CHAPTER

3

Functions, Scoping, Recursion, and Other Miscellany

OUTLINE

> *There is no way that this winter is *ever* going to end as long as this groundhog keeps seeing his shadow. I don't see any other way out. He's got to be stopped. And I have to stop him.*
>
> **–*"Phil Connors" in the movie* Groundhog Day**

CHAPTER POINTS

- Defining functions makes code reuseable.
- Docstrings are special comments that indicate how to use a function, and can be accessed using the `help` function.
- Functions have their own variables, as defined by scoping rules.
- Functions can be bundled into a module and called from many programs.

3.1 FUNCTIONS

In this chapter we are going to define our own functions to make life easier on ourselves. Defining functions will make our code more robust, less prone to errors, and more usable. When we define a function what we want to do is to create an abstract version of a concrete set of steps that we want to execute in our code. By creating abstract versions we will be able to run the same lines of code repeatedly without typing them over and over. Motivating why

Computational Nuclear Engineering and Radiological Science Using Python
DOI: 10.1016/B978-0-12-812253-2.00004-2

this is necessary for writing good code is a bit of an uphill battle because cutting and pasting code repeatedly can seem pretty easy.

To demonstrate why we might want to define a function, we will solve in code a particular example of a simple system of linear equations. Our example system is

$$4.5x + 3y = 10.5$$

$$1.5x + 3y = 7.5.$$

We could find the values of x and y that solve these equations with the following Python code that eliminates a variable by combining the equations to solve for y and then solving for x. Read the comments to see what is happening.

```
In [1]: """Python code to solve
        4.5 x + 3 y = 10.5
        1.5 x + 3 Y = 7.5
        by solving the second equation for y first,
        and then solving for x"""
        #step 1 solve for y, multiply equation 2 by
        #-3, and add to first equation
        LHS_coefficient = -3*3 + 3 #the coefficient for y
        RHS = -3*7.5 + 10.5 #the right-hand side
        #now divide right-hand side by left-hand side coefficient
        y = RHS / LHS_coefficient
        #plug y into first equation
        x = (10.5 - 3*y)/4.5
        #print the solution, note \n produces a linebreak
        print("The solution to:\n4.5 x + 3 y = 10.5\n
              1.5 x + 3 y = 7.5\n is x =",x,"y=",y)

The solution to:
4.5 x + 3 y = 10.5
1.5 x + 3 y = 7.5
 is x = 1.0 y= 2.0
```

Our code appears to work (you can check by plugging in the values into the system). Given that we have put the effort into solving that system, it is likely that we want to solve another 2 by 2 system with different coefficients. We could just take our old code and change out the coefficients and the right-hand sides, but there are many places that we need to change and it is likely that we will make a mistake.

What we would like to do is define a function that will solve the system for any coefficients and right-hand side (provided there is a solution). The definition of such a function to solve

$$a_1x + b_1y = c_1$$

$$a_2x + b_2y = c_2,$$

is given here

```
In [2]: def two_by_two_solver(a1,b1,c1,a2,b2,c2, LOUD=False):
            """Calculate the solution of the system
            a1 x + b1 y = c1,
            a2 x + b2 y = c2

            Args:
                a1: x coefficient in first equation (cannot be zero)
                b1: y coefficient in first equation
                c1: right-hand side in first equation
                a2: x coefficient in second equation
                b2: y coefficient in second equation
                c2: right-hand side in second equation
                LOUD: boolean that decides whether to print out the answer

            Returns:
                list containing the solution in the format [x,y]
            """
            #step one, eliminate x from the second equation
            #by multiplying first equation by -a2/a1
            #and then adding it to second equation
            new_b2 = b2 - a2/a1*b1
            new_c2 = c2 - a2/a1*c1
            #solve the new equation 2
            y = new_c2/new_b2
            #plug y into original equation 1
            x = (c1-b1*y)/a1

            if (LOUD):
                print("The solution to:\n",a1,"x +",b1,"y =",c1,"\n",a2,"x +",
                       b2,"y =",c2,"\n is x =",x,"y=",y)
            return [x,y]
```

After we define a function, we can call it to solve for the problem above by typing

```
In [3]:  two_by_two_solver(4.5,3,10.5,1.5,3,7.5,True)
```

This will give the output.

```
The solution to:
 4.5 x + 3 y = 10.5
 1.5 x + 3 y = 7.5
 is x = 1.0 y= 2.0

Out[3]:  [1.0, 2.0]
```

Given our function definition, when we type its name followed by the required input parameters, separated by commas, Python executes the code in the body of the function on those input parameters. Then at the end the function will output the values specified by the `return` statement. In this case the `return` statement creates a list that contains the values of x and y.

