CSC110 Supplemental Reading: Week03

Contents

1 Introduction to Functions

The new feature of programming for the week goes by many names: function, procedure, routine, subroutine, subprogram. For our purposes, we'll call them *functions*.

A **function** is a ***named group of Python statements*** that can be called upon and used as needed. Python allows for data to be passed to the function to use when it is called. Python also allows for a function to return data when it finishes executing.

1.1 Defining a Function

To create a function in Python, we use the **def** keyword, which is short for 'define'. The function has a name, just like any other thing in Python. The name must be a legal Python identifier. Then there are parentheses. (More about these later.) This followed by a colon : and a block of statements. Here is a schematic view of a simple function in Python.

```
def name ( ) :
      block_of_statements
```

What is a *block of statements*? A block of statements is a group of statements that Python will handle all together as a single unit. How do we make a block in Python? A block is one or more statements that are grouped together ***by being indented exactly the same amount***. The indentation is sort of indicated in the schematic above, the *block_of_statements* part is indented (goes in more) when compared with the **def**line.

Let's look at a simple example. Here, we're defining a function named, bob.

```
def bob ( ) :
  print ('Hi, my name is bob.')
  print ('Please note how I spell that.')
  print ('Most people misspell my name.')
```

Notice the first line is just what what's above: the keyword **def**, an identifier (in this case, bob), parentheses, and then a colon. Following this, there is a block of statements. These statements are bound together by their uniform indenting, in this case 2 spaces.

If we type this in IDLE and press ENTER, what do we have? Nothing seems to happen.

What happens if we type the following and press ENTER?

```
x = 15
```

Nothing seems to happen, but we have created a variable named 'x' and put the value 15 into the variable.

To use the variable, we just use its name.

```
y = x - 4
```

Now, y has the value 11, that is 15 - 4. No surprises.

In the same way, these four lines create a function named 'bob'.

```
def bob ( ) :
    print ('Hi, my name is bob.')
    print ('Please note how I spell that.')
    print ('Most people misspell my name.')
```

To use the function, we much ***call the function***. We call the function using its name and parentheses. So, we defined the function as "bob( )" and that's how we call the function.

```
bob()
```

This function call executes the three statements that make up the ***body of the function***, that is the block that was part of the definition.

1.2 Parameters & Arguments

Let's consider functions in math for a moment. Here is a typical simple function.

$f(x) = x^2 - 5x + 6$

This function has a ***parameter***, an input value. We can give the function a value to work with, known as an ***argument***: $f(5)$ is different than $f(3)$. $f(5) = 6$, while $f(3) = 0$.

We've already seen this in Python. When we call math.sqrt(100) 100 is the *argument*.

We can do the same thing with functions in Python. We create *parameters* for our functions by naming them within the parentheses.

```
def seuss (myName) :
  print ('I am.' , myName)
  print (myName, 'I am.')
  print ('Do you like green eggs and ham?')
```

This function has a parameter named 'myName'. Within the function, we can use the parameter *myName* like a variable. What is the value of *myName*? It doesn't have a value … yet.

Just like the function definition in math, the parameter x doesn't have a value.

$f(x) = x^2 - 5x + 6$

It gets a value only when you use the function, $f(5)$. Now, x has the value 5.

So, myName only gets a value when we call the function seuss().

```
seuss('Sam')
```

In this function call, we have supplied an argument for the parameter. So, the *parameter* is a *placeholder* that we use when we write a function. When the function is run, the parameter gets the value of the *argument* from the function call.

2 More on Functions

2.1 Terminology

Let's review the terminology used in connection with functions:

- **Function Call** -- the use of a function. For example, the statement

  ```
  num = math.sqrt( ans )
  ```

  contains a call to the sqrt() function.

- **Argument** -- the value that is passed to a function. Consider this example:
  ```
  y = len('alma mater')
  ```

  len() has one parameter, a string. The argument is 'alma mater'. This is the actual value sent to the function.

