Zelle 4e Chapter 6 Coding Assignment

General Instructions

My expectations for your work on coding assignment exercises will grow as we progress through the course. In addition to applying any new programming techniques that have been covered in the current chapter, I will be expecting you to follow all of the good programming practices that we have adopted in the preceding weeks. Here is a quick summary of good practices that we have covered so far:

- Include a Python Docstring that describes the intent of the program.
- Place your highest-level code in a function named *main*.
- Include a final line of code in the program that executes the *main* function.
- Follow all PEP-8 Python coding style guidelines enforced by the PyCharm Editor. For example, place two blank lines between the code making up a function and the code surrounding that function.
- Choose names for your variables that are properly descriptive.
- Define CONSTANT_VALUES and use them in place of *magic numbers*.
- Always use f-strings for string interpolation and number formatting.
- When processing items from Python lists and tuples, unpack the values into variables with meaningful variable names to avoid using indexed expressions in your code.
- Remember that your program should behave reasonably when it is not given any input. This might be the result of the user pressing enter at a console prompt. Or, it might be the result of the user providing an input file that is empty.
- Model your solution after the code that I demonstrate in the lecture videos.
- Make sure that your test input/output matches the sample provided.
- All functions that are not *main()* should have descriptive, action-oriented names.
- All functions should be of reasonable size.
- All functions should have high *cohesion*, and low *coupling*.
- Remember to test your program thoroughly before submitting your work.
- Your code must pass all relevant test cases. Make sure that it passes tests at the boundaries created by *if, else*, and *elif* conditions in your program (boundary value tests).
- Program modules (.py files) that contain a *main()* function and that will also be imported into other programs must use the more sophisticated version of the call to *main()* at the end of the program to prevent *main()* being run when the module is imported.

This Assignment Has a Different Approach Toward Tutorial Videos
There are no dedicated tutorial videos for this coding assignment. Instead, each exercise of this assignment has been associated with one of the sample programs that were included in the download named *Demonstrations to Supplement Zelle 4e Chapter 6*.

The instructions for each exercise below identify which sample program to consult. The instructions also provide a reference to a place in the Beyond the Textbook lecture videos (timecode provided) where that sample program is discussed. So, you may use that portion of the lecture video as your tutorial for the exercise.

Exercise 1 (Regular)
A foot race has been held for a large group of children. In keeping with modern thinking regarding children's competitions, every participant will receive a ribbon. The following table indicates which ribbon the participant should receive based up the place number in which they finished.

Place	Ribbon
1	Blue
2	Red
3	Orange
4	Gold
5	Green
6	Purple
>6	White

Create a program named *distribute_race_ribbons.py.* You will use this program to do manual unit testing. Later race organizers while use it while distributing the ribbons. Each time the program is run, it will prompt the user for an integer representing the place in which the child finished. Then, it will print the color of the ribbon that the child has earned.

Design your program such that the code that looks up the ribbon earned is in a separate function named *determine_ribbon()*. For this exercise, you can trust the user to enter an integer. Nevertheless, you need to check for inputs that are proper integers but do not represent proper finishing places. In this case, the function should return an error message instead of the description of the race award.

When creating this program, you may use the sample program *_25_lookup_in_function* as a model. This sample program is discussed in the *Beyond the Textbook Part 1* lecture at 40:36.

Your testing approach to this exercise should include both manual and automated testing. One practical aspect of this approach, is that the code at the bottom of your program that calls the *main()* function must use the more sophisticated form that was introduced in Chapter 6:

```
if __name__ == '__main__':
    main()
```

Another practical aspect of this approach is that the *main()* function of your program must include user interaction code that supports your unit testing work.