Functions are very flexible in both the inputs they can take, and the outputs they can return. Our use of Python to solve engineering problems will be rife with the use of functions, and we will see many examples of this flexibility. Before moving on, we will use this function to demonstrate more features of functions.

BOX 3.1 PYTHON PRINCIPLE

You can define a function using the syntax

```
def function_name([input_variables]):
        [code]
        return [output_variables]
```

In this definition the name of the function is `function_name` and the input variables to the function are entered, separated by commas, in the spot `[input_variables]`. This function returns the variable, or variables, listed in the spot `[output_variables]`.

Because we have defined a function, we can also solve other systems by changing the parameters when the function is called. We can also solve simple systems.

```
In [4]:  two_by_two_solver(1,0,3,0,1,2,True)

The solution to:
 1 x + 0 y = 3
 0 x + 1 y = 2
 is x = 3.0 y= 2.0

Out[4]: [3.0, 2.0]
```

This function cannot solve systems where a_1 is zero because our function divides by a_1. If we wanted to handle this case, we would have to make some changes to the way our functions works. As it stands, if we give the function a system with $a_1 = 0$, we will get an error:

```
In [5]: two_by_two_solver(0,1,2,1,0,3,True)


        ---------------------------------------------------------------------------
    ZeroDivisionError                         Traceback (most recent call last)

        <iPython-input-23-e8717fed1588> in <module>()
    ----> 1 two_by_two_solver(0,1,2,1,0,3,True)


        <iPython-input-19-25039de1b80f> in
        two_by_two_solver(a1, b1, c1, a2, b2, c2, LOUD)
    ---> 18     new_b2 = b2 - a2/a1*b1
         19     new_c2 = c2 - a2/a1*c1
         20     #solve the new equation 2


        ZeroDivisionError: division by zero
```

We will develop a fix for this problem later when we talk about pivoting. Giving an error when dividing by zero is a nice feature of Python for engineering calculations: if we accidentally divide by zero, Python tells the user where it happened, rather than giving a nonsensical answer.

3.1.1 Calling Functions and Default Arguments

In the above examples, we called our function two_by_two_solver by listing out the arguments in the order that it expects them `a1, b1, c1, a2, b2, c2, LOUD`. Nevertheless, Python allows you to call them in any order, as long as you are explicit in what goes where. In the next snippet of code we will specify the left-hand side coefficients first, and then the right-hand sides:

```
In [6]: two_by_two_solver(a1 = 4.5, b1 = 3, a2 = 1.5, b2 = 3,
                          c1 = 10.5, c2 = 7.5, LOUD = True)


The solution to:
 4.5 x + 3 y = 10.5
 1.5 x + 3 y = 7.5
 is x = 1.0 y= 2.0

Out[6]: [1.0, 2.0]
```

In this example we gave the values of the parameters explicitly: we told the function what each parameter was, rather than relying on the order that the parameters was listed.

BOX 3.2 LESSON LEARNED

It is often a good idea to call a function explicitly: that way if you mess up the order of the arguments, it does not matter.

In this example, there is also an example of a default parameter. Notice that in the function definition, the argument `LOUD` has `=False` after it. This indicates that if the function is called without a value for `LOUD`, it assumes the caller does not what the function to "be loud" and print out extra detail. Here we call the function without the `LOUD` parameter

```
In [7]:  two_by_two_solver(a1 = 4.5, b1 = 3, a2 = 1.5,
                          b2 = 3, c1 = 10.5, c2 = 7.5)

Out[7]:  [1.0, 2.0]
```

Notice that it did not print out its spiel about the system. The default behavior of not printing out extra information is common, because if we were going to call this function as part of a larger code many times, we do not want the screen filled with text to the point where it is indecipherable.

