Lab 7

CS161

For each of the problems below be sure to design your solution before you write any Python code. Follow the software development stages:

1. Analyze the problem
2. Determine the specifications for your solution
3. Create a design for your solution in pseudocode
4. Implement your design in Python
5. Test/debug your solution

Each Python module you turn in must begin with a block, called a *docstring*, that looks like the example below (see PEP 257 for more details). The module docstring must follow the shebang and coding statements in your module; i.e. it must be the first actual line of code in the module.

In your module docstring show the work you did following the software development stages above. Make notes about the problem as you analyze it; write specifications for what your program must do to find a solution; write your program out in pseudocode *before* you start to write Python; and design a set of test data that you can use to prove your program is producing correct results. Only after you've done all of this should you try to translate your pseudocode to Python.

```
#! /usr/bin/env python3
# coding=utf-8

"""
Brief, prescriptive, one sentence description of what the module does.

A more detailed explanation of what the module does. Use complete sentences and
proper grammar. Treat this as if it is documentation someone else will be reading
(because it is). Include any notes from your problem analysis as well as your
program specifications here.

Pseudo code, to be written before any actual Python code.

Assignment or Lab #

Firstname Lastname
Firstname Lastname of partner (if applicable)
"""
<two blank lines>
<your code>
```

Every function or method you write must also include its own docstring. That header will also use triple quotes and begin with a prescriptive, one sentence description of what the function or method does. If the function or method takes arguments, they should be described as well as any non-obvious return value(s). See examples below or PEP 257 for more details).

```
def print_greeting():
    """Print a greeting to the user."""
    print("Hello out there in userland!")


def hypotenuse(side_a, side_b):
    """
    Calculates the length of the hypotenuse of a right triangle using
    the formula c^2 = a^2 + b^2.

    side_a – the length of side A of the triangle
    side_b – the length of side B of the triangle
    """
    c = a**2 + b**2
    return math.sqrt(c)
```

Pair Programming

In this lab we are going to use **pair programming**.

> *Pair programming is an agile software development technique in which two programmers work together at one workstation. One, the **driver**, writes code while the other, the **observer** or **navigator**, reviews each line of code as it is typed in. The two programmers switch roles frequently.*
>
> *While reviewing, the observer also considers the "strategic" direction of the work, coming up with ideas for improvements and likely future problems to address. This is intended to free the driver to focus all of their attention on the "tactical" aspects of completing the current task, using the observer as a safety net and guide.*

Choose one partner in the lab to work with. You can pair program in one of two ways:

1. Two people can work at one computer, occasionally switching the driver and observer roles. It is *critical* that both members be engaged in the work. You should take turns being in the driver and observer roles with only the driver typing and the observer making comments or driving the direction of the code.
2. Two people can work at separate machines using Visual Studio Live Share. Live Share, available as a free plug-in for VS Code and already installed on the school's Surface devices, enables you and your partner to collaborate on the same codebase without the need any difficult configuration. When you share a collaborative session, your partner sees the context of the workspace in their editor. This means they can read the code you shared without having to clone a repo or install any dependencies your code relies on. They can use rich language features to navigate within the code; not only just opening other files as text but using semantic analysis-based navigation like Go to Definition or Peek. Again, you should take turns being in the driver and observer roles with only the driver typing and the observer making comments or driving the direction of the code.

Word Game

In this problem lab, you'll implement a version of a familiar word game! Don't be intimidated by the length of this lab. It's a lot of reading, but it is very doable.

Let's begin by describing the word game: This game is a lot like *Scrabble* or *Words With Friends*. Letters are dealt to players, who then construct one or more words using their letters. Each valid word earns the user points, based on the length of the word and the letters in that word.

The rules of the game are as follows. **Do not start coding yet – we will break this down into steps below!**

Dealing

- A player is dealt a hand of `HAND_SIZE` letters of the alphabet, chosen at random. This may include multiple instances of a particular letter.
- The player arranges the hand into as many words as they want out of the letters, but using each letter at most once.
- Some letters may remain unused, though the size of the hand when a word is played does affect its score.

