CMSC 201 Spring 2017

Lab 06 – Functions

Assignment: Lab 06 – Functions
Due Date: **During discussion**, March 6rd through March 9th
Value: 10 points (8 points during lab, 2 points for Pre Lab quiz)

This week's lab will put into practice the concepts you learned about functions: creating functions, passing parameters, scope, and returning information.

(Having concepts explained in a new and different way can often lead to a better understanding, so make sure to pay attention as your TA explains.)

Part 1A: Review – Scope

Everything in Python has a ***scope*** – the places in the program in which it is accessible. For example, you can create a constant outside of **`main()`**.

```
MAX_VAL = 8

def otherFxn():
    # code in otherFxn() has access to MAX_VAL

def main():
    # code in main() has access to MAX_VAL
main()
```

That constant is now a ***global*** constant, which means it can be accessed by any line of code in the file. So **`main()`** can access it, as well as any other functions that you might write. Remember, for this course you are only allowed to have constants be global – regular variables (that aren't constants) should only be declared inside functions.

Local variables are only accessible to code within their same scope. If a variable is declared in **`main()`**, another function called **`printInfo()`** will not be able to access it. In the same way, a variable in **`printInfo()`** will not be accessible to the code in **`main()`**.

```
def printInfo():
    # this variable can't be accessed by main()
    varForPrintInfo = 5

def main():
    # this variable can't be accessed by printInfo()
    varForMain = 17

main()
```

Part 1B: Review – Functions

A function in Python is a way of compartmentalizing our code: a well-written function does *one* thing, and does it very well. A function allows us to write a piece of code once, and to then use, or "call," the function whenever we want to use that code.

A function has a few key parts:

1. Function name
   - This is how we call the function. It tells Python that we want it to use that function and execute its…
2. Function body
   - This is the code that makes up the function. This is what the function does when called.
3. Formal parameters (optional)
   - A function uses parameters to take in information from the code that called it. This is one of the ways that data is passed from one piece of code to another. A function can have no parameters, one parameter, or it could have a hundred!
4. Return statement (optional)
   - A function can also return data to the code that called it. This is the other way that data is passed around your program. A function can return one or more variables, and a function with no **`return`** statement automatically returns **`None`**.

Let's take a look at some example functions and how they work:

```
def printName(name):
    print("Hello, my name is", name)
```

This function is called **`printName()`**; it takes in one formal parameter (**`name`**) and does not have a **`return`** statement. In order to use the code in this function, we must call the function and pass it an ***actual parameter***. The actual parameter could be a variable, or it could be a literal string (one with quotation marks around it).

Here's the **printName()** function again, but this time we also have a **main()** that calls the function multiple times.

```
def printName(name):
    print("Hello, my name is", name)

def main():
    userName = input("What's your name? ")
    prezUMBC = "Hrabowski"
    printName(userName)
    printName(prezUMBC)
    printName("John Jacob Jingleheimer Schmidt")

main()
```

Note that we have called the function multiple ways: both with variables and with a string literal. Note also that the variable names we passed as actual parameters (**prezUMBC** and **userName**) do not need to match the name of the formal parameter.

Here is the output for the code above:

```
Hello, my name is YOUR_NAME_HERE
Hello, my name is Hrabowski
Hello, my name is John Jacob Jingleheimer Schmidt
```

Part 1C: Review – Returning from a Function

Here is another example of a function.

```
def doubleNum(num):
    num = 2 * num
    return num
```

This function has a **`return`** statement, which means that it returns a value to the code calling it. In order to "catch" or "save" this value, the code calling the function must use the ***assignment operator*** to assign the value returned to a variable for later use. Let's examine what that would look like.

Here is a **`main()`** function that calls the **`doubleNum()`** function two times. What do you think the output of the following code will be?

```
def doubleNum(num):
    num = 2 * num
    return num

def main():
    num = 5
    print("num was first", num)
    doubleNum(num)
    print("num is now...", num)
    num = doubleNum(num)
    print("one last try:", num)

main()
```

Before we look at the output, let's look at the two calls that are made to the **`doubleNum()`** function. In the first one, we pass in **`num`** as our actual parameter – but we don't use the assignment operator in this statement. What will happen to the new value that is returned? Does **`num`** change in **`main()`**?

The second call to the **`doubleNum()`** function is similar, but this time we are using the assignment operator. What does that mean for the value of **`num`** in **`main()`**?

Here is the output:

```
num was first 5
num is now... 5
one last try: 10
```

From the output, we can see that the first call to the function didn't do anything to the value of **`num`** in **`main()`**! We **must** use the assignment operator if we want to save the values returned by our function. Even though **`main()`** and **`doubleNum()`** both have variables named **`num`**, they are in different ***scopes***, and so a change to one does not mean a change to the other.

Part 2: Exercise

In this lab, you'll be downloading a file and completing it by writing three function definitions, and then writing three function calls in **`main()`**.

The program you'll be coding will ask the user to create a list of integers, and will then print out two pieces of information about it: the sum of all of its numbers, and the product of all of its numbers.

