**Polymorphism in Python Programming**

Polymorphism is one of the core concepts of Object-Oriented Programming (OOP) that allows objects of different classes to be treated as objects of a common super class. It enables a single interface to be used for different data types.

In Python, polymorphism is achieved through **method overriding**, **method overloading (though not directly supported like in Java or C++)**, and **operator overloading**.

**Types of Polymorphism in Python**

**1️⃣ Method Overriding (Runtime Polymorphism)**

Method overriding occurs when a child class provides a specific implementation of a method already defined in its parent class.

🔹 **Example:**

class Animal:

def make\_sound(self):

return "Some sound"

class Dog(Animal):

def make\_sound(self):

return "Bark"

class Cat(Animal):

def make\_sound(self):

return "Meow"

# Creating objects

animals = [Dog(), Cat(), Animal()]

for animal in animals:

print(animal.make\_sound()) # Different outputs for each subclass

**Output:**

Bark

Meow

Some sound

Here, the make\_sound method is **overridden** in the Dog and Cat classes, allowing different implementations.

**2️⃣ Method Overloading (Compile-time Polymorphism)**

Python does not support traditional method overloading like Java or C++, but we can achieve it using **default arguments** or \*args/\*\*kwargs.

🔹 **Example using default arguments:**

class MathOperations:

def add(self, a, b, c=0): # Overloaded behavior using default arguments

return a + b + c

math = MathOperations()

print(math.add(5, 10)) # 15

print(math.add(5, 10, 20)) # 35

🔹 **Example using \*args:**

class MathOperations:

def add(self, \*args):

return sum(args)

math = MathOperations()

print(math.add(5, 10)) # 15

print(math.add(5, 10, 20)) # 35

print(math.add(1, 2, 3, 4)) # 10

Here, the add method works with a variable number of arguments.

**3️⃣ Operator Overloading**

Python allows operators like +, -, \*, and / to be **overloaded** using special methods (dunder methods).

🔹 **Example:**

class Vector:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_add\_\_(self, other): # Overloading the '+' operator

return Vector(self.x + other.x, self.y + other.y)

def \_\_str\_\_(self):

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

v1 = Vector(2, 3)

v2 = Vector(4, 5)

v3 = v1 + v2 # Uses the \_\_add\_\_ method

print(v3) # Output: Vector(6, 8)

Here, we overloaded the + operator to allow vector addition.

**Why Use Polymorphism?**

✅ **Code Reusability** – Same interface can be used for different data types.  
✅ **Flexibility & Scalability** – New classes can be easily added without modifying existing code.  
✅ **Easier Maintenance** – Makes the code cleaner and more maintainable.