Scoring

- The score for the hand is the sum of the score for each word formed.
- The score for a word is the **product** of two components:
    - First component: the sum of the points for letters in the word.
    - Second component: either `[7 * word_length - 3 * (n - word_length)]` or `1`, whichever value is greater, where:
        - *word_length* is the number of letters used in the word
        - *n* is the number of letters available in the current hand
- Letters are scored as in Scrabble; A is worth 1, B is worth 3, C is worth 3, D is worth2, E is worth 1, and so on. We have defined the dictionary `SCRABBLE_LETTER_VALUES` that maps each lowercase letter to its Scrabble letter value.
- Examples:
    - For example, if *n*=6 and the hand includes 1 'w', 2 'e's, and 1 'd' (as well as two other letters), playing the word 'weed' would be worth 176 points: (4+1+1+2) * (7*4 - 3*(6-4)) = 176. The first term is the sum of the values of each letter used; the second term is the special computation that rewards a player for playing a longer word and penalizes them for any leftover letters.
    - As another example, if n=7, playing the word 'it' would be worth 2 points:(1+1) * (1) = 2. The second component is 1 because 7*2 - 3*(7 - 2) = -1,which is less than 1.

Getting Started

1. The lab-7-files folder in the shared Google Drive class materials folder has the files you need to start. This includes the python file `word_game.py`, which should contain all your code, as it provides a set of initial procedures and templates for new procedures. Also in the folder is a file for testing your code, `test_word_game.py`, and a file of legitimate words `words.txt`.

2. Run `word_game.py`, without making any modifications to it, in order to ensure that everything is set up correctly. The code given to you loads a list of valid words from a file and then calls the `play_game` function. You will implement the functions it needs in order to work. If everything is okay, after a small delay, you should see the following printed out:

   ```
   Loading word list from file...
         83667 words loaded.
         play_game not yet implemented.
   ```

   If you see an `IOError` instead (e.g., *No such file or directory*), make sure you have saved `words.txt` in the same directory as `word_game.py`!

3. The file `word_game.py` has a number of already-implemented functions you can use while writing your solution. You can ignore the code between the following comments, though you should read and understand everything else.

   ```
   # -----------------------------------
   # Helper code
   # (you don't need to understand this helper code)
   .
   .
   .
   # (end of helper code)
   # -----------------------------------
   ```

4. This problem set is structured so that you will write a number of modular functions and then glue them together to form the complete game. Instead of waiting until the entire game is *ready*, you should test each function you write, individually, before moving on. This approach is known as *unit testing*, and it will help you debug your code.
5. Included are some hints about how you might want to implement some of the required functions in the included files. You don't need to remove them in your final submission.
6. Also provided are several test functions to get you started. As you make progress on the problem set, run `test_word_game.py` to check your work so far.

If your code passes the unit tests you will see a SUCCESS message; otherwise you will see a FAILURE message. **These tests aren't exhaustive. You may want to test your code in other ways too** (for example, with different test values)**.**

If you run `test_word_game.py` using the initially provided `word_game.py` skeleton, you should see that all the tests fail.

These are the provided test functions:

`test_get_word_score`

Test the `get_word_score` implementation.

`test_update_hand`

Test the `update_hand` implementation.

`test_is_valid_word`

Test the `is_valid_word` implementation.

`test_wildcard`

Test the modifications made to support wildcards. (more about those later on)

Problem 1: Word Scores

The first step is to implement a function that calculates the score for a single word. Fill in the code for `get_word_score` in `word_game.py` according to the function specifications.

As a reminder, here are the rules for scoring a word:

- The score for the hand is the sum of the score for each word formed.
- The score for a word is the **product** of two components:
    - First component: the sum of the points for letters in the word.
    - Second component: either `[7 * word_length - 3 * (n - word_length)]` or `1`, whichever value is greater, where:
        - *word_length* is the number of letters used in the word
        - *n* is the number of letters available in the current hand

You should use the `SCRABBLE_LETTER_VALUES` dictionary defined at the top of `word_game.py`. Do **not** assume that there are always 7 letters in a hand! The parameter *n* is the total number of letters in the hand when the word was entered.

