

CSE231 - Lab 12

Operator Overloading

Where we left off...

Last time, we talked about how to create a class that acts as a template for the creation of an *object* in Python. We discussed the constructor (`__init__()`), `self`, user-defined method functions, and some magic method functions (`__str__()` and `__repr__()`).

But it didn't really do much, right? Sure, we can pack a bunch of stuff inside our object, but can't lists just do that better?

This is where we talk about ***operator overloading***, using this concept gives you the ability to define *how* you want your object to interact with the Python language.

What does '+' *really* mean?

Let's first talk about the operators of Python, like '+' for instance. Symbolically, we think of this as *addition*, but it's just a symbol -- you can make it mean whatever you want to your class. We *tend* to think of it as addition, however, and so you'll commonly define '+' to "add" something with respect to your class.

```
print("Hello, " + "world!")      # 'Hello, world!'
```

```
print(1 + 2)      # 3
```

```
print( [1, 2, 3] + [4, 5, 6] )    # [1, 2, 3, 4, 5, 6]
```

```
print(1 + "hello")    # TypeError
```

Defining An Operator Overload

In order to tell Python the way we want our class to interact with other classes, we again have to use a magic method, and there is a magic method for *all of the operators*. If C1 and C2 are classes...

C1 + C2 # __add__()

C1 - C2 # __sub__()

str(C1) # __str__()

len(C1) # __len__()

Defining An Operator Overload

In order to tell Python the way we want our class to interact with other classes, we again have to use a magic method, and there is a magic method for *all of the operators*. If C1 and C2 are classes...

C1 + C2 # __add__() → __add__(self, other)

C1 - C2 # __sub__() → __sub__(self, other)

str(C1) # __str__() → __str__(self)

len(C1) # __len__() → __len__(self)

Defining An Operator Overload

In order to tell Python the way we want our class to interact with other classes, we again have to use a magic method, and there is a magic method for *all of the operators*. If C1 and C2 are classes...

C1 + C2 # __add__() → C1.__add__(C2)

C1 - C2 # __sub__() → C1.__sub__(C2)

str(C1) # __str__() → C1.__str__()

len(C1) # __len__() → C1.__len__()

Math-like Operators		
Expression	Method name	Description
$x + y$	<code>__add__()</code>	Addition
$x - y$	<code>__sub__()</code>	Subtraction
$x * y$	<code>__mul__()</code>	Multiplication
x / y	<code>__div__()</code>	Division <code>__truediv__()</code>
$x == y$	<code>__eq__()</code>	Equality
$x > y$	<code>__gt__()</code>	Greater than
$x \geq y$	<code>__ge__()</code>	Greater than or equal
$x < y$	<code>__lt__()</code>	Less than
$x \leq y$	<code>__le__()</code>	Less than or equal
$x \neq y$	<code>__ne__()</code>	Not equal
Sequence Operators		
<code>len(x)</code>	<code>__len__()</code>	Length of the sequence
<code>x in y</code>	<code>__contains__()</code>	Does the sequence y contain x?
<code>x[key]</code>	<code>__getitem__()</code>	Access element <i>key</i> of sequence x
<code>x[key]=y</code>	<code>__setitem__()</code>	Set element <i>key</i> of sequence x to value y
General Class Operations		
<code>x=myClass()</code>	<code>__init__()</code>	Constructor
<code>print(x), str(x)</code>	<code>__str__()</code>	Convert to a readable string
	<code>__repr__()</code>	Print a Representation of x
	<code>__del__()</code>	Finalizer, called when x is garbage collected

TABLE 12.1 Python Special Method Names

Example

```
class ComplexNumber(object):  
  
    def __init__(self, real, imaginary):  
  
        self.real_ = real  
  
        self.imaginary_ = imaginary  
  
    def __add__(self, other):    # Assuming `other` is another ComplexNumber() instance  
  
        real = self.real_ + other.real_  
  
        imaginary = self.imaginary_ + other.imaginary_  
  
        return ComplexNumber(real, imaginary)  
  
    def __str__(self):  
  
        return "{}+{}i".format(self.real_, self.imaginary_)
```


Example (cont.)

```
C1 = ComplexNumber(3, 6)
```

```
print(C1)      # '3+6i'
```

```
C2 = ComplexNumber(4, 5)
```

```
print(C2)      # '4+5i'
```

```
C3 = C2 + C1
```

```
print(C3)      # '7+11i'
```

```
class ComplexNumber(object):
```

```
    def __init__(self, real, imaginary):
```

```
        self.real_ = real
```

```
        self.imaginary_ = imaginary
```

```
    def __add__(self, other):    # Assuming `other` is an
```

```
        real = self.real_ + other.real_
```

```
        imaginary = self.imaginary_ + other.imaginary_
```

```
        return ComplexNumber(real, imaginary)
```

```
    def __str__(self):
```

```
        return "{}+{}i".format(self.real_, self.imaginary_)
```

Example (cont.)

```
C1 = ComplexNumber(3, 6)
```

```
print(C1)      # '3+6i'
```

```
C2 = ComplexNumber(4, 5)
```

```
print(C2)      # '4+5i'
```

```
C3 = C2 + C1
```

```
print(C3)      # '7+11i'
```

```
class ComplexNumber(object):
```

```
    def __init__(self, real, imaginary):
```

```
        self.real_ = real
```

```
        self.imaginary_ = imaginary
```

```
    def __add__(self, other):    # Assuming `other` is an
```

```
        real = self.real_ + other.real_
```

```
        imaginary = self.imaginary_ + other.imaginary_
```

```
        return ComplexNumber(real, imaginary)
```

```
    def __str__(self):
```

```
        return "{}+{}i".format(self.real_, self.imaginary_)
```

