L7

June 20, 2018

1 Lecture 7: Functions

CSCI 1360E: Foundations for Informatics and Analytics

1.1 Overview and Objectives

In this lecture, we'll introduce the concept of *functions*, critical abstractions in nearly every modern programming language. Functions are important for abstracting and categorizing large codebases into smaller, logical, and human-digestable components. By the end of this lecture, you should be able to:

- Define a function that performs a specific task
- Set function arguments and return values
- Differentiate *positional* arguments from *keyword* arguments
- Write a function from scratch to answer questions in JupyterHub!

1.2 Part 1: Defining Functions

A *function* in Python is not very different from a function as you've probably learned since algebra.

"Let f be a function of x"...sound familiar? We're basically doing the same thing here.

A function (f) will [usually] take something as input (x), perform some kind of operation on it, and then [usually] return a result (y). Which is why we usually see $f(x) = y$.

A function, then, is composed of *three main components*:

1: **The function itself**. A [good] function will have one very specific task it performs. This task is usually reflected in its name. Take the examples of `print`, or `sqrt`, or `exp`, or `log`; all these names are very clear about what the function does.

2: **Arguments (if any)**. Arguments (or parameters) are the *input* to the function. It's possible a function may not take any arguments at all, but often at least one is required. For example, `print` has 1 argument: a string.

3: **Return values (if any)**. Return values are the *output* of the function. It's possible a function may not return anything; technically, `print` does not return anything. But common math functions like `sqrt` or `log` have clear return values: the output of that math operation.

1.2.1 Philosophy

A core tenet in writing functions is that **functions should do one thing, and do it well** (with apologies to the Unix Philosophy).

Writing good functions makes code *much* easier to troubleshoot and debug, as the code is already logically separated into components that perform very specific tasks. Thus, if your application is breaking, you usually have a good idea where to start looking.

It's very easy to get caught up writing "god functions": one or two massive functions that essentially do everything you need your program to do. But if something breaks, this design is very difficult to debug.

1.2.2 Functions vs Methods

You've probably heard the term "method" before, in this class. Quite often, these two terms are used interchangeably, and for our purposes they are pretty much the same.

BUT. These terms ultimately identify different constructs, so it's important to keep that in mind. Specifically:

- *Methods* are functions defined inside classes (sorry, not being covered in 1360E).
- *Functions* are not inside classes.

Otherwise, functions and methods work identically.

So how do we write functions? At this point in the course, you've probably already seen how this works, but we'll go through it step by step regardless.

First, we define the function *header*. This is the portion of the function that defines the name of the function, the arguments, and uses the Python keyword `def` to make everything official:

```
In [1]: def our_function():
            pass

In [2]: def our_function():
            pass
```

That's everything we need for a working function! Let's walk through it:

- `def` keyword: required before writing any function, to tell Python "hey! this is a function!"
- **Function name**: one word (can "fake" spaces with underscores), which is the name of the function and how we'll refer to it later
- **Arguments**: a comma-separated list of arguments the function takes to perform its task. If no arguments are needed (as above), then just open-paren-close-paren.
- **Colon**: the colon indicates the end of the function header and the start of the actual function's code.
- `pass`: since Python is sensitive to whitespace, we can't leave a function body blank; luckily, there's the `pass` keyword that does pretty much what it sounds like--no operation at all, just a placeholder.

Admittedly, our function doesn't really do anything interesting. It takes no parameters, and the function body consists exclusively of a placeholder keyword that also does nothing. Still, it's a perfectly valid function!

```
In [3]: # Call the function!

        our_function()

        # Nothing happens...no print statement, no computations, nothing.
        # But there's no error either...so, yay?
```

1.2.3 Other notes on functions

- You can define functions (as we did just before) almost anywhere in your code. Still, good coding practices behooves you to generally group your function definitions together, e.g. at the top of your Python file.

- Invoking or activating a function is referred to as *calling* the function. When you call a function, you type its name, an open parenthesis, any arguments you're sending to the function, and a closing parenthesis. If there are no arguments, then calling the function is as simple as typing the function name and an open-close pair of parentheses (as in our previous example).

1.3 Part 2: Function Arguments

Arguments (or parameters), as stated before, are the function's input; the "x" to our "f", as it were.

You can specify as many arguments as want, separating them by commas:

```
In [4]: def one_arg(arg1):
            print(arg1)

        def two_args(arg1, arg2):
            print(arg1, arg2)

        def three_args(arg1, arg2, arg3):
            print(arg1, arg2, arg3)

        # And so on...
```

Like functions, you can name the arguments anything you want, though also like functions you'll probably want to give them more meaningful names besides `arg1`, `arg2`, and `arg3`. When these become just three functions among hundreds in a massive codebase written by dozens of different people, it's helpful when the code itself gives you hints as to what it does.

