Object-Oriented Programming Concepts for Optimization (Python vs Java)

Operations Research Students

January 29, 2025

Introduction

In this document, we will explore key concepts in Object-Oriented Programming (OOP) and how they can be applied to solving optimization problems commonly used in Operations Research. OOP is a programming paradigm that organizes software design around data, or objects, rather than functions and logic.

We will provide examples of OOP concepts in both Python and Java, including the following:

- Definition of a Class
- Attributes and Methods
- Polymorphism
- Inheritance
- Abstract Classes
- Access Modifiers (Private and Protected)
- Encapsulation
- Multiple Inheritance

1. Class Definition

A class is a blueprint for creating objects. It defines a set of attributes and methods that the created objects will have. In OOP, a class provides the structure for an object.

Python Example:

Listing 1: Python Class Definition

```python
class Solver:
    def __init__(self, objective_function):
        self.objective_function = objective_function  # Attribute

    def solve(self):  # Method
        pass  # This will be overridden in child classes
```

Java Example:

Listing 2: Java Class Definition

```java
public class Solver {
    private Function<Double, Double> objectiveFunction; // Attribute

    public Solver(Function<Double, Double> objectiveFunction) {
        this.objectiveFunction = objectiveFunction;
    }

    public void solve() {
        // This will be overridden in child classes
    }
}
```

2. Attributes and Methods

Attributes (also known as fields or properties) are variables associated with an object. Methods are functions that define the behaviors of the objects created from a class.

Python Example:

Listing 3: Attributes and Methods in Python

```python
class Solver:
    def __init__(self, objective_function):
        self.objective_function = objective_function  # Attribute

    def solve(self):  # Method
        print("Solving...")
        return self.objective_function(10)  # Example usage
```

Java Example:

Listing 4: Attributes and Methods in Java

```java
public class Solver {
    private Function<Double, Double> objectiveFunction; // Attribute

    public Solver(Function<Double, Double> objectiveFunction) {
        this.objectiveFunction = objectiveFunction;
    }

    public double solve() { // Method
        System.out.println("Solving...");
        return objectiveFunction.apply(10.0); // Example usage
    }
}
```

3. Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It is commonly used to call methods from different classes using the same interface.

Python Example:

Listing 5: Polymorphism in Python

```python
class ZerothOrderMethod(Solver):
    def solve(self):
        print("Solving using Zeroth-Order Method")
        return "Zeroth Order Solution"

class FirstOrderMethod(Solver):
    def solve(self):
        print("Solving using First-Order Method")
        return "First Order Solution"

# Example of polymorphism
solver = ZerothOrderMethod(lambda x: x**2)
print(solver.solve())

solver = FirstOrderMethod(lambda x: x**2)
print(solver.solve())
```

Java Example:

Listing 6: Polymorphism in Java

```java
public class ZerothOrderMethod extends Solver {
    public ZerothOrderMethod(Function<Double, Double> objectiveFunction
        ) {
        super(objectiveFunction);
    }

    @Override
    public String solve() {
        System.out.println("Solving using Zeroth-Order Method");
        return "Zeroth Order Solution";
    }
}

public class FirstOrderMethod extends Solver {
    public FirstOrderMethod(Function<Double, Double> objectiveFunction)
        {
        super(objectiveFunction);
    }

    @Override
    public String solve() {
        System.out.println("Solving using First-Order Method");
        return "First Order Solution";
    }
```

```
}

// Polymorphism example
Solver solver = new ZerothOrderMethod(x -> x * x);
System.out.println(solver.solve());

solver = new FirstOrderMethod(x -> x * x);
System.out.println(solver.solve());
```

4. Inheritance

Inheritance is a mechanism where one class acquires the attributes and methods of another class. The class that is inherited from is called the superclass, and the class that inherits is the subclass.

Python Example:

Listing 7: Inheritance in Python

```python
class Solver:
    def __init__(self, objective_function):
        self.objective_function = objective_function

    def solve(self):
        pass

class FirstOrderMethod(Solver):
    def solve(self):
        return "First Order Solution"
```

Java Example:

Listing 8: Inheritance in Java

```java
public class Solver {
    protected Function<Double, Double> objectiveFunction;

    public Solver(Function<Double, Double> objectiveFunction) {
        this.objectiveFunction = objectiveFunction;
    }

    public String solve() {
        return "Solving using base Solver class";
    }
}

public class FirstOrderMethod extends Solver {
    public FirstOrderMethod(Function<Double, Double> objectiveFunction)
        {
        super(objectiveFunction);
    }

    @Override
    public String solve() {
        return "First Order Solution";
    }
}
```

5. Abstract Classes

An abstract class cannot be instantiated on its own and must be subclassed. It is used to define methods that must be implemented in derived classes.