Example (cont.)

```
C1 = ComplexNumber(3, 6)
```

```
print(C1)      # '3+6i'
```

```
C2 = ComplexNumber(4, 5)
```

```
print(C2)      # '4+5i'
```

```
C3 = C2 + C1
```

```
print(C3)      # '7+11i'
```

```
class ComplexNumber(object):
```

```
    def __init__(self, real, imaginary):
```

```
        self.real_ = real
```

```
        self.imaginary_ = imaginary
```

```
    def __add__(self, other):    # Assuming `other` is an
```

```
        real = self.real_ + other.real_
```

```
        imaginary = self.imaginary_ + other.imaginary_
```

```
        return ComplexNumber(real, imaginary)
```

```
    def __str__(self):
```

```
        return "{}+{}i".format(self.real_, self.imaginary_)
```

Defining An Operator Overload (cont.)

In the example you just saw, what's secretly happening is a mapping akin to what your average, everyday user-defined method function would do:

$$C1 + C2 \rightarrow C1.__add__(C2)$$

In fact, all operators are really doing, are invoking functions that are defined within the classes they're being used on. The only reason we have operators is to make invoking these functions faster to write -- that's all it is 🙄

Of course, you would have to customly define behaviour for your class if you wanted the operator to work with other classes that *aren't* another instance of the *same* class.

Example

```
class ComplexNumber(object):  
  
    def __init__(self, real, imaginary):  
  
        self.real_ = real  
  
        self.imaginary_ = imaginary  
  
    def __add__(self, int_):    # Now we're assuming the second parameter to be an int  
  
        real = self.real_ + int_  
  
        return ComplexNumber(real, self.imaginary_)  
  
    def __str__(self):  
  
        return "{}+{}i".format(self.real_, self.imaginary_)
```

Example (cont.)

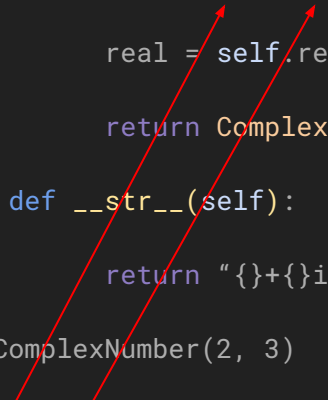
```
class ComplexNumber(object):  
  
    def __init__(self, real, imaginary):  
  
        self.real_ = real  
  
        self.imaginary_ = imaginary  
  
    def __add__(self, int_):    # Now we're assuming the second parameter to be an int  
  
        real = self.real_ + int_  
  
        return ComplexNumber(real, self.imaginary_)  
  
    def __str__(self):  
  
        return "{}+{}i".format(self.real_, self.imaginary_)
```

```
C1 = ComplexNumber(2, 3)
```

```
C2 = C1 + 2
```

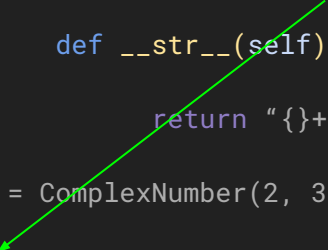
Example (cont.)

```
class ComplexNumber(object):  
  
    def __init__(self, real, imaginary):  
  
        self.real_ = real  
  
        self.imaginary_ = imaginary  
  
    def __add__(self, int_):    # Now we're assuming the second parameter to be an int  
  
        real = self.real_ + int_  
  
        return ComplexNumber(real, self.imaginary_)  
  
    def __str__(self):  
  
        return "{}+{}i".format(self.real_, self.imaginary_)  
  
C1 = ComplexNumber(2, 3)  
C2 = C1 + 2
```



Example (cont.)

```
class ComplexNumber(object):  
  
    def __init__(self, real, imaginary):  
  
        self.real_ = real  
  
        self.imaginary_ = imaginary  
  
    def __add__(self, int_):    # Now we're assuming the second parameter to be an int  
  
        real = self.real_ + int_  
  
        return ComplexNumber(real, self.imaginary_)  
  
    def __str__(self):  
  
        return "{}+{}i".format(self.real_, self.imaginary_)  
  
C1 = ComplexNumber(2, 3)  
C2 = C1 + 2    # '4+3i'
```



Class Mutability

Because we have full access to the data members, including the ability to edit them within the class, does that mean user-defined classes are mutable? 🤔

Yes.

In fact, you can treat the operators *as mutable operations* if you so choose. Think of what happens when you call `.sort()` on a `list`, for example. We don't reassign the list to be the return of a `.sort()` call because `.sort()` will edit the contents of the instance *for us*.

We can treat the operators in just the same way if we code it.

Example

```
class ComplexNumber(object):  
  
    def __init__(self, real, imaginary):  
  
        self.real_ = real  
  
        self.imaginary_ = imaginary  
  
    def __add__(self, int_):    # Still assuming parameter two is an int  
  
        self.real_ += int_  
  
    def __str__(self):  
  
        return "{}+{}i".format(self.real_, self.imaginary_)
```

Example (cont.)

```
class ComplexNumber(object):

    def __init__(self, real, imaginary):

        self.real_ = real

        self.imaginary_ = imaginary

    def __add__(self, int_):    # Still assuming parameter two is an int

        self.real_ += int_

    def __str__(self):

        return "{}+{}i".format(self.real_, self.imaginary_)

C1 = ComplexNumber(2, 3)

C1 + 2

print(C1)    # '4+3i'
```