Plugging Your Classes into Python Syntax

You know how to use several of Python's operators. For example, we have used the + operator for:

- adding 2 numbers together,
- concatenating two string, and
- joining two lists.

You've also used *, /, -, **, and other operators.

We're going to show you how to use operators like + with your own types.

Special Methods

All classes are built on an existing class called `object`. Class `object` has many existing "underscore" methods, known as *special* or *magic* methods. We can see their names by calling `dir` on `object`:

```
>>> dir(object)
['__class__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__']
```

These special methods "plug in" to Python syntax, often with operators such as + and ==. Here are a few examples:

- Method `__init__` is called after an object is created, in order to initialize that object.
- Method `__str__` is called to get a string representation of an object (such as by `str()` or `print()`).
- Method `__eq__` is called whenever the operator == is used.

Adding to the CashRegister Class

We're going to add the `__eq__` method to our `CashRegister` class so that we can compare two cash register objects using ==.

We will consider two cash registers to be equal if they contain the same amount of money. Here is the main program that we'd like to run:

```
if __name__ == '__main__':

    cr1 = CashRegister(2, 0, 0, 0, 0)
    cr2 = CashRegister(0, 1, 0, 0, 0)
    cr3 = CashRegister(1, 1, 0, 0, 0)
    print(cr1 == cr2)
    print(cr3 == cr2)
```

When we run the program, both the comparisons for equality produce `False`:

```
>>>
```

```
False
False
```

Since we have not supplied an `__eq__` method in class `CashRegister`, `object`'s `__eq__` method is used. Class `object`'s `__eq__` method works by comparing memory addresses. Since `cr1, cr2`, and `cr3` contain different memory locations, they are not equal to one another.

However, since we want our `CashRegister` objects to be considered equal if the amount of money is the same, we need to write our own method.

Writing `__eq__`

The first parameter to our `__eq__` method is `self`, and the second parameter refers to the `CashRegister` object that `self` will be compared with.

When calling the `__eq__` method, the object on the left-hand side of the equality operator corresponds with `self`, and the operator on the right-hand side correponds with `other` (for example, for expression `cash1 == cash2`, `self` will refer to `cash1`, and `other` will refer to `cash2`)

For this method, we need to check whether the value produced by `self.get_total()` is equal to the value produced by `other.get_total()`:

```
def __eq__(self, other):
    """ (CashRegister, CashRegister) -> bool

    Return True iff this CashRegister has the same amount of money as other.

    >>> reg1 = CashRegister(2, 0, 0, 0, 0)
    >>> reg2 = CashRegister(0, 1, 0, 0, 0)
    >>> reg1 == reg2
    True
    """

    return self.get_total() == other.get_total()
```

Now, when the program is executed, we get the following results:

```
>>>
True
False
```

In General

Whenever defining special methods that correpond with Python's operators, it's up to you to decide what these operators will do.

In this case we decided that two `CashRegister` objects were equal when the amount of money stored in them was equal. We could have also chosen to consider them equal only when they had the exact same number of loonies, toonies, fives, tens and twenties, or chosen to go with the default - making them equal only when they referred to the same object in memory.

The choice is up to you.

Jennifer Campbell • Paul Gries
University of Toronto