Finally, you may find the `str.lower` function helpful:

```
s = "My string"
print(s.lower())
>>> "my string"
```

If you don't know what this does you could try typing help(str.lower) in the Python shell to see the documentation for the functions.

Testing: If this function is implemented correctly, and you run `test_word_game.py`, the `test_get_word_score()` tests will pass. You should also test your implementation of `get_word_score` yourself, using some reasonable English words. Note that the wildcard tests will crash due to a KeyError. This is fine for now - you will fix this in Problem 4.

Problem 2: Dealing with hands

****Please read problem 2 entirely before you begin coding your solution**** Most of the functions described below have been implemented for you already.

Representing hands

A hand is the set of letters held by a player during the game. The player is initially dealt a set of random letters. For example, the player could start out with the following hand: **a**, **q**, **l**, **m**, **u**, **i**, **l**. In our program, a hand will be represented as a dictionary: the keys are (lowercase) letters and the values are the number of times the particular letter is repeated in that hand. For example, the above hand would be represented as:

```
hand = {'a':1, 'q':1, 'l':2, 'm':1, 'u':1, 'i':1}
```

Notice how the repeated letter 'l' is represented. With a dictionary representation, the usual way to access a value is hand['a'], where 'a' is the key we want to find. However, this only works if the key is in

the dictionary; otherwise, we get a `KeyError`. To avoid this,we can instead use the function call `hand.get('a',0)` . This is the "safe" way to access a value if we are not sure the key is in the dictionary. `d.get(key,default)` returns the value for key if key is in the dictionary `d`, else it returns default . If default is not given, it returns `None`, so that this method never raises a `KeyError`.

Converting words into dictionary representation

One useful function defined for you is `get_frequency_dict`, defined near the top of `word_game.py`. When given a string of letters as an input, it returns a dictionary where the keys are letters and the values are the number of times that letter is represented in the input string. For example:

```
>>>get_frequency_dict("hello") {'h': 1, 'e': 1, 'l': 2, 'o': 1}
```

As you can see, this is the same kind of dictionary we use to represent hands.

Displaying a hand

Given a hand represented as a dictionary, we want to display it in a user-friendly way. You've been given the implementation for this in the `display_hand` function. Take a few minutes now to read through this function carefully and understand what it does and how it works.

Generating a random hand

The hand a player is dealt is a set of letters chosen at random. You've been provided with a function that generates a random hand, `deal_hand`. The function takes as input a positive integer *n*, and returns a new dictionary representing a hand of *n* lowercase letters. Again, take a few minutes to read through this function carefully and understand what it does and how it works.

Removing letters from a hand (you implement this!)

The player starts with a full hand of *n* letters. As the player spells out words, letters from the set are used up. For example, the player could start with the following hand: **a**, **q**, **l**, **m**, **u**, **i**, **l**. The player could choose to play the word **quail**. This would leave the following letters in the player's hand: l, m.

You will now write a function that takes a hand and a word as inputs, uses letters from that hand to spell the word, and returns a new hand containing only the remaining letters. Your function should **not** modify the input hand. For example:

```
>>>hand = {'a':1, 'q':1, 'l':2, 'm':1, 'u':1, 'i':1}
>>>display_hand(hand)
a q l l m u i
>>new_hand = update_hand(hand, 'quail')
>>>new_hand
{'l': 1, 'm': 1}
>>>display_hand(new_hand)
l m
>>>display_hand(hand)
a q l l m u i
```

(**NOTE:** Alternatively, in the above example, after the call to `update_hand` the value of `new_hand` could be the dictionary `{'a':0, 'q':0, 'l':1, 'm':1, 'u':0, 'i':0}`. The exact value depends on your implementation; but the output of `display_hand()` should be the same in either case.)

