Polymorphism in Python OOPs can be categorized into two main types: compile-time polymorphism (also known as static or method overloading) and runtime polymorphism (also known as dynamic or method overriding). Let's explore each type with examples:

1. Compile-Time Polymorphism (Static Method Overloading):

Compile-time polymorphism involves using the same method or function name but with different parameters or arguments. The decision about which method to call is made at compile time based on the method signature.

Example:

```
class MathOperations:
    def add(self, x, y):
        return x + y

    def add(self, x, y, z):
        return x + y + z


math_ops = MathOperations()
result1 = math_ops.add(2, 3)
result2 = math_ops.add(2, 3, 4)

print("Result 1:", result1)  # Output: Result 1: 5
print("Result 2:", result2)  # Output: Result 2: 9
```

In this example, the `**MathOperations**` class has two `**add**` methods with different numbers of parameters. The appropriate method is chosen at compile time based on the number and types of arguments passed during the method call.

2. Runtime Polymorphism (Dynamic Method Overriding):

Runtime polymorphism occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. This is achieved through method overriding, where the subclass provides its own version of a method with the same signature as the superclass.

Example:

```
class Animal:
    def make_sound(self):
        print("Generic animal sound")

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

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


dog = Dog()
cat = Cat()

dog.make_sound()  # Output: Woof! Woof!
cat.make_sound()  # Output: Meow!
```

In this example, the **`Animal`** class has a method called **`make_sound`**. The **`Dog`** and **`Cat`** classes, which are subclasses of **`Animal`**, provide their own implementations of the **`make_sound`** method. The method to be executed is determined at runtime based on the type of the object.

Runtime polymorphism is achieved through method overriding, and it enhances code flexibility by allowing different classes to provide their own implementations of common methods.

Compile-time vs Runtime Polymorphism in Python OOPs

Feature	Compile-time Polymorphism (Method Overloading)	Runtime Polymorphism (Method Overriding)
Mechanism	Multiple methods with the same name but different parameter lists (number, type, order)	Subclass providing its own implementation of a method inherited from a parent class
Binding time	Compile time	Runtime, based on the actual object type at execution
Inheritance	Not required	Requires inheritance hierarchy
Flexibility	Limited to different parameters	More flexible, allows specific behavior changes
Performance	Faster, compiler chooses the correct method early	Slower, virtual table lookup at runtime

Additional Notes:

- Python does not support true function overloading due to dynamic typing, but similar behavior can be achieved with methods and different parameter combinations.

- Method overriding requires the overridden method to have the same name, parameter list, and return type as the parent class method.
- Runtime polymorphism is often used for implementing more dynamic and flexible behavior in object-oriented systems.