Advanced Python Objects, map()

Up to this point, we haven't spoken much about object-oriented Python.
While functions play a big role in the Python ecosystem, Python does have classes which can have attached methods, and be instantiated as objects.

First, you can define a class using a class keyword, and ending with a colon.

Classes in Python are generally named using camel case, which means the first character of each word is capitalized.

You don't declare variables within the object, you just start using them.
Class variables can also be declared. These are just variables which are shared across all instances.
So in this example, we're saying that the default for all people is at the school of information.

An example of a class in python:

```
In [*]: class Person:
            department = 'School of Information' #a class variable

            def set_name(self, new_name): #a method
                self.name = new_name
            def set_location(self, new_location):
                self.location = new_location
```

To define a method, you just write it as you would have a function.
The one change is that to have access to the instance which a method is being invoked upon, you must include self, in the method signature.

Similarly, if you want to refer to instance variables set on the object, you prepend them with the word self, with a full stop.

In this definition of a person, for instance, we have written two methods.
Set name and set location.
And both instances bound variables, called name and location respectively.

When we run this cell, we see no output.
The class exists, but we haven't created any objects yet.
We can instantiate this class by calling the class name with empty parenthesis behind it.

```
person = Person()
person.set_name('Christopher Brooks')
person.set_location('Ann Arbor, MI, USA')
print('{} live in {} and works in the department {}'.format(person.name, person.location, person.department))
```

Then we can call functions and print out attributes of the class using the dot notation, common in most languages.

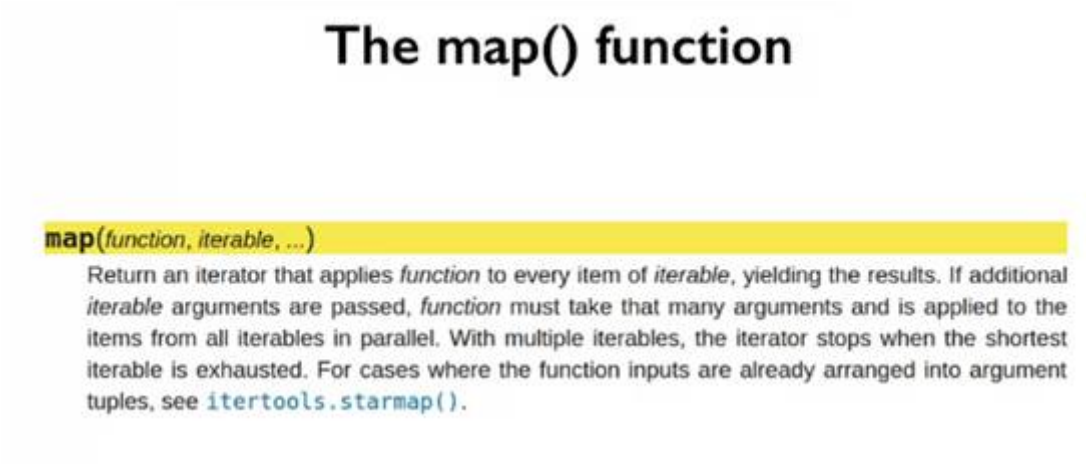
There are a couple of implications of object-oriented programming in Python that you should take away from this very brief example.

- First, objects in Python do not have private or protected members.
  If you instantiate an object, you have full access to any of the methods or attributes of that object.
- Second, there's no need for an explicit constructor when creating objects in Python.
  You can add a constructor if you want to by declaring the __init__ method.

"__init__" is a reserved method in python classes. It is called as a constructor in object oriented terminology. This method is called when an object is created from a class and it allows the class to initialize the attributes of the class.

The map function is one of the bases for functional programming in Python.
Functional programming is a programming paradigm in which you explicitly declare all parameters which could change through execution of a given function.
Thus functional programming is referred to as being side-effect free, because there is a software contract that describes what can actually change by calling a function.
Now, Python isn't a functional programming language in the pure sense.
Since you can have many side effects of functions, and certainly you don't have to pass in the parameters of everything that you're interested in changing.

But functional programming causes one to think more heavily while chaining operations together.
And this really is a sort of underlying theme in much of data science and date cleaning in particular.
So, functional programming methods are often used in Python, and it's not uncommon to see a parameter for a function, be a function itself.
The **map** built-in function is one example of a functional programming feature of Python, that I think ties together a number of aspects of the language.
The map function signature looks like this.
map(function, iterate,…)



The first parameters of function that you want executed, and the second parameter, and every following parameter, is something which can be iterated upon.

All the iterable arguments are unpacked together, and passed into the given function.
That's a little cryptic, so let's take a look at an example.

Here's an example of mapping the `min` function between two lists.

```
In [1]: store1 = [10.00, 11.00, 12.34, 2.34]
        store2 = [9.00, 11.10, 12.34, 2.01]
        cheapest = map(min, store1, store2)
        cheapest

Out[1]: <map at 0x7f909c2e1f60>
```

Imagine we have two lists of numbers, maybe prices from two different stores on exactly the same items.
And we wanted to find the minimum that we would have to pay if we bought the cheaper item between the two stores.
To do this, we could iterate through each list, comparing items and choosing the cheapest.

With map, we can do this comparison in a single statement.

But when we go to print out the map, we see that we get an odd reference value instead of a list of items that we're expecting.
This is called lazy evaluation.
In Python, the map function returns to you a map object.
It doesn't actually try and run the function min on two items, until you look inside for a value.
This is an interesting design pattern of the language, and it's commonly used when dealing with big data.
This allows us to have very efficient memory management, even though something might be computationally complex.

Maps are iterable, just like lists and tuples, so we can use a for loop to look at all of the values in the map.

Now let's iterate through the map object to see the values.

In [2]:
```
for item in cheapest:
    print(item)
```

```
9.0
11.0
12.34
2.01
```

This passing around of functions and data structures which they should be applied to is a hallmark of functional programming.
It's very common in data analysis and cleaning.

Here is a list of faculty teaching this MOOC. Can you write a function and apply it using map() to get a list of all faculty titles and last names (e.g. ['Dr. Brooks', 'Dr. Collins-Thompson', …]) ?