3.1.2 Return Values

At the end of the function we have a `return` statement. This tells Python what the function is returning to the caller. In this case we return a list that has the solution for x and y. We can store this in a new variable, or do whatever we like with it.

```
In [8]:  answer = two_by_two_solver(a1 = 4.5, b1 = 3,
                                  a2 = 1.5, b2 = 3,
                                  c1 = 10.5, c2 = 7.5)
          #store in the variable x the first value in the list
          x = answer[0]
          #store in the variable y the first value in the list
          y = answer[1]
          print("The list",answer,"contains",x,"and",y)

The list [1.0, 2.0] contains 1.0 and 2.0
```

We can do even fancier things, if we are so bold. For example, we can grab the first entry in the list returned by the function using square brackets on the end of the function call. We can also assign the two entries in the list to two variables in a single line.

```
In [9]: #just get x
        x = two_by_two_solver(a1 = 4.5, b1 = 3, a2 = 1.5,
                              b2 = 3, c1 = 10.5, c2 = 7.5)[0]
        print("x =",x)

        #assign variables to the output on the fly
        x,y = two_by_two_solver(a1 = 4.5, b1 = 3, a2 = 1.5,
                                b2 = 3, c1 = 10.5, c2 = 7.5)
        print("x =",x,"y =",x)

x = 1.0
x = 1.0 y = 1.0
```

These examples are more advanced, and they are designed to show you some of the neat tricks you can do in Python.

3.2 DOCSTRINGS AND HELP

Our 2 × 2 solver code had a long, and a detailed comment at the beginning of it. This comment is called a docstring and it is meant to tell the user of the function how to call the function and what it does. The user will need to know, for example, what the function will return to prepare to use that information. The user can get the information from the docstring by using the `help` function:

```
In [10]: help(two_by_two_solver)

Help on function two_by_two_solver in module __main__:

two_by_two_solver(a1, b1, c1, a2, b2, c2, LOUD=False)
    Calculate the solution of the system
    a1 x + b1 y = c1,
    a2 x + b2 y = c2
    Args:
        a1: x coefficient in first equation (cannot be zero)
        b1: y coefficient in first equation
        c1: right-hand side in first equation
        a2: x coefficient in second equation
        b2: y coefficient in second equation
        c2: right-hand side in second equation
        LOUD: boolean that decides whether to print out the answer

    Returns:
        list containing the solution in the format [x,y]
```

The point of this long comment is to tell the client (or caller) of the function what the function expects, in terms of arguments, and what the client should expect in terms of what is going to be returned. In this example we can see that we need to provide at least 6 numbers, and possibly an optional boolean.

You may wonder why a docstring is important, when you have the code defining the function right in front of you. The answer is that the user of a function will not always have the definition of the function readily available (perhaps somebody else wrote it). If you want to call that function properly, you can refer to the docstring.

BOX 3.3 PYTHON PRINCIPLE

Docstrings are long comments at the beginning of the body of a function that tells the user what the function needs as input parameters and what the function returns. These useful comments can be obtained by a user of the function by calling

```
help(function_name)
```

where `function_name` is the name of a function.

Let us look at the docstring for some members of the `math` module and the `random` module.

```
In [11]:import math
        help(math.fabs)

Help on built-in function fabs in module math:

fabs(...)
    fabs(x)

    Return the absolute value of the float x.

In [12]:import random
        help(random.uniform)

Help on method uniform in module random:

uniform(a, b) method of random.Random instance
    Get a random number in the range [a, b) or [a, b] depending on rounding.
```

We do not have the source code for these functions in front of us, but if we want to know how to call them, the docstring tells us what to do.

The docstrings for random.uniform and math.fabs are a bit different that the one we used in our function for solving a linear system. The format that we used is derived from the Google coding standards for Python docstrings (https://google-styleguide.googlecode.com/svn/trunk/pyguide.html#Comments).

