() # Output: Tracking location...
```