Writing Special Method `__str__`

In this lecture, you will learn about the special method `__str__`. As you saw in the previous lecture, special methods start and end with a double underscore __. So far, our `class CashRegister` has two special methods, `__init__` and `__eq__`. It also has three regular methods, `get_total`, `add`, and `remove`.

Method `__str__`

Let us create a few `CashRegister` objects:

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

At this point, when the print function is called on `CashRegister` objects, the memory addresses of the objects are printed:

```
>>> print(cr1)
<__main__>.CashRegiter object at 0x101d7a550
>>> print(cr2)
<__main__>.CashRegiter object at 0x101d7ac90
```

The function `print` calls the special method `__str__` in order to get a string to print. Let us now implement the `__str__` method in our `CashRegister` class in order to get nicer output from the `print` function call:

```
def __str__(self):
    """ (CashRegister) -> str

    Return a string representation of this CashRegister.

    >>> reg1 = CashRegister(1, 2, 3, 4, 5)
    >>> reg1.__str__()
    CashRegister: $160 ($1x1, $2x2, $5x3, $10x4, $20x5)
    """

    return 'CashRegister: $' + self.get_total() + ' ($1x' + self.loonies + \
           ', $2x' + self.toonies + ', $5x' + self.fives + ', $10x' + \
           self.tens + ', $20x' + self.twenties + ')'
```

According to the type contract, the `__str__` method takes a `CashRegister` object and returns an `str` object. If we run the `CashRegister` module with the above `__str__` method, we will get an error that says `TypeError: Can't Convert 'int' object to str implicitly`. The problem is that we are applying the `+` operator to an `int` and a `str`. That can be fixed by calling function `str` on the `int`s to get string representations of them. Here is the updated code:

```
def __str__(self):
    """ (CashRegister) -> str

    Return a string representation of this CashRegister.

    >>> reg1 = CashRegister(1, 2, 3, 4, 5)
    >>> reg1.__str__()
    CashRegister: $160 ($1x1, $2x2, $5x3, $10x4, $20x5)
    """

    return 'CashRegister: $' + str(self.get_total()) + ' ($1x' + str(self.loonies) + \
```

```
            ', $2x' + str(self.toonies) + ', $5x' + str(self.fives) + ', $10x' + \
            str(self.tens) + ', $20x' + str(self.twenties) + ')'
```

If we run the `CashRegister` module now, we see that it runs without any problems. Now, if we print a `CashRegister` object, here is what we get:

```
>>> cr1 = CashRegister(2, 0, 0, 0, 0)
>>> cr2 = CashRegister(0, 1, 0, 0, 0)
>>> print(cr1)
CashRegister: $2 ($1x2, $2x0, $5x0, $10x0, $20x0)
>>> print(cr2)
CashRegister: $2 ($1x0, $2x1, $5x0, $10x0, $20x0
```

However, notice that our current `__str__` method looks long and messy. We can improve this by using the `str.format` method call.

Method `str.format`

The `str.format` uses *placeholders*, which are locations in the string that we want to replace with actual values. These placeholders are curly braces with an integer between them. The placeholders correspond with the arguments passed to `str.format` between the parentheses (as opposed to the argument to the left of the dot). Placeholder `{0}` corresponds with the first argument, and `{1}` corresponds with the second argument, and so on. Here is the updated `__str__` method that uses `str.format`:

```
def __str__(self):
    """ (CashRegister) -> str

    Return a string representation of this CashRegister.

    >>> reg1 = CashRegister(1, 2, 3, 4, 5)
    >>> reg1.__str__()
    CashRegister: $160 ($1x1, $2x2, $5x3, $10x4, $20x5)
    """

    return 'CashRegister: ' + \
           '${0} ($1x{1}, $2x{2}, $5x{3}, $10x{4}, $20x{5})'.format(
               self.get_total(), self.loonies, self.toonies,
               self.fives, self.tens, self.twenties)
```

In the above function, `self.get_total()` corresponds with `{0}`, `self.loonies` corresponds with `{1}`, and so on.

Jennifer Campbell • Paul Gries
University of Toronto
