Python for Rapid Engineering Solutions
Homework #1

Feel free to discuss the problems with your classmates. While you may compare answers and help each other with coding issues, you must write your own code. Make sure you consult the coding guidelines in Module 0 as your grade will be affected if you don't follow them!
Remember to use comments!
Use constants where appropriate!
Constants are to be all upper case!
Variables are to be all lower case!

You will turn in THREE separate Python scripts. When submitting to Canvas, submit one at a time, choosing submit an additional file to add the second and third files. Note that you must follow the file and function names when specified.

Part 1 (40 points)

Implement a quadratic equation solver. It takes 3 inputs: a, b, and c where these are the coefficents of the equation $ax^2+bx+c=0$. Your script should return the values of x which make the equation true. Recall the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

But this can present a problem if $b^2 - 4ac < 0$. So, you'll need to determine if this is the case. If it is, then use the sqrt function from a package called cmath. If it is not, that is, if the value is equal to, or greater than 0, then use the sqrt function from numpy. If the value is really close to 0, then consider it to be zero. It is up to you to decide what "really close" means. If the value is 0, then you have a double root and should indicate that in your answer.

Where do a, b, and c come from? Ask the user to enter them. For this exercise, you can assume that a, b, and c will always be integer values.

Here are some example executions of the program.

> python hw1_quad.py
Input coefficient a: **1**
Input coefficient b: **2**
Input coefficient c: **1**

Double root: -1.0

> python hw1_quad.py
Input coefficient a: **2**
Input coefficient b: **-10**
Input coefficient c: **12**

Root 1: 3.0
Root 2: 2.0

> python hw1_quad.py
Input coefficient a: **1**
Input coefficient b: **1**
Input coefficient c: **4**

Root 1: (-0.5+1.9364916731037085j)
Root 2: (-0.5-1.9364916731037085j)

Your script should be called hw1_quad.py.

Part 2 (30 points)

The objective of this homework set is to get you started writing Python code. Newman, exercise 2.12, on page 83 in my edition. For output, ONLY print the list of primes. That is, your output code should look something like:

print(my_prime_list)

No, you don't have to use that variable name, but do use one that has some meaning. The only output from your program should be the list, and it'll look something like:

[2, 3, 5, ...]

Where the ... represents all the rest of the list of primes that you've found.

Your program MUST be called hw1_12.py and only the program need be turned in for this part of HW1.

Part 3 (30 points)

The objective of this homework set is to get you started writing Python code. First, check out the factorial program in Newman, in exercise 2.13 (page 83 in my edition). (We're just using the factorial portion of the problem – do **NOT** do the other parts of the problem!)

1. Include the factorial function shown in the first part of the problem so you confirm that you've got the recursion thing down. Make sure you have it running. Once you have it running, comment it out, but leave it in the file. That way, it'll be there for reference, but won't execute when your run the square root calculator below.

2. There are a number of neat ways to calculate the square root of a number without the square root function. An ancient method known to the Babylonians is this recursive method:

Let N be the number whose square root is to be computed. Guess that some integer n_0 is the square root of N. Calculate the following:

$$n_1 = \frac{n_0 + \frac{N}{n_0}}{2}$$

and n_1 will be a better approximation. You can keep iterating by having a function that computes:

$$n_{i+1} = \frac{n_i + \frac{N}{n_i}}{2}$$

Until $n_{i+1} - n_i < \epsilon$, where ϵ is some suitably small number.

Implement your own function, called my_sqrt, that calls itself recursively until the square root is computed. Print the answer to the precision of your tolerance. That is, if you have $\epsilon = 0.01$, then only print 2 places past the decimal point.

Request the number whose root is to be computed and an initial guess. Your script will report the square root. You may assume that the number and the guess will be positive integers, where the guess is smaller than the number.

NOTE: You may NOT use a Python square root function, such as the one from numpy, in your code. You may, however, use the np.abs() function to compute the absolute value in order to determine if you are within the tolerance you've picked.

Here are some example executions assuming a tolerance of 2 places past the decimal:

> python hw1_root.py
Enter a number whose square root is desired: **16**
Enter an initial guess: **3**
The square root of 16 is 4.0

> python hw1_root.py
Enter a number whose square root is desired: **15**
Enter an initial guess: **6**
The square root of 15 is 3.87

> python hw1_root.py
Enter a number whose square root is desired: **937**
Enter an initial guess: 2

The square root of 937 is 30.61

Your program must be called hw1_root.py.