Exercise 40: Modules, Classes, and Objects

Python is something called an "object-oriented programming language." What this means is there's a construct in Python called a `class` that lets you structure your software in a particular way. Using classes, you can add consistency to your programs so that they can be used in a cleaner way, or at least that's the theory.

I am now going to try to teach you the beginnings of object-oriented programming, classes, and objects using what you already know about dictionaries and modules. My problem though is that object-oriented programming (aka OOP) is just plain weird. You have to simply struggle with this, try to understand what I say here, type in the code, and then in the next exercise I'll hammer it in.

Here we go.

Modules Are Like Dictionaries

You know how a dictionary is created and used and that it is a way to map one thing to another. That means if you have a dictionary with a key 'apple' and you want to get it then you do this:

```
mystuff = {'apple': "I AM APPLES!"}
print mystuff['apple']
```

Keep this idea of "get X from Y" in your head, and now think about modules. You've made a few so far, and used them, and you know they are:

1. A Python file with some functions or variables in it.
2. You then import that file.
3. And then you can access the functions or variables in that module with the '.' (dot) operator.

Imagine if I have a module that I decide to name `mystuff.py` and I put a function in it called `apple`. Here's the module `mystuff.py`:

```
# this goes in mystuff.py
def apple():
    print "I AM APPLES!"
```

Once I have that, I can use that module with `import` and then access the `apple` function:

```
import mystuff

mystuff.apple()
```

I could also put a variable in it named `tangerine` like this:

```
def apple():
    print "I AM APPLES!"

# this is just a variable
tangerine = "Living reflection of a dream"
```

Then again I can access that the same way:

```
import mystuff

mystuff.apple()
print mystuff.tangerine
```

Refer back to the dictionary, and you should start to see how this is similar to using a dictionary, but the syntax is different. Let's compare:

```
mystuff['apple'] # get apple from dict
mystuff.apple() # get apple from the module
mystuff.tangerine # same thing, it's just a variable
```

This means we have a *very* common pattern in Python of this:

1. Take a key=value style container.
2. Get something out of it by the key's name.

In the case of the dictionary, the key is a string and the syntax is `[key]`. In the case of the module, the key is an identifier, and the syntax is `.key`. Other than that they are nearly the same thing.

Classes are Like Modules

A way to think about a module is that it is a specialized dictionary that can store Python code so you can get to it with the `'.'` operator. Python also has another construct that serves a similar purpose called a `class`. A `class` is a way to take a grouping of functions and data and place them inside a container so you can access them with the `'.'` (dot) operator.

If I were to create a class just like the `mystuff` module, I'd do something like this:

```
class MyStuff(object):

    def __init__(self):
        self.tangerine = "And now a thousand years
between"

    def apple(self):
        print "I AM CLASSY APPLES!"
```

That looks complicated compared to modules, and there is definitely a lot going on by comparison, but you should be able to make out how this is like a "mini-module" with `MyStuff` having an `apple()` function in it. What is probably confusing with this is the `__init__()` function and use of `self.tangerine` for setting the `tangerine` variable.

Here's why classes are used instead of modules: You can take the above class and use it to craft many of them, millions at a time if you want, and they won't interfere with each other. With modules, when you import there is only one for the entire program unless you do some monster hacks.

Before you can understand this though, you need to know what an "object" is and how to work with `MyStuff` just like you do with the `mystuff.py` module.

Objects are Like Mini-Imports

If a `class` is like a "mini-module," then there has to be a similar concept to `import` but for classes. That concept is called "instantiate" which is just a fancy, obnoxious, overly smart way to say "create." When you instantiate a class what you get is called an `object`.

The way you do this is you call the `class` like it's a function, like this:

```
thing = MyStuff()
thing.apple()
print thing.tangerine
```

The first line is the "instantiate" operation, and it's a lot like calling a function. However, when you call this there's a sequence of events that Python coordinates for you. I'll go through them using the above code for `MyStuff`:

1. Python looks for `MyStuff()` and sees that it is a `class` you've defined.
2. Python crafts an empty object with all the functions you've specified in the class using `def`.
3. Python then looks to see if you made a "magic" `__init__` function, and if you have it calls that function to initialize your newly created empty object.
4. In the `MyStuff` function `__init__` I then get this extra variable `self`, which is that empty object Python made for me, and I can set variables on it just like you would with a module, dict, or other object.
5. In this case, I set `self.tangerine` to a song lyric and then I've initialized this object.
6. Now Python can take this newly minted object and assign it to the `thing` variable for me to work with.

That's the basics of how Python does this "mini-import" when you call a `class` like a function. Remember that this is *not* giving you the class, but instead it is using the class as a *blueprint* for how to build a copy of that type of thing.

Keep in mind that I'm giving you a slightly inaccurate idea for how these work so that you can start to build up an understanding of classes based on what you know of modules. The truth is, classes and objects suddenly diverge from modules at this point. If I were being totally honest, I'd say something more like this:

- Classes are like blueprints or definitions for creating new mini-modules.
- Instantiation is how you make one of these mini-modules *and* import it at the same time.
- The resulting created mini-module is called an `object` and you then assign it to a variable to work with it.

After this though classes and objects become very different from modules and this should only serve as a way for you to bridge over to understanding classes.

Getting Things from Things

I now have three ways to get things from things:

```
# dict style
mystuff['apples']

# module style
mystuff.apples()
print mystuff.tangerine

# class style
thing = MyStuff()
thing.apples()
print thing.tangerine
```

A First Class Example

You should start seeing the similarities in these three key=value container types and probably have a bunch of questions. Hang on with the questions, as the next exercise is going to hammer home your "object-oriented vocabulary." In this exercise, I just want you to type in this code and get it working so that you have some experience before moving on.

```
1  class Song(object):
2
3      def __init__(self, lyrics):
4          self.lyrics = lyrics
5
6      def sing_me_a_song(self):
7          for line in self.lyrics:
8              print line
9
10 happy_bday = Song(["Happy birthday to you",
11                   "I don't want to get sued",
12                   "So I'll stop right there"])
13
14 bulls_on_parade = Song(["They rally around the family",
15                         "With pockets full of shells"])
16
17 happy_bday.sing_me_a_song()
18
19 bulls_on_parade.sing_me_a_song()
```

What You Should See

```
$ python ex40.py
Happy birthday to you
I don't want to get sued
So I'll stop right there
They rally around the family
With pockets full of shells
```

Study Drills

1. Write some more songs using this and make sure you understand that you're passing a list of strings as the lyrics.
2. Put the lyrics in a separate variable, then pass that variable to the class to use instead.
3. See if you can hack on this and make it do more things. Don't worry if you have no idea how, just give it a try, see what happens. Break it, trash it, thrash it, you can't hurt it.
4. Search online for "object-oriented programming" and try to overflow your brain with what you read. Don't worry if it makes absolutely no sense to you. Half of that stuff makes no sense to me too.

Common Student Questions

Why do I need `self` when I make `__init__` or other functions for classes?

If you don't have `self`, then code like `cheese = 'Frank'` is ambiguous. That code isn't clear about whether you mean the *instance*'s `cheese` attribute, *or* a local variable named `cheese`. With `self.cheese = 'Frank'` it's very clear you mean the instance attribute `self.cheese`.