Python Functions

Jupyter notebook

The Jupyter notebook allows you to Put your code up into what they call cells, and execute these cells on demand.

For now, let's dig in to some basic Python.
First, Python is a high level language, which means it's optimized for reading by people instead of machines.
It's also an interpreted language which means it isn't compiled directly to machine code and importantly is commonly used in an interactive fashion. This might be quite different if you're used to programming in something like Java or C, where you write your code, compile it, run it, and watch the output.
  In Python, you can start the interactive interpreter and begin writing code, line by line, with the interpreter evaluating each statement as you write it. This turns out to be very useful for tasks that require a lot of investigation, versus those that require a lot of design. Shell scripting is one example of this. And data cleaning is another.
Or you can write scripts that execute programs like you might be used to another languages. We're going to use Python throw specialization in this interactive fashion.

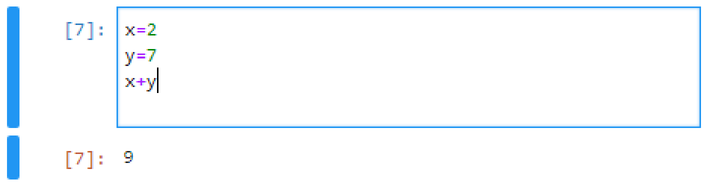
dynamic typing

A common surprise for some programmers coming from a Java or C background is that Python is a dynamically typed language, similar to languages like JavaScript. This means that when you declare a variable, you can assign it to be an integer on one line, and a string on the next line.

Since there is no compilation step, you don't have anyone to help you manage types. You need to either check for the presence of functionality when you go to use it or try and use the functionality and catch any errors that occur. The dynamic typing of Python is particularly nice when used in an interactive fashion, as it allows you to quickly set and modify variable contents without having to worry about the underlying syntactic definition of the variable.

Okay, so let's jump in with an example

Python is very little boiler plate code. In fact, if you just wanted to set the value of a couple of variables, and output the results of these variables added together, you could do so in three lines. In this example, I'll write three statements. The first two set the variables x and y, each to be some integer value. Then we'll do some addition.



```
[7]: x=2
     y=7
     x+y

[7]: 9
```

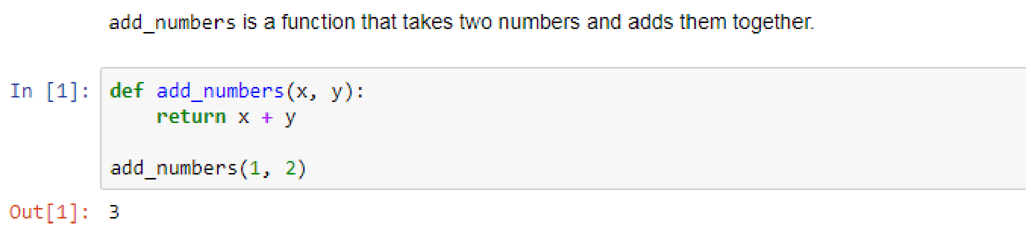
We can run this cell by hitting shift+enter or by clicking on the play head icon in the tool bar. The output from the statement is immediately printed. If you were using Python in a non interactive mode, nothing would print. But since we're using it in interactive mode, we get the value immediately. What's happening underneath is that the browser is sending your Python code across to a machine in the cloud, which executes the code in a Python three interpreter, and sends the results back.

We can see that after executing a cell, a new empty cell is created for us below. If we just put in x and execute, we get the value of 1. So it's important to know that the Python interpreter is stateful. That is, that your variables exist between cells.

Beyond that, if we go back and change something in a previous cell, we have to re-execute the script to make those changes take place. The restart and run all function is particularly useful, as it wipes the interpreter state and reruns all of the cells in the current notebook.

You'll notice that Python doesn't require the use of keywords like var to declare a variable name or semicolons at the end of lines which are commonly used in other languages. Python leverages white space to understand the scope of functions and loops and end of line markers to understand the end of statements.

Of course, Python has traditional software structures like functions. Here's an example, refactoring that previous code into a function. You'll see the *def* statement indicates that we're writing a function. Then each line that is part of the function needs to be indented with a tab character or a couple of spaces.



add_numbers is a function that takes two numbers and adds them together.

```
In [1]: def add_numbers(x, y):
            return x + y

        add_numbers(1, 2)

Out[1]: 3
```

Again, because we're in an interactive environment, when the statement is evaluated on a shift+enter, the results are printed out immediately below.

Here's our first bit of interaction. Why don't you try and change this function to accept three parameters instead of two and return the sum of all three of those.

Okay, functions are great but they're a bit different than you might find in other languages and here are some of subtleties involved. First, since there's no typing, you don't have to set your return type. Second, you don't have to set user return statement at all actually. There's a special value called none that's returned. None a similar to null in Java and represents the absence of value.

Third, in Python, you can have default values for parameters. Here's an example. In this example, we can rewrite the add numbers function to take three parameters, but we could set the last parameter to be none by default. This means that you can call add numbers with just two values or with three, and you don't have to rewrite the function signature to overload it.

`add_numbers` updated to take an optional 3rd parameter. Using `print` allows printing of multiple expressions within a single cell.

```
In [ ]: def add_numbers(x,y,z=None):
            if (z==None):
                return x+y
            else:
                return x+y+z

        print(add_numbers(1, 2))
        print(add_numbers(1, 2, 3))
```

labeled parameters

This is an important implication. All of the optional parameters, the ones that you got default values for, need to come at the end of the function declaration. It also means that you can pass an optional parameters as labeled values. Here's an example of a labeled parameters.
We can rewrite that function so that there's two optional parameters. If we want to call the function with just two numbers but also set the flag value, we have to explicitly name and set the flag parameter to true when invoking the function. By now, you've also seen the use of the print statement. We're going to dive into strings in more detail, but print will take an item, and convert it to a string and print the output. We don't use this much in the interactive mode of the interpreter, but it's useful when we want to print out multiple values for a single cell.

`add_numbers` updated to take an optional flag parameter.

```
In [ ]: def add_numbers(x, y, z=None, flag=False):
            if (flag):
                print('Flag is true!')
            if (z==None):
                return x + y
            else:
                return x + y + z

        print(add_numbers(1, 2, flag=True))
```

Okay, a final word on the basics of functions in Python. In Python, you can assign a variable to a function. This might seem either completely normal to you or completely odd depending upon on your programming background. By assigning a variable to a function, you can pass that variable into other functions allowing some basic functional programming. We'll talk about that a little bit later on in the course. But here's an example where we define a function to add numbers, then we assign that function to a variable, and then we invoke the variable.

Assign function `add_numbers` to variable `a`.

```
In [ ]: def add_numbers(x,y):
            return x+y

        a = add_numbers
        a(1,2)
```