Kenneth Love
writes on October 27, 2014

Operator Overloading in Python

Python is an interesting language. It's meant to be very explicit and easy to work with. But what happens when how you want or need to work with Python isn't just what the native types expose? Well, you can build classes and give those classes attributes or methods that let you use them, but then you end up having to call methods instead of being able to just add items together where it makes sense.

But what if you *could* just add them together? What if your `Area` class instances could just be added together to make a larger area? Well, thankfully, you can. This practice is called Operator Overloading because you're overloading, or overwriting, how operators work. By "operator", I

Book Club

Many of us here at Treehouse like to read and I think it would be neat to have a way to measure the books we've read. Let's ignore the fact that several services already exist to do this very thing. I want to do it locally and in Python. Obviously I should start with a `Book` class.

```
class Book:
    title = ''
    pages = 0

    def __init__(self, title='', pages=0):
        self.title = title
        self.pages = pages

    def __str__(self):
        return self.title
```

Nothing here that we didn't cover in *Object-Oriented Python*. We make an instance and set some attributes based on the passed-in values. We also gave our class a `__str__` method so it'll give us the title when we turn it into a string.

What I want to be able to do, though, is something

me an error about `"TypeError: unsupported operand type(s) for + : 'int' and 'Book'"`. That's because our `Book` class has no idea what to do with a plus sign. Let's fix that.

Reverse Adding

So the `sum()` function in Python is pretty swell. It takes a list of numbers and adds them all together. So if you have a bunch of, say, baseball inning scores, you can add them together in one method without having to have a counter variable or anything like that.

But, `sum()` does something you probably don't expect. It starts with `0` and then adds the first itme in the list to that. So if the first item doesn't know how to add itself to 0, Python fails. But before it fails, Python tries to do a reversed add with the operators.

Basically, remember how `2 + 5` and `5 + 2` are the same thing due to the commutative property of addition? Python takes advantage of that and swaps the operators. So instead of `0 + Book`, it tries `Book + 0`. `0 + Book` won't work because the

OK, let's try it.

```
>>> from books import Book
>>> book1 = Book('Fluency', 381)
>>> book2 = Book('The Martian', 385)
>>> book3 = Book('Ready Player One', 386)
>>> sum([book1, book2, book3])
1152
```

Excellent!

But what if we want to add two `Book` instances together directly? If we do:

```
>>> book1 + book2
```

from our above example, we get another `TypeError` about `+` being unsupported for the `Book` type. Well, yeah, we told Python how to add them in reverse, but no matter how Python tries to put these together, one of them has to be in front, so `__radd__` isn't being used.

Adding

Time for regular adding, then. As you might have guessed, we override the `__add__` method.

pretty well down up. But what if we want to compare books to each other? Let's override a few more methods so we can use `<`, `>`, and friends.

Comparative Literature

There's a handful of methods we have to override to implement the comparison operators in our class. Let's just do them all at once.

```
def __lt__(self, other):
    return self.pages < other

def ___le__(self, other):
    return self.pages <= other

def __eq__(self, other):
    return self.pages == other

def __ne__(self, other):
    return self.pages != other

def __gt__(self, other):
    return self.pages > other

def __ge__(self, other):
    return self.pages >= other
```

This works fine for `<`, `>`, `==`, and `!=`, but blows up on `<=` and `>=` because we haven't said what to


```
            return NotImplemented

    def __ge__(self, other):
        if isinstance(other, Book):
            return self.pages >= other.pages
        elif isinstance(other, (int, float)):
            return self.pages >= other
        else:
            return NotImplemented
```

Yes, this is more work but it makes our code smarter. If we're comparing two `Book` instances, we'll use their `pages` attributes. If not, but we're comparing against a number, we'll compare that like normal. Then, finally, we'll return a `NotImplemented` error for anything else. Let's try it out.

```
>>> book1 <= book3
True
>>> book3 > book2
True
>>> book3 > 500
False
```

Great! Now we can add books together to get total page counts and we can compare books to each other.

overriding `__iadd__` and `__isub__`. To see the entire list of magic methods that can be overridden, check the Python documentation. I've also posted the code from this article. See you next time!

Check out my Python courses at Treehouse.

Photo from Loughborough University Library.

code object-oriented oo

oop operator Python

7 Responses to "Operator Overloading in Python"

you need a proofreader.

Reply

Nimesh K Verma on **February 18, 2016 at 11:50 pm** said:

Under the section “Adding”, the definition of__add__ should be

def __add__(self, other):
return self.pages + other.pages

Reply

> **fred** on **August 3, 2016 at 7:59 pm** said:
> yeah i noticed that too and was confused for a while and figured it must’ve been a typo. your comment confirmed it to me. these tutorials need to be very carefully proofread because any mistake can make it very confusing for a novice.
>
> **Reply**

John Wu on **October 30, 2015 at 10:26 pm** said:

Hey!

Great tutorial! Just wondering, why is it different for `=`? Why you need to do type inspection for them but not for others (“ etc)?

Kenneth Love on August 17, 2015 at 9:40 am said:

Hey Thomas,

You don't *have* to overload both but most of the time you'll want to. `__radd__` lets Python know how to add things when the second type doesn't match its expectations and this happens more often than you think. The good thing is, most of the time, `__add__` and `__radd__` are very similar, often just inverses of each other!

Reply

Leave a Reply

Name *

Email Address (will not be published) *

Website

Submit Comment

Email Address

Subscribe