Python Operator Overloading

In Python, we can change the way operators work for user-defined types.

For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.

Python Special Functions

Class functions that begin with double underscore `__` are called special functions in Python.

The special functions are defined by the Python interpreter and used to implement certain features or behaviors.

They are called **"double underscore"** functions because they have a double underscore prefix and suffix, such as `__init__()` or `__add__()`.

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

Example: + Operator Overloading in Python

To overload the `+` operator, we will need to implement `__add__()` function in the class.

With great power comes great responsibility. We can do whatever we like inside this function. But it is more sensible to return the `Point` object of the coordinate sum.

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

    def __str__(self):
        return "({0},{1})".format(self.x, self.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)

print(p1+p2)

# Output: (3,5)
```

In the above example, what actually happens is that, when we use `p1 + p2`, Python calls `p1.__add__(p2)` which in turn is `Point.__add__(p1,p2)`. After this, the addition operation is carried out the way we specified.

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 only. We can overload comparison operators as well.

Here's an example of how we can overload the `<` operator to compare two objects 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
```

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.