Python Operator Overloading

As we know, the + operator can perform addition on two numbers, merge two lists, or concatenate two strings.

With some tweaks, we can use the + operator to work with user-defined objects as well. This feature in Python, which allows the same operator to have different meanings depending on the context is called **operator overloading.**

Python Special Functions

In Python, methods that have two underscores, __, before and after their names have a special meaning. For example, `__add__()`, `__len__()` etc.

These special methods can be used to implement certain features or behaviors.

Let's use the `__add__()` method to add two numbers instead of using the + operator.

```
number1 = 5

# similar to number2 = number1 + 6
number2 = number1.__add__(6)

print(number2)  # 11
```

It is possible to use the `__add__()` method on integers because:

- Everything in Python is an object, including integers.
- Integers have the `__add__()` method defined that we can use.

In fact, the + operator internally calls the `__add__()` method to add integers and floats.

Here are some of the special functions available in Python:

Function	Description
`__init__()`	Initialize the attributes of the object.
`__str__()`	Returns a string representation of the object.
`__len__()`	Returns the length of the object.
`__add__()`	Adds two objects.
`__call__()`	Call objects of the class like a normal function.

How to Use Operator Overloading?

Suppose we want to use the + operator to add two user-defined objects.

Since the + operator internally calls the `__add__()` method, if we implement this method in a class, we can make objects of that class work with the + operator.

Example: Add Two Coordinates (Without Overloading)

Let's first write a program to add two co-ordinates (without using + operator overloading).

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

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


p1 = Point(1, 2)
p2 = Point(2, 3)

p3 = p1.add_points(p2)

print((p3.x, p3.y))   # Output: (3, 5)
```

In the above example, we created the `add_points()` method to add two points. To call this method, we have used `p1.add_points(p2)`.

Let's write the same code using the + operator to add two points.

Example: Add Two Coordinates (With Overloading)

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

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


p1 = Point(1, 2)
p2 = Point(2, 3)

# this statment calls the __add__() method
p3 = p1 + p2

print((p3.x, p3.y))   # Output: (3, 5)
```

Here, this code `p1 + p2` calls the `__add__(self, other)` method. The `self` parameter takes *p1*, and the *other* parameter takes *p2* as arguments.

Don't Misuse Operators

In the above program, we could have easily used the + operator for subtraction like this:

```
def __add__(self, other):
    x = self.x - other.x
    y = self.y - other.y
    return Point(x, y)
```

Now the + operator in the above code performs subtraction of points. Even though the program works without errors, you should absolutley avoid this. We should always use oeprators appropriately during operator overloading.

Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator	Expression	Internally
Addition	`p1 + p2`	`p1.__add__(p2)`
Subtraction	`p1 - p2`	`p1.__sub__(p2)`
Multiplication	`p1 * p2`	`p1.__mul__(p2)`
Power	`p1 ** p2`	`p1.__pow__(p2)`
Division	`p1 / p2`	`p1.__truediv__(p2)`
Floor Division	`p1 // p2`	`p1.__floordiv__(p2)`
Remainder (modulo)	`p1 % p2`	`p1.__mod__(p2)`
Bitwise Left Shift	`p1 << p2`	`p1.__lshift__(p2)`
Bitwise Right Shift	`p1 >> p2`	`p1.__rshift__(p2)`
Bitwise AND	`p1 & p2`	`p1.__and__(p2)`

Bitwise OR	`p1 | p2`	`p1.__or__(p2)`
Bitwise XOR	`p1 ^ p2`	`p1.__xor__(p2)`
Bitwise NOT	`~p1`	`p1.__invert__()`

Overloading Comparison Operators

Python does not limit operator overloading to arithmetic operators. We can overload comparison operators as well.

Here's an example of how we can overload the < operator to compare two objects of the *Person* class based on their `age`:

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

    # overload < operator
    def __lt__(self, other):
        return self.age < other.age

p1 = Person("Alice", 20)
p2 = Person("Bob", 30)

print(p1 < p2)  # prints True
print(p2 < p1)  # prints False
```

Output

```
True
False
```

Here, `__lt__()` overloads the < operator to compare the *age* attribute of two objects.

The `__lt__()` method returns:

- `True` - if the first object's *age* is less than the second object's *age*
- `False` - if the first object's *age* is greater than the second object's *age*

Similarly, the special functions that we need to implement to overload other comparison operators are tabulated below.

Operator	Expression	Internally
Less than	`p1 < p2`	`p1.__lt__(p2)`
Less than or equal to	`p1 <= p2`	`p1.__le__(p2)`
Equal to	`p1 == p2`	`p1.__eq__(p2)`
Not equal to	`p1 != p2`	`p1.__ne__(p2)`
Greater than	`p1 > p2`	`p1.__gt__(p2)`
Greater than or equal to	`p1 >= p2`	`p1.__ge__(p2)`

Advantages of Operator Overloading

Here are some advantages of operator overloading:

- Improves code readability by allowing the use of familiar operators.
- Ensures that objects of a class behave consistently with built-in types and other user-defined types.
- Makes it simpler to write code, especially for complex data types.
- Allows for code reuse by implementing one operator method and using it for other operators.

Also Read:

- Python Classes and Objects
- Self in Python, Demystified
- Precedence and Associativity of Operators in Python

Table of Contents