Communion: Python Lesson Two

These lessons are meant to give you an introduction to Python, not to teach you everything there is about it. Please do your own exploration if you're curious. This can be done through looking at the official Python tutorials, StackOverflow (a community driven QA site for coding), or just through experimentation.

Lesson Two Contents

Functions/Methods/Definitions

Before talking about the topic for this section, let's start with an example:

```
foo = [1, 2, 3, 4]
foo.reverse()
print(foo)
# do a bunch of stuff with foo

foo = [12, 13, 14, 15]
foo.reverse()
print(foo)
# do a bunch of stuff with foo

# repeat ad infinitum
```

You may think that putting the `reverse` and `print` into a loop of some kind would reduce the length of the code, and you would be right, except we have code running between each iteration and it's not given that that code is going to be the same each time. Since loops run the lines of code one right after another, this is not really the best solution. Additionally, the loop would require us to know all the `foo` lists beforehand, and in many cases, that's just not true. To solve this problem, we use `def` (short for definition, but most commonly called *functions* and sometimes *methods* in other languages). I will call them functions from here on out as this is what they're most commonly called throughout all languages and even Python programmers will often call them functions (not definitions).

Anatomy of a Function

```
def <name>([<arg>[, <arg> [...]]]):
    <code>
    <code>
```

```
    <...>
    [return [<obj>, [<obj> [...]]]]
```

Pieces encapsulated with `[]` are optional.

Functions are special when it comes to runtime as they are not run when read by the interpreter, like all of the code that we've been writing up to this point. Instead, it's loaded into memory as a function and has to be called. They're special because they can replace multiple code blocks that only vary by a few variables. Often, I'll write my code without thinking too much about adding functions in intelligent places, but when I find myself writing essentially the same code over and over, I'll replace the block with a function.

Another interesting thing to note concerning the top-down execution nature of many languages is that a function must be defined above any code that calls it. If not, then the interpreter will not have read that part of the code yet and will throw a `NameError`. For this reason, I always have all of my functions and classes at the top of a script and the actual code that gets executed at the bottom. That way, everything is loaded by the time it gets called.

The Pieces

- `<name>`: The name of your function. It's a good idea/practice to pick a typing convention for variables, functions, and classes (we'll get to these in the next section) so that when you're reading your code, it's readily visible what it is from the variable name. My preferences are:
    - variables: Lower-case with underscores. If I'm manipulating the same variable and it's mainly changing types, I'll tack the type to the end.
        - example: `this_is_my_bad_variable_name`
        - example: `foo_str` and `foo_int` would be the same data, just cast as string and integer, respectively.
    - functions: Camel-case with the first letter lower case.
        - example: `thisIsMyBadFunctionName`
    - classes: Camel-case with the first letter upper case.
        - example: `ThisIsMyBadClassName`
- `<arg>`: These are the variables that your function will have available to use (in addition to `global` variables). In programming languages, the concept of variable scope is qutie important. If a variable is defined within a function/class, it will only exist within that function/class. These are called local variables. If a variable is defined outside a function/class, then it will be a global variable. As the names imply, local variables can only be used locally (to the function that you are in) and global variables can be used anywhere in the code (including other scripts that import the code containing the global variable). Since we want to reduce the probability of name collision (using the same variable name in multiple places) and code unwittingly changing our variables against our will, it's good to use local variables wherever possible (which is almost always). This is where function arguments come in. This is essentially the way to pass variables from one function to another (don't worry, there'll be lots of examples).
- `<code>`: This is where you'll put your code. Note that it is all indented the same way that loops are.
- `<return>`: In the same vein of sharing variables to functions, it's important to share back. What good is sharing if you don't get anything in return? The `return` statement is where the function sends data back to the place that called it. The `return` can send back any number of items (including zero).

Examples

Rewriting our bad example from the beginning of this section with a function, we get:

```
def reverseAndPrint(data_list):
    data_list.reverse()
    print(data_list)
    return data_list
```

```
foo = [1, 2, 3, 4]
foo = reverseAndPrint(foo)
# do a bunch of stuff with foo

foo = [12, 13, 14, 15]
foo = reverseAndPrint(foo)
# do a bunch of stuff with foo
```

This is not a really great example of how a function can save you writing/shorten your code, but the idea of moving twenty or even thirty lines of code that's repeated multiple times into a function is quite powerful.

```
def unpacker(data_list):
    if type(data_list) == list:
        a = data_list[0]
        b = data_list[1]
        c = data_list[2]
        return a, b, c
    else:
        return 0, 0, 0

foo = [1, 2, 3]
f1, f2, f3 = unpacker(foo)
print(f1, f2, f3)

bar = 'hey there!'
b1, b2, b3 = unpacker(bar)
print(b1, b2, b3)
```

This example is a little more complex, but you can see that the function is effectively unpacking the list into individual elements and returning the elements. Many times this kind of function is very useful as data can get quite complex.