3.3 SCOPE

The variables that we define in our code store information in the computer's memory. The computer divides the memory that you access into different sections based on scoping rules. Scoping rules are, in essence, a way for the program to separate information in memory and control access to that information. Understanding scoping rules is important when we define functions. Functions have their own scope in memory that is different than the memory used by other parts of a code. That is the memory used by a function is separate from the rest of the program and only knows about the outside world through the parameters it gets passed. In practice, what this means is that any variables used by the function (including those passed to the function) are *completely different* than the variables outside the function. When a function is called, it creates its own copy of the variables that get passed to it.

Here is a simple, but illustrative, example of how a function makes its own copy of the data it gets passed.

```
In [13]: def scope_demonstration(input_variable):
             x = input_variable*3
             return x

         #now call the function after defining some variables
         x = "oui "
         y = "no "

         new_x = scope_demonstration(x)
         new_y = scope_demonstration(y)
         print("x =",x,"\nnew_x =",new_x)
         print("y =",y,"\nnew_y =",new_y)
```

```
x = oui
new_x = oui oui oui
y = no
new_y = no no no
```

Now let us analyze what happened in the code. Before we called the function, we defined a variable x to be the string "oui". Then we called scope_demonstration, passing it x. Notice that even though scope_demonstration defines a variable x as input_variable*3, the value of x that exists outside the function is not changed. This is because when I call scope_demonstration it creates its own memory space and any variable I create in there is different than in the rest of the program, even if the variables have the same name.

In this particular example, the function first copies the value passed to the function into the variable `input_variable`, and then manipulates that copy of the data.

There are many subtleties in scoping rules, but this example outlines the main pitfall for a neophyte programmer. There are extra rules we will need, but these will be covered as we need them.

3.4 RECURSION

The idea behind recursion is that a function can call itself. Recursion enables some neat tricks for the programmer and can lead to very short code for some complicated tasks. In many cases there is often a faster way of doing things than using recursion, but it can be a useful tool for a programmer. Here is an example of computing the factorial of n,

$$n! = 1 \times 2 \times \cdots \times (n-1) \times n,$$

with both a recursive and non-recursive implementation.

```
In [14]: def factorial(n, prev=1):
             if not((n==1) or (n==0)):
                 prev = factorial(n-1,prev)*n
             elif n==0:
                  return 1
             else
                  return prev

         def factorial_no_recursion(n):
             output = 1;
             #can skip 1 because x*1 = 1
             for i in range(2,n+1):
                 output *= i
             return output
         x = 12
         print(x,"! =",factorial(x))
         print(x,"! =",factorial_no_recursion(x))

12 ! = 479001600
12 ! = 479001600
```

We can time the functions to see which is faster. To make the amount of time it takes run large enough to measure well, we will compute the factorials of 0 through 20, 100,000 times. (These are timings on my computer, if you run this example for yourself you may see differences based on your computer hardware and other demands on the system.)

```
In [15]: for times in range(10**5):
             for n in range(21):
                 factorial(n)
```

The time it took to run this was 16 μs. Compare this to

```
In [16]: for times in range(10**5):
             for n in range(21):
                 factorial_no_recursion(n)
```

which takes 6 µs.

The no recursion version, while not as neat, is nearly 50% faster. Even though we're talking microseconds, if my code was going to do this millions of times, the difference would matter. Part of the difference is that every time a function is called, there is an overhead in terms of creating the new memory space, etc. With recursion this extra work has to be done when the function recursively calls itself.

