Tutorial Six: Functional Programming

Imad Pasha
Chris Agostino

March 11, 2015

1 Introduction

Earlier in the semester we went over how to define a function within python, which you can then call to do a specific task. Functions lend themselves well to a series of manipulations which you want to perform on multiple objects. For example say you write a block of code which can take an image of a galaxy, perform corrections to it, and perform some sort of analysis. In this format, the variable for the image will be used throughout the code, and somewhere at the top you will have a line which sets image equal to a loaded fits file. If you wanted to run your code block on a different image of another galaxy (or the same galaxy), you would have to go in to edit your code and change the load statement to match the new file.

Alternatively, you could "functionalize" this block of code. By enclosing it in a def statement, and making a filename one of the inputs, you can then run this code once in the interpreter, and call the function to all sorts of different images quickly.

1.1 Standalone Functions

Often when doing research you write blocks of code such as the example above, and it may take several hundred lines of code (all within a single function, perhaps with smaller functions defined and called within).

Python gives you the ability to save that document so that you can actually import your own function into a different .py script. Basically, you can make one python file where you just define a bunch of functions, and then import them into your actual research code to be used, to reduce clutter. To import the functions written in one .py file into another, simply ensure both files are in the same directory and in the research code import the name of the other file (something like defined_functions). You can also import specific functions from a file by typing "from filename import functionname".

2 Writing Functions

Functions are all about efficiency: saving yourself repetitive work. The reason you haven't maybe felt the strong need for them thus far in the class is you have had relatively simple

coding problems, which don't require repeated calculations.
Now imagine you have a data set and need to perform some complex calculation on them with slightly changed parameters each time, it quickly becomes quite tedious to type out every single time. Therefore, it behooves us to create our own functions such that we can input almost any variable arguments and the function will return our desired value. In a previous tutorial, we asked you to write a script which, with user inputs, uses the quadratic formula to solve for the roots of the equation you enter. In this tutorial, you will do something similar but the result will be a function. As a reminder, the syntax for writing a function goes a little something like this

```
def funcname(*args):
        perform calculations
        return desired values
```

Return statements are very important for functions, especially when you begin to assign variable names to be equal to the output of functions.

3 Some Basic Functions

1. Write a function which takes two positions (like 2 x and 2 y coordinates) and uses the distance function to return the distance between the points. E.g. I plug in 1,2,4,5 to get the distance between (1,2) and (4,5).

2. Write a function which takes in a distance of a planet in AU and returns the its period via Kepler's Third Law. Then use the distances below and a for loop to create an array of periods.

$$distances = .39, .72, 1.0, 1.52, 5.2, 9.54, 1918, 30.06, 39.52$$

   Plot the distances against the periods on a log log scale. Then, write a function that takes in this array and returns a dictionary of the planet corresponding to it's period. Keep in mind, everything is assumed to be in years here so do not worry about units.

3. Write a function which takes in two lists of strings containing first and last names respectively, then returns one list containing both.

4 A Recursive Function

RECURSION TIME! We are going to make a function that calculates a factorial using recursion. What is recursion? Recursion allows you to call a the function you are writing in the middle of the function you are writing. Sound weird? It is. It is useful for things like factorials, which are in fact *defined* recursively. For example, the definition of n! is

$$0! = 1 \tag{1}$$

$$n! = n(n-1)! \tag{2}$$

Notice how n! is quite literally defined in terms of the factorial of the n below it. To calculate a factorial, say, 3!, you do 3(2!)... so now we need to do 2!, which is 2*1!... so now we need to do 1!, which is 1*0!... ah, finally we have one that is actually defined. so 1! is 1, 2! is 2, and 3! is 6, working back up.

Amazingly, we can actually have our function we write call itself in the middle of its own execution, which will allow us to recursively calculate the answer. You will be making a function that returns the nth digit of the famous Fibonacci sequence. But before sending you off on that, lets work through an example of recursion on the factorial.

4.1 A recursive factorial function

```
def factorial(n):
    if n==0:
        return 1
    else:
        recursive = factorial(n-1)
        result = n * recursive
        return result
```

Above we have a fully functional, recursive factorial function. Note that the word recursive used as a variable name is not ”special”, we could’ve use any variable names for anything in the function. Notice how if you plug in an n greater than 0, it calls itself on one below your n, and if THAT is greater than 0, it does it again, and again, until finally, n-1 is 0. Then when it plugs in n=0, it finally returns one. But all of this is in the control flow of the function, so each time it re calls itself, it multiplies n by the output of the function itself called on n-1, in result. This is really mind-boggling, but think about it slowly for a while. Check it yourself to see that it works. Once you’re ready, you can try it yourself!

4.2 The Fibonacci Sequence

The Fibonacci sequence is familiar to most of you, it goes 1,1,2,3,5,8,13,....

It is defined by the following rules:

$$fibonacci(0) = 0 \tag{3}$$

$$fibonacci(1) = 1 \tag{4}$$

$$fibonacci(n) = fibonacci(n-1) + fibonacci(n-2) \tag{5}$$

That is, each n in fibonacci is the sum of the previous two, excluding 0 and 1.

Write a recursive function like the one above to determine the fibonacci number of any n. Before you run your new function, it might be helpful to implement some safeguards. Safeguards protect our code from things like infinte recursion (somewhat like an infinite while loop). As the n entered is a positive integer, things will be fine. Your first step is to check if it’s greater than 0- this isn’t new, just have an if statement for it first thing, and if it isn’t, print an error string and return none. You also want to check the datatype entered- which i’ll tell you you can do like this:

```
if not isinstance(n,int):
    print "Fibonacci is only for integers, sorry'
    return none
```

With those safeguards in place, you can check out your function and see if it properly returns the sequence.

4.3 Recursion with lists

Say we have some sort of sorted list with numbers going from smallest to largest. like

```
numbers = [3,8,15,19,24,29,32,35,37,40,43,45,47,95]
```

similar to an average physics course's midterm distribution. Now let's say we want to reverse said list such that the numbers go from largest to smallest. There is an inherent python function which reverses lists but that's no fun so in this problem we ask you to write a function, preferably a recursive one, that takes in a list and returns the reversed form of it.
In a similar manner, do this on the list of distances given in the Kepler problem to confirm you have succeeded.
Though you do not need to do this, think of a way in which you could sort a given list of unsorted numbers using recursion

5 Global and Local Variables

Thus far in our coding adventure, most of the variables we have declared are what are called "global" variables- that is, they can be accessed/retrieved in any part of your code. For example, when you define an empty list and then start a for-loop, you can access that list within the for-loop to append to it, as we have often done. There are certain variables that when declared are only "local", or accessible within a certain part of your code. For example, when you implicitly declare a variable "i" in a for loop by saying "for i in blah", that "i" is only usable within the for loo. If later in your code you try to use i, python won't know what you mean.

The other main place where we have locally defined variables is in functions. When you define a function, you can declare whatever variables you like within it to accomplish the intermediary tasks needed to achieve the output. But none of those variables are global to your code- you can't reference and use them outside your function. I.e., if I had the recursion function above, I couldn't try to use the variable "recursive" anywhere in the body of my code- it is limited to use within the function.

Now, as we have seen, you specify which arguments your functions can take. However, any globally defined variable in your code can be used within a function in your code, without passing it as an argument. NEVER DO THIS. The whole point of a function is that it is self sufficient- ideally, the ONLY things your function should need in order to work are the arguments you pass to it. (I.e., if your function were the only thing in the whole python file, it should still work when you try to use it). So try to make sure your functions have as arguments everything they need in order to function.