

Background Information

This assignment tests your understanding of and ability to apply the programming concepts we have covered throughout the unit. The concepts covered in the second half of the unit build upon the fundamentals covered in the first half of the unit.

Assignment Overview

You are required to design and implement two related programs:

- **“word_find.py”**, a CLI program that allows the user to play a word game. If the user scores high enough in the game, save data about the game in a text file. Develop this program before “log_viewer.py”.
- **“log_viewer.py”**, a GUI program that lets the user view the data in the text file. Develop this program after “word_find.py”.

The following pages describe the requirements of both programs in detail.

Starter files for both of these programs are provided along with this assignment brief, to help you get started and to facilitate an appropriate program structure. *Please use the starter files.*

The following pages describe the requirements of both programs in detail.

Overview of “word_find.py”

“word_find.py” is a program with a Command-Line Interface (CLI) like that of the programs we have created throughout the first half of the unit. The program can be implemented in under 175 lines of code (although implementing optional additions may result in a longer program). *This number is not a limit or a goal.* Everything you need to know in order to develop this program is covered in the first 7 modules. This program should be developed before “log_viewer.py”.

The program is a single-player word game, where the user enters as many words as they are able to think of using a selection of 9 letters. The user can play the game in “easy mode” or “hard mode”.

The program first selects 9 letters of the alphabet at random. The same letter can appear multiple times, and the selection process uses the frequency of letters in the game of Scrabble when choosing letters, so that the 9 letters chosen are more likely to contain letters that are common in English words, e.g. the letters are much more likely to contain “E”, “I” or “A” than “Z”, “Q” or “X”.

The program displays the 9 letters in a 3x3 grid and prompts the user for input:

```
Score: 0.  Your letters are:

      A | H | R
      -----
      T | R | Y
      -----
      J | E | H

Enter a word, [s]huffle letters, [l]ist words, or [e]nd game): |
```

Note: The grid is only for presentation – words do not need to be made up of adjoining letters.

When prompted for input, the user has the following options (these should *not* be case-sensitive):

1. Enter a word, e.g. "TRAY" would be a valid word given the letters pictured above.
 - The program will check that the word is valid and award points to the user if it is.
2. Enter "S" to randomly re-order the letters.
 - This is just to help the user think of more words, by seeing the letters in a new layout.
3. Enter "L" to display a list of the words that they have already entered during this game.
4. Enter "E" to end the game.

The program shows the grid of letters and prompts the user for input until they enter "E" to end the game. This allows them to keep entering words, shuffling the letters and viewing the list of words until they cannot think of any more words to enter, and choose to end the game.

Whenever the user enters a word, the program needs to check that:

1. The word has a minimum length of 3 characters.
2. The word has not already been used in this game.
3. The word is made up of the letters selected in this game.
4. The word is a recognised English word.

If the word is *not* valid, the program displays a message telling the user which piece of criteria was not met. If the game is being played in “hard mode”, this causes the game to end immediately. If the game is being played in “easy mode”, the program simply returns to the input prompt.

If the word *is* valid, the user is awarded points using the letter values in the game of Scrabble, where less common letters are awarded more points, e.g.



Note: The pictures of tiles are only to help you visualise how points are calculated – they do not appear in the game.

The program will use a Python package ‘pyenchant’ (feel free to use any other package if you are confident about it) to check if the word is a valid English word and to calculate the amount of points that it is worth. Follow these instructions to install and use this package.

Go to command prompt and type:

```
pip install pyenchant
```

Open your Python program and type:

```
import enchant
```

```
d = enchant.Dict("en_US") # US dictionary. Other option is en_GB
```

```
d.check("Hello") # checks if the input parameter is a valid English word.
```

Once the user enters “E” to end the game, the program should show their final score.

If their score is at least 50, the program should congratulate the user and record a “log” of the game in a text file named “logs.txt”. Use the “`json`” module to read and write data from/to the text file in JSON format (see Reading 7.1). Each log of a game should be a *dictionary* consisting of three keys:

- **“letters”**: the list of letters used in the game
- **“words”**: the list of words entered by the user
- **“score”**: the final score of the game

The logs should be stored in a list, resulting in the file containing a *list of dictionaries*. The example below demonstrates a list of two logs (the word list has been truncated to fit onto the page):

JSON

```
[
  {
    "letters": ["A", "A", "B", "D", "F", "I", "L", "O", "R"],
    "words":   ["AFRAID", "BALD", "BOAR", "BOLD", "DAB", "FAB", "FAIL", "FAR", ...],
    "score":   102
  },
  {
    "letters": ["B", "D", "I", "J", "L", "N", "O", "O", "S"],
    "words":   ["BIN", "BINS", "BOLD", "BOLDS", "DIN", "DINS", "JOB", "JOIN", ...],
    "score":   76
  }
]
```

The program should then show their final score, print “Thank you for playing!”, and end.