The following is an example of expected input/output on the console from a successful user interaction when performing manual unit testing:

```
Please enter place finished (1, 2, 3...): 5

Ribbon Awarded: Green
```

The following is an example of expected input/output on the console from a user interaction the results in an error when performing manual unit testing:

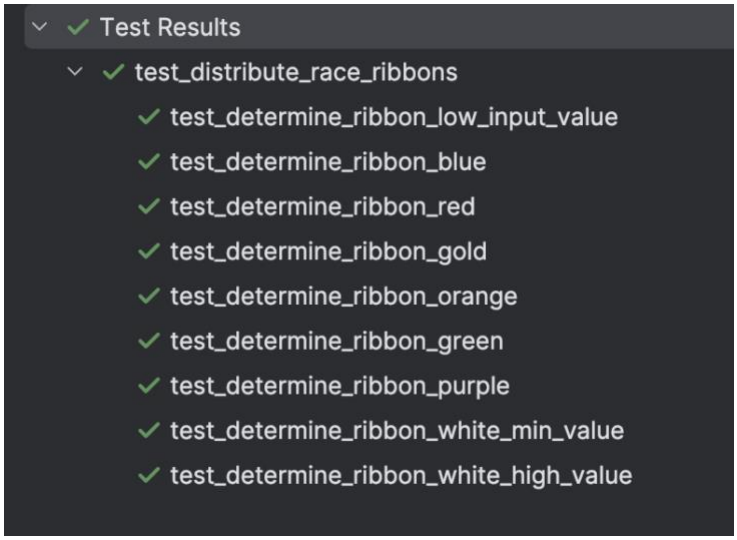
```
Please enter place finished (1, 2, 3...): -9

Ribbon Awarded: ERROR - Place must be greater than zero.
```

When you have concluded your manual unit testing, perform automated unit testing on this program as well. To do so, take the following approach:

- Use the PyCharm features demonstrated in the lecture videos to create a test program file for *distribute_race_ribbons.py*. This test program file should be named *test_distribute_race_ribbons.py*.
- Create individual test case functions for each test case. Remember to include a test case for each ribbon color and a test case for error values.
- Run each test case as you create it to confirm that it passes.
- Correct and re-run any failing test cases.
- Conclude testing by running all test cases to confirm that they all pass.

When you run all test cases and get passing results, the PyCharm test results output panel should resemble the following:



Exercise 2 (Regular)
A state department of motor vehicles needs to calculate annual registration fees for vehicles registered in the state. Fees are based upon the vehicle type (car or truck) and the vehicle weight.

Create a python program named *dmv_system_common.py*. Eventually, this Python module will contain several functions used by various programs in the DMV System. For now, it will only contain the function that implements the annual fee calculation function.

When creating this program, you may use the sample program *_35_nested_in_function* as a model. This sample program is discussed in the *Beyond the Textbook Part 1* lecture at 1:04:21.

Within *dmv_system_common.py*, create a Python function named *determine_annual_registration_fee()*. It should accept two input parameters: vehicle-type and weight. It should return the annual fee as a float value. The annual fee will be based upon the following table:

Vehicle Type	Weight	Annual Fee
Car	< 3000	125.00
Car	>= 3000	200.00
Truck	< 4000	250.00
Truck	>= 4000	350.00

In normal circumstances, any calls made to the determine_annual_registration_fee function should be for a car or a truck. Nevertheless, the code should check the vehicle-type for unexpected values. If an unexpected vehicle type value is detected, the code should raise a *ValueError* with a descriptive message.

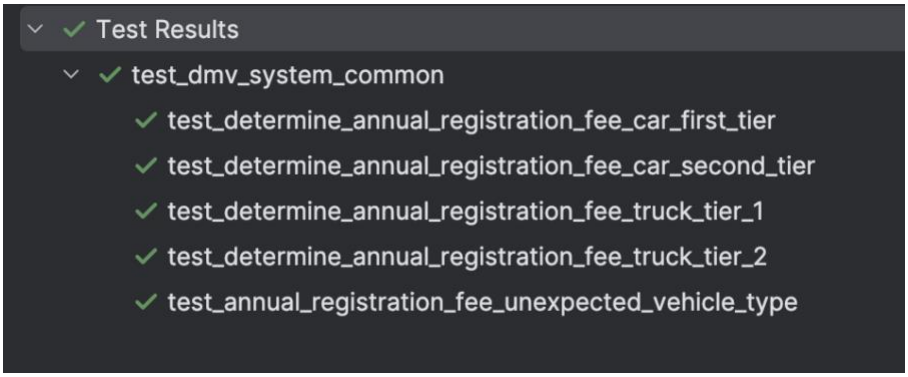
PLEASE NOTE: Because *dmv_system_common.py* contains only functions that will be imported by other programs and called by them, this program will not contain a main function. Also, it will not contain code that calls the *main()* function.