Python Example:

Listing 9: Abstract Class in Python

```python
from abc import ABC, abstractmethod

class Solver(ABC):
    @abstractmethod
    def solve(self):
        pass

class FirstOrderMethod(Solver):
    def solve(self):
        return "First Order Solution"
```

Java Example:

Listing 10: Abstract Class in Java

```java
public abstract class Solver {
    protected Function<Double, Double> objectiveFunction;

    public Solver(Function<Double, Double> objectiveFunction) {
        this.objectiveFunction = objectiveFunction;
    }

    public abstract String solve();
}

public class FirstOrderMethod extends Solver {
    public FirstOrderMethod(Function<Double, Double> objectiveFunction)
        {
        super(objectiveFunction);
    }

    @Override
    public String solve() {
        return "First Order Solution";
    }
}
```

6. Access Modifiers: Private, Protected, Public

In OOP, access modifiers control the visibility of class members (fields and methods). These modifiers are:

- **Private**: Can only be accessed within the class.
- **Protected**: Can be accessed within the class and by subclasses.
- **Public**: Can be accessed from anywhere.

In Python and Java, the concept of private and protected fields is handled differently. Let's explore how both languages manage field visibility.

Python:

In Python, there is no strict enforcement of private or protected fields. However, Python uses a naming convention to indicate the visibility of a class member.

- A single underscore (e.g., `_protected_field`) indicates that the attribute is intended to be protected.
- A double underscore (e.g., `__private_field`) triggers name mangling, which makes the attribute harder (but not impossible) to access.

Java:

In Java, access modifiers are enforced. You can define fields as `private`, `protected`, or `public` to control access from other classes.

- `private` restricts access to the same class.
- `protected` allows access in subclasses and classes in the same package.
- `public` allows access from anywhere.

Python Example:

Listing 11: Private and Protected Members in Python

```python
class Solver:
    def __init__(self, objective_function):
        self.__objective_function = objective_function  # Private attribute

    def solve(self):
        print("Solving...")
        return self.__objective_function(10)

class FirstOrderMethod(Solver):
    def __init__(self, objective_function):
        super().__init__(objective_function)

    def solve(self):
        print("Solving with First Order Method")
        return super().solve()
```

Java Example:

Listing 12: Private and Protected Members in Java

```java
public class Solver {
    private Function<Double, Double> objectiveFunction; // Private
        attribute

    public Solver(Function<Double, Double> objectiveFunction) {
        this.objectiveFunction = objectiveFunction;
    }

    public String solve() {
        System.out.println("Solving...");
        return objectiveFunction.apply(10.0);
    }
}

public class FirstOrderMethod extends Solver {
    public FirstOrderMethod(Function<Double, Double> objectiveFunction)
        {
        super(objectiveFunction);
    }

    @Override
    public String solve() {
        System.out.println("Solving with First Order Method");
        return super.solve();
    }
}
```

7. Multiple Inheritance

Multiple inheritance refers to a feature of object-oriented programming languages in which a class can inherit attributes and methods from more than one class. This can be useful when a class needs to inherit behaviors from multiple sources.

Python:

Python supports multiple inheritance, meaning a class can inherit from more than one class. While this allows for greater flexibility, it can lead to complexity, particularly when there is ambiguity in method resolution (e.g., when two parent classes have methods with the same name).

Python Example:

Listing 13: Multiple Inheritance in Python

```python
class Optimization:
    def optimize(self):
        print("Optimizing...")

class Solver:
    def solve(self):
        print("Solving...")

class FirstOrderMethod(Solver, Optimization):
    def solve(self):
        print("Solving with First Order Method")
        super().solve()  # Calling method from Solver
        self.optimize()  # Calling method from Optimization
```

Java:

Java does not support multiple inheritance for classes. However, Java allows a class to implement multiple interfaces, which is somewhat similar to multiple inheritance.

Java Example:

Listing 14: Multiple Inheritance in Java (using interfaces)

```java
interface Optimization {
    void optimize();
}

interface Solver {
    void solve();
}

public class FirstOrderMethod implements Solver, Optimization {
    @Override
    public void solve() {
        System.out.println("Solving with First Order Method");
    }
```

```
14
15      @Override
16      public void optimize() {
17          System.out.println("Optimizing...");
18      }
19  }
```

Exercise Description

1. **Base Class Creation:**
   - Define an abstract class `Solver` with:
     - A protected/private attribute `objective_function`.
     - An abstract method `solve()`.
     - A method `evaluate(x)` to compute the objective function value at a given `x`.
2. **Inheritance and Implementation:**
   - Create two subclasses:
     - `ZerothOrderSolver`: Implements `solve()` to evaluate the function at predefined points.
     - `FirstOrderSolver`: Implements `solve()` using gradient descent.
3. **Polymorphism:**
   - Implement a function `run_solver(solver)` that accepts any subclass of `Solver` and calls its `solve()` method.
4. **Encapsulation and Access Modifiers:**
   - In Python:
     - Make `objective_function` a private attribute.
     - Add a protected method `_prepare()` for preprocessing.
   - In Java:
     - Use `private` for `objectiveFunction`.
     - Use `protected` methods for preprocessing.
5. **Multiple Inheritance (Python Only):**
   - Create a mixin class `Logger` with a `log(message)` method.
   - Modify `FirstOrderSolver` to inherit from both `Solver` and `Logger`.

