Computer Science 1 — CSci 1100
Lab 3 — Functions
Fall Semester 2016

Lab Overview

The main goal of this lab is to learn to define and use functions in our programs, learn good program structure and also build some skills in reading error messages.

A good program is easy to read and debug. Your program should be well-structured. Similarly, your variables and functions should have meaningful names. This will help you and others debug code.

First, in structuring your code, follow the guidelines we presented during lecture:

1. A general comment describing the program.
2. All import statements.
3. All function definitions.
4. The main body of your program.

We will work on this throughout this lab. Look at the code we provided you for examples.

Second, we would like to see comments that explain the main purpose of the program and also tricky steps if there are any. As Python programs are short and generally very easy to understand, you do not need a lot of comments if you have well-structured code and meaningful variable names, but you will need some, and it always helps to define your functions and the meaning of the function parameters.

The final thing is debugging code. Often programmers see debugging as a chore. In fact, debugging is really 80% of programming. So, you must learn to solve problems, read error messages, and figure out how to relate error messages to your code. Debugging is a little like solving a puzzle or being a detective. You need to find clues and think through what could be causing the problem. The first checkpoint will test this.

Checkpoint 1: Using existing functions

First, we will experiment with using functions and debugging code. Create a new directory in your dropbox for Lab 3, and copy from Piazza the program called `lab3_check1.py`. Run this code and see that it gives an error message.

Let's look closely what the code does: it creates a function that takes four values: `x1,y1,x2,y2`, and returns a floating point value that represents the Euclidean distance between two points with Cartesian coordinates `(x1,y1)` and `(x2,y2)`. The program starts from an initial point and then reads the next `(x,y)` coordinates of an object. It computes the distance between these two and outputs the result.

The code contains many bugs. Often, the program aborts after the first error and you need to fix it and rerun the program. So, we would like you to go through this process with this program. In fact, even if you can see a bug in the program, we recommend that you do not fix it until you see the error it produces first.

Each time you run the program, read the error message. Sometimes it is easy to interpret and sometimes not. But very often it will tell you the line number and the exact location. Learn to read these messages and interpret them. You will learn a lot from fixing mistakes, so do not let others tell you what each error means. If you are having trouble figuring them out, read the relevant course notes and think. If you figure out it yourself, you will learn so much more.

Please take the time to figure out all the bugs. Also, format the output nicely so that it looks like this:

```
The next x value ==> 12
The next y value ==> 12
The line has moved from (10,10) to (12,12)
Total length traveled is: 2.83
```

After you fixed al the bugs, show the corrected program to a TA or a mentor. They will ask you about different errors and discuss strategies for figuring out what they mean.

To complete Checkpoint 1: Show the TA the working version of the program. Be prepared to discuss with them debugging methods and ask for advice.

Checkpoint 2: Restructuring code

In this section, we will restructure code with the help of a function. First start by downloading the program called `lab3_check2.py` and save it in your lab3 folder. Create a copy of it `lab3_check2v2.py` by saving it a second time with the new name. You will be modifying this copy and comparing its output to the original.

Run the original program and make sure you understand what it does. You can see that this program contains repetitive code. If you found a bug in one computation, you would have to fix it in 4 places. Instead, you can put all this computation into a function improving both readibility and maintainability.

Write a function that takes as input all the necessary data for the repeated code, does the computation and prints the results. Your function definition must come before any other code in the program. What should the arguments for this function be?

Now, rewrite the rest of the code by simply calling this function four times. Check and make sure it provides the same output as the original program.

To complete Checkpoint 2: Show the TA your refactored program. Make sure that your program follows the required structure: function first, followed by function calls.

Checkpoint 3: Prey and Predator Population

Now that we have experimented with functions, you will write a full program that defines functions and then uses them.

Suppose you are trying to understand how the population of two types of animals will evolve over time. You have bunnies and foxes. Unfortunately, foxes like to eat bunnies. So, even though the bunny population grows every year when baby bunnies are born, it also goes down because some of them are eaten. Would the fox population keep increasing in size until there are no bunnies left? Not so. The fox population is directly linked to the bunny population, their main food source.

What will happen over time if you start with a given population of bunnies and foxes? Who will win out? Will they balance each other? This is what you will investigate in this part.

Suppose `bpop, fpop` are the population of bunnies and foxes currently. Then, next year's population of bunnies (`bpop_next`) and foxes (`fpop_next`) are given by:

```
bpop_next = (10*bpop)/(1+0.1*bpop) - 0.05*bpop*fpop
fpop_next = 0.4 * fpop + 0.02 * fpop * bpop
```

What will be the population of bunnies and foxes next year? Your program should first read the population of bunnies and foxes using `input`, and then compute and print the next years' population for both based on this current value. Its output initially should look like this:

```
Number of bunnies ==> 100
100
Number of foxes ==> 10
10
Year 1: 100 10
Year 2: 40 24
```

Let's think about how to solve this problem. Start by creating a new file in the Wing IDE to work with. You will need variables to store the current population of bunnies and foxes. You need one function for computing the population of bunnies for the next year and another function for the population of foxes. Now, call these functions once to find the population of both animals in the next year and print.

Remember two crucial things:

- Population is an integer, not a float. You never see half bunnies running around.
- The second one is a bit tricky. The population of animals cannot be negative (something worse than extinct?). Don't listen to people around you who are telling you to use an if statement. **You are forbidden to use an if statement.** Think about how to use the `max()` function to make sure that your function does not return a negative number.

Now, for the final challenge, use Year 2 values to find Year 3 populations of bunnies and foxes. Do not create new variables, just use the same variables you currently have. Once you figure this out, you can repeat this part 2 more times to find the population in Year 5. When completed, your program should look like this:

```
Number of bunnies ==> 100
100
Number of foxes ==> 10
10
Year 1: 100 10
Year 2: 40 24
Year 3: 32 28
Year 4: 31 29
Year 5: 30 29
```

In the next few weeks, we will see that we can use loops to make this calculation much simpler. Loops will allow you avoid repeatedly typing (or copying-and-pasting) the same code and make

your programs shorter, clearer, more general, and less likely to contain errors. For now, let's use this lab to learn how to use the same variables again and again in computations.

To complete Checkpoint 3: show a TA or a mentor the completed solution and illustrate its use with the values given above.

Make sure your program follows the structure we outlined earlier in this lab: function definitions first, then your variables `bpop, fpop` are assigned their initial value, and finally the code to compute and print the year-by-year population.