Tasks

Starting:

- ☐ Copy the **`given_nums.py`** file from Dr. Gibson's **`pub`** directory
  - ☐ It should have been renamed to be **`nums.py`**
- ☐ Complete the file header comment at the top

Functions:

- ☐ Write the code for **`sumList()`**
- ☐ Write the code for **`prodList()`**
- ☐ Write the **`while`** loop for **`getValidInt()`**

`main():`

- ☐ Call the function **`getValidInt()`** inside the **`while`** loop, and store the returned values in the **`numbers`** list
- ☐ Call the function **`sumList()`**
- ☐ Call the function **`prodList()`**

General:

- ☐ Run and test your code as needed
- ☐ Show your work to your TA

Part 3A: Downloading the File

First, create the **`lab6`** folder using the **`mkdir`** command -- the folder needs to be inside your **`Labs`** folder as well.

Next, copy a file into your **`lab6`** folder using the **`cp`** command.

```
cp /afs/umbc.edu/users/k/k/k38/pub/cs201/given_nums.py nums.py
```

This will copy the files **`given_nums.py`** from Dr. Gibson's public folder into your current folder, and will change the file's name to **`nums.py`** instead.

The first thing you should do in your file is complete the file header comment, filling in your name, section number, email, and the date.

Part 3B: Creating Functions

At this point, if you try to run the file, you will get an error. That is because the file is only partially completed for you.

You will need to update the file to complete the three function definitions and three function calls. If you open the file, you should see comments boxed in by **#** signs – these are where you need to write new code. Read the function header comments to see the details about the three functions.

You should have written code similar to all of these functions during in-class exercises. Both **`sumList()`** and **`prodList()`** are relatively simple pieces of code that you should be able to reproduce. The big difference is that since they are functions, they will need to **return** their result, so that the function that called them has that information.

The code for **`getValidInt()`** should also be familiar to you. About half of the function has already been written for you; the only part you need to write is the sentinel loop that re-prompts the user.

You will also need to write calls to each of these functions. The places where these calls need to happen in **`main()`** are indicated for you. You shouldn't need to write any other code.

(PROTIP: it is highlihy recommended ot test as you go)

(See the next page for sample output.)

Here is some sample output of the completed program, with user input in **blue**. (Yours does not have to match this word for word, but it should be similar.)

```
bash-4.1$ python nums.py
Welcome to the number program!
We'll be creating a list of 8 integers today!
Enter a number between -1000 and 1000 (inclusive): 9999
That number is not allowed.  Please try again!
Enter a number between -1000 and 1000 (inclusive): -9999
That number is not allowed.  Please try again!
Enter a number between -1000 and 1000 (inclusive): 999
Enter a number between -1000 and 1000 (inclusive): 21
Enter a number between -1000 and 1000 (inclusive): 86
Enter a number between -1000 and 1000 (inclusive): -4
Enter a number between -1000 and 1000 (inclusive): 201
Enter a number between -1000 and 1000 (inclusive): 762
Enter a number between -1000 and 1000 (inclusive): 5
Enter a number between -1000 and 1000 (inclusive): 1
The sum of the list is 2071
The product of the list is -5526679228560
Thank you for using the number program!  Come again!


bash-4.1$ python nums.py
Welcome to the number program!
We'll be creating a list of 8 integers today!
Enter a number between -1000 and 1000 (inclusive): 0
Enter a number between -1000 and 1000 (inclusive): 1
Enter a number between -1000 and 1000 (inclusive): 2
Enter a number between -1000 and 1000 (inclusive): 3
Enter a number between -1000 and 1000 (inclusive): 4
Enter a number between -1000 and 1000 (inclusive): 5
Enter a number between -1000 and 1000 (inclusive): 6
Enter a number between -1000 and 1000 (inclusive): 7
The sum of the list is 28
The product of the list is 0
Thank you for using the number program!  Come again!
```

Part 4: Completing Your Lab

Since this is an in-person lab, you do not need to use the **`submit`** command to complete your lab. Instead, raise your hand to let your TA know that you are finished.

They will come over and check your work – they may ask you to run your program for them, and they may also want to see your code. Once they've checked your work, they'll give you a score for the lab, and you are free to leave.

Tasks

Starting:

- ☐ Copy the **`given_nums.py`** file from Dr. Gibson's **`pub`** directory
  - ☐ It should have been renamed to be **`nums.py`**
- ☐ Complete the file header comment at the top

Functions:

- ☐ Write the code for **`sumList()`**
- ☐ Write the code for **`prodList()`**
- ☐ Write the **`while`** loop for **`getValidInt()`**

`main():`

- ☐ Call the function **`getValidInt()`** inside the **`while`** loop, and store the returned values in the **`numbers`** list
- ☐ Call the function **`sumList()`**
- ☐ Call the function **`prodList()`**

General:

- ☐ Run and test your code as needed
- ☐ Show your work to your TA

IMPORTANT: If you leave the lab without the TA checking your work, you will receive a **zero** for this week's lab. Make sure you have been given a grade before you leave!