Functions

Introduction to Functions

This lecture will consist of explaining what a function is in Python and how to create one. Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.

So what is a function?

Formally, a function is a useful device that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.

On a more fundamental level, functions allow us to not have to repeatedly write the same code again and again. If you remember back to the lessons on strings and lists, remember that we used a function len() to get the length of a string. Since checking the length of a sequence is a common task you would want to write a function that can do this repeatedly at command.

Functions will be one of most basic levels of reusing code in Python, and it will also allow us to start thinking of program design (we will dive much deeper into the ideas of design when we learn about Object Oriented Programming).

def Statements

Let's see how to build out a function's syntax in Python. It has the following form:

In [1]:
```
def name_of_function(arg1,arg2):
    '''
    This is where the function's Document String (docstring) goes
    '''
    # Do stuff here
    # Return desired result
```

We begin with `def` then a space followed by the name of the function. Try to keep names relevant, for example len() is a good name for a length() function. Also be careful with names, you wouldn't want to call a function the same name as a built-in function in Python (such as len).

Next come a pair of parentheses with a number of arguments separated by a comma. These arguments are the inputs for your function. You'll be able to use these inputs in your function and reference them. After this you put a colon.

Now here is the important step, you must indent to begin the code inside your function correctly. Python makes use of *whitespace* to organize code. Lots of other programing languages do not do this, so keep that in mind.

Next you'll see the docstring, this is where you write a basic description of the function. Using iPython and iPython Notebooks, you'll be able to read these docstrings by pressing Shift+Tab after a function name. Docstrings are not necessary for simple functions, but it's good practice to put them in so you or other people can easily understand the code you write.

After all this you begin writing the code you wish to execute.

The best way to learn functions is by going through examples. So let's try to go through examples that relate back to the various objects and data structures we learned about before.

Example 1: A simple print 'hello' function

In [2]:
```
def say_hello():
    print('hello')
```

Call the function:

In [3]:
```
say_hello()
```

```
hello
```

Example 2: A simple greeting function

Let's write a function that greets people with their name.

In [4]:
```
def greeting(name):
    print('Hello %s' %(name))
```

In [5]:
```
greeting('Jose')
```

```
Hello Jose
```

Using return

Let's see some example that use a `return` statement. `return` allows a function to *return* a result that can then be stored as a variable, or used in whatever manner a user wants.

Example 3: Addition function

In [6]:
```
def add_num(num1,num2):
    return num1+num2
```

In [7]:
```
add_num(4,5)
```

Out[7]: 9

In [8]:
```
# Can also save as variable due to return
result = add_num(4,5)
```

In [9]:
```
print(result)
```

```
9
```

What happens if we input two strings?

In [10]:
```
add_num('one','two')
```

Out[10]: 'onetwo'

Note that because we don't declare variable types in Python, this function could be used to add numbers or sequences together! We'll later learn about adding in checks to make sure a user puts in the correct arguments into a function.

Let's also start using `break`, `continue`, and `pass` statements in our code. We introduced these during the `while` lecture.

Finally let's go over a full example of creating a function to check if a number is prime (a common interview exercise).

We know a number is prime if that number is only evenly divisible by 1 and itself. Let's write our first version of the function to check all the numbers from 1 to N and perform modulo checks.

In [11]:
```
def is_prime(num):
    '''
    Naive method of checking for primes.
    '''
    for n in range(2,num):
        if num % n == 0:
            print(num,'is not prime')
            break
    else: # If never mod zero, then prime
        print(num,'is prime!')
```

In [12]:
```
is_prime(16)
```

```
16 is not prime
```

In [13]:
```
is_prime(17)
```

```
17 is prime!
```

Note how the `else` lines up under `for` and not `if`. This is because we want the `for` loop to exhaust all possibilities in the range before printing our number is prime.

Also note how we break the code after the first print statement. As soon as we determine that a number is not prime we break out of the `for` loop.

We can actually improve this function by only checking to the square root of the target number, and by disregarding all even numbers after checking for 2. We'll also switch to returning a boolean value to get an example of using return statements:

In [14]:
```
import math

def is_prime2(num):
    '''
    Better method of checking for primes.
    '''
    if num % 2 == 0 and num > 2:
        return False
    for i in range(3, int(math.sqrt(num)) + 1, 2):
        if num % i == 0:
            return False
    return True
```

In [15]:
```
is_prime2(18)
```

Out[15]:
```
False
```

Why don't we have any `break` statements? It should be noted that as soon as a function *returns* something, it shuts down. A function can deliver multiple print statements, but it will only obey one `return`.

Great! You should now have a basic understanding of creating your own functions to save yourself from repeatedly writing code!