More about Classes and Objects

Everything is an object

In Python, *everything* is an object. That means that it has a class type, and it has methods, even if you don't see them directly. In the Python REPL, you can get the type and also information about the methods of an object by using the function `dir()`.

```
>>> d = None
>>> type(d)
<class 'NoneType'>
>>> dir(d)
['__bool__', '__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__']
>>>
```

```
>>> def plus_one(num):
...     """Add one to num and return it."""
...     return num + 1
...
```

```
>>> type(plus_one)
<class 'function'>
>>> dir(plus_one)
['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__defaults__',
'__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get__',
'__getattribute__', '__globals__', '__gt__', '__hash__', '__init__', '__kwdefaults__', '__le__',
'__lt__', '__module__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>>
```

You can see that the function `plus_one` is an object, just like any variable. We often say that functions are "first-class objects", meaning they can be treated as objects just like variables and class instances. In week 2, we saw the `map()` and `filter()` built-in functions. These functions take other functions as arguments. It would not be possible if functions were not first-class objects.

This is a very important concept in Python, and enables a lot of cool features, like decorators.

Also note that the function has a docstring; the string on the line immediately following the function definition. This string is now part of the object's definition:

```
>>> plus_one.__doc__
'Add one to num and return it.'
```


Another note about writing classes

Never apply an attribute to the class unless you are *absolutely* sure that you need it! Attributes are almost always instance attributes; attach them to the `self` object inside the methods.

Class methods

Sometimes you want to be able to have a method that operates on a class instead of an instance. There is a decorator, `@classmethod`, that is used for this purpose. It tells Python that the method it is decorating is meant for the class, and not for instances. One common use case is making an alternate constructor for a class.

Here we have the bare bones of a Circle object:

```
class Circle:
    """Class to create Circle objects"""
    def __init__(self, radius=1):
        self.radius = radius
```

Now suppose we want to be able to create a circle using the area instead of the radius? We can create an alternate Circle constructor as a class method:

```
    @classmethod
    def from_area(cls, area):
        return cls(math.sqrt(area / math.pi))
```

Let's look at what's happening here. The `@classmethod` decorator tells Python that this method will act on the class and not a class instance. The first parameter is implicit, just as the `self` parameter on instance methods is implicit, but in this case, it is the class and not the instance. Although it is not required, there is a strong convention for using `cls` as the name of this parameter, just like we always use `self` for the instance variable in regular methods. The second parameter is the `area` value. Then we convert the area value to a radius value, and using the implicit class parameter `cls`, create a circle using the calculated radius.

We can now do this with our new Circle class:

```
>>> from shapes import Circle
>>> c = Circle(7.5)
>>> c.area
176.71458676442586
>>> circle = Circle.from_area(176.71458676442586)
>>> circle.radius
7.5
```

Dictionary objects have an alternate constructor `fromkeys()` that is implemented using the `@classmethod` decorator.

Static methods

A static method on a class is indicated with the `@staticmethod` decorator. A static method on a class is one that is related to the class, but does not act on the class or class instances. An example would be for a class that has multiple inputs, perhaps we want to have a way to validate that a set of inputs fits within some restricted set of values, before using them to try to create a class object. We could create a method `is_valid()`, which when given a set of input values, will return True or False. This method would be given a `@staticmethod` decorator. Then the static method could be used before attempting to create an instance.

Here is a simple example, with a snippet of a class called `Construct`, that takes 3 arguments. There is some relationship between the values of the arguments, and we want the user of the Construct class to be able to determine whether a set of variables is valid for the Construct class:

```
class Construct:
    """Class for the Construct object."""

    def __init__(self, part_a, part_b, part_c):
        """Initialize Construct after validation of input."""
        if not Construct.is_valid_construct(part_a, part_b, part_c):
            raise ValueError('Invalid inputs for Construct!')
        self.part_a = part_a
        self.part_b = part_b
        self.part_c = part_c

    @staticmethod
    def is_valid_construct(a, b, c):
        """Return True if Construct variables are valid."""
        if a > b:
            return False
        if a * 2 < c:
            return False
        return True
```

With this static method, we can check to see if our values will make a valid Construct object before we try to make it:

```
>>> from example import Construct
>>> Construct.is_valid_construct(2, 5, 3)
True
>>> Construct.is_valid_construct(1, 2, 3)
False
>>> construct = Construct(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/diane/DevProjects/ucsd/code_files/example.py", line 7, in __init__
    def __init__(self, part_a, part_b, part_c):
ValueError: Invalid inputs for Construct!
>>> construct = Construct(2, 5, 3)
>>> construct.part_a
2
```

It wouldn't make sense to have this validate function independent of the Construct class, because it is only related to that class and doesn't make sense in other contexts.