Module-3

PPT-4

Operator Overloading

- Operator Overloading means giving extended meaning beyond their predefined operational meaning.
- For example operator + is used to add two integers as well as join two strings and merge two lists.
- It is achievable because ‘+’ operator is overloaded by int class and str class.
- You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called Operator Overloading.

```
print(1 + 2)# concatenate two strings
print("Geeks"+"For") # Product two numbers
print(3 * 4)
print("Geeks"*4)# Repeat the String
```

To change the behavior of len(), you need to define the __len__() special method in your class. Whenever you pass an object of your class to len(), your custom definition of __len__() will be used to obtain the result.

```
class Order:
    def __init__(self, cart, customer):
        self.cart = list(cart)
        self.customer = customer

    def __len__(self):
        return len(self.cart)

order = Order(['banana', 'apple', 'mango'], 'Real Python')
print(len(order))
```

When you don’t have the __len__() method defined but still call len() on your object, you get a TypeError:

```
class Order:
    def __init__(self, cart, customer):
        self.cart = list(cart)
        self.customer = customer


order = Order(['banana', 'apple', 'mango'], 'Real Python')
len(order)  # Calling len when no __len__
```

But, when overloading len(), you should keep in mind that Python requires the function to return an integer. If your method were to return anything other than an integer, you would get a TypeError.

```
class Order:
    def __init__(self, cart, customer):
        self.cart = list(cart)
        self.customer = customer

    def __len__(self):
        return float(len(self.cart))  # Return type changed to float

order = Order(['banana', 'apple', 'mango'], 'Real Python')
len(order)
```

Any change is made directly to self and it is then returned. What happens when you return some random value, like a string or an integer?

```
class Order:
    def __init__(self, cart, customer):
        self.cart = list(cart)
        self.customer = customer
    def __iadd__(self, other):
        self.cart.append(other)
        return 'Hey, I am string!'
order = Order(['banana', 'apple'], 'Real Python')
order += 'mango'
order
```

Even though the relevant item was appended to the cart, the value of order changed to what was returned by __iadd__(). Python implicitly handled the assignment for you. This can lead to surprising behavior if you forget to return something in your implementation:

```
class Order:
    def __init__(self, cart, customer):
        self.cart = list(cart)
        self.customer = customer
    def __iadd__(self, other):
        self.cart.append(other)
order = Order(['banana', 'apple'], 'Real Python')
order += 'mango'
print(order)  # No output
```

operator overloading for add function

```
class Mock:
    def __init__(self, num):
        self.num = num
    def __add__(self, other):
        return Mock(self.num + other)

mock = Mock(5)
mock = mock + 6
print(mock.num)
```

Cont... operator overloading for add function

```
class Order:
    def __init__(self, cart, customer):
        self.cart = list(cart)
        self.customer = customer
    def __iadd__(self, other):
        self.cart.append(other)
        return self #using self
order = Order(['banana', 'apple'], 'Real Python')
order += 'mango'
print(order.cart)
```

Binary Operators

Operator	Method
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	*object.__floordiv__(self, other)*
/	object.__div__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

Assignment Operators:

Operator	Method
`+=`	`object.__iadd__(self, other)`
`-=`	`object.__isub__(self, other)`
`*=`	`object.__imul__(self, other)`
`/=`	`object.__idiv__(self, other)`
`//=`	*`object.__ifloordiv__(self, other)`*
`%=`	`object.__imod__(self, other)`
`**=`	`object.__ipow__(self, other[, modulo])`
`<<=`	`object.__ilshift__(self, other)`
`>>=`	`object.__irshift__(self, other)`
`&=`	`object.__iand__(self, other)`
`^=`	`object.__ixor__(self, other)`
`|=`	`object.__ior__(self, other)`

Unary Operators:

Operator	Method
-	object.__neg__(self)
+	object.__pos__(self)
abs()	object.__abs__(self)
~	object.__invert__(self)
complex()	object.__complex__(self)
int()	object.__int__(self)
long()	object.__long__(self)
float()	object.__float__(self)
oct()	object.__oct__(self)
hex()	object.__hex__(self)

Comparison Operators

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

```python
# Python Program illustrate how
# to overload an binary * operator

class A:
    def __init__(self, a):
        self.a = a

    # adding two objects
    def __mul__(self, o):
        return self.a * o.a
ob1 = A(2)
ob2 = A(3)
ob3 = A("SideBayes!")

print(ob1 * ob2)
print(ob2 * ob3)
```

```
6
SideBayes!SideBayes!SideBayes!
```