Chapter 5 - Operator Overloading

1. Introduction

Operator Overloading is a feature in Python that allows objects of user-defined classes to interact with built-in operators (`+`, `-`, `*`, `/`, `==`, etc.). By **overloading operators**, we can define how they behave when applied to objects of a particular class.

For example, in mathematical operations, `+` is used for addition:

```
print(2 + 3)  # Output: 5
```

But for objects, Python does not understand how to perform `+` unless we define it explicitly.

1.1 Why Use Operator Overloading?

- **Enhances readability**: Writing `a + b` is more intuitive than calling `a.add(b)`.
- **Custom behaviors**: Allows user-defined classes to behave like built-in types.
- **Improves code reusability**: Enables reuse of operators in object operations.

---

2. Magic Methods (Dunder Methods)

Python provides special **"magic methods"** (also called **dunder methods**, meaning "double underscore") that allow us to define custom behavior for operators.

2.1 Common Operator Overloading Methods

Operator	Method
`+`	`__add__(self, other)`
`-`	`__sub__(self, other)`
`*`	`__mul__(self, other)`
`/`	`__truediv__(self, other)`
`//`	`__floordiv__(self, other)`
`%`	`__mod__(self, other)`
`**`	`__pow__(self, other)`

Operator	Method
`==`	`__eq__(self, other)`
`!=`	`__ne__(self, other)`
`<`	`__lt__(self, other)`
`<=`	`__le__(self, other)`
`>`	`__gt__(self, other)`
`>=`	`__ge__(self, other)`

---

3. Implementing Operator Overloading

3.1 Overloading the `+` Operator

Let's say we have a class `Vector`, and we want to add two vectors using `+`:

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

    def __add__(self, other):  # Overloading +
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1 + v2  # Calls __add__
print(v3)  # Output: Vector(6, 8)
```

3.2 Overloading the `*` Operator

We can overload `*` to scale a vector:

```
class Vector:
    def __init__(self, x, y):
```

```
        self.x = x
        self.y = y

    def __mul__(self, scalar):  # Overloading *
        return Vector(self.x * scalar, self.y * scalar)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v = Vector(3, 4)
v_scaled = v * 2  # Calls __mul__
print(v_scaled)  # Output: Vector(6, 8)
```

3.3 Overloading Comparison Operators

We can overload comparison operators like `>`, `<`, `==`:

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def __gt__(self, other):  # Overloading >
        return self.area() > other.area()

    def __eq__(self, other):  # Overloading ==
        return self.area() == other.area()

rect1 = Rectangle(5, 10)
rect2 = Rectangle(4, 12)

print(rect1 > rect2)  # Output: True
print(rect1 == rect2) # Output: False
```

3.4 Overloading `__str__` and `__repr__`

By default, printing an object gives an unreadable output:

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p = Person("Alice", 25)
print(p)  # Output: <__main__.Person object at 0x...>
```

We can override `__str__` and `__repr__` for a readable output:

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):  # Readable output for users
        return f"Person(name={self.name}, age={self.age})"

    def __repr__(self):  # Debugging output for developers
        return f"Person({self.name}, {self.age})"

p = Person("Alice", 25)
print(str(p))   # Output: Person(name=Alice, age=25)
print(repr(p))  # Output: Person(Alice, 25)
```

---

4. Operator Overloading in Django Models

Django models often need custom operators, especially for comparison or mathematical operations.

4.1 Overloading `__str__` in Django Models

Django **automatically** calls `__str__` when displaying objects in the admin panel.

```python
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.FloatField()
```

```
    def __str__(self):  # Overloading __str__
        return f"{self.name} - ${self.price}"
```

4.2 Overloading Comparison Operators in Django Models

For comparing products by price:

```
class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.FloatField()

    def __gt__(self, other):  # Overloading >
        return self.price > other.price
```

---

5. Exercises

Exercise 1: Overload the `+` operator

Create a class `BankAccount` that allows adding two accounts using `+` to merge their balances.

```
class BankAccount:
    def __init__(self, balance):
        self.balance = balance

    def __add__(self, other):
        # Your code here

# Test your implementation
acc1 = BankAccount(500)
acc2 = BankAccount(300)
merged_acc = acc1 + acc2
print(merged_acc.balance)  # Expected Output: 800
```

Exercise 2: Overload the `>`, `<`, `==` operators

Create a class `Student` where students can be compared based on their grades.

```python
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    # Your code here (Overload >, <, ==)

# Test your implementation
s1 = Student("Alice", 85)
s2 = Student("Bob", 90)
print(s1 > s2)  # Expected Output: False
print(s1 == s2) # Expected Output: False
```

Exercise 3: Overload `__str__` and `__repr__`

Modify the `Student` class to print readable output when using `print(student)`.