Zelle 4e 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.
- Remember that your program should behave reasonably when it is not given any input.
- Model your solution after the code that I demonstrate in the tutorial videos.
- Make sure that your test input/output matches the sample provided.
- Remember to test your program thoroughly before submitting your work.
- 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).
- Use of the *break* statement is allowed.
- Use of the *continue* statement is forbidden.

Exercise 1 (Regular)
Using the design pattern that I demonstrated in the tutorial video for this exercise, create a program named *numeric_character_analytics.py.* It should meet the following requirements:

1. Prompt the user to enter a line of text to be analyzed.

2. For each word in the line of text entered, print the quantity of numeric digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) included in the word. Remember to adjust the wording of your output to account for having found "1 digit" as compared to "5 digits".

3. Print the number of words that were analyzed.

The following is an example of expected input/output on your console from a typical test:

```
Please enter a line of text to be analyzed: I ate 5 hamburgers and
drank 6 beers at Studio54 on 54th Street.

"I" contains 0 numeric characters.
"ate" contains 0 numeric characters.
"5" contains 1 numeric character.
"hamburgers" contains 0 numeric characters.
"and" contains 0 numeric characters.
"drank" contains 0 numeric characters.
"6" contains 1 numeric character.
"beers" contains 0 numeric characters.
"at" contains 0 numeric characters.
"Studio54" contains 2 numeric characters.
"on" contains 0 numeric characters.
"54th" contains 2 numeric characters.
"Street." contains 0 numeric characters.

13 words were analyzed.
```

Your program should behave reasonably when no input is provided. The following is an example of expected input/output on your console from a test with empty input:

```
Please enter a line of text to be analyzed:


0 words were analyzed.
```

Exercise 2 (Regular)
You will be creating several functions for the Acme System that will eventually be imported and called by other programs in the Acme System. Using the design pattern that I demonstrated in the tutorial video for this exercise, create a module named *acme_common.py* to hold these functions and any constants that they may require.

Within *acme_common.py* create a function named *validate_username_components()* that will be used to validate data values used to create a username in the Acme System. This function will accept the following parameters:

- Firstname (str): First name.
- Lastname (str): Last name.
- Year Hired (str): The 4-digit year in which the user was hired.

Validate each of these values and return a list of any error messages that apply. If there are no errors, pass back an empty list of error messages.

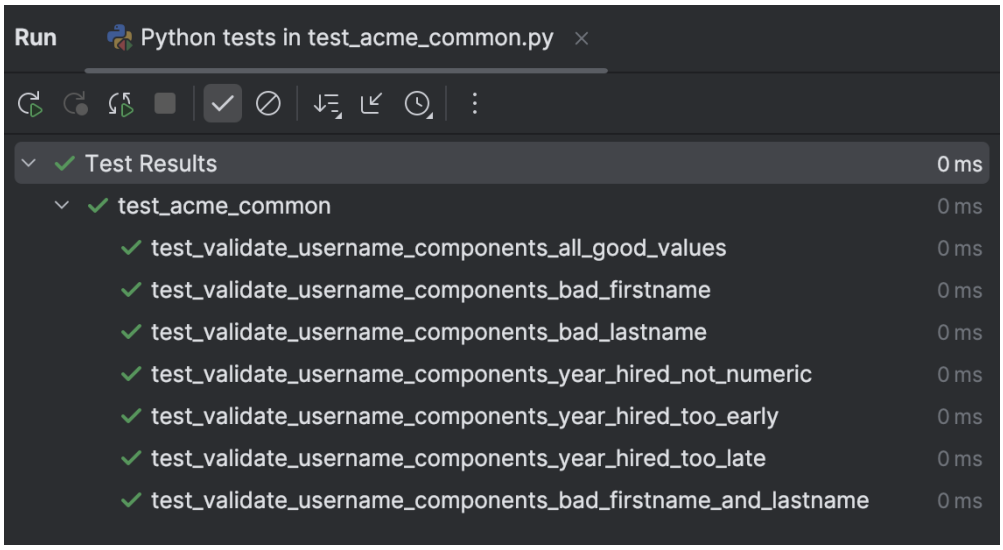
While validating, check for the following:

- Firstname must be alphabetic without any spaces or special characters.
- Lastname must be alphabetic without any spaces or special characters.
- Year Hired must be numeric without any spaces or special characters.
- Year Hired must not be prior to the founding year for Acme (1999).
- Year Hired must not be after the current year.

