L8

June 20, 2017

1 Lecture 8: Functions II

CSCI 1360E: Foundations for Informatics and Analytics

1.1 Overview and Objectives

In the previous lecture, we went over the basics of functions. Here, we'll expand a little bit on some of the finer points of function arguments that can both be useful but also be huge sources of confusion. By the end of the lecture, you should be able to:

- Differentiate *positional* arguments from *keyword* arguments
- Construct functions that take any number of arguments, in positional or key-value format
- Explain "pass by value" and contrast it with "pass by reference", and why certain Python types can be modified in functions while others can't

1.2 Part 1: Keyword Arguments

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

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

In [2]: 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 [3]: pet1 = "Rocco"
        pet2 = "Lucy"

In [4]: 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 [5]: # 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 [6]: # Here, we've specified both arguments, using the keyword format.
        z = pos_def(x = 10, y = 20)
        print(z)

30

In [7]: # 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 [36]: # 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-36-2668ae51c8c4> 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'
```

1.3 Part 2: Passing an Arbitrary Number of Arguments

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):

```
In [9]: def make_pizza(*toppings):
            print("Making a pizza with the following toppings:")
            for topping in toppings:
                print(" - ", topping)
```

```
In [10]: make_pizza("pepperoni")
```

```
Making a pizza with the following toppings:
 -  pepperoni
```

```
In [11]: make_pizza("pepperoni", "banana peppers", "green peppers", "mushrooms")
```

```
Making a pizza with the following toppings:
 -  pepperoni
 -  banana peppers
 -  green peppers
 -  mushrooms
```

Inside the function, the arguments are 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. But that argument could go either way depending on the person and the circumstance, most likely.

With variable-length arguments, you may very well ask: this is cool, but it doesn't seem like I can make keyword arguments work in this setting? And to that I would say, *absolutely correct!*

So we have a slight variation to accommodate keyword arguments in the realm of including arbitrary numbers of arguments:

```
In [12]: def build_profile(**user_info):
             profile = {}
             for key, value in user_info.items():  # This is how you iterate through the key/val
                 profile[key] = value
             return profile
```

```
In [13]: profile = build_profile(firstname = "Shannon", lastname = "Quinn", university = "UGA")
         print(profile)
```

```
{'university': 'UGA', 'firstname': 'Shannon', 'lastname': 'Quinn'}
```

```
In [14]: profile = build_profile(name = "Shannon Quinn", department = "Computer Science")
         print(profile)
```

```
{'name': 'Shannon Quinn', 'department': 'Computer Science'}
```

Instead of one star (*) in the function header, there are two (**). And yes, instead of a list when we get to the inside of the function, now we basically have a dictionary!

Arbitrary arguments (either "lists" or "dictionaries") can be mixed with positional arguments, as well as with each other.

```
In [15]: def build_better_profile(firstname, lastname, *nicknames, **user_info):
             profile = {'First Name': firstname, 'Last Name': lastname}
             for key, value in user_info.items():
                 profile[key] = value
             profile['Nicknames'] = nicknames
             return profile
```

```
In [16]: profile = build_better_profile("Shannon", "Quinn", "Professor", "Doctor", "Master of Sc
                                        department = "Computer Science", university = "UGA")
         for key, value in profile.items():
             print(key, ": ", value)
```

```
First Name :  Shannon
university :  UGA
Last Name :  Quinn
department :  Computer Science
Nicknames :  ('Professor', 'Doctor', 'Master of Science')
```

That last example had pretty much everything.

- We have our positional or keyword arguments (they're used as positional arguments here) in the form of `firstname` and `lastname`
- `*nicknames` is an arbitrary list of arguments, so anything beyond the positional / keyword (or default!) arguments will be considered part of this aggregate
- `**user_info` is comprised of any key-value pairs that are *not* among the default arguments; in this case, those are `department` and `university`

(Don't worry--we won't use these mechanisms too often in this class. They're also not used too often in practice, but it's still good to know how to *read* them when they do show up)

1.4 Part 2: Pass-by-value vs Pass-by-reference

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:

```
In [17]: def magic_function(x):
             x = 20
             print("Inside function: ", x)
```

```
In [18]: x = 10
         print("Before function: ", x)
```

```
Before function:  10
```

```
In [19]: magic_function(x)
```

```
Inside function:  20
```

If we wrote another `print` statement, what would print out? 10? 20? Something else?