Another drawback to recursion is that it is possible to have too many levels of recursion. The number of levels of recursion, that is, the number of times the function can call itself, is limited to make sure the computer has enough memory to keep track of all the functions that have been called. In this example we demonstrate such an error by having a function call itself about 1000 times:

```
In [17]: x = 1000
         #this won't work and prints ~1000 errors
         #the errors are not repeated here
         print(x,"! =",factorial(x))

In [18]: x = 1000
         #this works
         print(x,"! =",factorial_no_recursion(x))

1000 ! = 40238726007709377354370243392300398571937486421071
46325437999104299385123986290205920442084869694048004799
86101971960586316668729948085589013238296699445909974245
40870737599188236277271887325197795059509952761208749754
62497043601418278094646496291056393887437886487337119181
04582578364784997701247663288983595573543251318532395846
30755574091142624174743493475534286465766116677973966688
20291207379143853719588249808126867838374559731746136085
37953452422158659320192809087829730843139284440328123155
86110369768013573042161687476096758713483120254785893207
67169132448426236131412508780208000261683151027341827977
70478463586817016436502415369139828126481021309276124489
63599287051149649754199093422215668325720808213331861168
11553615836546984046708975602900950537616475847728421889
67964624494516076535340819890138544248798495995331910172
33555566021394503997362807501378376153071277619268490343
52625200015888535147331611702103968175921510907788019393
17811419454525722386554146106289218796022383897147608850
62768629671466746975629112340824392081601537808898939645
18263243671616762179168909779911903754031274622289988005
19544441428201218736174599264295658174662830295557029902
43241531816172104658320367869061172601587835207515162842
25540265170483304226143974286933061690897968482590125458
32716822645806652676995865268227280707578139185817888965
22081643483448259932660433676601769996128318607883861502
79465955131156552036093988180612138558600301435694527224
20634463179746059468257310379008402443243846565724501440
28218852524709351906209
```

```
902313649327349756551395872055965422874977401141334696271
542284586237738753823048386568897646192738381490014076731
044664025989949022222176590433990188601856652648506179970
235619389701786004081188972991831102117122984590164192106
888438712185564612496079872290851929681937238864261483965
738229112312502418664935314397013742853192664987533721894
069428143411852015801412334482801505139969429015348307764
456909907315243327828826986460278986432113908350621709500
259738986355427719674282224875758676575234422020757363056
949882508796892816275384886339690995982628095612145099487
170124451646126037902930912088908694202851064018215439945
715680594187274899809425474217358240106367740459574178516
082923013535808184009699637252423056085590370062427124341
690900415369010593398383577793941097002775347200000000000
000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000
0000000000
```

3.5 MODULES

Often we have to define many functions and do not want to have one giant source file. Also, once we define a function, we do not want to have to copy and paste functions that we have already defined into each file we create. To make functions defined in one Python file available in others, we can import the functions from a file (called a module).

To demonstrate this we define several functions inside a Python file. In particular consider a file called sphere.py. This file defines two functions volume and surface_area that compute the volume and surface area of a sphere. Because the file is called sphere.py we can import those functions using import sphere leaving off the .py part of the name. The code inside of sphere.py is

```
def volume(radius):
    ‘‘’’‘‘compute volume of a sphere
    Args:
    radius: float giving the radius of the sphere

    Returns:
    volume of the sphere as a float
    """
    return 4.0/3.0*math.pi*radius**3

def surface_area(radius):
    ‘‘’’‘‘compute surface area of a sphere
    Args:
    radius: float giving the radius of the sphere

    Returns:
    surface area of the sphere as a float
    """
    return 4.0*math.pi*radius**2
```

After we import the module, we can pass it to the help function to find out what is in the module:

```
In [19]: import sphere
         help(sphere)

Help on module sphere:

NAME
    sphere

FUNCTIONS
    surface_area(radius)
        compute surface area of a sphere

        Args:
        radius: float giving the radius of the sphere

        Returns:
        surface area of the sphere as a float

    volume(radius)
        compute volume of a sphere

        Args:
        radius: float giving the radius of the sphere

        Returns:
        volume of the sphere as a float

FILE
    /Users/mcclarren/sphere.py
```

With the module imported, I can call the functions inside the module by preceding the function name by `sphere.`

```
In [20]: r = 1.0
         print("The volume of a sphere of radius",r,"cm is",
               sphere.volume(r),"cm**3")
         print("The surface area of a sphere of radius",r,"cm is",
               sphere.surface_area(r),"cm**2")

The volume of a sphere of radius 1.0 cm is 4.1887902047863905 cm**3
The surface area of a sphere of radius 1.0 cm is 12.566370614359172 cm**2
```

Modules will be useful for us because our numerical algorithms will build on each other. Using modules will make our source code manageable and eliminate copy/paste errors.

3.6 FILES

It will be useful for us occasionally to use files for input and output from our codes. In this section we will cover some rudimentary uses of files for reading and writing. This discussion is not exhaustive, but will give you the necessary ingredients for file input and output.

It is simple to read in text files in Python. Just about any file that is plain text can be read by Python. One simple way to read in a file is to read it in a line at a time. This can be done by having a `for` loop, loop over the file: each pass through the loop body will read in a new line of the file.

