Basic Programming with Python

Prepared By:

Professor Dr. Md. Mijanur Rahman
Department of Computer Science & Engineering
Jatiya Kabi Kazi Nazrul Islam University, Bangladesh.
www.mijanrahman.com

8

OOP Concepts in Python

CONTENTS

8.11. POLYMORPHISM IN PYTHON

8.11.1 What is Polymorphism?

Polymorphism in Python refers to the ability of different objects to be treated as instances of a common interface. This means that different classes can define methods with the same name, but with different implementations. When we call a method on an object, Python determines which implementation of the method to execute based on the type of the object.

Two main types of polymorphism in Python:

1. **Method Overriding:** This occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. When we call the method on an instance

of the subclass, the subclass's implementation is executed instead of the superclass's implementation.

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

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

animal = Animal()
animal.sound()  # Output: Animal makes a sound

dog = Dog()
dog.sound()  # Output: Dog barks
```

2. **Method Overloading:** Unlike some other programming languages, Python does not support method overloading by default (i.e., defining multiple methods with the same name but different signatures). However, we can achieve a similar effect using default argument values or variable arguments.

```
class Math:
    def add(self, a, b):
        return a + b

    def add(self, a, b, c):
        return a + b + c

math = Math()
print(math.add(2, 3))
print(math.add(2, 3, 4))
```

8.11.2 In-built Polymorphic Functions

In Python, there are several built-in functions that exhibit polymorphic behavior, meaning they can operate on different types of objects and produce different results based on the type of the input. Some of the key built-in polymorphic functions in Python include:

- len(): This function returns the length of a sequence (such as a string, list, tuple, or dictionary). It works polymorphically on different types of sequences.

```
print(len("hello"))      # Output: 5
print(len([1, 2, 3, 4]))  # Output: 4
print(len((1, 2, 3)))     # Output: 3
print(len({"a": 1, "b": 2}))  # Output: 2 (number of keys)
```

- str(): The str() function converts an object into a string representation. It works polymorphically on various types of objects.

```
print(str(123))    # Output: '123'
print(str([1, 2])) # Output: '[1, 2]'
print(str({"a": 1})) # Output: "{'a': 1}"
```

- max() and min(): These functions return the maximum and minimum values from a sequence. They work polymorphically on different types of sequences as long as the elements can be compared.

```
print(max(1, 2, 3))          # Output: 3
print(max([1, 2, 3, 4, 5]))   # Output: 5
print(min("hello"))          # Output: 'e'
print(min((5, 3, 9, 1)))      # Output: 1
```

- sum(): The sum() function returns the sum of all elements in a sequence. It works polymorphically on different types of sequences containing numeric values.

```
print(sum([1, 2, 3]))    # Output: 6
print(sum((4, 5, 6)))    # Output: 15
```

- sorted(): The sorted() function returns a new sorted list from the elements of any iterable. It works polymorphically on different types of iterables.

```
print(sorted([3, 1, 2]))      # Output: [1, 2, 3]
print(sorted("hello"))        # Output: ['e', 'h', 'l', 'l', 'o']
print(sorted({3, 1, 2}))      # Output: [1, 2, 3]
```

Python program demonstrating the use of some in-built polymorphic functions:

```
# Function to demonstrate polymorphism using len()
def demo_len(obj):
    print(f"Length of {obj}: {len(obj)}")

# Function to demonstrate polymorphism using str()
def demo_str(obj):
    print(f"String representation of {obj}: {str(obj)}")

# Function to demonstrate polymorphism using max() and min()
def demo_max_min(seq):
    print(f"Maximum value in {seq}: {max(seq)}")
    print(f"Minimum value in {seq}: {min(seq)}")

# Function to demonstrate polymorphism using sum()
def demo_sum(seq):
    print(f"Sum of elements in {seq}: {sum(seq)}")

# Function to demonstrate polymorphism using sorted()
def demo_sorted(iterable):
    print(f"Sorted version of {iterable}: {sorted(iterable)}")

# Main function
```

```
def main():
    # Different types of objects
    string_obj = "hello"
    list_obj = [1, 2, 3, 4, 5]
    tuple_obj = (5, 4, 3, 2, 1)
    set_obj = {3, 1, 2}
    dictionary_obj = {"a": 1, "b": 2, "c": 3}