IMPORTANT: If the player guesses a word that is invalid, either because it is not a real word or because they used letters that they don't actually have in their hand, they still lose the letters from their hand

that they did guess as a penalty. Make sure that your implementation accounts for this! Do not assume that the word you are given only uses letters that actually exist in the hand. For example:

```
>>>hand = {'j':2, 'o':1, 'l':1, 'w':1, 'n':2}
>>>display_hand(hand)
j j o l w n n
>>>hand = update_hand(hand, 'jolly')
>>>hand
{'j':1, w':1, 'n':2}
>>>display_hand(hand)
j w n n
```

Note that one 'j', one 'o', and one 'l' (despite that facts that the player tried to use two, because only one existed in the hand) were used up. The 'y' guess has no effect on the hand, because 'y' was not in the hand to begin with. Also, the same note from above about alternate representations of the hand applies here.

Implement the **`update_hand`** function according to the specifications in the skeleton code.

HINT: You may wish to review the documentation for the ".copy" method of Python dictionaries.

Testing: Make sure the test_update_hand tests pass. You may also want to test yourimplementation of update_hand with some reasonable inputs.

Problem 3. Valid words

At this point, we have not written any code to verify that a word given by a player obeys the rules of the game. A *valid* word is in the word list (we ignore the case of words here) **and** it is composed entirely of letters from the current hand.

Implement the **`is_valid_word`** function according to its specifications.

Testing: Make sure the **`test_is_valid_word`** tests pass. You should also test your implementation with some reasonable inputs. In particular, you may want to test your implementation by calling it multiple times on the same hand - what should the correct behavior be?

Problem 4. Wildcards

We want to allow hands to contain wildcard letters, which will be denoted by an asterisk (*). **Wildcards can only replace vowels**. Each hand dealt should initially contain exactly one wildcard as one of its letters. The player **does not** receive any points for using the wildcard (unlike all the other letters), though it **does** count as a used or unused letter when scoring.

During the game, a player wishing to use a wildcard should enter "*" (without quotes) instead of the intended letter. The word-validation code should not make any assumptions about what the intended vowel should be, but should verify that at least one valid word can be made with the wildcard as a vowel in the desired position.

The examples below show how wildcards should behave in the context of playing a hand, which you will implement in Problem 5 below. Don't worry about that part yet - just pay attention to how the wildcard is handled.

Example #1: A valid word made without the wildcard

```
Current Hand: c o w s * z
Enter word, or "!!" to indicate that you are finished: cows
"cows" earned 198 points. Total: 198 points

Current Hand: * z
Enter word, or "!!" to indicate that you are finished: !!
Total score: 198 points
```

Example #2: A valid word made using the wildcard

```
Current Hand: c o w s * z
Enter word, or "!!" to indicate that you are finished: c*ws
"c*ws" earned 176 points. Total: 176 points

Current Hand: o z
Enter word, or "!!" to indicate that you are finished: !!
Total score: 176 points
```

Example #3: An invalid word with a wildcard

```
Current Hand: c o w s * z
Enter word, or "!!" to indicate that you are finished: c*wz
That is not a valid word. Please choose another word.

Current Hand: o s
Enter word, or "!!" to indicate that you are finished: !!
Total score: 0 points
```

Example #4: Another invalid word with a wildcard

```
Current Hand: c o w s * z
Enter word, or "!!" to indicate that you are finished: *ows
That is not a valid word. Please choose another word.

Current Hand: c z
Enter word, or "!!" to indicate that you are finished: !!
Total score: 0 points
```

Modify the `deal_hand` function to support always giving one wildcard in each hand. Note that `deal_hand` currently ensures that one third of the letters are vowels and the rest are consonants. Leave the consonant count intact, and replace one of the vowel slots with the wildcard. You will also need to modify one or more of the constants defined at the top of the file to account for wildcards.

Then modify the `is_valid_word` function to support wildcards. **Hint:** Check to see what possible words can be formed by replacing the wildcard with other vowels. You may want to review the documentation for string module's `find()` function and make note of its behavior when a character is not found. The constant `VOWELS` defined for you at the top of the file may be helpful as well.

Testing: Make sure the `test_wildcard` tests pass. You may also want to test your implementation with some reasonable inputs.

