COMP 1510 Programming Methods Lab 08

Christopher Thompson
chris_thompson@bcit.ca

BCIT CST — March 2022

1 Welcome!

Welcome to your eighth COMP 1510 lab.

For lab eight, 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 March 11th 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/DXrz4yc7`

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. Edsgar Dijkstra, who I introduced during our first history lesson, developed a fun problem he called the **Dutch National Flag problem**. Given a list of strings, each of which is either 'red', 'white', or 'blue' (each can be repeated in the list), rearrange the list so that the strings are in the order of the Dutch national flag: all the 'red' strings first, then all the 'white' strings, then all the 'blue' strings.

   (a) **Implement a function called dijkstra in a file called dijkstra.py.** Create a flowchart in a file called dijkstra.PDF.

   (b) Your function much accept a non-empty list that contains randomly shuffled strings 'red', 'white', and 'blue'. Your function is not responsible for how it responds to anything else.

   (c) Sort the strings in the original list. Your function does not return or print anything. It sorts the elements in the list passed to the function. Your code must be efficient.

   (d) Remember there is no return value. Your function can be used and tested like this:

   ```
   dutch = ['white', 'blue', 'blue', 'red', 'white', 'red', 'white']
   dijkstra(dutch)
   print(dutch)
   ['red', 'red', 'white', 'white', 'white', 'blue', 'blue']
   ```

2. **Implement a suite of unit tests in the accompanying test_dijkstra/py file** that prove your function will work when it is provided arguments that pass the precondition(s).

3. **Implement the sample usage above in a simple doctest in your dijkstra function's doctest.**

That's it! There are solutions to the Dutch Flag Problem all over the internet. I've seen many of them, and I encourage you to search for and evaluate them too. There are lots of sorting algorithms out there, but I encourage you to be creative. You can implement a sort if you like, sure, but there are simple, "Pythonic" solutions too!

Good luck, and see you soon!