BOX 3.4 PYTHON PRINCIPLE

Using a `for` loop to iterate over a file, the looping variable will contain an entire line of the file each pass through the loop.

In the folder that the following code is saved in, I have created a file called `fifth_republic.txt` that has the names of the presidents of France's fifth republic, one per line. Using a `for` loop, we can read that file and print it to the screen line by line:

```
In [21]: #open fifth_republic.txt for reading ('r')
         file = open('fifth_republic.txt', 'r')
         for line in file:
             print(line)
         file.close()

Charles de Gaulle

Georges Pompidou

Valéry Giscard d'Estaing

François Mitterrand

Jacques Chirac

Nicolas Sarkozy

François Hollande

Emmanuel Macron
```

Notice how the for loop can iterate through each line of the file. You can also read a line at a time:

```
In [22]: #open fifth_republic.txt for reading ('r')
         file = open('fifth_republic.txt', 'r')
         first_line = file.readline()
         second_line = file.readline()
```

```
        print(first_line)
        print(second_line)
        file.close()

Charles de Gaulle

Georges Pompidou
```

It is also possible to write to a file. A straightforward way of doing this is to open a file for writing using the `open` function. With the file open, we then can write to the file much like we use the `print` statement. To end a line we use `\n`. Also, note that if we open a file for writing and it exists, the file will be wiped clean (sometimes called clobbered) upon opening it.

```
In [23]: #open hats.txt to write (clobber if it exists)
         writeFile = open("hats.txt","w")
         hats = ["fedora","trilby","porkpie","tam o'shanter",
                 "Phrygian cap","Beefeaters' hat","sombrero"]
         for hat in hats:
             writeFile.write(hat + "\n") #add the endline
         writeFile.close()

         #now open file and print
         readFile = open("hats.txt","r")
         for line in readFile:
             print(line)

fedora

trilby

porkpie

tam o'shanter

Phrygian cap

Beefeaters' hat

sombrero
```

There are more sophisticated ways to use files for input and output, but these examples will enable the basic functionality we need for file manipulation.

PROBLEMS

Short Exercises

3.1. Write a function for Python that simulates the roll of two standard, six-sided dice. Have the program roll the dice a large number of times and report the fraction of rolls that are each possible combination, 2–12. Compare your numbers with the probability of each possible roll that you calculate by hand.

3.2. What is the value of `x` at the end of the following code snippet:

```
def sillyFunction(input_var):
    x = 1.0
    return input_var

x = 10.0
sillyFunction(x)
```

3.3. Write a function that takes as a parameter the name of the file, and a parameter that gives the number of lines to read from the file. The function should open the file, read the lines and print them to the screen, and then close the file.

Programming Projects

1. Monte Carlo Integration

The exponential integral, $E_n(x)$, is an important function in nuclear engineering and health physics applications. This function is defined as

$$E_n(x) = \int_1^\infty dt\, \frac{e^{-xt}}{t^n}.$$

One way to compute this integral is via a Monte Carlo procedure for a general integral,

$$\int_a^b dy\, f(y) \approx \frac{b-a}{N} \sum_{i=1}^{N} f(y_i),$$

where

$$y_i \sim U[a,b],$$

or in words, y_i is a uniform random number between a and b. For this problem you may use the `random` or `numpy` modules.

3.1. Make the substitution $\mu = 1/t$ in the integral to get an integral with finite bounds.

3.2. Write a Python function to compute the exponential integral. The inputs should be n, x, and N in the notation above. Give estimates for $E_1(1)$ using $N = 1, 10, 100, 1000$, and 10^4.

3.3. Write a Python function that estimates the standard deviation of several estimates of the exponential integral function from the function you wrote in the previous part. The formula for the standard deviation of a quantity g given L samples is

$$\sigma_g = \sqrt{\frac{1}{L-1} \sum_{l=1}^{L} (g_l - \bar{g})^2},$$

where $\bar{g}$ is the mean of the L samples. Using your function and L of at least 10, estimate the standard deviation of the Monte Carlo estimate of $E_1(1)$ using $N = 1, 10, 100, 1000, 10^4$, and 10^5.