  An argument can be a more complex expression; the expression is first evaluated and that value is passed to the function. Every time a function is called, different arguments may be used. Here are 3 different calls to math.sqrt() each with different arguments

  ```
  x = 3
  y = 2
  math.sqrt(100) # here the argument is 100. The function call returns the value 10.0
  math.sqrt(3**2) # here the argument is 3 ** 2. That expression has has value 9. The function call returns the value 3.0.
  math.sqrt(x + y + 11) # here the argument is x + y + 11. That expression has value 16. The function call returns the value 4.0
  ```

- **Parameter** -- placeholder variable used to hold data passed to a function. Found in a function definition
- **Function Definition** -- the naming of a group of statements. Uses the keyword **def**.
- **Return Value** -- the value sent back, or returned, by a function. The 3 calls to math.sqrt() above return the values listed.

  When calling a function or method, it is important to be aware of what information is provided in the return value and what data type is provided. Lack of awareness of a return value's data type is a cause of some common errors when writing code.

2.2 Using Arguments

When calling a function that has parameters, it is important to be aware of the following things:

- how many parameters the function needs,
- what the data types of the parameters are,
- what the order of the parameters are

This information is critical in order to use a function properly. When calling a function and passing in arguments:

- the number of arguments must match the number of parameters
- the data types of the arguments must match the data types of the parameters
- the order of the arguments must match the order of the parameters.

(Note, this assumes we're not using keyword arguments. You can read more about them in the text)

2.3 How Return Values Work

A function call that returns a value may be used anywhere an expression is required or as a part of a longer expression. When a function call is completed, the return value effectively replaces the function call in the expression. For example, let's consider the how the following statement would execute:

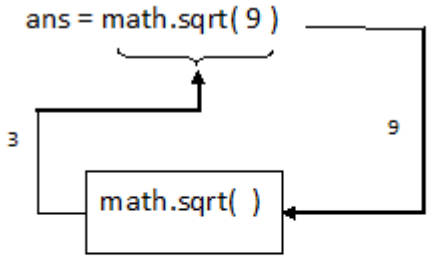
Original statement	`x = math.sqrt(math.ceil(3.8) * 9)`
Order of operations say we must do what's inside the parentheses first. Therefore, the computer calls math.ceil(3.8). This returns a 4.0. So consider the 4.0 replacing the call to math.ceil():	`x = math.sqrt (4.0 * 9)`
Now, the multiplication happens. 36.0 replaces 4.0 * 9:	`x = math.sqrt( 36.0 )`
Then, the computer calls math.sqrt(36) to calculate the square root of 36. math.sqrt() returns a 6.0. So 6 would replace the call to the function math.sqrt()	`x = 6`
Lastly, the computer stores the 6 in the variable x.	

2.4 A Function is a Mini-Program

A function can be thought of as a miniature program. It has the characteristics we have come to associate with a program:

it receives **input** -- through its parameters
it performs **processing** -- using the statements contained in the function
it produces **output** -- the return value

The drawing to the right illustrates this:



The argument 9 is passed to the function. The function performs its calculation, then returns the result 3. The return value 3 is substituted for the expression math.sqrt(9). Lastly, the 3 is stored to the variable ans.

Important points:

- arguments appear in function calls -- they provide data to be used when the function executes
- parameters appear in function definitions -- they are identifiers (names) chosen as variables that will hold the information provided by the arguments. Parameters only exist as long as the function is executing. Once the function finishes, all of its variables (including the parameters) are destroyed.
- A return statement is found in a function definition. It is used to return a value to the function caller. The keyword **return** is followed by any expression that evaluates to the desired return value.
- As soon as a return statement is encountered, the function stops executing (even if other statements follow), and the specified value is returned to the function caller.
- If a function does not have a return statement, then the function ends when execution reaches the bottom of the function.

3 Program Structure

Up until now we have used a pretty basic structure for our programs: input --> processing --> output. This pattern will continue to be valuable to us in the future. However, now that we will be using programmer-defined functions, it will be useful to impose another layer of organization on our programs. Please follow the structure defined here in all programs that make use of programmer-defined functions.

