COMP 1510 Programming Methods Lab 07

Christopher Thompson
chris_thompson@bcit.ca

BCIT CST — February 2022

1 Welcome!

Welcome to your seventh COMP 1510 lab.

For lab seven, I'd like to help you construct a very simple game. This will help you with your A4 text-based adventure game.

Let's begin!

2 Submission Requirements

1. This lab is due no later than **Saturday March 5th at or before 12:00:00 (that's noon.**
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/k78luoJu`

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. We will make a very simple game. The user will begin at the upper left hand corner of the environment aka row 0 column 0 aka coordinates (0, 0), and the user must reach the lower right hand corner without running out of HP. For lab 07, we will make a small board, 3 x 3 square. That means the user wants to successfully reach coordinates (2, 2).

2. Begin by creating a function called make_board inside the littlegame.py module. This function must accept two positive, non-zero integers called rows and columns that are equal to or greater than 2. This function is not required to behave correctly if it receives anything else. This function must create and return a dictionary that contains rows * columns keys, where each key is a tuple that contains a set of coordinates, and each value is a short string description. I made four empty rooms here, but maybe you can use a list of interesting descriptions and randomly assign one to each location. For example:

```
>>> rows = 2
>>> columns = 2
>>> board = make_board(rows, columns)
>>> board
{(0, 0): 'Empty room', (0, 1): 'Empty room', (1, 0): 'Empty room', (1, 1): 'Empty room'}
```

3. Create another function called make_character. This function must create and return a dictionary that contains the following key:value pairs: "X-coordinate": 0, "Y-coordinate": 0, "Current HP": 5. For example:

```
>>> player = make_character()
>>> player
{"X-coordinate": 0, "Y-coordinate": 0, "Current HP": 5}
```

4. Define a function called game() which accepts no parameters. Calling the game function must be the only line of code in your main function.

5. Inside the game() function, you need a game loop. Go ahead and copy this code snippet. It's a simple game loop:

```
def game(): # called from main
    rows = 3
    columns = 3
    board = make_board(rows, columns)
    character = make_character( )
    achieved_goal = False
    while not achieved_goal:
        // Tell the user where they are
        describe_current_location(board, character)
        direction = get_user_choice( )
        valid_move = validate_move(board, character, direction)
        if valid_move:
            move_character(character)
            describe_current_location(board, character)
            there_is_a_challenger = check_for_foes()
            if there_is_a_challenger:
                guessing_game(character)
            achieved_goal = check_if_goal_attained(board, character)
        else:
            // Tell the user they can't go in that direction
    // Print end of game stuff like congratulations or sorry you died
```

   Look at this code snippet carefully, Note what I have done here. I have assembled the parts of the game, i.e., a board that is 3 by 3, and a character. I've also declared a variable called achieved_goal and set it to False. We will not change this variable to True until the character reaches coordinates (2, 2), aka the bottom-right corner, aka the SE corner.

6. Inside the game-loop, the first thing we do is describe the current location. Implement a function called describe_current_location. My version of this function accepts the board and the character, but you may decide to make your function a bit differently. Use the character's coordinates to retrieve the information about that location from the board dictionary, and print the description.

7. The next thing we do in the loop is ask the user where they wish to go. Create a function called get_user_choice. This function must print a numbered list of directions and ask the user to enter the direction they wish to travel, and return the direction. Choose a system and stick to it, i.e., North-East-South-West, or Up-Down-Left-Right. Don't let me enter junk. If I enter something that is not a number from the list, make me choose again. Reject all user input that is not correct. Tell me to try again. And again. And again. Keep looping while my input is not correct.

8. The user is not allowed to cross the boundaries of the game board. Implement a function called validate_move. The version I've created here accepts three parameters, but yours may be a little different. This function must determine where the player is on the board and whether they can, in fact, travel in their desired direction. If so, return True. If not, return False. You will need to use some clever arithmetic here to ensure I don't go off the board!

9. If the move is valid, invoke a function called move_character which accepts the character dictionary and updates the character's X- and Y-coordinates appropriately.

10. After the character has moved, describe the new location. Then check to see if they have reached the bottom right hand corner of the game, i.e., their coordinates are now (rows - 1, columns - 1).

If so, they have made it to the end of the game without perishing. They have achieved their goal! Create a function called check_if_goal_attained(board, character) which checks and returns True if the character has made it to their destination, else False.

11. You will see in the game loop that if the user has reached the bottom right hand corner, and the achieved_goal is now True, we will break out of the game loop. The flow of control will now move to the code AFTER the while loop, and this is where you should print your congratulatory message.

12. I left something out. Did you notice? There is a function called check_for_foes. After each move, generate a random number. Ensue that 25% of the time, I encounter a foe. This function must return True if I am about to encounter a foe, else False.

13. What happens when I encounter a foe? I have to play a guessing game. The foe will randomly pick a number between 1 and 5 inclusive. I have 1 chance to guess it. If I guess it correctly, game play continues as usual. If I don't guess it correctly, I lose one HP. and then I may continue on my merry way.

14. There is some sample code in our course repo with a simple guessing game. I suggest you make a guessing_game function and invoke it if the check_for_foes function has returned True. The guessing_game function must reduce my HP by 1 if I guess incorrectly. If I do guess correctly, do not reduce my HP.

15. I left out one crucial thing. What if I die? Modify the guard condition in the while loop. Ensure game play ends when my HP reaches zero, even if I have not reached the end of the game. Perhaps you will make a function called is_alive which accepts the character and returns True if my HP is not 0, else False, like this:

    ```
    while is_alive(character) and not achieved_goal:
    ```

That's it! Good luck, and see you soon!