

Functions

January 28, 2020

1 Lecture 8: Functions

CBIO (CSCI) 4835/6835: Introduction to Computational Biology

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-digestible 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
- Construct functions that take any number of arguments, in positional or key-value format

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.

WARNING: 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 inside classes (not really covered in this course).
- *Functions* are not inside classes. In some sense, they're "free" (though they may be found inside specific modules; however, since a module != a class, they're still called functions).

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:

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

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

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

- **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!

```
[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. As we'll see when we get to functional programming, you can literally define functions in the middle of a line of code. Still, good coding practices behooves you to generally group your function definitions together, e.g. at the top of your module.
- Invoking or activating a function is referred to as *calling* the function.
- Functions can be part of modules. You've already seen some of these in action: the `numpy.array()` functionality is indeed a function.
- Though not recommended, it's possible to import *only* select functions from a module, so you no longer have to specify the module name in front of the function name when calling the function. This uses the `from` keyword during import:

```
[4]: from numpy import array
```

Now the `array()` method can be called directly without prepending the package name `numpy` in front. **USE THIS CAUTIOUSLY:** if you accidentally name a variable `array` later in your code, you will get some very strange errors!

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:

```
[5]: def one_arg(arg1):
      pass

      def two_args(arg1, arg2):
```

```

    pass

def three_args(arg1, arg2, arg3):
    pass

# 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.

```
[6]: one_arg("some arg")
```

```
[7]: two_args("some arg")
```

```

-----

TypeError                                Traceback (most recent call last)

<ipython-input-7-161408af5f75> in <module>
----> 1 two_args("some arg")

TypeError: two_args() missing 1 required positional argument: 'arg2'

```

```
[8]: two_args("some arg", "another arg")
```

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. If the function header specifies a positional argument, then every single call to that functions needs to have that argument specified.

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:

```
[9]: def func_with_default_arg(positional, default = 10):
      print("'" + positional + "' with default arg '" + str(default) + "'")

      func_with_default_arg("Input string")
      func_with_default_arg("Input string", default = 999)
```

```
'Input string' with default arg '10'
'Input string' with default arg '999'
```

If you look through the NumPy online documentation, you'll find most of its functions have entire books' worth of default arguments.

The `numpy.array` function we've been using [has quite a few](#); the only positional (required) argument for that function is some kind of list/array structure to wrap a NumPy array around. Everything else it tries to figure out on its own, *unless* the programmer explicitly specifies otherwise.

```
[10]: import numpy as np
      x = np.array([1, 2, 3])
      y = np.array([1, 2, 3], dtype = float) # Specifying the data type of the array,
      ↳ using "dtype"

      print(x)
      print(y)
```

```
[1 2 3]
[1. 2. 3.]
```

Notice the decimal points that follow the values in the second array! This is NumPy's way of showing that these numbers are *floats*, not integers!

1.3.2 Keyword Arguments

Keyword arguments are a something of a superset of positional and default arguments.

By the names, *positional* seems to imply a relationship with position (specifically, position in the list of arguments), and *default* seems obvious enough: it takes on a default value unless otherwise specified.

Keyword arguments can overlap with both, in that they can be either required or default, but provide a nice utility by which you can ensure the variable you're passing into a function is taking on the exact value you want it to.

Let's take the following function.

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

      pet1 = "King"
```

```
pet2 = "Reginald"
pet_names(pet1, pet2)
pet_names(pet2, pet1)
```

```
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 function as before, `pet_names`, only this time we'll use the names of the arguments themselves (aka, *keywords*):

```
[12]: pet1 = "Rocco"
      pet2 = "Lucy"

      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, 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`.

Even though keyword arguments somewhat obviate the need for strictly *positional* arguments, keyword arguments are extremely useful when it comes to default arguments.

If you take a look at any NumPy API—even the documentation for `numpy.array`—there are LOTS of default arguments. Trying to remember their ordering is a pointless task. What's much easier is to simply remember the name of the argument—the *keyword*—and use that to override any default argument you want to change.

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!

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

(you can't really mix-and-match the ordering of positional and default arguments anyway, so hopefully this isn't a rude awakening)

Here's an example of this behavior in action:

```
[13]: # Here's our function with a default argument.
def pos_def(x, y = 10):
    return x + y
```

