Python Operator Overloading

programiz.com/python-programming/operator-overloading

Join our newsletter for the latest updates.

You can change the meaning of an operator in Python depending upon the operands used. In this tutorial, you will learn how to use operator overloading in Python Object Oriented Programming.

Python operators work for built-in classes. But the same operator behaves differently with different 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.

So what happens when we use them with objects of a user-defined class? Let us consider the following class, which tries to simulate a point in 2-D coordinate system.

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


p1 = Point(1, 2)
p2 = Point(2, 3)
print(p1+p2)
```

Output

```
Traceback (most recent call last):
  File "<string>", line 9, in <module>
    print(p1+p2)
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

Here, we can see that a `TypeError` was raised, since Python didn't know how to add two `Point` objects together.

However, we can achieve this task in Python through operator overloading. But first, let's get a notion about special functions.

Python Special Functions

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

These functions are not the typical functions that we define for a class. The `__init__()` function we defined above is one of them. It gets called every time we create a new object of that class.

There are numerous other special functions in Python. Visit Python Special Functions to learn more about them.

Using special functions, we can make our class compatible with built-in functions.

```
>>> p1 = Point(2,3)
>>> print(p1)
<__main__.Point object at 0x00000000031F8CC0>
```

Suppose we want the `print()` function to print the coordinates of the `Point` object instead of what we got. We can define a `__str__()` method in our class that controls how the object gets printed. Let's look at how we can achieve this:

```
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)
```

Now let's try the `print()` function again.

```
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)


p1 = Point(2, 3)
print(p1)
```

Output

```
(2, 3)
```

That's better. Turns out, that this same method is invoked when we use the built-in function `str()` or `format()`.

```
>>> str(p1)
'(2,3)'

>>> format(p1)
'(2,3)'
```

So, when you use `str(p1)` or `format(p1)`, Python internally calls the `p1.__str__()` method. Hence the name, special functions.

Now let's go back to operator overloading.

Overloading the + Operator

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 a `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)
```

Now let's try the addition operation again:

```
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)
```

What actually happens is that, when you 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.

Suppose we wanted to implement the less than symbol `<` symbol in our `Point` class.

Let us compare the magnitude of these points from the origin and return the result for this purpose. It can be implemented as follows.

```
# overloading the less than operator
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 __lt__(self, other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag

p1 = Point(1,1)
p2 = Point(-2,-3)
p3 = Point(1,-1)

# use less than
print(p1<p2)
print(p2<p3)
print(p1<p3)
```

Output

```
True
False
False
```

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)`