Python's "Main"

In many languages there is a `main` function that is the inception point of the program. As stated earlier, Python originated as a scripting language and one of their tennants is that the code executes from top down, line after line. In order to create a "starting place" for your code, it's often considered best practice to have all of your code encapsulated by functions and use `if __name__ == '__main__':` as your main function.

Example:

```
def function1():
    pass

def function2():
    pass
```

```
def function3():
    pass

# ...

if __name__ == '__main__':
    function1()
    function2()
    function3()
    # ...
```

In this framework, I would not have any code without an initial tab, unless it is a function definition. That way, I can ensure that I don't have any unwanted global variables and I can track where my code is executing more readily. There are quite a few other reasons for using `if __name__ == '__main__':` in your code, but their explanation exceeds the scope of what our topics will include. In general, it's a good idea to use it.

Note: `pass` is a way to tell the code to do nothing. Since Python relies on indents to denote code blocks, it makes sense that the code would have a hard time knowing what's going on if there was a conditional statement, loop, or function definition without any accompanying indented code.

Named Arguments

Let's consider the following example to explain named arguments:

```
def personInfo(name, gender, height_feet, height_inches, weight, hair_length, hair_color,
eye_color, ...):
    print('name', name)
    print('gender', gender)
    # ...

personInfo('John Doe', 'Male', 6, 4, 200, 'short', 'dark brown', 'dark brown', ...)
```

What if there were 50 arguments to this function? Certainly there are easily 50 different characteristics that we can define for a given individual. As the programmer, how embarassing would it be if you accidentally coded this such that the two height arguments were backwards? It can certainly be difficult to make sure that the arguments are lined up correctly every time. Additionally, what if some of the arguments are optional or you want default values? This is where named arguments come in.

A named argument is essentially adding a default value to your arguments which provides two important benefits:

1. if no value is provided, the function has one to work with and
2. you can call the function and provide just the arguments that you want.

Let's rewrite the example using named arguments:

```
def personInfo(name='default name', gender='undecided', height_feet='more than zero',
height_inches='less than 12', weight='more than the dog', hair_length='bald?',
hair_color='invisible...', eye_color='mauve', ...):
    pass

personInfo(name='Andy', weight=210, gender='Male')
```

In this rewrite, anywhere I didn't specify one of the arguments and pass a value, the defaults in the function declaration are used. Additionally, you can see that I didn't have to put the arguments in my call in any particular order. Since they're named, I can put them wherever I want. Any named argument is effectively optional.

Note: It is possible to mix named and non-named arguments. This way you can have required and optional arguments. The caveat is that non-named arguments need to be first in the function declaration and .

Classes

Again, before we go into detail about classes, let's reconsider the person example from above. Wouldn't it be difficult to make sure you keep all the data for a single person all in one place? What if we also wanted multiple functions that would only be used on that data (e.g. BMI calculation, estimated date of full hair-loss, etc)? It can be messy to just have all these functions floating around in your script for any piece of the code to call, as well as needing to pull together all this data each time you want to do anything with it. To solve this, classes and objects come into play.

First off, the relationship between a class and an object is that of architectural drawing and constructed house: the class defines how the object will look and behave. For example, somewhere deeply rooted in the Python install that you did lives the definition of how a string behaves. That is the string's class. Each time that you create a string variable, your instantiating that class and creating a string object. (By the way, *everything* in Python is a class/object at some level.)

Anatomy of a Class

```
class <name>[(<parent>)]:
    def <function_name>(self[, <arg>[, <arg> [...]]]):
        <code>
    [...]
```

```
class Person:
    def __init__(self, name, gender=None, height_feet=None, height_inches=None,
weight=None, hair_length=None, hair_color=None, eye_color=None):
        self.name = name
        self.sex = gender
        self.h_ft = height_feet
        self.h_in = height_inches
        self.lbs = weight
        self.hair_ln = hair_length
        self.hair_clr = hair_color
        self.eye_clr = eye_color

    def badlyCalculateBMI(self):
        self.BMI = ((self.h_ft*12) + self.h_in) / self.lbs

    # ...

if __name__ == '__main__':
    andy_obj = Person('Andy', height_inches=10, height_feet=5, weight=343241,
gender='guess')
```

```
    andy_obj.badlyCalculateBMI()
    print(andy_obj.BMI)
```