```
In [20]: print("After function: ", x)
```

```
After function:  10
```

It prints **10**. Before explaining, let's take another example.

```
In [21]: def magic_function2(x):
             x[0] = 20
             print("Inside function: {}".format(x))
```

```
In [22]: x = [10, 10]
         print("Before function: {}".format(x))
```

```
Before function: [10, 10]
```

```
In [23]: magic_function2(x)
```

```
Inside function: [20, 10]
```

What would a `print` statement now print out? `[10, 10]`? `[20, 10]`? Something else?

```
In [24]: print("After function: ", x)
```

```
After function:  [20, 10]
```

It prints `[20, 10]`.
To recap, what we've seen is that

1. We tried to modify an integer function argument. It worked *inside* the function, but once the function completed, the old value returned.
2. We modified a list element of a function argument. It worked inside the function, and the changes were still there after the function ended.

Explaining these seemingly-divergent behaviors is the tricky part, but to give you the punchline:

- ## 2 1 (attempting to modify an integer argument) is an example of pass by value, in which the *value* of the argument is copied when the function is called, and then discarded when the function ends, hence the variable retaining its original value.

- ## 3 2 (attempting to modify a list argument) is an example of pass by reference, in which a *reference* to the list--not the list itself!--is passed to the function. This reference still points to the original list, so any changes made inside the function are also made to the original list, and therefore persist when the function is finished.

StackOverflow has a great gif to represent this process in pictures.

- In pass by value (on the right), the cup (argument) is outright copied, so any changes made to it inside the function are made on the *copy*, not the *original*. So when the function ends and the copy vanishes, so do the edits you made on it.

- In pass by reference (on the left), only a reference to the cup is given to the function. This reference, however, "refers" to the original cup--as in, the original version is NOT copied!--so changes made to the reference are propagated back to the original, and therefore persist even when the function ends.

3.0.1 What are "references"?

So what are these mysterious references?

Imagine you're throwing a party for some friends who have never visited your house before. They ask you for directions (or, given we live in the age of Google Maps, they ask for your home address).

Rather than try to hand them your entire house, or put your physical house on Google Maps (I mean this quite literally), what do you do? You **write down your home address on a piece of paper** (or, realistically, send a text message).

This is not your house, but it is a *reference* to your house. It's small, compact, and easy to give out--as opposed to your physical, literal home--while intrinsically providing a path to the real thing.

So it is with references. They hearken back to ye olde computre ayge when fast memory was a precious commodity measured in kilobytes, which is not enough memory to store even the Facebook home page.

It was, however, enough to store the address. These addresses, or *references*, would point to specific locations in the larger, much slower main memory hard disks where all the larger data objects would be saved.

Scanning through larger, extremely slow hard disks looking for the object itself would be akin to driving through every neighborhood in the city of Atlanta looking for a specific house. Possible, sure, but not very efficient. Much faster to have the address in-hand and drive directly there whenever you need to.

3.0.2 That doesn't explain the differing behavior of lists versus integers as function arguments

Very astute. This has to do with a subtle but important difference in how Python passes variables of different types to functions.

- For the "primitive" variable types--int, float, string--they're passed by value. These are [typically] small enough to be passed directly to functions. However, in doing so, they are *copied* upon entering the function, and these copies vanish when the function ends, along with any changes you made to them inside the function.

- For the "object" variable types--lists, sets, dictionaries, NumPy arrays, generators, and pretty much anything else that builds on "primitive" types--they're passed by reference. This means you can modify the values inside these objects while you're still in the function, and those modifications will persist even after the function ends (because the original object was never copied--you were, in a sense, working directly on the original).

Think of references as "arrows"--they refer to your actual objects, like lists or NumPy arrays. The name with which you refer to your object is the reference.

