Simon Fraser University
School of Computing Science

Lab Week 9

(No submission required for this lab.)

Objective

In this week's lab, our objectives are:

1. Understanding function calls, arguments and parameters, variable scope, etc…
2. Practice creating user-defined functions, on their own as well as part of a whole program.
3. Examine the flow of execution when a Python interprets a recursive function.

Notes

1. For this lab, you are free to work on our own or with a partner.
2. This is a long lab. If you do not have enough time to finish these lab exercises within our lab session, no problem! You can finish the lab on our own outside the lab session either
   a. by coming back to CSIL and using a CSIL workstation in a lab room where there is no scheduled lab taking place, or
   b. by using our own computer at home or on campus or elsewhere.
3. Let's make sure we show our work to one of the TA's in the lab to ensure we understand the concepts (see Objectives above) exercised in this lab.

Lab Prep

Once we have logged into a CSIL Windows workstation, let's access our **U: drive/CMPT120** and create a directory called **Lab6**. Let's store any files we create as part of this lab into this directory.

In all the exercises of this lab, feel free to use the **visualizer** (see its link on the Home page of our course web, week 8) to view the execution flow as it executes our functions.

Exercise 1 – Understanding functions

1. Let's have a look at the Python program called **AreaCalculator_4.py** posted under the Code and Examples column of this week's lecture on the course web site.

   Either on a piece of paper or using a text editor, make a table with 7 columns.

   Start of table

   a) In the first leftmost column, list all the functions, one function per row. Add a final row for the Main part of the program.

   b) In the 2nd column, write the main purpose of the function listed on the left.

   c) In the 3rd column, write the name of the parameters this function has, if any. Note that the Main part of the program does not have parameters.

   d) In the 4th column, write the name of the variables local to this function, if there are any and for each variable, state its scope, i.e., whether the scope of a variable is a particular function (if so, which section of the function) or the whole program (if so, which section of the program).

   e) In the 5th column, write the value the function returns, if any.

   f) In the 6th column, write the number of times this function is called. Be careful, it is not always the case that functions are called from the Main part of a program. Sometimes, functions call other functions.

   g) In last column, write the name of the function(s) this function calls, if any.

   End of table

Question 1: Using this **AreaCalculator_4.py** program as an example, can you point out a function that was created using the guideline of generalization?

Question 2: Using this **AreaCalculator_4.py** program as an example, can you point out a function that was created using refactoring and encapsulation?

Question 3: What are the three aspects of function interface design we need to address (make decisions about) when we create our own functions?

2. Finally, finish the AreaCalculator program by implementing the steps of the algorithm that were not yet implemented in **AreaCalculator_4.py** (see the comments at the end of this program).

Exercise 2 – User-defined functions

Whenever we are asked to implement a function, we also need to implement a "main part of the program" to test it. For an example of such "main part of the program" that tests a function, please, see the Python program **reverseString.py** posted on our course web site for Lecture 25.

1. Do Exercise 5.2 in Section 5.14 – Exercises from our online textbook.

2. Do Exercise 5.3 in Section 5.14 - Exercises from our online textbook.

   Once we have written the function described in part 1) of Exercise 5.3, modify that function such that instead of printing Yes or No, **it returns True or False**, depending on whether we can or cannot form a triangle from sticks with the given lengths. For example, if we call `is_triangle(3, 4, 5)`, this function call should return `True` and if we call `is_triangle(12, 1, 2)` , this function call should return `False`since 1+2 < 12.

3. Do the exercise at the end of Section 6.1 from our online textbook. It starts like this: "As an exercise, write a compare function …".

4. Do the exercise at the end of Section 6.4 from our online textbook. It starts like this: “As an exercise, write a function `is_between(x,y,z)` …”.

5. Do Exercise 3.3 (1. and 2.) in Section 3.14 - Exercises from our online textbook.

6. Write a function that capitalizes every third letter in a word (starting at the first letter of the word). The word is passed to the function as a parameter. The function returns the newly created string. For example, if the word passed as a parameter is **supercalifragilisticexpialidocious**, then the function returns **SupErcAliFraGilIstIceXpiAliDocIouS**.

   Make sure our function guards itself from empty strings and strings containing other characters than letters.

9. Read the section 10.13 on debugging. Do Exercise 10.2 to 10.7 in Section 10.15 - Exercises from our online textbook.

Exercise 3 – Executing recursive functions

1. Copy and paste the Python program below:

```
# factorial.py
def factorial( n ):
  if n == 1 or n == 0:      # 1! = 1 or 0! = 1
    result = 1
  else :                    # n! = n * (n-1)!
    result = n * factorial(n-1)
  return result



# Main
# Test Case 1
testData = 1
result = factorial(testData)
print("factorial({}) = {}".format(testData, result))
```

```
# Test Case 2
testData = 0
result = factorial(testData)
print("factorial({}) = {}".format(testData, result))


# Test Case 3
testData = 5
result = factorial(testData)
print("factorial({}) = {}".format(testData, result))
```

in the visualizer and step through it execution. Pay particular attention to the number of stack frames called *factorial*, their content (value of parameters, of variables and the returned value) and where the execution flow returns to when we are “recursing back”, i.e., returning to the caller (either factorial or the main part of the program) from a call to factorial.

Have fun!