Python Solution

Base Class and Subclasses

Listing 15: Base Class and Subclasses in Python

```python
from abc import ABC, abstractmethod

# Mixin class for logging
class Logger:
    def log(self, message):
        print(f"[LOG]: {message}")

class Solver(ABC):
    def __init__(self, objective_function):
        self.__objective_function = objective_function  # Private attribute

    def evaluate(self, x):
        return self.__objective_function(x)  # Evaluate function

    @abstractmethod
    def solve(self):
        pass  # To be implemented in subclasses

    # Protected method for preprocessing
    def _prepare(self):
        print("Preparing solver...")

class ZerothOrderSolver(Solver):
    def solve(self):
        print("Solving using Zeroth Order Method")
        points = [1, 2, 3]
        best_point = min(points, key=self.evaluate)
        print(f"Optimal solution: x = {best_point}, f(x) = {self.evaluate(best_point)}")
        return best_point

class FirstOrderSolver(Solver):
    def solve(self):
        print("Solving using Gradient Descent")
        x = 5.0  # Starting point
        for _ in range(10):  # Perform gradient descent for 10 iterations
            gradient = 2 * (x - 3)
            x -= 0.1 * gradient  # Update step
            print(f"Iteration: x = {x}, f(x) = {self.evaluate(x)}")
        print(f"Optimal solution: x = {x}, f(x) = {self.evaluate(x)}")
        return x
```

Polymorphism Example

Listing 16: Polymorphism Example in Python

```python
def run_solver(solver):
    solver.solve()

objective = lambda x: (x - 3)**2
run_solver(ZerothOrderSolver(objective))
run_solver(FirstOrderSolver(objective))
```

Java Solution

Base Class and Subclasses

Listing 17: Base Class and Subclasses in Java

```java
import java.util.function.Function;

abstract class Solver {
    private Function<Double, Double> objectiveFunction; // Private attribute

    public Solver(Function<Double, Double> objectiveFunction) {
        this.objectiveFunction = objectiveFunction;
    }

    protected double evaluate(double x) {
        return objectiveFunction.apply(x); // Evaluate function
    }

    public abstract void solve();
}

class ZerothOrderSolver extends Solver {
    public ZerothOrderSolver(Function<Double, Double> objectiveFunction) {
        super(objectiveFunction);
    }

    @Override
    public void solve() {
        System.out.println("Solving using Zeroth Order Method");
        double[] points = {1, 2, 3};
        double bestPoint = points[0];
        for (double point : points) {
            if (evaluate(point) < evaluate(bestPoint)) {
                bestPoint = point;
            }
        }
        System.out.printf("Optimal solution: x = %.2f, f(x) = %.2f%n", bestPoint, evaluate(bestPoint));
    }
}

class FirstOrderSolver extends Solver {
```

```java
    public FirstOrderSolver(Function<Double, Double> objectiveFunction) {
        super(objectiveFunction);
    }

    @Override
    public void solve() {
        System.out.println("Solving using Gradient Descent");
        double x = 5.0; // Starting point
        for (int i = 0; i < 10; i++) {
            double gradient = 2 * (x - 3);
            x -= 0.1 * gradient; // Update step
            System.out.printf("Iteration: x = %.2f, f(x) = %.2f%n", x,
                evaluate(x));
        }
        System.out.printf("Optimal solution: x = %.2f, f(x) = %.2f%n",
            x, evaluate(x));
    }
}
```

Polymorphism Example

Listing 18: Polymorphism Example in Java

```java
public class Main {
    public static void runSolver(Solver solver) {
        solver.solve();
    }

    public static void main(String[] args) {
        Function<Double, Double> objective = x -> Math.pow(x - 3, 2);
        runSolver(new ZerothOrderSolver(objective));
        runSolver(new FirstOrderSolver(objective));
    }
}
```

Conclusion

This document covered the fundamental concepts of Object-Oriented Programming (OOP) using Python and Java, with a focus on optimization and solving problems in Operations Research. By understanding and applying concepts like inheritance, polymorphism, encapsulation, and multiple inheritance, you can design more efficient and modular solutions for complex problems.