COMP 1510 Programming Methods Lab 05

Christopher Thompson
`cthompson98@bcit.ca`

BCIT CST — October 2019

Welcome!

Welcome to your fifth COMP 1510 lab. In today's lab we will start working with lists.

You will have two hours to work on the lab on Tuesday. On Thursday in lab, I will spend two minutes with each student in a lightning-round face to face marking meeting. After a short break, we'll spend some time reviewing some important topics from the lab and lectures, and finish with our weekly quiz.

Take your time. Read each step. Don't skip anything. Good luck, and have fun!

1 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 points) Correctly implement the coding requirements in this lab.
2. (1 point) Correctly format and comment your code.
3. (1 point) Correctly generate unit tests for your code.

2 Project Setup

Please complete the following:

1. For your fifth lab, all files must go into the Lab05 folder.
2. After you complete each task, commit and push your change to version control. In order to earn full marks, you must commit and push after each function is complete. Your commit message must be a short message to me that describes what you completed for this commit, i.e., "deconstructed base_conversion", or "debugged function_name", etc.
3. Since I am already a collaborator I will pull your future work automatically.

3 Requirements

1. Implement the following functions. The names you must use are below. Do not modify the names I have provided.
2. Include correctly formatted docstrings that include doctests (where possible) for each function. These doctests must explain to the user how the function is meant to be used, and what the expected output will be when input is provided.
3. Each function must also be tested in a separate unit test file by a separate collection of unit tests. Ensure your test cases and functions have descriptive, meaningful names, and ensure that every possible valid input is tested for each function.
4. 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
   (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
   (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
    :postcondition: calculates the correct factorial
    :return: factorial of number

    >>> my_factorial(0)
    1
    >>> my_factorial(1)
    1
    >>> my_factorial(5)
    120
    """
    return math.factorial(number)
```

5. Create a function called roll_die. We will create a function without any limitations of number of rolls or number of sides:
   (a) Die is the singular form of the word dice. (Suppose I have two dice in my hand. If I remove one, I now only have one die.)
   (b) This function accepts two parameters, both integers. The first is called number_of_rolls, and the second is called number_of_sides.

   (c) This function must simulate rolling a die of the specified size the specified number of times. That is, if I invoke the function like this: roll_die(3, 6), then I mean to roll a six-sided die three times, for a random total that will be anywhere from 3 to 18 inclusive [3, 18].
   (d) This function must return that random total. Note that both parameters must be positive integers or the function must return 0. It's impossible to roll a zero with any known die in this universe, so this is a good return value that signifies something went wrong.
   (e) When you test this function, since this function will return a value that is random, we cannot test it with a doctest. We can only test this function using unit tests. I suggest you write two tests where each contains an assertion that the value generated is greater than (or less than) some other boundary values.

6. Create a function called choose_inventory:
   (a) choose_inventory must accept two parameters, a list called inventory and an int called selection. The purpose of this function is to accept a list x and an integer y, select y elements at random from x, and return these y elements in a new sorted list z.
   (b) If the list is empty and selection is equal to zero, return an empty list.
   (c) If the parameter called selection is negative, print a warning message and return an empty list.
   (d) If the parameter called selection is positive but larger than the size of the list, print a warning message and return a sorted copy of the list (not the original though).
   (e) If the parameter called selection is equal to the size of the list, return a sorted copy of the list (not the original though).
   (f) If none of the above is true, randomly select the specified number of elements from the inventory and return them in a new sorted list.
   (g) You may find the sample or choices functions in the random library helpful. They can be studied here: `https://docs.python.org/3/library/random.html`
   (h) When you test this function, since this function will return a value that is random, we cannot test it with a doctest. We can only test this function using unit tests. When you look at the different kinds of assertion statements we can use, do you see any that would be helpful? Here is a helpful link: `https://docs.python.org/3.7/library/unittest.html#assert-methods`

7. Create a function called generate_name. This function must accept a single parameter, a positive integer called syllables. This function has a pre-condition: the function will only work for positive non-zero integers.
   (a) In order to make our names a little easier to understand, we will generate syllables. Each syllable must consist of a consonant followed by a vowel.
   (b) Create a helper function called generate_vowel. This function should randomly select and return a single vowel (including y).
   (c) Create a helper function called generate_consonant. This function should randomly select and return a single consonant (including y).
   (d) Create a helper function called generate_syllable. This function should use generate_consonant and generate_vowel to acquire a random consonant and vowel, concatenate the two letters, and return the two-letter syllable.
   (e) Finally, generate_name should use the parameter it was passed and the generate_syllable function to generate a name composed of the specified number of syllables. Return this name.
   (f) Remember you can use the sample or choices functions in the random library helpful. They can be studied here: `https://docs.python.org/3/library/random.html`.
   (g) The string module also has some helpful things inside it. Check it out: `https://docs.python.org/3/library/string.html`.

8. Create a function called create_characte:
   (a) This function will create a Dungeons and Dragons character and return it.

(b) This function accepts a single parameter called name_length.

(c) This function must create a list. The first element in the list must be a random name whose length is specified by the parameter passed to the function. Don't duplicate code – invoke the create_name function you have already created and store the return value in a new list.

(d) A Dungeons and Dragons character has 6 core attributes: Strength, Dexterity, Constitution, Intelligence, Wisdom, and Charisma. Each of these attributes is initialized by rolling three six-sided dice. We call this 3d6 in D&D.

(e) For each of these attributes, create a mini-list that contains 2 items: a string that contains the attribute name, and the initial value.

(f) Each of these attribute mini-lists must be added to the new list that contains the randomly-generated name. Make sure they are added in order. That is, the first element in the list is the name. The second element in the list is a mini-list of two things, the string Strength and a value in the range [3, 18]. The third element in the list is a mini-list of two things, the string Intelligence and a value in the range [3, 18], and so on.

(g) If the parameter passed to this function is anything other than a positive integer, this function is not required to do anything special, i.e., let the function fail (possibly miserably)!

9. Create a function called print_character:

   (a) This function will accept a single parameter called character, which is a list formatted by the create_character function, and will print it to the screen in a lovely and easy-to-read manner.

   (b) You may assume this function will always accept a character that has been formatted exactly as described in the section about the create_character function. You do not have to test for poorly formatted input. You do not have to worry about malformed input.

10. Create a module called game_driver.py in your lab. This module should only contain a main function. In the main function, create a short interactive program for the user that demonstrates how your code works:

   (a) Print enough output so I know what is going on

   (b) Every function must be showcased

   (c) To demonstrate choose_inventory, create a list of strings. The list should contain tools and artifacts a hypothetical D&D player might find in a supply shoppe. Invite the user to select a number, use the function to randomly choose this number of elements from the inventory list, and show the user which items they got.

   (d) Bonus (1 mark): In your main function, append the list of purchased elements to the character list, and then print the entire character by passing it to print_character. Make sure you modify the print_character function so that it will work correctly whether or not the character has an inventory list appended to the end of the list.

That's it. Not so hard, was it?