Zelle 3e Chapter 8 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.
- Close all files before the conclusion of the program.
- 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 a an input file that is empty.
- Model your solution after the code that I demonstrate in the tutorial videos.
- Make sure that your test input/output matches the sample provided.
- Create a sub-directory named *data* within your PyCharm project to hold data files.
- Remember to submit all data files with your PyCharm project – including the files that were provided as starter files to this assignment.
- 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).
- Use of the *break* statement is allowed but not encouraged.
- Use of the *continue* statement is forbidden.

Exercise 1 (Regular)
Create a program named *flexible_float_adder.py*. It should modeled after the program that I demonstrated in the tutorial *(_05_flexible_user_interaction_using_while.py*). Your program should be different in the following respects:

1. The user should be prompted to enter float values.

2. The program should accumulate float values.

3. When printing the accumulated sum, the program should display 2 decimal places.

When running a test where the user provides no input, you should expect the following input/output on your console:

```
Welcome to The Flexible Float Adding Machine.

Please enter a float value to be added (<Enter> to stop):

No entries were provided.
```

When running a test with more typical input, you should expect the following input/output on your console:

```
Welcome to The Flexible Float Adding Machine.

Please enter a float value to be added (<Enter> to stop): 1.11
Please enter a float value to be added (<Enter> to stop): 2.22
Please enter a float value to be added (<Enter> to stop): 4.44
Please enter a float value to be added (<Enter> to stop):

The sum of these 3 entries is 7.77
```

Exercise 2 (Regular)
Create a program named *usable_float_adder.py*. It should modeled after the program that I demonstrated in the tutorial (*_08_recovering_from_bad_user_input_using_while.py*). Your program should be different in the following respects:

1. The user should be prompted to enter float values.

2. The program should accumulate float values.

3. When printing the accumulated sum, the program should display 2 decimal places.

When running a test where the user provides no input, you should expect the following input/output on your console:

```
Welcome to The Usable Float Adding Machine.

Please enter a float value to be added (<Enter> to stop):

No entries were provided.
```

When running a test where the user provides bad input, you should expect the following input/output on your console:

```
Welcome to The Usable Float Adding Machine.

Please enter a float value to be added (<Enter> to stop): Hi, Mom!
A float value was expected. You entered Hi, Mom!
Please enter a float value to be added (<Enter> to stop): 1.11
Please enter a float value to be added (<Enter> to stop): 2.22
Please enter a float value to be added (<Enter> to stop): 4.44
Please enter a float value to be added (<Enter> to stop):

The sum of these 3 entries is 7.77
```

When running a test where the user provides typical input, you should expect the following input/output on your console:

```
Welcome to The Usable Float Adding Machine.

Please enter a float value to be added (<Enter> to stop): 1.11
Please enter a float value to be added (<Enter> to stop): 2.22
Please enter a float value to be added (<Enter> to stop): 4.44
Please enter a float value to be added (<Enter> to stop):

The sum of these 3 entries is 7.77
```

Exercise 3 (Regular)
Create a program named *identify_slot_machine_winner.py*. It should modeled after the program that I demonstrated in the tutorial *(_10_searching_using_while.py*). Your program should be different in the following respects:

1. It should search for a line in a text file that contains three occurrences of the word "Red" and 2 occurrences of the word "Blue". This line represents a slot machine win. Hint: you can use *list.count()* to count occurrences.

2. It should provide the expected results when searching the following starter files (see expected results below):

   - `empty_file.txt`
   - `slot_values.txt`
   - `tricky_slot_values.txt`

3. It should recognize the winning line even when that line is the last line in the file.

When running a test with `empty_file.txt`, you should expect the following input/output on your console:

```
Please enter the input filename: empty_file.txt

No winning slot values found.
0 lines read from empty_file.txt
```

When running a test with `tricky_slot_values.txt`, you should expect the following input/output on your console:

```
Please enter the input filename: tricky_slot_values.txt

Winner found on line 10: Red Blue Red Red Blue
```

When running a test with `slot_values.txt`, you should expect the following input/output on your console:

```
Please enter the input filename: slot_values.txt

Winner found on line 3458: Red Blue Red Red Blue
```

Exercise 4 (Regular)
Create a program named *create_name_badges.py*. It should modeled after the program that I demonstrated in the tutorial *(_15_processing_data_records_using_while.py*).
Your program should be different in the following respects:

1. It will read a text file with a single person name on each line. This name will be formatted in natural order.

2. It will write a text file with name values on each line that can be used by a word processing program to create name badges. The first name will be in all caps. It will be followed directly by a comma. That will be followed directly by the last name in mixed case. An example of this output format would be:

```
ANTHONY,Parr
```

3. The name of the output file should be created automatically by prepending "name_badges_" to the name of the input file. For example, when the input filename is `empty_file.txt`, the output filename will be `name_badges_empty_file.txt`.

This program should produce the expected results (see below) when processing the following input files provided as starter files:

- `empty_file.txt`
- `person_names.txt`

When running a test with `empty_file.txt`, you should expect the following input/output on your console:

```
Please enter the input filename: empty_file.txt

0 records were read from empty_file.txt
0 records were written to name_badges_empty_file.txt
```

When running a test with `person_names.txt`, you should expect the following input/output on your console:

```
Please enter the input filename: person_names.txt

125 records were read from person_names.txt
125 records were written to name_badges_person_names.txt
```

Exercise 5 (Challenge)
Create a program named *rock_paper_scissors.py*. It should modeled after the Chapter 8 demonstration program that I distributed named *_20_playing_a_game_using_while.py*. Your program should be different in the following respects:

1. It will play *The Rock-Paper-Scissors Game* rather than *The Simple Coin Flipping Game.*

2. This program should implement the rules of the rock-paper-scissors game as presented in https://wrpsa.com/the-official-rules-of-rock-paper-scissors/ .

The following is a simple example of output that you should expect on your console when running a test:

```
Welcome to The Rock-Paper-Scissors Game.

Player A throws SCISSORS. Player B throws PAPER.
Player A wins.
```

The following is another simple example of output that you should expect on your console when running a test:

```
Welcome to The Rock-Paper-Scissors Game.

Player A throws SCISSORS. Player B throws ROCK.
Player B wins.
```

The following is a more complex example of output that you should expect on your console when running a test:

```
Welcome to The Rock-Paper-Scissors Game.

Player A throws ROCK. Player B throws ROCK.
Player A throws SCISSORS. Player B throws SCISSORS.
Player A throws ROCK. Player B throws ROCK.
Player A throws PAPER. Player B throws PAPER.
Player A throws SCISSORS. Player B throws PAPER.
Player A wins.
```

Tools
Use PyCharm 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_3e_chapter_08
```

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

```
trainor_kevin_exercises_zelle_3e_chapter_08
```

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_3e_chapter_08.zip
```

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

```
trainor_kevin_exercises_zelle_3e_chapter_08.zip
```

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

Last Revised
2022-06-02