To help you visualise the program/game, here is an annotated screenshot of it being played:

```
Welcome to Word Find.
Come up with as many words as possible from the letters below!

Do you wish to play [e]asy mode or [h]ard mode? g
Invalid input, please select a mode.

Do you wish to play [e]asy mode or [h]ard mode? h
Hard mode selected. Entering an invalid word will end the game!

Score: 0. Your letters are:

  N | D | A
  ---
  A | E | O
  ---
  M | R | R

Enter a word, [s]huffle letters, [l]ist words, or [e]nd game): l
You have not yet entered any words.

Score: 0. Your letters are:

  N | D | A
  ---
  A | E | O
  ---
  M | R | R

Enter a word, [s]huffle letters, [l]ist words, or [e]nd game): dream
DREAM accepted - 8 points awarded. Your score is now 8.

Score: 8. Your letters are:

  N | D | A
  ---
  A | E | O
  ---
  M | R | R

Enter a word, [s]huffle letters, [l]ist words, or [e]nd game): mare
MARE accepted - 6 points awarded. Your score is now 14.

Score: 14. Your letters are:

  N | D | A
  ---
  A | E | O
  ---
  M | R | R

Enter a word, [s]huffle letters, [l]ist words, or [e]nd game): l

Previously entered words:
- DREAM
- MARE
```

The program welcomes the user and asks them to select a difficulty. Oops! They typed "g" the first time – they are re-prompted, and choose hard mode.

The program displays the letter grid and waits for input. The user entered "L", demonstrating what happens if you choose this option when you have not entered any words.

The user enters a couple of valid words, which are accepted and awarded points.

They enter "L" again, and are shown a list of the accepted words that they have entered.

Score: 14. Your letters are:

```
N | D | A
-----
A | E | O
-----
M | R | R
```

Enter a word, [s]huffle letters, [l]ist words, or [e]nd game): s

Shuffling letters...

Score: 14. Your letters are:

```
N | R | D
-----
M | O | R
-----
A | E | A
```

Enter a word, [s]huffle letters, [l]ist words, or [e]nd game): road

ROAD accepted - 5 points awarded. Your score is now 19.

Score: 19. Your letters are:

```
N | R | D
-----
M | O | R
-----
A | E | A
```

Enter a word, [s]huffle letters, [l]ist words, or [e]nd game): rod

ROD accepted - 4 points awarded. Your score is now 23.

Score: 23. Your letters are:

```
N | R | D
-----
M | O | R
-----
A | E | A
```

Enter a word, [s]huffle letters, [l]ist words, or [e]nd game): door

Invalid character(s) used! Game over!

Your final score was 23.

Thank you for playing!

The user enters "S" to shuffle the letters, making them appear in different positions when the letters are displayed from now on.

Another couple of valid words are entered...

The user enters an invalid word (there is only one "O" available) and the game ends since they are playing in hard mode.

Requirements of “word_find.py”

In the following information, numbered points describe a *requirement* of the program, and bullet points (in italics) are *additional details, notes and hints* regarding the requirement. Ask your tutor if you do not understand the requirements or would like further information. The requirements are:

1. The first things the program should do welcome the user, and then create some variables that will be needed in the game. The following variables are needed:
 - A *“score”* variable, to store the user’s score. This should be set to 0 at this point.
 - A *“used_words”* variable, to store the words that the user has entered. This should be set to an empty list at this point.
 - A *“letters”* variable, to store the letters selected for this game. Use the *“select_letters()”* function (detailed below) to produce an appropriate list of letters for this variable.
2. Next, the program should ask the user to select easy mode or hard mode by entering “E” or “H”. The program should re-prompt the user for a choice until they enter “E” or “H”. In easy mode, entering an invalid word does not end the game. In hard mode, an invalid word ends the game.
 - The user’s input should not be case-sensitive, e.g. “e” or “E” can be used to select easy mode.
 - If easy mode is selected, set a *“hard_mode”* variable to False. If hard mode is selected, set it to True. This variable will be used to determine what happens in Requirements 3.5 – 3.8.
3. The program should then enter a loop that will repeat until the user chooses to end the game. The body of this loop must:
 - 3.1. Print the user’s score and “Your letters are:”, display the letters in a 3x3 grid, and then get input from the user with the following prompt:
“Enter a word, [s]huffle letters, [l]ist words, or [e]nd game): ”

- Use the `display_letters()` function (detailed below) to display the letters in a 3x3 grid.
- Remember that the user's input should not be case-sensitive, e.g. "e" or "E" can end the game.
- Convert the user's input to uppercase and store it like that to make the following steps easier.