```
In [25]: some_list = [1, 2, 3]
```

```
        # some_list -> reference to my list
        # [1, 2, 3] -> the actual, physical list
```

Whenever you operate on some_list, you have to traverse the "arrow" to the object itself, which is separate. Again, think of the house analogy: whenever you want to clean your house, you have to follow your reference to it first.

This YouTube video isn't exactly the same thing, since C++ handles this much more explicitly than Python does. But if you substitute "references" for "pointers", and ignore the little code snippets, it's more or less describing precisely this concept.

There's a slight wrinkle in this pass-by-value, pass-by-reference story... Using what you've learned so far, what do you think the output of this function will be?

```
In [26]: def set_to_none(some_list):
             some_list = None  # sets the reference "some_list" to point at nothing
             print("In function: ", some_list)

In [27]: a_list = [1, 2, 3]
         print("Before function: {}".format(a_list))

Before function: [1, 2, 3]


In [28]: set_to_none(a_list)

In function:  None
```

Now the function has finished; if we print out the list, what will we get?

```
In [29]: print(a_list)

[1, 2, 3]
```

"But," you begin, "you said objects like lists are pass-by-reference, and therefore any changes made in functions are permanent!"

And you'd be... mostly right. Because references are tricky little buggers.

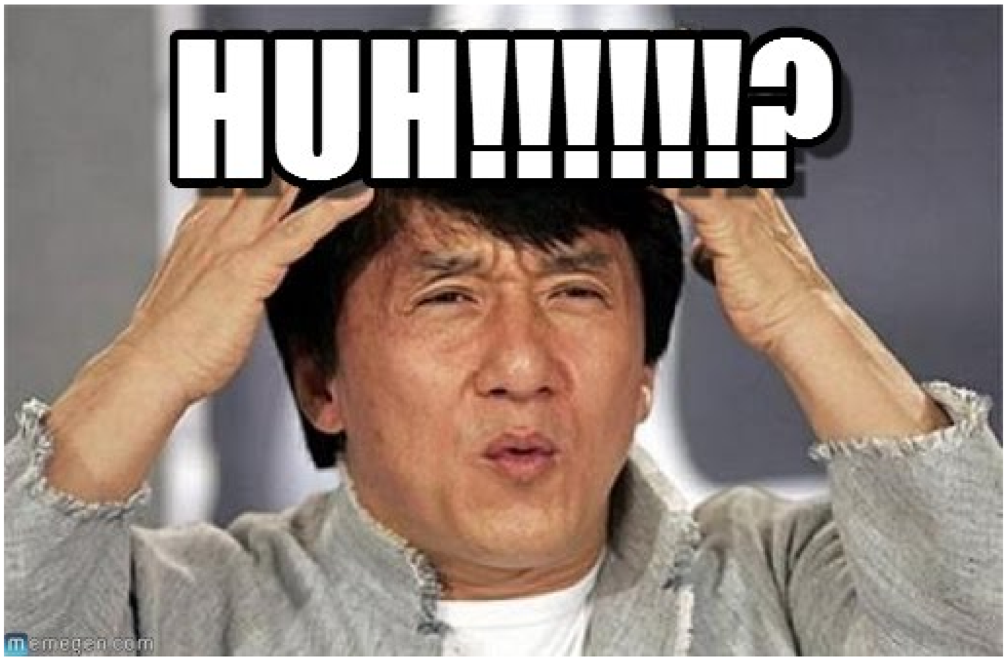
Here's the thing: ***everything* in Python is pass-by-value**. But **references to non-"primitive" objects, like lists, still exist and have specific effects.**

Let's parse this out.

We already know any "basic" data type in Python is passed by value, or copied. So any modifications we make inside a function go away outside the function (unless we return it, of course). This has not changed.

