Operator Overloading and Dunder Methods

Intermediate Application Development

Otago Polytechnic
Dunedin, New Zealand
Kaiako: Tom Clark

Let's add two integers

Suppose we want to add two integers, say 2 and 3. Addition of integers is a function that takes two arguments and returns their sum. We might expect to write it this way:

```
add(2,3)
```

or, if we're feeling particularly object oriented,

```
2.add(3)
```

Let's add two integers

But we don't do that. We generally we write it this way:

```
2 + 3
```

because somebody did it like that in 1360 and people liked it. The "+" symbol in this expression is an *operator*.

Wow, that's handy

The "+" operator notation is so handy that we use it in other places

```
> [3, 1, 4] + [22, 11]
[3, 1, 4, 22, 11]
```

It's not just "+" that gets this treatment.

```
> 3 * 'cat'
'catcatcat'
```

What if we wanted to do this for our own classes?

Dunder Methods

Every object in Python inherits some "dunder" methods. They're also commonly called "magic methods". We've already seen two of these, `__init__()` and `__str__()`. There are also other such methods that we can choose to implement.

It's bad practice to define new dunder methods, and it won't generally do what we want anyway. Also, it's possible to abuse dunder methods by implementing them in unexpected ways. Don't do that.

Operator overloading

Suppose we define a class and would like to be able to add two objects of that class using "+". This would be an example of *opertator overloading*.

```
class Bill:
    def __init__(self, items):
        self.line_items = items # a list of item objects

    def __add__(self, other):
        new_list = self.line_items + other.line_items
        return Bill(new_list)
.
.
.
b1 = Bill(some_items)
b2 = Bill(some_other_items)
b3 = b1 + b2
```

Programming Activity

1. Pull the course materials repo.
2. Create a new branch, `04-practical` in your practicals repo.
3. Add a subdirectory, `04-practical` and copy `04-practical.ipynb` from the class materials into it.
4. Open a shell, cd to this directory, and run `jupyter notebook` to open the notebook. Complete the first questions.
5. We will discuss results in 30ish minutes.

Other Dunder Methods

There are other handy dunder methods to implement that aren't strictly used for operator overloading

__STR__ AND __REPR__

We've already seen that we can use the `__str__()` method to control what happens when our object is passed as an argument to `print()`. The general idea is that `__str__()` should return a user-friendly string.

`__repr__()` is similar, but it should return a programmer-friendly string. A good idea is to return a string that matches a call to the constructor method that returns the given object.

```
class Cat:
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return f'Cat("{self.name}")'

c  = Cat("Larry")
repr(c)
```

__LEN__

We have seen that we can call `len()` on a list or string to get its length. If we want to make this work for our own classes, we implement `__len__()`.

```
class Bill:
    def __init__(self, items):
        self.line_items = items # a list of item objects

    def __len__(self):
        return len(self.line_items}
```

Conclusions

Operator overloading is somewhat controversial. Not all languages support it and some programmers think it's a bad idea. I find it to be useful.

Implementing appropriate dunder methods is a very "Pythonic" way to do things. Well implemented dunder methods can make your classes easy and productive to use.

Some refs:
`https://dbader.org/blog/python-dunder-methods`
`https://docs.python.org/3/reference/datamodel.html#special-method-names`