```
[14]: # Using keywords in the same order they're defined is totally fine.
z = pos_def(x = 10, y = 20)
print(z)
```

30

```
[15]: # Mixing their ordering is ok, as long as I'm specifying the keywords.
z = pos_def(y = 20, x = 10)
print(z)
```

30

```
[16]: # Only specifying the default argument is a no-no.
z = pos_def(y = 20)
print(z)
```

TypeError

Traceback (most recent call last)

```
<ipython-input-16-1e241f325129> in <module>
    1 # 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'

1.3.3 Arbitrary Argument Lists

There are instances where you'll want to pass in an arbitrary number of arguments to a function, a number which isn't known until the function is called and could change from call to call!

On one hand, you could consider just passing in a single list, thereby obviating the need. That's more or less what actually happens here, but the syntax is a tiny bit different.

Here's an example: a function which lists out pizza toppings. Note the format of the input argument(s):

```
[17]: def make_pizza(*toppings):  
        print("Making a pizza with the following toppings:")  
        for topping in toppings:  
            print(" - " + topping)
```

```
[18]: make_pizza("pepperoni")  
        make_pizza("pepperoni", "banana peppers", "green peppers", "mushrooms")
```

Making a pizza with the following toppings:

- pepperoni

Making a pizza with the following toppings:

- pepperoni
- banana peppers
- green peppers
- mushrooms

Inside the function, it's basically treated as a list: in fact, it *is* a list.

So why not just make the input argument a single variable which is a list?

Convenience.

In some sense, it's more intuitive to the programmer calling the function to just list out a bunch of things, rather than putting them all in a list structure first.

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.”

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

```
[19]: def identity_function(in_arg):  
        return in_arg  
  
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:

```
[20]: def compute_square(number):  
      square = number ** 2  
      return square
```

```
[21]: start = 3  
      end = compute_square(start)  
      print("Square of " + str(start) + " is " + str(end))
```

Square of 3 is 9

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

```
[22]: import numpy.random as r  
  
      def square_and_rand(number):  
          square = compute_square(number)  
          rand_num = r.randint(0, 100)  
          return rand_num, square
```

```
[23]: retvals = square_and_rand(3)  
      print(retvals)
```

(96, 9)

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: A Note on Modifying Arguments

This is arguably one of the trickiest parts of programming, so **please** ask questions if you’re having trouble.

Let’s start with an example to illustrate what’s this is. Take the following code:

```
[24]: def magic_function(x):  
      x = 20  
      print("Inside function: x = " + str(x))
```

```
[25]: x = 10  
      print("Before calling 'magic_function': x = " + str(x))
```

```
# Now, let's call magic_function(). What is x = ?
```

Before calling 'magic_function': x = 10

```
[26]: magic_function(x)
```

Inside function: x = 20

Once the function finishes running, what is the value of x?

```
[27]: print(x)
```

10

It prints 10. Can anyone explain why?

Let's take another, slightly different, example.

```
[28]: def magic_function2(x):  
      x[0] = 20  
      print("Inside function: x = " + str(x))
```

```
[29]: x = [10, 10]  
      print("Before function: x = " + str(x))  
  
      # Now, let's call magic_function2(x). What is x = ?
```

Before function: x = [10, 10]

```
[30]: magic_function2(x)
```

Inside function: x = [20, 10]

Once the function finishes running, what is the value of x?

```
[31]: print(x)
```

[20, 10]

It prints [20, 10]. Can anyone explain why?

This is one of the trickiest aspects of programming and isn't something I want to get into (look up *pass by value* and *pass by reference* if you're curious about the theory).

However, I bring this up because you still need to understand good programming practices when writing functions so your code doesn't do weird things.

- In general, when you write functions that accept arguments, you **should NOT modify the arguments themselves**.
- Instead, treat them as constants, and **return** any new values you want to use later.

1.6 Review Questions

Some questions to discuss and consider:

- 1: From where do you think the term “positional argument” gets its name?
- 2: 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).
- 3: Name a couple of functions in your experience that would benefit from being implemented with default arguments (hint: mathematical functions).
- 4: Give some examples for when we’d want to use keyword arguments *and* positional arguments.

1.7 Administrivia

- **How is Assignment 2 going?** Due next week!
- This is the last major Python topic! There will be others, but in terms of fundamental language basics, this is it!

1.8 Additional Resources

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