Make sure to include a DocString for this function using the Google format conventions.

Create a test module for *acme_common.py* named *test_acme_common.py*. Create and execute Pytest test cases that adequately cover boundary values for the *validate_username_components()* function.

When you execute the unit test cases for the *validate_username_components()* function, the PyCharm test output should resemble the following:



Exercise 3 (Regular)
Within *acme_common.py* create a function named *create_username()* that will be used to create a username in the Acme System. You may use my tutorial as a design pattern for this function. This function will accept the following parameters:

- Firstname (str): First name.
- Lastname (str): Last name.
- Year Hired (str): The 4-digit year in which the user was hired.

Programs that call c*reate_username()* are expected to have validated the values being used by having previously called *validate_username_components()* and having received an empty error messages list as the return value.

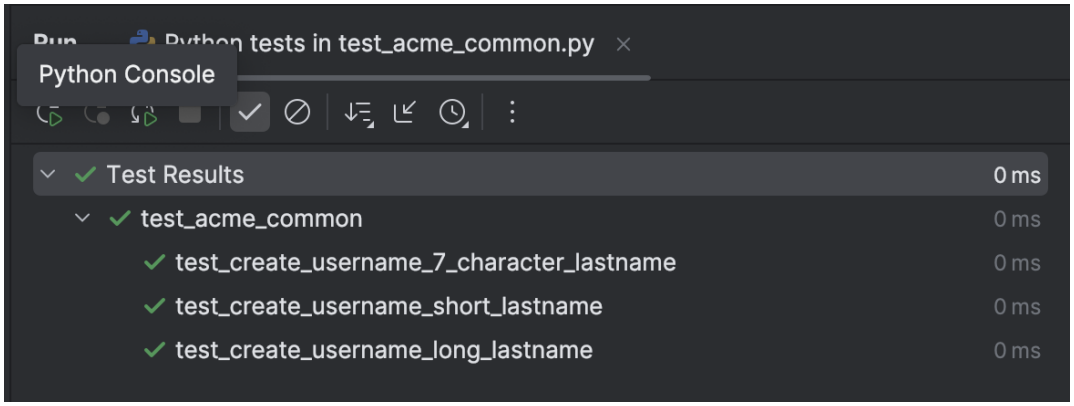
Within the create_username() function, use the following specifications to assemble and return a proper username value:

- The firstname component should be the first character only.
- The lastname component should be the first portion of the lastname up to 7 characters.
- The year hired component should be the last two digits of the 4-digit year.
- The order of the components should be: firstname, lastname, year hired.

Make sure to include a DocString for this function using the Google format conventions.

Create and execute Pytest test cases that adequately cover boundary values for the *create_username()* function.

When you execute the unit test cases for the *create_username()* function, the PyCharm test output should resemble the following:



Note that to generate test output that matches this example you may need to place test case functions that do not apply into a comment block.

Exercise 4 (Challenge)
Create a stand-alone program module named *acme_username_builder.py*. While there is no tutorial video for this exercise, I did demonstrate a similar program during my tutorials for exercises 2 and 3. So, use my comments in those tutorials as a guide.

The program should prompt the user at the console for one set of inputs from which to create a username, including:

- Firstname
- Lastname
- Year Hired

It should then call the previously created *validate_username_components()* to validate these values. If the validation is successful, the program should print the results (see example below). If the validation is unsuccessful, the program should print a list of errors (see example below).

This program should expect to receive input for exactly 1 user. While it will re-prompt at the console when it receives invalid input, it will not offer to create further usernames after the first success.

The following is an example of console input/output for a scenario that includes one round of errors and a subsequent success:

```
Please enter the user's first name: 4Hector
Please enter the user's last name: Rodriguez/
Please enter the user's year hired: 1998

You will need to correct the following errors before re-
entering values:
      First name must be made up of alphabetic characters only.
      Last name must be made up of alphabetic characters only.
      Year hired must not be prior to 1999.

Please enter the user's first name: Hector
Please enter the user's last name: Rodriguez
Please enter the user's year hired: 1999
The username for Hector Rodriguez will be hrodrigu99
```

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_08
```

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

```
trainor_kevin_exercises_zelle_4e_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_4e_chapter_08.zip
```

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

```
trainor_kevin_exercises_zelle_4e_chapter_08.zip
```

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

Last Revised
2025-10-01