Polymorphism

Polymorphism in Python, just like in other object-oriented programming languages, allows objects of different classes to be treated as objects of a common superclass. This can be done using method overriding (runtime polymorphism) and method overloading (though Python doesn't directly support method overloading, it achieves similar behavior).

Here's a detailed look at **Polymorphism in Python** with examples:

1. Runtime Polymorphism (Method Overriding) in Python

Runtime polymorphism occurs when a method in a subclass overrides a method in a superclass. The method that gets called is determined at runtime based on the object's type, not the reference type.

Example:

```
python

class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

class Cat(Animal):
    def speak(self):
        print("Cat meows")

# Polymorphism in action
def animal_sound(animal):
    animal.speak()  # The appropriate speak() method is called based on the
object's actual type

# Create instances of Dog and Cat
dog = Dog()
cat = Cat()

# Using the same function, polymorphism allows us to call the appropriate
method
animal_sound(dog)  # Outputs: Dog barks
animal_sound(cat)  # Outputs: Cat meows
```

Explanation:

- `animal_sound()` function accepts an `Animal` object, but it can call the `speak()` method on any subclass of `Animal` (like `Dog` or `Cat`).
- At runtime, the correct `speak()` method is called depending on the type of the object (`dog` or `cat`), demonstrating **runtime polymorphism**.

2. Method Overloading Simulation in Python

Python does not support method overloading in the traditional sense (like Java or C++), meaning you cannot define multiple methods with the same name but different arguments. However, you can simulate method overloading using default arguments, variable-length arguments (`*args` and `**kwargs`), or by checking the number and type of arguments inside the method.

Example:

```
python

class Calculator:
    def add(self, *args):
        result = 0
        for num in args:
            result += num
        return result

# Creating an instance of Calculator
calc = Calculator()

# Calling the add method with different numbers of arguments
print(calc.add(1, 2))           # Outputs: 3
print(calc.add(1, 2, 3))        # Outputs: 6
print(calc.add(1, 2, 3, 4, 5))  # Outputs: 15
```

Explanation:

- The `add()` method uses `*args` to accept a variable number of arguments.
- This simulates method overloading by allowing the method to work with any number of parameters, giving you flexibility in how it's called.

3. Polymorphism Using Inheritance and Interfaces (Duck Typing)

Python uses a concept called **duck typing**, which is a type of polymorphism where an object's behavior is determined by its methods and properties, rather than its actual class. It's named after the saying: *"If it looks like a duck, swims like a duck, and quacks like a duck, it must be a duck."*

In Python, we don't need an explicit interface to declare polymorphism — if an object implements the required methods, we can treat it like the expected type.

Example:

```
python
```

```
class Duck:
    def speak(self):
        print("Quack")

class Person:
    def speak(self):
        print("Hello")

def make_speak(entity):
    entity.speak()  # Calls the speak method on any object that has a 'speak'
method

# Polymorphism in action
duck = Duck()
person = Person()

make_speak(duck)    # Outputs: Quack
make_speak(person)  # Outputs: Hello
```

Explanation:

- Both `Duck` and `Person` have a `speak()` method, so they can be treated interchangeably in the `make_speak()` function.
- Python doesn't enforce any formal interface or inheritance for polymorphism, but it relies on the principle of "duck typing" — as long as the object has the required method, it can be used as the expected type.

4. Polymorphism in Python with Abstract Base Classes (ABCs)

While Python doesn't have traditional interfaces like other languages, you can use the `abc` module to define abstract base classes, which enforce that derived classes implement certain methods.

Example:

```
python

from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

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

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

```
# Polymorphism with abstract base class enforcement
def animal_sound(animal: Animal):
    animal.speak()

# Create instances of Dog and Cat
dog = Dog()
cat = Cat()

animal_sound(dog)  # Outputs: Woof!
animal_sound(cat)  # Outputs: Meow!
```

Explanation:

- The `Animal` class is an abstract base class that defines the `speak()` method as an abstract method, forcing any subclass to implement `speak()`.
- The `animal_sound()` function accepts any object that is an instance of `Animal` or its subclasses, showing how polymorphism can work with abstract classes.

Key Takeaways from Polymorphism in Python:

1. **Runtime Polymorphism**: Achieved using method overriding (subclass methods replacing superclass methods).
2. **Method Overloading Simulation**: Python doesn’t support method overloading directly, but you can use default arguments or variable-length arguments (`*args`).
3. **Duck Typing**: Python supports polymorphism without the need for explicit interfaces; if an object has the required methods, it can be used like any other object of the expected type.
4. **Abstract Base Classes**: You can enforce polymorphism with abstract methods using Python’s `abc` module to define common interfaces for subclasses.