While you will not be performing manual unit testing on *dmv_system_common.py*, you will be performing automated unit testing using the following approach:

- Use the PyCharm features demonstrated in the lecture videos to create a test program file for *dmv_system_common.py*. This test program file should be named *test_dmv_system_common.py*.

- Create individual test case functions for each test case. Remember to include a test case for each combination of vehicle type and weight using the principle of boundary value testing.
- Also create a test case for an unexpected vehicle type.
- Run each test case as you create it to confirm that it passes.
- Correct and re-run any failing test cases.
- Conclude testing by running all test cases to confirm that they all pass.

When you run all test cases and get passing results, the PyCharm test results output panel should resemble the following:



Exercise 3 (Regular)
Create a program named *detect_input_error.py.* This program will be based upon the demonstration program *decisions_40_try.py.* This sample program is discussed in the *Beyond the Textbook Part 2* lecture at 0:00. Feel free to copy the sample program to provide a starting point for your code.

Your program should be different from the sample program in the following respects:

1. Instead of only prompting the user for 1 integer, your program should use a for/in loop to prompt the user for an integer 5 times.

2. Your program should print polite messages to the user at the start of the program so that the user knows how many integers they will be prompted for and what kind of output to expect.

3. When your program detects a value error, it should issue an error message for that problem user input. Then, it should continue prompting and processing the remainder of the expected 5 user inputs.

Make sure that your program catches any bad input, prints the appropriate error message, and makes an immediate graceful exit.

PLEASE NOTE: You will only be doing manual unit testing for this program.

The following is an example of expected input/output on the console from a manual unit test in which proper integer values are entered by the user:

```
This program prompts you for 5 integers.
Valid integer inputs are echoed back to you on the console.
If you enter an invalid input, the program will print a warning message.

Please enter an integer: 11
You have entered the integer 11

Please enter an integer: 22
You have entered the integer 22

Please enter an integer: 33
You have entered the integer 33

Please enter an integer: 44
You have entered the integer 44

Please enter an integer: 55
You have entered the integer 55

Thanks for playing.
```

The following is an example of expected input/output on the console from a manual unit test in which some invalid values are entered by the user:

```
This program prompts you for 5 integers.
Valid integer inputs are echoed back to you on the console.
If you enter an invalid input, the program will print a warning message.

Please enter an integer: 111
You have entered the integer 111

Please enter an integer: 2.22
An integer was expected. You entered "2.22".

Please enter an integer: 333
You have entered the integer 333

Please enter an integer: Hi, Mom!
An integer was expected. You entered "Hi, Mom!".

Please enter an integer: 555
You have entered the integer 555

Thanks for playing.
```

Exercise 4 (Challenge)
Create a program named *panama_canal_system_common.py*. This program will be based upon the sample program *_80_validation_using_function_and_messages.py*. This sample program is discussed in the *Beyond the Textbook Part 2* lecture at 44:19. Feel free to copy the sample program to provide a starting point for your code.

Eventually, this Python module will contain several functions used by various programs in the Panama Canal System. For now, it will only contain the function that determines the eligibility of ships to transit the Panama Canal. Your program should implement a single function named *determine_new_panamax_eligibility()*. This function will be imported and called by various programs in the Panama Canal System when they need to check ship eligibility. Your program will be different from the sample program in the following respects.

