Sharif University of Technology
Department of Computer Engineering

Fundamentals of Programming

Python Language

Arman Malekzadeh
PhD Candidate in Artificial Intelligence

Table of contents

Class Polymorphism

Polymorphism in Python

- Polymorphism in Python refers to the ability of different types of objects to respond to the same method or operator in different but appropriate ways.
- It's a key concept in object-oriented programming (OOP) that enables functions or methods to process objects differently based on their class or data type.
- The term "polymorphism" comes from the Greek words "poly" (many) and "morph" (form), indicating the ability to take on many forms.

Applications of Polymorphism

- **Flexibility**: Polymorphism allows for flexibility and the integration of new features without modifying existing code.
- **Code Reusability**: It promotes code reusability through the use of common interfaces.
- **Extensibility**: Systems can be easily extended by adding new classes that implement existing methods.
- **Simplifying Code**: It helps in simplifying the code as one interface can handle a variety of data types.

Polymorphism in Python: Duck Typing

- Duck typing is a concept related to dynamic typing, where the type or the class of an object is less important than the methods it defines.
- When you use duck typing, you do not check types at all. Instead, you check for the presence of a given method or attribute.
- Duck typing is usually explained by saying "If it looks like a duck and quacks like a duck, then it must be a duck."

Method Overriding

In Python, polymorphism often manifests through method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass.

```
class Bird:
    def fly(self):
        print("Some birds can fly")

class Ostrich(Bird):
    def fly(self):
        print("Ostriches cannot fly")

bird = Bird()
ostrich = Ostrich()

bird.fly()  # Output: Some birds can fly
ostrich.fly()  # Output: Ostriches cannot fly
```

Operator Overriding

The following example shows how the + operator can be defined for two custom objects.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

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

p1 = Point(1, 2)
p2 = Point(2, 3)
result = p1 + p2
print(result.x, result.y)  # Output: 3 5
```

Class Inheritance Polymorphism

- Class inheritance polymorphism is a fundamental concept in object-oriented programming (OOP) where a subclass inherits properties and behavior (methods) from a parent class but can also modify or extend these behaviors.
- This is a form of polymorphism because the subclass can be treated as an instance of the parent class, but it can also behave differently.

Class Inheritance Polymorphism: Applications

- **Creating Extensible and Maintainable Code**: By using class inheritance polymorphism, you can write more general code in the superclass and specify only what is different in the subclasses.
- **Code Reusability**: It allows for reusing code of the superclass, reducing redundancy.
- **Flexibility**: Polymorphic code can work with objects of different classes as long as they inherit from the same superclass. This makes the code more flexible and adaptable to change.

Class inheritance polymorphism is a powerful tool in Python for creating organized, reusable, and adaptable code, especially in large and complex software systems. It encapsulates the principles of OOP, making it easier to manage and scale the codebase.

Class Inheritance Polymorphism: Example

```
class Animal:
    def speak(self):
        raise NotImplementedError("Subclass must implement abstract method")

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

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

animals = [Dog(), Cat()]

for animal in animals:
    print(animal.speak())
```

Polymorphism Complex Example 1: Part 1

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

class SwimmingCreature:
    def swim(self):
        return "I can swim!"

class Duck(FlyingCreature, SwimmingCreature):
    pass

class Airplane:
    def fly(self):
        return "Flying at high altitudes!"

class Submarine:
    def swim(self):
        return "Diving deep!"
```

Polymorphism Complex Example 1: Part 2

```
def adventure(traveler):
    try:
        print(traveler.fly())
    except AttributeError:
        print("Cannot fly!")
    try:
        print(traveler.swim())
    except AttributeError:
        print("Cannot swim!")

duck = Duck()
airplane = Airplane()
submarine = Submarine()

adventure(duck)         # Duck can fly and swim
adventure(airplane)     # Airplane can fly, but can't swim
adventure(submarine)    # Submarine can swim, but can't fly
```

Polymorphism Complex Example 1: Trickiness

- The **adventure** function is designed to work with any object that has **fly** or **swim** methods, regardless of the object's class.
- This demonstrates duck typing: it's not the type of the object that matters, but whether it has the required methods.
- The **Duck** class inherits from both **FlyingCreature** and **SwimmingCreature**. Python determines the method resolution order at runtime, which can be complex in multiple inheritance scenarios.
- The **adventure** function dynamically calls methods (**fly** and **swim**) based on the passed object.
- The binding of these methods happens at runtime, demonstrating **dynamic binding**.
- This allows flexibility but can be tricky to debug if the behavior of the methods varies significantly across classes.

Introduction to Method Resolution Order (MRO)

- Method Resolution Order (MRO) is a principle in object-oriented programming.
- Determines the order in which base classes are searched for a method.
- Crucial for understanding multiple inheritance in Python.

How MRO Works

- Python uses C3 linearization to calculate MRO.
- Two rules:
  1. Child classes are checked before parent classes.
  2. In multiple inheritance, classes are searched in the order they are specified.
- MRO can be viewed using the `mro()` method or `__mro__` attribute.

Example Demonstrating MRO

```
class A:
    def do_something(self):
        return "Method defined in A"

class B(A):
    def do_something(self):
        return "Method defined in B"

class C(A):
    def do_something(self):
        return "Method defined in C"

class D(B, C):
    pass

d_instance = D()
print(d_instance.do_something())
# Output: "Method defined in B"
```

MRO in Class D

- MRO for class D: [D, B, C, A, object]
- Method `do_something` from class B is called.
- MRO dictates method overriding in multiple inheritance.

References

References I

[1] B Downey, A. (2015). Think Python: How to Think Like a Computer Scientist-2nd Edition.

[2] Deitel, H. M., & Deitel, P. J. (2004). C: How to program. Pearson Educacion.

Sharif University of Technology
Department of Computer Engineering

Arman Malekzadeh

Fundamentals of Programming

Python Language