Inheritance and Polymorphism in Python

Inheritance and polymorphism are fundamental concepts in object-oriented programming (OOP) that help to promote code reuse and flexibility. In Python, these concepts are implemented in a straightforward manner due to the language's dynamic nature and its support for multiple inheritance.

**Inheritance**

**Inheritance** is a mechanism where a new class (derived or child class) inherits the attributes and methods from an existing class (base or parent class). This allows the child class to reuse code from the parent class and extend or override its functionality.

**Single Inheritance**

In single inheritance, a class inherits from one base class.

class Animal:

def speak(self):

print("Animal speaks")

class Dog(Animal):

def bark(self):

print("Dog barks")

# Usage

dog = Dog()

dog.speak() # Output: Animal speaks

dog.bark() # Output: Dog barks

**Multiple Inheritance**

In multiple inheritance, a class can inherit from more than one base class.

class Mammal:

def feed\_milk(self):

print("Mammal feeds milk")

class Bird:

def lay\_eggs(self):

print("Bird lays eggs")

class Platypus(Mammal, Bird):

pass

# Usage

platypus = Platypus()

platypus.feed\_milk() # Output: Mammal feeds milk

platypus.lay\_eggs() # Output: Bird lays eggs

**Polymorphism**

**Polymorphism** allows objects of different classes to be treated as objects of a common super class. It is achieved by having multiple classes implement the same method or attribute. Polymorphism promotes flexibility and makes it easier to integrate components.

**Method Overriding**

Method overriding allows a child class to provide a specific implementation for a method that is already defined in its parent class.

class Animal:

def speak(self):

print("Animal speaks")

class Dog(Animal):

def speak(self):

print("Dog barks")

class Cat(Animal):

def speak(self):

print("Cat meows")

# Usage

animals = [Dog(), Cat()]

for animal in animals:

animal.speak()

# Output:

# Dog barks

# Cat meows

**Polymorphism with Functions**

Polymorphism can be demonstrated by functions that can take objects of different classes and perform actions based on their methods.

def make\_animal\_speak(animal):

animal.speak()

# Usage

make\_animal\_speak(Dog()) # Output: Dog barks

make\_animal\_speak(Cat()) # Output: Cat meows

**Example: Real-World Scenario**

Consider a real-world scenario where you have different types of shapes, and you want to compute their areas.

class Shape:

def area(self):

pass

class Rectangle(Shape):

def \_\_init\_\_(self, width, height):

self.width = width

self.height = height

def area(self):

return self.width \* self.height

class Circle(Shape):

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

return 3.14 \* self.radius \*\* 2

# Usage

shapes = [Rectangle(2, 3), Circle(5)]

for shape in shapes:

print(shape.area())

# Output:

# 6

# 78.5

In this example:

* The Shape class defines a common interface with the area method.
* The Rectangle and Circle classes inherit from Shape and provide specific implementations of the area method.
* The shapes list contains different types of shapes, and the code can compute their areas without knowing their specific types, demonstrating polymorphism.

**Summary**

* **Inheritance** allows a new class to inherit attributes and methods from an existing class, promoting code reuse.
  + **Single Inheritance**: A class inherits from one base class.
  + **Multiple Inheritance**: A class inherits from more than one base class.
* **Polymorphism** allows objects of different classes to be treated as objects of a common super class, promoting flexibility and integration.
  + **Method Overriding**: A child class provides a specific implementation for a method already defined in its parent class.
  + **Polymorphism with Functions**: Functions can operate on objects of different classes that implement the same interface.

By leveraging inheritance and polymorphism, you can create more modular, reusable, and maintainable code in .