- The function definitions in your program should be listed one after another in sequence. Do not place one function definition inside of another. If you follow all the recommendations listed here, you may place your function definitions in any order, except that the **main** function should be listed either first or last. Choose an order for your function definitions that makes sense to you and helps a reader understand the program.
- Each function definition should be preceded by a comment block that describes what the function does and provides information about any required parameters.
- In general, no global variables should be used in your programs unless specifically authorized in an assignment. This is just good programming practice.
- Global constants may be used if appropriate. The names of global constants should be shown in all upper-case letters with underscores between words, like this: CONTRIBUTION_RATE.
- The names of functions and local variables should follow one of two patterns: all lower case (e.g. calculate_tip, total_to_pay) or "camel case" (e.g. calculateTip, totalToPay). Notice that variable and function names always should begin with a lower-case letter.
- Always use a function named **main** as the entry point for your program. As stated above, the **main** function should be either the first or the last function defined.
- The last statement in the program, and in general the only statement that is not contained inside a function definition, should call the **main** function:

  ```
  main()
  ```

Following these rules will help you avoid confusing errors. Because Python is an interpreted programming language, it must read a function definition before it can process a call to that function. This rule could force you to place the function definitions is a specific order. However, if the call to 'main' is the only statement outside of a function definition, and if you place that call as the last line of your program, you can list your function definitions in any order.

Sample program 5–15 in the text (starting on page 194) demonstrates the use of the guidelines listed above.

4 Tracing function calls

Following a few simple rules will make it easy for you to trace the operation of a program that uses functions, and that ability will help you understand how they work:

- When a function definition is encountered in a program, the statements inside the body of the definition are **not** executed. Rather, the definition is actually declaring the name of the function so it can be called later.
- Each time a function is called:
    1. the **arguments** in the function call are evaluated.
    2. the values of the **arguments** are used to **initialize the parameters** in the function definition,
    3. then flow of control moves to the first line of the function
    4. then the statements in the body of the function are executed
    5. then the function returns to the place where the call came from (the function caller). A value may or may not be returned.

A function call changes the flow of execution. Here is a trace of the execution of a program containing a function. The code is on the left and the trace, or the order that the lines are executed and the different values of the variables, is on the right. Notice that the first line to execute is line 8:

```
line 1 def cubeRoot(n):
line 2   ans = n**(1.0 / 3.0)
line 3   return ans

line 4 def main():
line 5   x = 8
line 6   y = cubeRoot(x)
line 7   z = cubeRoot(35 - x)

line 8 main()
```

Line #	x	y	z	n	ans
8	---	---	---	---	---
4					
5	8				
6...					
1				8	
2					2.0
3				---	---
...6		2.0			
7...					
1				27	
2					3.0
3				---	---
...7			3.0		

Notice the following things about the trace shown above:

- The first line that executes in the trace is line 8. That is because Python does not execute the statements in a function definition until the function is called. At this point, all of the variables are undefined (noted by the ---)
- Line 6 contains a function call. That means that the statement on this line cannot be completed until the function body is executed. Therefore, the entry in the table ('6...') suggests that execution of line 6 has been interrupted temporarily. Control is then transferred to the cubeRoot function header.
- When line 1 is executed as the result of a function call, the effect is to assign a value to the parameter **n**. **n** is initialized with the value of the argument in the function call. We say the argument initializes the parameter in the function definition. Since the argument's value is 8, the parameter's value is 8. This value can now be used by the statements in the body of the function.
- Line 2 executes and stores the number 2.0 in the variable **ans**.

- When line 3 is executed, a return statement is encountered, and this ends the function call. The expression that appears after the keyword return is evaluated, and that value is sent back to the calling statement ('function caller'). Do you remember where this call to cubeRoot() came from? Line 6. Then, on the next line of the trace ('...6'), the value 2.0 (the return value) is assigned to the variable **y.** Notice how the program remembers exactly where to go back.
- Line 7 contains another function call. In this case, the argument is an expression. The expression is evaluated before the function is called. Then that value is used to initialized the parameter, so **n** is set to 27 when line 1 is executed.
- The variables **n** and **ans** have local scope. They only exist while the function is being executed. (We say that the 'lifetime' of these variables is the duration of one execution of the function.) So, when the return statement is encountered and control passes back to the function caller, these variables become undefined. We represent the value 'undefined' in the table by '---'. All variables are undefined before they are first assigned a value, as shown in the first row of data in the table.