Lecture 4: Code organization: Functions

As you write more and more code, you will come to realize how important organization is for understanding your code and reusing it. This lecture will focus on two methods to accomplish this, writing scripts and creating functions.

Functions

One of the most useful aspects of programming language is our ability to create functions. As we have seen, we often use a function to do something over and over. For example, it's nice to have a function to return the methods/attributes of an object (dir()) by typing 5 extra characters. Functions DO something, often taking in arguments, performing some sort of analysis and then returning an output value. They are useful to define as a way to organize your code and also prevent the unnecessary rewriting of code. You should be familiar with using functions, as we have used some so far. In Python, you can create functions like so:

```
>>> def add(a,b):
...     c = a + b
...     return c
...
>>> add(1,2)
3
```

A function takes in 0 or more parameters. In performs some sort of analysis on these parameters and then returns 0 or more outputs. It can also perform other types of work, such as printing information to a screen or modifying a variable that it has access to.

We have defined this function to have a name called add (which is what you use to call it in your program) that takes in two parameters, called a and b. If we use an inappropriate amount of parameters, we get an error:

```
>>> add()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() missing 2 required positional arguments: 'a' and 'b'
```

A function typically has five required parts:

1. def is a Python keyword that lets it know you are about to define a function (similar to how ‘’ or [] implicitly define strings and lists).
2. a name of the function to call it within the program. len, type, and dir are three function names you have seen before. add is the name of the function we just defined.
3. 0 or more parameters. These parameters can be instances of any type. For example, you could pass an integer, a string, and a list to the function. You can even pass a function to the function.
4. Code – The function must know what code to perform on the parameters. In other languages, this is designated using open and closing braces {}. Python is a bit unusual in that it uses white-space to indicate this. After the first line, you must end it with a colon. Then, the following line or lines are all indented by the same amount of white-space. Typically it's easiest to use the tab to force each line to be indented by the same amount.
5. Output. A function can either output a built-in constant called None or a single object of any type. What is returned is defined by the return keyword. In this case, c is returned. If nothing follows the return keyword, None is returned.

Scoping and immutability

One of the confusing parts of a function are which variables are available for it to see.

```
def print_info():
    print(x)
```

```
x = 6
print_info()
print(x)
```

```
(base) ➜  ~ python3 function_examples.py
6
6
```

Here we can see that a function can access variable that are not passed to it or defined within it (x was defined in a completely different part of the code). But in general you shouldn't ever try access those variables as unexpect things will happen. For example:

```
def print_info():
    x = 2
    print(x)

x = 6
print_info()
print(x)
```

```
(base) ➜  ~ python3 function_examples.py
2
6
```

Notice that we did not change the value of x in the main part of the script. That is because a copy of the variable was created.

```
def print_info(info_variable):
     print(info_variable)

x = 6
print_info(x)
print(x)
```

```
(base) ➜  ~ python3 function_examples.py
6
6
```

In general, you should always pass the data that the function needs as an argument to the function. See the above code for how to do that.

And remember, the function makes copies of all the data you send it. You can't modify the original variable by modifying the argument like this:

```
def print_info(info_variable):
     info_variable = 2
     print(info_variable)

x = 6
print_info(x)
print(x)
```

```
(base) ➜  ~ python3 function_examples.py
2
6
```

If you want to modify the original variable, return the value from the function and set it in the main part of the code:

```
def print_info(info_variable):
      info_variable = 2
      print(info_variable)
      return info_variable

x = 6
x = print_info(x)
print(x)
```

```
(base) ➜  ~ python3 function_examples.py
2
2
```

One exception to how this works is lists. Because you only pass shallow copies of the list, any changes you make in the function are permanent.

```
def print_info(info_variable):
      info_variable[1] = 1
      print(info_variable)

x = [1,2,3,4]
print_info(x)
print(x)
```

```
(base) ➜  ~ python3 function_examples.py
[1,1,3,4]
[1,1,3,4]
```

Take home message: I hope you understand better now how functions work. The complexity of how variables get passed and modified means that you should follow a few rules when you create your own functions.

1. You should pass in (as parameters) all variables/data that your function needs. DO NOT access a variable that isn't passed into the function.
2. All output of a function should be returned using the return statement. DO NOT try to return output by modifying the parameters that are passed in.
3. Be very careful modifying lists (or later dictionaries and objects) in your functions. This is permanent and will modify the original data you sent in.