CMSC 201 Fall 2016
Lab 08 – Functions

Assignment: Lab 08 – Functions
Due Date: During discussion, October 24th through 27th
Value: 10 points

Part 1: 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 main():
    # etc
```

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 2: 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; 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():
    prezUMBC = "Hrabowski"
    prezUSA  = "Obama"
    printName(prezUMBC)
    printName(prezUSA)
    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 **prezUSA**) do not need to match the name of the formal parameter.

Here is the output for the code above:

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

Part 3: Returning from Functions

Here is another example of a function.

```
def attemptChange(num):
    num = 7 * 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 **`attemptChange()`** function two times. What do you think the output of the following code will be?

```
def attemptChange(num):
    num = 7 * num
    return num

def main():
    num = 5
    print("num was first", num)
    attemptChange(num)
    print("num is now...", num)
    num = attemptChange(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 **`attemptChange()`** 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 **`attemptChange()`** 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: 35
```

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 **`attemptChange()`** 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 4A: Writing Your Program

After logging into GL, navigate to the **`Labs`** folder inside your **`201`** folder. Create a folder there called **`lab8`**, and go inside the newly created **`lab8`** directory.

```
linux2[1]% cd 201/Labs
linux2[2]% pwd
/afs/umbc.edu/users/k/k/k38/home/201/Labs
linux2[3]% mkdir lab8
linux2[4]% cd lab8
linux2[5]% pwd
/afs/umbc.edu/users/k/k/k38/home/201/Labs/lab8
linux2[6]%
```

Once you're in the folder, you will need to copy the starter file from my public directory. Type (all on one line – don't forget the period at the end!):

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

To open the file for editing, type

```
emacs nums.py
```

and hit enter.

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

Now you can start writing your code for the lab, following the instructions in Part 4B.

Part 4B: 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 the code for two of these functions during in class exercises or for a previous homework. Both **`evenOrOdd()`** and **`posOrNeg()`** 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 reprompts 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.

Sample output is on the next page;

Sample output for this lab, with the user input in blue.

```
bash-4.1$ python nums.py
Welcome to the number program!
Please enter a number between -1000000 and 1000000
(inclusive): -17
The number -17 is negative
The number -17 is odd
Thank you for using the number program!  Come again!

bash-4.1$ python nums.py
Welcome to the number program!
Please enter a number between -1000000 and 1000000
(inclusive): 123456789
Please enter a number between -1000000 and 1000000
(inclusive): -123456789
Please enter a number between -1000000 and 1000000
(inclusive): 123456
The number 123456 is positive
The number 123456 is even
Thank you for using the number program!  Come again!

bash-4.1$ python nums.py
Welcome to the number program!
Please enter a number between -1000000 and 1000000
(inclusive): 0
The number 0 is zero
The number 0 is even
Thank you for using the number program!  Come again!
```

Part 5: Completing Your Lab

To test your program, make sure that you've enabled Python 3, then run **`nums.py`**. Try a few different inputs to see how well your program works.

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.

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!