Notebook

February 7, 2019

1 Functions and Scope

Recall how in the first Python lesson we looked at the `while` loop and saw how it allows us to repeat instructions to the computer as many times as you want.

The next step up from a loop is a function, which allows us to wrap up a series of commands into a single command on its own. Let's take a look at an example.

```
In [1]: def plus_one(number):
            return number + 1
```

That code gives us the basic form of a function. Let's break it down.

- First is the `def` keyword, which just say that we're defining a function.
- following that is a space, and then the name of the function, in this case `plus_one`
- immediately following that (no space) are names for the *parameters* that the function accepts, separated by commas and enclosed in parentheses. This function takes just one parameters, named `number`.
- following that is a colon signifying the beginning of a new *scope* — we'll talk about the concept of scope later, but just think of it as a new block like in a loop.
- following that is the *body* of the function, indented as per usual. Each of the parameters to the function is available as a variable in the body. This function just has one line in the body, and it adds 1 to whatever is assigned to `number`.
- this function, and most functions, end with a `return` statement: a statement that starts with the keyword `return` and then includes an expression that evaluates to some value. This is the value that, as the name suggests, the function returns. In this case, returns means that the function evaluates to it when it's executed.

Let's take a look at what happens when we execute it.

```
In [2]: two = plus_one(1)
        print(two)
```

```
2
```

We *call* a function by entering its name followed by parentheses containing values for the function's parameters—we call these values *arguments*. So here, we're calling the function `plus_one` with the value `1`. That value gets passed into the body of the function attached to the name in the appropriate place in the list of parameters, so it gets assigned `number` in the function.

The function then executed, and evaluates to its return statement, if any. Here, of course, that just means that it evaluates to the number 2, which is assigned to the variable and printed as normal.

Functions don't need to have any parameters:

```
In [3]: def cat_noise():
            return "MEOW"

        print(cat_noise())
```

```
MEOW
```

They also don't need to have a return statement. If they don't have a return statement, they return the special value `None`.

Why would you want to have a function with no return statement? Well, some functions can do things that don't require returning. For example, they might modify other values in memory (though that gets a little complicated and we'll talk about it later) or they might print something, make an internet request, delete a file, etc.

```
In [4]: def say_cat_noise():
            print("MEOW")

In [5]: my_variable = say_cat_noise()
```

```
MEOW
```

```
In [6]: print(my_variable)
```

```
None
```

Of course, functions can have multiple parameters.

```
In [7]: def greet(name, punctuation):
            greeting = "Hello " + name + punctuation
            return greeting

        print(greet("Leonidas, the best cat", "!!!!"))
```

```
Hello Leonidas, the best cat!!!!
```

```
In [8]: print(greet("Rex", "."))
```

```
Hello Rex.
```

Normally you pass arguments by position, i.e., the first argument is the first parameter in the function, and so forth, and so forth. But you can also pass arguments by name, by putting the names in the call to the function and then an equals sign. If you do so, you don't have to do it in order.

```
In [9]: print(greet(punctuation="?", name="students"))

Hello students?
```

Normally, you get an error if you pass fewer arguments to a function than its number of parameters.

```
In [10]: greet("nobody")


         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-10-2b2b4e8a714b> in <module>
     ----> 1 greet("nobody")


         TypeError: greet() missing 1 required positional argument: 'punctuation'
```

However, you can define a function with *default arguments* for some parameters, which get applied if you don't give an argument for that spot.

```
In [11]: def greet2(name, punctuation="!"):
             return "Hello " + name + punctuation

In [12]: print(greet2("Dean Washburn"))

Hello Dean Washburn!


In [13]: print(greet2("President Harreld", "?!!?"))

Hello President Harreld?!!?
```

1.1 Scope

Learning about functions also requires understanding the really important concept of *scope*. Every variable exists in a scope, and isn't defined, or has a different definition, outside of that scope.

Functions create their own scope. What that means is that a variable defined in a function, or passed into it as one of its arguments, doesn't exist outside that scope. Here's an example:

```
In [14]: x = 5
         y = 100

         def plus_10(x):
             y = 50
             print("inside the function, y = " + str(y))
             return x + 10

In [15]: print(x)

5


In [16]: print(plus_10(20))

inside the function, y = 50
30


In [17]: print(x)

5


In [18]: print(y)

100
```

What we can see here is that even though we assigned `x` and `y` to name in the scope of the `plus_10` function, that didn't touch the values of those variables assigned outside the function. We got to use those values inside the function without clashing with the external names. This is how scope works.

If a function can't find a variable in its own scope, it looks outside in its *enclosing scope* for it. Here's another example:

```
In [19]: def plus_100(x):
             return x + y

In [20]: print(plus_100(50))

150
```

y wasn't defined in the body or the arguments of `plus_100`, so it went out into the *global scope* to find a value for it. Since we'd assigned y to 100 earlier, it was findable by the function.

Let's reassign it now.

```
In [21]: y = 99

In [22]: plus_100(50)

Out[22]: 149
```

When we called `plus_100` again, it had to go looking for y in the global scope again, and it found a different value! So it used that.

This is also a good practical programming clue for you. It's a bad idea to rely too much on variables in global scope (called *global variables*), because if you accidentally change their names, it can lead to surprising bugs, like a function that you meant to add 100 to things actually adding 99 to them.

So what if you actually want to change a global variable inside a function? Well, you could use the `global` declaration in the body of the function, before you refer to that variable, in order to do that. But that's a bad idea, don't do it.

```
In [23]: # if you find outself tempted to try something like this,
         # you usually should rethink your code.
         def insist_on_plus_100(x):
             global y
             y = 100
             return x + y

In [24]: y = 99
         print(y)
         print(insist_on_plus_100(50))
         print(y)

99
150
100
```

You can have nested scope—scope within scope within scope. The outermost scope is the global scope, and a reference to a variable will look within the its own scope, and then within the next outer scope, and the next, and so forth, until it gets to the global scope.

```
In [25]: x = "global scope"

         def inner():
             x = "innermost scope"
             print(x)

         def middle():
             x = "middle scope"
```

```
        inner()
        print(x)

    def outer():
        middle()
        print(x)
```

Can you guess what calling `outer()` will do? Take a minute to figure it out, and then run this code yourself and see if you're right.