Lesson 33. Customizing classes

After reading lesson 33, you'll be able to

- Add special Python methods to your classes
- Use special operators such as +, –, /, and * on your classes

You've been working with classes defined in the Python language since you wrote your first Python program. The most basic type of objects in the Python language, called *built-in types*, allowed you to use special operators on these types. For example, you used the + operator between two numbers. You were able to use the `[]` operator to index into a string or a list. You were able to use the `print()` statement on any of these types of objects, as well as on lists and dictionaries.

Consider this

- Name five operations you can do between integers.
- Name one operation you can do between two strings.
- Name one operation you can do between a string and an integer.

Answer:

- `+, –, *, /, %`
- `+`
- `*`

Each of these operations is represented in shorthand using a symbol. However, the symbol is only a shorthand notation. Each operation is actually a method that you can define to work with an object of a specific type.

33.1. OVERRIDING A SPECIAL METHOD

Every operation on an object is implemented in a class as a method. But you may have noticed that you use several shorthand notations when you work with simple object types such as `int`, `float`, and `str`. These shorthand notations are things like using the + or – or * or / operator between two such objects. Even something like `print()` with an object in the parentheses is shorthand notation for a method in a class. You can implement such methods in your own classes so that you can use shorthand notation on your own object types.

Table 33.1 lists a few special methods, but there are many more. Notice that all these special methods begin and end with double underscores. This is specific to the Python language, and other languages may have other conventions.

Table 33.1. A few special methods in Python

Category	Operator	Method name
Mathematical operations	+ - * /	__add__ __sub__ __mul__ __truediv__
Comparisons	== < >	__eq__ __lt__ __gt__
Others	print() **and** str() Create an object —for example, some_object = ClassName()	__str__ __init__

To add the capability for a special operation to work with your class, you can *override* these special methods. Overriding means that you'll implement the method in your own class and decide what the method will do, instead of the default behavior implemented by the generic Python object.

Begin by creating a new type of object, representing a fraction. A fraction has a numerator and a denominator. Therefore, the data attributes of a `Fraction` object are two integers. The following listing shows a basic definition of how the `Fraction` class might look.

Listing 33.1. Definition for the `Fraction` class

```
class Fraction(object):
    def __init__(self, top, bottom):          1
        self.top = top                        2
        self.bottom = bottom                  2
```

- ***1* The initialization method takes in two parameters.**
- ***2* Initializes data attributes with the parameters**

With this definition, you can now create two `Fraction` objects and try to add them together:

```
half = Fraction(1,2)
quarter = Fraction(1,4)
print(half + quarter)
```

Adding 1/2 and 1/4 should give 3/4. But when you run the snippet, you get this error:

```
TypeError: unsupported operand type(s) for +: 'Fracti
```

This tells you that Python doesn't know how to add two `Fraction` objects together. This error makes sense, because you never defined this operation for a `Fraction` object type.

To tell Python how to use the + operator, you need to implement the special method `__add__` (with double underscores before and after the name `add`). Addition works on two objects: one is the object you're calling the method on, and the other is a parameter to the method. Inside the method, you perform the addition of two `Fraction` objects by referencing the numerators and denominators of both objects, as in the following listing.

Listing 33.2. Methods to add and multiply two `Fraction` objects

```
class Fraction(object):
    def __init__(self, top, bottom):
        self.top = top
        self.bottom = bottom
    def __add__(self, other_fraction):
        new_top = self.top*other_fraction.bottom  +
                  self.bottom*other_fraction.top
        new_bottom = self.bottom*other_fraction.botto
        return Fraction(new_top, new_bottom)
    def __mul__(self, other_fraction):
        new_top = self.top*other_fraction.top
        new_bottom = self.bottom*other_fraction.botto
        return Fraction(new_top, new_bottom)
```

- ***1* Defines special method to implement the + operator between two Fraction objects**
- ***2* Breaks up the line into two lines by using a backslash**
- ***3* Calculates the numerator from the addition**
- ***4* Calculates the denominator from the addition**
- ***5* Returns a new Fraction object, created using the new numerator and denominator**
- ***6* Method to multiply two Fraction objects**

Quick check 33.1

Q1:

Write a method for the `Fraction` object to use the – operator between two `Fraction` objects.

33.2. OVERRIDING PRINT() TO WORK WITH YOUR CLASS

Now that you defined the + operator between `Fraction` objects, you can try the same code as before:

```
half = Fraction(1,2)
quarter = Fraction(1,4)
```

This code doesn't give an error anymore. Instead, it prints the type of the object and its memory location:

```
<__main__.Fraction object at 0x00000200B8BDC240>
```

But this isn't informative at all. You'd rather see the value of the fraction! You need to implement another special function, one that tells Python how to print an object of your type. To do this, you implement the special method `__str__`, as in the next listing.