When you call a function, you'll need to provide the same number of arguments in the function call as appear in the function header, otherwise Python will yell at you.

```
In [5]: one_arg(10)  # "one_arg" takes only 1 argument

10


In [6]: one_arg(10, 5)  # "one_arg" won't take 2 arguments!


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

        TypeError                                 Traceback (most recent call last)

        <ipython-input-6-4f5a74bb73ae> in <module>()
    ----> 1 one_arg(10, 5)  # "one_arg" won't take 2 arguments!
```

```
        TypeError: one_arg() takes 1 positional argument but 2 were given
```

```
In [7]: two_args(10, 5)  # "two_args", on the other hand, does take 2 arguments
```

```
10 5
```

```
In [8]: two_args(10, 5, 1)   # ...but it doesn't take 3
```

```
        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)

        <ipython-input-8-ebd661bc7294> in <module>()
    ----> 1 two_args(10, 5, 1)   # ...but it doesn't take 3


        TypeError: two_args() takes 2 positional arguments but 3 were given
```

To be fair, it's a pretty easy error to diagnose, but still something to keep in mind--especially as we move beyond basic "positional" arguments (as they are so called in the previous error message) into optional arguments.

1.3.1 Default arguments

"Positional" arguments--the only kind we've seen so far--are required whenever you call a function. If the function header specifies a positional argument, then every single call to that functions needs to have that argument specified.

In our previous example, `one_arg` is defined with 1 positional argument, so *every time you call `one_arg`, you HAVE to supply 1 argument*. Same with `two_args` defining 2 arguments, and `three_args` defining 3 arguments. Calling any of these functions without exactly the right number of arguments will result in an error.

There are cases, however, where it can be helpful to have optional, or *default*, arguments. In this case, when the function is called, the programmer can decide whether or not they want to override the default values.

You can specify default arguments in the function header:

```
In [9]: def func_with_default_arg(positional, default = 10):
            print(positional, default)
```

```
In [10]: func_with_default_arg("pos_arg")
```

```
pos_arg 10
```

```
In [11]: func_with_default_arg("pos_arg", default = 999)
```

```
pos_arg 999
```

Can you piece together what's happening here?

Note that, in the function header, one of the arguments is set equal to a particular value:

`def func_with_default_arg(positional, default = 10):`

This means that you can call this function **with only 1 arguments**, and if you do, the second argument will take its "default" value, aka the value that is assigned in the function header (in this case, 10).

Alternatively, you can specify a different value for the second argument if you supply 2 arguments when you call the function.

Can you think of examples where default arguments might be useful?

Let's do one more small example before moving on to return values. Let's build a method which prints out a list of video games in someone's Steam library.

```
In [12]: def games_in_library(username, library):
             print("User '{}' owns: ".format(username))
             for game in library:
                 print("\t{}".format(game))
```

You can imagine how you might modify this function to include a default argument--perhaps a list of games that everybody owns by simply registering with Steam.

```
In [13]: games_in_library('fps123', ['DOTA 2', 'Left 4 Dead', 'Doom', 'Counterstrike', 'Team For
```

```
User 'fps123' owns:
        DOTA 2
        Left 4 Dead
        Doom
        Counterstrike
        Team Fortress 2
```

```
In [14]: games_in_library('rts456', ['Civilization V', 'Cities: Skylines', 'Sins of a Solar Empi
```

```
User 'rts456' owns:
        Civilization V
        Cities: Skylines
        Sins of a Solar Empire
```

```
In [15]: games_in_library('smrt789', ['Binding of Isaac', 'Monaco'])
```

```
User 'smrt789' owns:
        Binding of Isaac
        Monaco
```

In this example, our function `games_in_library` has two positional arguments: `username`, which is the Steam username of the person, and `library`, which is a list of video game titles. The function simply prints out the username and the titles they own.

1.4 Part 3: Return Values

Just as functions [can] take input, they also [can] return output for the programmer to decide what to do with.

Almost any function you will ever write will most likely have a return value of some kind. If not, your function may not be "well-behaved", aka sticking to the general guideline of doing one thing very well.

There are certainly some cases where functions won't return anything--functions that just print things, functions that run forever (yep, they exist!), functions designed specifically to test other functions--but these are highly specialized cases we are not likely to encounter in this course. Keep this in mind as a "rule of thumb": **if your function doesn't have a `return` statement, you may need to double-check your code.**

To return a value from a function, just use the `return` keyword:

```
In [16]: def identity_function(in_arg):
             return in_arg

In [17]: x = "this is the function input"
         return_value = identity_function(x)
         print(return_value)

this is the function input
```

This is pretty basic: the function returns back to the programmer as output whatever was passed into the function as input. Hence, "identity function."

Anything you can pass in as function parameters, you can return as function output, including lists:

```
In [18]: def explode_string(some_string):
             list_of_characters = []
             for index in range(len(some_string)):
                 list_of_characters.append(some_string[index])
             return list_of_characters

In [19]: words = "Blahblahblah"
         output = explode_string(words)
         print(output)

['B', 'l', 'a', 'h', 'b', 'l', 'a', 'h', 'b', 'l', 'a', 'h']
```

This function takes a string as input, uses a loop to "explode" the string, and returns a list of individual characters.