    # Demonstrating polymorphism using various built-in functions
    demo_len(string_obj)
    demo_len(list_obj)
    demo_len(tuple_obj)
    demo_len(dictionary_obj)

    demo_str(123)
    demo_str([1, 2, 3])
    demo_str({"a": 1, "b": 2})

    demo_max_min([5, 3, 9, 1])
    demo_max_min("hello")
    demo_max_min({3, 1, 2})

    demo_sum([1, 2, 3, 4, 5])
    demo_sum((4, 5, 6))

    demo_sorted([3, 1, 2])
    demo_sorted("hello")
    demo_sorted({3, 1, 2})

if __name__ == "__main__":
    main()
```

8.11.3 Polymorphism with Class Methods

Polymorphism with class methods in Python involves defining methods in different classes with the same name but different implementations. When these methods are called on instances of different classes, Python determines which implementation to execute based on the type of the object. This concept is fundamental to object-oriented programming and allows for more flexible and modular code.

Here's an example demonstrating polymorphism with class methods in Python:

```
class Animal:
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
```

```
        print("Dog barks")

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

class Bird(Animal):
    def make_sound(self):
        print("Bird chirps")

# Function to make an animal sound
def make_animal_sound(animal):
    animal.make_sound()

# Main function
def main():
    dog = Dog()
    cat = Cat()
    bird = Bird()

    make_animal_sound(dog)  # Output: Dog barks
    make_animal_sound(cat)  # Output: Cat meows
    make_animal_sound(bird) # Output: Bird chirps

if __name__ == "__main__":
    main()
```

In this example:

- There is a base class **Animal** with a method **make_sound()** which serves as the common interface.
- There are three subclasses **Dog, Cat,** and **Bird,** each with its own implementation of the **make_sound()** method.
- The **make_animal_sound()** function takes an **Animal object** as input and calls its **make_sound()** method.
- When **make_animal_sound()** is called with instances of Dog, Cat, and Bird, polymorphism ensures that the appropriate **make_sound()** method associated with each object is executed.

8.11.4 Polymorphism with Inheritance

Polymorphism with inheritance in Python refers to the ability of different classes to share a common interface through inheritance while providing their own specific implementations of methods. This allows objects of different classes to be treated interchangeably when interacting with shared methods or functions, promoting code reuse and flexibility.

Here's an example demonstrating polymorphism with inheritance in Python:

```
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

# Function to calculate the area of a shape
def calculate_area(shape):
    print("Area:", shape.area())

# Main function
def main():
    rectangle = Rectangle(5, 4)
    circle = Circle(3)

    calculate_area(rectangle)  # Output: Area: 20
    calculate_area(circle)     # Output: Area: 28.26 (approximately)

if __name__ == "__main__":
    main()
```

In this example:

- There is a base class **Shape** with a method **area()** serving as the common interface.
- There are two subclasses **Rectangle** and **Circle**, each providing its own implementation of the **area()** method to calculate the area of the respective shape.
- The **calculate_area()** function takes a **Shape** object as input and calls its **area()** method.
- When **calculate_area()** is called with instances of **Rectangle** and **Circle**, polymorphism ensures that the appropriate **area()** method associated with each object is executed.

8.11.5 Polymorphism with a Function and Objects

Polymorphism in Python can also be achieved through functions that accept objects of different classes, treating them uniformly through a shared interface. Here's an example demonstrating polymorphism with a function and objects in Python:

```
class Vehicle:
    def move(self):
        pass

class Car(Vehicle):
    def move(self):
        return "Car is driving"

class Bicycle(Vehicle):
    def move(self):
        return "Bicycle is cycling"

class Boat(Vehicle):
    def move(self):
        return "Boat is sailing"

# Function to make a vehicle move
def make_vehicle_move(vehicle):
    if isinstance(vehicle, Vehicle):
        print(vehicle.move())
    else:
        print("Not a valid vehicle object")

# Main function
def main():
    car = Car()
    bicycle = Bicycle()
    boat = Boat()

    make_vehicle_move(car)      # Output: Car is driving
    make_vehicle_move(bicycle)  # Output: Bicycle is cycling
    make_vehicle_move(boat)     # Output: Boat is sailing

    # Passing an invalid object
    make_vehicle_move("Airplane") # Output: Not a valid vehicle object

if __name__ == "__main__":
    main()
```

In this example:

- There is a base class **Vehicle** with a method **move()** serving as the common interface.

- There are three subclasses **Car, Bicycle,** and **Boat,** each providing its own implementation of the **move()** method.
- The **make_vehicle_move()** function accepts an object as input and checks if it's an instance of **Vehicle**. If it is, it calls the **move()** method of that object.
- When **make_vehicle_move()** is called with instances of Car, Bicycle, and Boat, polymorphism ensures that the appropriate move() method associated with each object is executed.
- If an object that is not an instance of **Vehicle** is passed to **make_vehicle_move()**, it outputs a message indicating that it's not a valid vehicle object.

⌘⌘⌘