Listing 33.3. Method to print a `Fraction` object

```
class Fraction(object):
    def __init__(self, top, bottom):
        self.top = top
        self.bottom = bottom
    def __add__(self, other_fraction):
        new_top = self.top*other_fraction.bottom  +
                  self.bottom*other_fraction.top
        new_bottom = self.bottom*other_fraction.botto
        return Fraction(new_top, new_bottom)
    def __mul__(self, other_fraction):
        new_top = self.top*other_fraction.top
        new_bottom = self.bottom*other_fraction.botto
        return Fraction(new_top, new_bottom)
    def __str__(self):
        return str(self.top)+"/"+str(self.bottom)
```

- ***1* Defines method to print a Fraction object**
- ***2* Returns a string, what to print**

Now, when you use `print` on a `Fraction` object, or when you use `str()` to convert your object to a string, it'll call the method `__str__`. For example, the following code prints `1/2` instead of the memory location:

```
half = Fraction(1, 2)
print(half)
```

The following creates a string object with the value 1/2:

```
half = Fraction(1, 2)
half_string = str(half)
```

Quick check 33.2

Q1:

Change the `__str__` method for a `Fraction` object to print the numerator on one line, two dashes on the next, and the denominator on a third line. The line `print(Fraction(1,2))` prints this:

```
1
--
2
```

33.3. BEHIND THE SCENES

What exactly happens when you use a special operator? Let's look at the details, and what happens when you add two `Fraction` objects:

```
half = Fraction(1,2)
quarter = Fraction(1,4)
```

Consider this line:

```
half + quarter
```

It takes the first operand, `half`, and applies the special method `__add__` to it. That's equivalent to the following:

```
half.__add__(quarter)
```

Additionally, every method call can be rewritten by using the class name and explicitly giving the method a parameter for the `self` parameter. The preceding line is equivalent to this:

```
Fraction.__add__(half, quarter)
```

Despite being called a *special method*, all the methods that start and end with double underscores are regular methods. They're called on an object, take parameters, and return a value. What makes them special is that there's another way to call the methods. You can either call them using a special operator (for example, a mathematical symbol) or using a fairly well-known function (for example, `len()` or `str()` or `print()`, among others). This shorthand notation is often more intuitive for others if they're reading code than if they were to read the formal function call notation.

Thinking like a programmer

One nice goal that you should have as a programmer is to make life easier for other programmers that may use classes you define. This involves documenting your classes, methods, and whenever possible, implementing special methods that allow others to use your class in an intuitive way.

Quick check 33.3

Rewrite each of the following lines in two ways: by calling the method on an object and by calling the method by using the class name. Assume you start with this:

```
half = Fraction(1,2)
quarter = Fraction(1,4)
```

1

```
quarter * half
```

2

```
print(quarter)
```

3

```
print(half * half)
```

33.4. WHAT CAN YOU DO WITH CLASSES?

You've seen the details and the syntax behind creating your own object types using Python classes. This section will show you examples of classes that you may want to create in certain situations.

33.4.1. Scheduling events

Say that you're asked to schedule a series of events. For example, you're going to a movie festival and you want to arrange the movies in your schedule.

Without using classes

If you didn't use classes, you could use one list to hold all the movies you want to see. Each element in the list is a movie to see. The relevant information regarding a movie includes its name, its start time, end time, and perhaps a critic's rating. This information could be stored in a tuple as the element of the list. Notice that almost right away the list becomes cumbersome to use. If you wanted to access the ratings of every movie, you'd rely on indexing twice—first into the list of movies and then into the tuple to retrieve the rating.

Using classes

Knowing what you know about classes, it's tempting to make every object into a class. In the scheduling problem, you could make the following classes:

- `Time` class representing a time object. An object of this type would have data attributes: hours (`int`), minutes (`int`), and seconds (`int`). Operations on this object could be to find the difference between two times, or to convert to total number of hours, minutes, or seconds.
- `Movie` class representing a movie object. An object of this type would have data attributes: name (`string`), start time (`Time`), end time (`Time`), and rating (`int`). Operations on this class would be to check whether two movies overlap in time, or whether two movies have a high rating.

With these two classes, you can abstract away some of the annoying details

list of `Movie` objects. If you need to index into the list (to access a rating, for example), you can use nicely named methods defined in the movie class.

Using too many classes

It's important to understand how many classes are too many. For example, you could create a class to represent an `Hour`. But this abstraction doesn't add any value because its representation would be an integer, in which case you can use the integer itself.

SUMMARY

In this lesson, my objective was to teach you how to define special methods that allow you to use multiple objects and use operators on your object types. Here are the major takeaways:

- Special methods have a certain name and use double underscores before and after the name. Other languages may take different approaches.
- Special methods have a shorthand notation.

Let's see if you got this...

Q33.1

Write a method to allow you to use the `print` statement on a `Circle` and a `Stack`. Your `Stack`'s `print` should print each object in the same way that `prettyprint` does in lesson 32. Your `Circle print` should print the string `"circle: 1"` (or whatever the radius of the circle is). You'll have to implement the `__str__` method in the `Stack` class and the `Circle` class. For example, the following lines

```
circles = Stack()
one_circle = Circle()
one_circle.change_radius(1)
circles.add_one(one_circle)
two_circle = Circle()
two_circle.change_radius(2)
circles.add_one(two_circle)
print(circles)
```

should print this:

```
|_ circle: 2 _|
|_ circle: 1 _|
```