Website: https://www.programiz.com/python-programming/operator-overloading

**Python Operator Overloading**

In this tutorial, we will learn about operator overloading in Python with the help of examples.

In Python, we can change the way [operators](https://www.programiz.com/python-programming/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.

Let's see an example,

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)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

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

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**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.