1. Your program will evaluate candidate vessels that wish to transit the Panama Canal using the General characteristics New Panamax limits as shown in the Wikipedia article at https://en.wikipedia.org/wiki/Panamax

2. Your program should allow for floating point values to be entered for each of the characteristics.

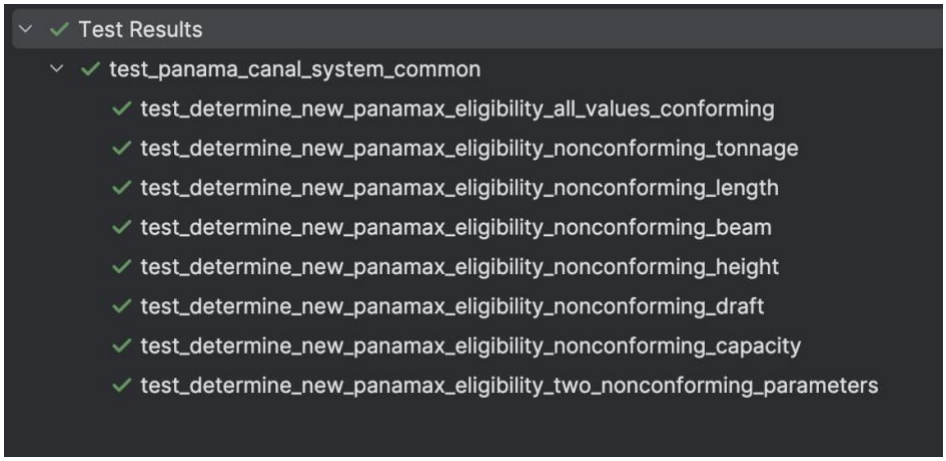
PLEASE NOTE: Because *panama_canal_system_common.py* contains only functions that will be imported by other programs and called by them, this program will not contain a main function. Also, it will not contain code that calls the *main()* function.

While you will not be performing manual unit testing on *panama_canal_system_common.py*, you will be performing automated unit testing using the following approach:

- Use the PyCharm features demonstrated in the lecture videos to create a test program file for *panama_canal_system_common.py*. This test program file should be named *test_panama_canal_system_common.py*.
- Create individual test case functions for each test case.
- Remember to include a test case where all values of the parameters are conforming:
    - Tonnage (in DWT)
    - Length (in feet)
    - Beam (in feet)
    - Height (in feet)
    - Draft (in feet)
    - Capacity (in TEU)
- Remember to create a test case for each parameter where it has a non-conforming value and the other parameters are conforming.

- Remember to create a test case that has more than one non-conforming value and generates more than one error message.
- Run each test case as you create it to confirm that it passes.
- Correct and re-run any failing test cases.
- Conclude testing by running all test cases to confirm that they all pass.

When you run all test cases and get passing results, the PyCharm test results output panel should resemble the following:



Tools
Use PyCharm (and Pytest) to create and test all Python programs.

Submission Method
Follow the process that I demonstrated in the tutorial video on submitting your work. This involves:

- Locating the properly named directory associated with your project in the file system.
- Compressing that directory into a single .ZIP file using a utility program.
- Submitting the properly named zip file to the submission activity for this assignment.

File and Directory Naming
Please name your Python program files as instructed in each exercise. Please use the following naming scheme for naming your PyCharm project:

surname_givenname_exercises_zelle_4e_chapter_06

If this were my own project, I would name my PyCharm project as follows:

trainor_kevin_exercises_zelle_4e_chapter_06

Use a zip utility to create one zip file that contain the PyCharm project directory. The zip file should be named according to the following scheme:

surname_givenname_exercises_zelle_4e_chapter_06.zip

If this were my own project, I would name the zip file as follows:

trainor_kevin_exercises_zelle_4e_chapter_06.zip

Due By
Please submit this assignment by the date and time shown in the Weekly Schedule.

Last Revised
2025-09-15