You can even return multiple values *simultaneously* from a function. They're just treated as tuples!

```
In [20]: def list_to_tuple(inlist):
             return [10, inlist]  # Yep, this is just a list.
```

```
In [21]: print(list_to_tuple([1, 2, 3]))
```

```
[10, [1, 2, 3]]
```

```
In [22]: print(list_to_tuple(["one", "two", "three"]))
```

```
[10, ['one', 'two', 'three']]
```

This two-way communication that functions enable--arguments as input, return values as output--is an elegant and powerful way of allowing you to design modular and human-understandable code.

1.5 Part 4: Keyword Arguments

In the previous lecture we learned about positional arguments. As the name implies, position is key:

```
In [23]: def pet_names(name1, name2):
             print("Pet 1: ", name1)
             print("Pet 2: ", name2)
```

```
In [24]: pet1 = "King"
         pet2 = "Reginald"
         pet_names(pet1, pet2)  # pet1 variable, then pet2 variable
         pet_names(pet2, pet1)  # notice we've switched the order in which they're passed to the
```

```
Pet 1:  King
Pet 2:  Reginald
Pet 1:  Reginald
Pet 2:  King
```

In this example, we switched the ordering of the arguments between the two function calls; consequently, the ordering of the arguments inside the function were also flipped. Hence, positional: position matters.

In contrast, Python also has *keyword* arguments, where order no longer matters **as long as you specify the keyword**. We can use the same `pet_names` function as before.

Only this time, we'll use the names of the arguments themselves (aka, *keywords*):

```
In [25]: pet1 = "Rocco"
         pet2 = "Lucy"
```

```
In [26]: pet_names(name1 = pet1, name2 = pet2)
         pet_names(name2 = pet2, name1 = pet1)
```

```
Pet 1:  Rocco
Pet 2:  Lucy
Pet 1:  Rocco
Pet 2:  Lucy
```

As you can see, we used the names of the arguments from the function header itself (go back to the previous slide to see the definition of `pet_names` if you don't remember), setting them equal to the variable we wanted to use for that argument.

Consequently, *order doesn't matter*--Python can see that, in both function calls, we're setting `name1 = pet1` and `name2 = pet2`.

Keyword arguments are extremely useful when it comes to default arguments.

Ordering of the keyword arguments doesn't matter; that's why we can specify some of the default parameters by keyword, leaving others at their defaults, and Python doesn't complain.

Here's an important distinction, though:

- Default (optional) arguments are **always** keyword arguments, but...
- Positional (required) arguments **MUST** come before default arguments, both in the function header, and whenever you call it!

In essence, you can't mix-and-match the ordering of positional and default arguments using keywords.

Here's an example of this behavior in action:

```
In [27]: # Here's our function with a default argument.
         # x comes first (required), y comes second (default)
         def pos_def(x, y = 10):
             return x + y

In [28]: # Here, we've specified both arguments, using the keyword format.
         z = pos_def(x = 10, y = 20)
         print(z)
```

```
30
```

```
In [29]: # We're still using the keyword format, which allows us to reverse their ordering.
         z = pos_def(y = 20, x = 10)
         print(z)
```

```
30
```

```
In [30]: # But *only* specifying the default argument is a no-no.
         z = pos_def(y = 20)
         print(z)
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-30-dcc9ab41d414> in <module>()
  1 # But *only* specifying the default argument is a no-no.
```

```
----> 2 z = pos_def(y = 20)
      3 print(z)


TypeError: pos_def() missing 1 required positional argument: 'x'
```

1.6 Review Questions

Some questions to discuss and consider:

1: You're a software engineer for a prestigious web company named after a South American rain forest. You've been tasked with rewriting their web-based shopping cart functionality for users who purchase items through the site. Without going into too much detail, quickly list out a handful of functions you'd want to write with their basic arguments. Again, no need for excessive detail; just consider the workflow of navigating an online store and purchasing items with a shopping cart, and identify some of the key bits of functionality you'd want to write standalone functions for, as well as the inputs and outputs of those functions.

2: From where do you think the term "positional argument" gets its name?

3: Write a function, `grade`, which accepts a positional argument `number` (floating point) and returns a letter grade version of it ("A", "B", "C", "D", or "F"). Include a second, default argument that is a string and indicates whether there should be a "+", "-", or no suffix to the letter grade (default is no suffix).

4: Name a couple of functions in your experience that would benefit from being implemented with default arguments (hint: mathematical functions).

5: Give some examples for when we'd want to use keyword arguments *and* positional arguments.

1.7 Course Administrivia

- How did Assignment 2 go?
- **Assignment 3 was released yesterday.** Good luck!
- **Assignment 4 is released tomorrow.**

1.8 Additional Resources

1. Matthes, Eric. *Python Crash Course*. 2016. ISBN-13: 978-1593276034