Problem 5. Playing a hand

You are now ready to begin writing the code that interacts with the player. Implement the `play_hand` function. This function allows the user to play out a single hand. You'll first need to implement the helper function `calculate_handlen`, which can be done in under five lines of code.

To end the hand early, the player must type "!"(two exclamation points).

Note that after the line `# BEGIN PSEUDOCODE` there is a bunch of, well, pseudocode! This is to help guide you in writing your function. Check out the Why Pseudocode? resource to learn more about the What and Why of Pseudocode before you start this problem.

Testing: Try out your implementation as if you were playing the game: run your program and call the `play_hand` function from your shell with a hand and the `word_list`.

Note: Your output should match the examples below. You should not print extraneous `"None"` messages.

Example #1

```
Current Hand: a j e f * r x

Enter word, or "!!" to indicate that you are finished: jar

"jar" earned 90 points. Total: 90 points


Current Hand: * f x e

Enter word, or "!!" to indicate that you are finished: f*x

"f*x" earned 216 points. Total: 306 points


Current Hand: e

Enter word, or "!!" to indicate that you are finished: !!

Total score: 306 points
```

Example #2

```
Current Hand: a c f i * t x
Enter word, or "!!" to indicate that you are finished: fix
"fix" earned 117 points. Total: 117 points

Current Hand: a c t *
Enter word, or "!!" to indicate that you are finished: ac
That is not a valid word. Please choose another word.

Current Hand: t *
Enter word, or "!!" to indicate that you are finished: *t
"*t" earned 14 points. Total: 131 points

Ran out of letters. Total score: 131 points
```

Problem 6. Playing a game

A game consists of playing multiple hands. You need to implement two final functions to complete your word game. Implement the **`substitute_hand`** and **`play_game`** functions according to their specifications. For the game, you should use the **`HAND_SIZE`** constant to determine the number of letters in a hand.

Do **not** assume that there will always be 7 letters in a hand! The goal is to keep the code modular - if you want to try playing your word game with 10 letters or 4 letters you will be able to do it by simply changing the value of **`HAND_SIZE`**!

When implementing substitution, you might want to check the methods associated with dictionaries, such as **`.keys`**, or review the **`del`** keyword. You may also want to look at the code for **`deal_hand`** to see how **`random.choice`** can be used to select an element at random from a set of elements (such as a string).

Note that you are note provided pseudocode for this problem. However, as you are deciding how to implement these functions, you may want to write your own as a guideline.

Testing: Try out this implementation as if you were playing the game. Try out different values for **`HAND_SIZE`** with your program and be sure that you can play the word game with different hand sizes by modifying only the variable **`HAND_SIZE`**.

Example

```
Enter total number of hands: 2
Current hand: a c i * p r t
Would you like to substitute a letter? no

Current hand: a c i * p r t
Please enter a word or '!!' to indicate you are done: part
"part" earned 114 points. Total: 114 points

Current hand: c i *
Please enter a word or '!!' to indicate you are done: ic*
"ic*" earned 84 points. Total: 198 points

Ran out of letters
Total score for this hand: 198
----------
Would you like to replay the hand? no
Current hand: d d * l o u t

Would you like to substitute a letter? yes
Which letter would you like to replace: l

Current hand: d d * a o u t
Please enter a word or '!!' to indicate you are done: out
"out" earned 27 points. Total: 27 points

Current hand: d d * a
Please enter a word or '!!' to indicate you are done: !!
Total score for this hand: 27
----------
```

```
Would you like to replay the hand? yes
Current hand: d d * a o u t
Please enter a word or '!!' to indicate you are done: d*d
"d*d" earned 36 points. Total: 36 points

Current hand: a o u t
Please enter a word or '!!' to indicate you are done: out
"out" earned 54 points. Total: 90 points

Current hand: a
Please enter a word or '!!' to indicate you are done: !!
Total score for this hand: 90
----------
Total score over all hands: 288
```

Exercise 2

"

"

Submitting Your Lab

Submit all your Python files, and any other necessary files, in the appropriate folder in Google Drive as demonstrated by your instructor.