COMP 1510 Programming Methods Lab 09

Christopher Thompson
chris_thompson@bcit.ca

BCIT CST — March 2022

1 Welcome!

Welcome to your ninth COMP 1510 lab.

For lab nine, I challenge you to be creative and solve this brainteaser for me. Let's begin!

2 Submission Requirements

1. This lab is due no later than **Friday April 1st at or before 12:00:00 (that's noon on Friday!)**
2. Late submissions will not be accepted for any reason.
3. **This is a collaborative lab. I strongly encourage you to share ideas and concepts. You have permission to show each other your code, but you must submit your own work!**

3 Grading

Figure 1: This lab is graded out of 5

This lab will be marked out of 5. For full marks this week, you must:

1. (3.0 points) Correctly implement the coding requirements in this lab.
2. (1.0 point) Correctly format and comment your code.
3. (1.0 point) Correctly generate unit tests for your code.
4. (-1.0 point penalty) I will withhold one mark if your commit messages are not clear and specific. Tell me EXACTLY what you did. You must:
   (a) Start your commit message with a verb in title case in imperative tense (just like docstrings). Implement/Test/Debug/Add/Remove/Rework/Update/Polish/Write/Refactor/Change/Move...
   (b) Remove unnecessary punctuation – do not end your message with a period
   (c) Limit the first line of the commit message to 50 characters
   (d) Sometimes you may want to write more. This is rare, but it happens. Think of docstrings. We try to describe a function in a single line, but sometimes we need more. If you have to do this, leave a blank line after your 50-char commit message, and then explain what further change(s) you have made and why you made it/them.
   (e) Do not assume the reviewer understands what the original problem was, ensure your commit message is self-explanatory
   (f) Do not assume your code is self-explanatory.

4 Set up

1. Visit this URL to copy the template I created to your personal GitHub account, and then clone your repository to your laptop:

   `https://classroom.github.com/a/kcXNDnnb`

2. Populate the README.md with your information. Commit and push your change.

5 Style Requirements

1. You must observe the code style warnings produced by PyCharm. **You must style your code so that it does not generate any warnings.**

   When code is underlined in red, PyCharm is telling us there is an error. Mouse over the error to (hopefully!) learn more.

   When code is underlined in a different colour, we are being warned about something. We can click the warning triangle in the upper right hand corner of the editor window, or we can mouse over the warning to learn more.

2. **You must comment each function you implement with correctly formatted docstrings.** Include informative doctests **where necessary and possible** to demonstrate to the user how the function is meant to be used, and what the expected reaction will be when input is provided.

3. In Python, functions must be atomic. They must only do one thing. If a function does more than one logical thing, break it down into two or more functions that work together. Remember that helper functions may help more than one function, so we prefer to use generic names as much as possible. Don't hard code values, either. Make your functions as flexible as possible.

4. Don't create functions that just wrap around and rename an existing function, i.e., don't create a divide function that just divides two numbers because we already have an operator that does that called `/` .

5. **Ensure that the docstring for each function you write has the following components (in this order)**:

   (a) Short one-sentence description that begins with a verb in imperative tense and ends with a period.

   (b) One blank line

   (c) Additional comments if the single line description is not sufficient (this is a good place to describe in detail how the function is meant to be used)

   (d) One blank line if additional comments were added

   (e) PARAM statement for each parameter which describes what the user should pass to the function

   (f) PRECONDITION statement for each precondition which the user promises to meet before using the function

   (g) POSTCONDITION statement for each postcondition which the function promises to meet if the precondition is met

   (h) RETURN statement which describes what will be returned from the function if the preconditions are met

   (i) One blank line

   (j) And finally, the doctests. Here is an example:

```
def my_factorial(number):
    """
    Calculate factorial.

    A simple function that demonstrates good comment construction.

    :param number: a positive integer
    :precondition: number must be a positive integer equal to or
                   greater than zero
    :postcondition: calculates the correct factorial
    :return: factorial of number

    >>> my_factorial(0)
    1
    >>> my_factorial(1)
    1
    >>> my_factorial(5)
    120
    """
    if number == 0:
        return 1
    else:
        return number * factorial(number - 1)
```

6 Requirements

Please complete the following:

1. Remember Eratosthenes and his sieve? I'd like to introduce you to a new algorithm developed by Heron of Alexandria `https://en.wikipedia.org/wiki/Hero_of_Alexandria`.
2. Heron's algorithm helps us find the square root of a number without using a calculator or any existing functions in the Python libraries. The square root of 9 is 3. The square root of 100 is 10. But what's the square root of 2? Or 17? You're about to learn one of the neatest tricks in the math books.
3. The algorithm works like this:
   (a) Suppose I want to find the square root of 42. I'll start by guessing the number itself. Let's start by guessing that the square root of 42 is 42. Note that it doesn't have to be 42. I can start with 100. I can start with 5. This algorithm will still work! But there is an elegance and a mathematical beauty to starting with the original number. The code is simpler, too.
   (b) I start by calculating the average of 42 and 42 / 42. The average of 42 and 42 / 42 = the average of 42 and 1 = 43 / 2 = 21.5
   (c) I'm going to perform the same operation. I'll calculate the average of 21.5 and 42 / 21.5, which is 11.726744186046512 or something like that.
   (d) Let's do it again. Let's calculate the average of 11.726744186046512 and 42 / 11.726744186046512. This give us 7.654150476761481 or so.
   (e) Let's do it again. Let's calculate the average of 7.654150476761481 and 42 / 7.654150476761481. This give us 6.570684743283658 or so.
   (f) Go look up the square root of 42. I dare you. Look how close we are already!
   (g) We usually going until the difference between the old value and the new value is negligible. Voila! A very, very good approximation of the square root!
   (h) Pause for applause.
4. Create a file called exceptions.py. Inside the file implement a function called heron. This function must accept a number and return the square root, which will be a floating point number.

5. Once you have this working, try calculating the square root of a negative number: heron(-1). What happens? Python will probably raise some sort of Exception.

6. Here's the catch. In Python we do NOT check the data type of the incoming argument. That's not Pythonic. So we don't ensure the incoming value is positive. We can use a precondition, but we may also choose to use exception handling to recover from invalid arguments.

7. Modify your function code to catch the Exception before it propagates to the top of the call stack and causes our program to crash. Use a try-except control structure and make sure you only put a single line of code in the try block. When you catch the Exception, I think it makes sense to print a warning message and return -1. It's impossible for -1 to be the square root of a non-complex number, so this is a safe return value that says, "No, I can't do that, so here's some other number that means I can't do that!"

8. One more thing. I'd also like you to implement the following function:

```
def find_an_even(input_list):
    """
    Return the first even number in input_list.

    param input_list: a list of integers
    precondition: input_list must be a list of integer
    postcondition: return the first even number in input_list
    raises ValueError: if input_list does not contain an even number
    return: first even number in input_list
    """
```

9. Don't forget to comment your two functions, and create four or five unit tests for each one. Ensure you prove that exceptions are raised when they are supposed to be raised.

10. That's it!

Good luck and have fun!