```
In [30]: def modify_int(x):
             x = 9238493  # Works as long as we're in this function...once we leave, it goes awa

In [31]: x = 10
         modify_int(x)
         print(x)  # Even though we set x to some other value in the function, that was only a c
```



waaaat

```
10
```

When it comes to more "complicated" data types--strings, lists, dictionaries, sets, tuples, generators--we have to deal with two parts: the reference, and the object itself.

When we pass one of the objects into a function, the *reference* is passed-by-value...meaning the *reference* is copied!

But since it points to the **same original object** as the original reference, any modifications made to the object persist, even though the copied reference goes away after the function ends.

It's like I sent out *two* text messages with my home address. I've copied the reference, though I clearly have not copied my house. And if a recipient deletes that text message, well--they only deleted the reference, not my house.

Upshot being: to persist modifications made inside a function to an object, you have to modify *the object itself*, not the reference. Setting the reference to `None` is a modification of the reference, not the object.

```
In [32]: def modify_list(x):
             x[0] = 9238493  # Here, we're modifying a specific part of the object, so this will
```

```
In [33]: a_list = [1, 2, 3]
         print(a_list)  # Before

         modify_list(a_list)
         print(a_list)  # After
```

```
[1, 2, 3]
[9238493, 2, 3]
```

Think of it this way:

- `x[0]` modifies the *list itself*, of which there is only 1.
- `x = None` is modifying *the reference to the list*, of which we have only a COPY of in the function. This copy then goes away when the function ends, and we're left with the *original* reference, which still points to the *original* list!

Clear as mud? Excellent!

Here is one more example, which illustrates why it's better to think of Python variables as two parts: one part *bucket of data* (the actual object), and one part *pointer to a bucket* (the name).

```
In [34]: x = [1, 2, 3]    # Create a list, assign it to x.
         y = x            # Assign a new variable y to x.
         x.append(4)      # Append an element to the list x.
         print(y)         # Print ** y **

[1, 2, 3, 4]
```

Notice how we called `append` on the variable `x`, and yet when we print `y`, we see the 4 there as well!

This is because `x` and `y` are both references that **point to the same object**. As such, if we reassign `x` to point to something else, `y` will still point to the first list.

```
In [35]: x = 5   # reassign x
         print(x)
         print(y)   # same as before!

5
[1, 2, 3, 4]
```

3.1 Review Questions

Some questions to discuss and consider:

1: Give some examples for when we'd want to use keyword arguments, arbitrary numbers of arguments, and key-value arguments.

2: Let's say I wanted to write a function to store travel information on a per-person basis: this function would let people specify an unbounded number of cities they'd visited, as well as their name, to tie the locations to them. Would this be a legal way of defining the function? Why or why not? `def places_visited(*cities, name_of_traveler):`

3: Let's say I wanted to write a function, `add1()`, which takes an integer as input and adds 1 to it. We know that integers are passed by value, so therefore any changes made to the argument inside the function are discarded when the function finishes. Are there any additional changes I could make so that this function will indeed give me a value that is 1 + the input argument?

4: Write a function `swap(x, y)` that swaps the values of the arguments `x` and `y`, without using a `return` statement. Can you do this with integer data types, or do you need to "augment" their type?

5: Describe what's happening in the code below. What will the value of `x` be at the two `print` statements? Why?

```
x = [1, 2, 3]
def func(x = 10):
    x = 20

func()
print(x)
func(x)
print(x)
```

3.2 Course Administrivia

- **How is Assignment 3 going?**
- **Assignment 4 comes out tomorrow!** It's also the last assignment before the midterm. Assignment 5 will come out after the exam.
- **Speaking of midterms, ours is coming up!** It's one week from Thursday, on July 29. It will be *entirely online*, on JupyterHub in fact! But it'll be flexible; you'll be able to do it somewhat at your leisure. More details to come.

3.3 Additional Resources

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