3.2. Start an "if/elif/else" statement to respond appropriately to the user's input. If the user enters "E", print "Ending game..." and end the loop (from Requirement 3).

- By ending this loop, the game is ended and the code reaches Requirement 4.

3.3. Otherwise, if the user enters "S", print "Shuffling letters..." and randomly re-order the `letters` list. This will re-position the letters in the 3x3 grid the next time it is displayed.

- The `random.shuffle()` function will do this. Remember that this function does not return a shuffled list – it actually shuffles the list that you pass it.

3.4. Otherwise, if the user enters "L", check if the `used_words` list is empty. If so, print an appropriate message. Otherwise, sort the list alphabetically and then print all of the words.

- Print "Previously entered words:" and show each word on a separate line, as per the screenshot.
- The `.sort()` list method can be used to sort a list.

Requirements 3.2 – 3.4 handle the “commands” that the user can enter – ending the game (“E”), shuffling the letters (“S”) or listing the used words (“L”). Any other input that they enter is considered to be a word – Requirements 3.5 – 3.8 are all about checking that the word is valid.

- 3.5.** Otherwise, if the user’s input is **less than 3 characters long**, print an appropriate message. If `hard_mode` is True, print “Game over!” and end the loop to end the game.
- 3.6.** Otherwise, if the user’s input is **in the `used_words` list**, print an appropriate message. If `hard_mode` is True, print “Game over!” and end the loop to end the game.
- Use an “in” comparison to check if the word is in the `used_words` list – see Lecture 3.
- 3.7.** Otherwise, if the user’s input is **not valid**, print an appropriate message. If `hard_mode` is True, print “Game over!” and end the loop to end the game.
- A valid word must be made up of the available `letters`.
 - Use the “`validate_word()`” function (detailed below) to determine if the word is valid. This function returns True or False – you can use it directly in the condition of the “`elif`” statement.
- 3.8.** Otherwise (this is the final “else” part of the “if/elif/else” statement), check if the user’s input is a recognised English word (this will be done by using “pyenchant” package) and obtain its Scrabble score. If the word is recognised, print a “word accepted” message that tells the user how many points it is worth, and add the points to their score, and append the word to the `used_words` list.
- If the word is not recognised, print an appropriate message. If `hard_mode` is True, print “Game over!” and end the loop to end the game.

4. Once the game ends (by choice or due to an invalid word in hard mode), print their final score. If their score is 50 or more, congratulate them and record a log of the game as detailed on Page 4 of this assignment brief. This will involve first opening a text file in read mode to read existing log data, appending a log of the game to the data, and then writing the entire data to the file.
 - First create a dictionary with keys of "letters", "words" and "score" and values of the `letters` variable, the `used_words` list and the `score` variable.
 - Then, try to open a file named "logs.txt" in read mode, use the "[`json.load\(\)`](#)" function to load JSON data from the file into a "logs" variable, and then close the file. If any exceptions occur, set `logs` to an empty list. This will occur the first time you try to record a log, since the file will not exist.
 - Finally, append the log dictionary to the `logs` list, and then open "logs.txt" in write mode, use the "[`json.dump\(\)`](#)" to write the `logs` list to the file in JSON format, and then close the file.
 - It may seem that appending data to the file would be easier, but this approach is more appropriate and less prone to errors due to the data being stored in JSON format.
5. Finally, print "Thank you for playing!"
 - This concludes the core requirements of "word_find.py". The following pages detail the functions mentioned above and optional additions and enhancements that can be added to the program.

Functions in “word_find.py”

The program requirements above mentioned 3 user-defined functions - “select_letters()” (Requirement 1), “display_letters()” (Requirement 3.1), and “validate_word()” (Requirement 3.7). As part of “word_find.py”, you must define and use these functions.

1. The “select_letters()” function does not take any parameters. The function randomly selects 9 letters and returns them as a list of 9 strings. The same letter can appear multiple times, and the selection process uses the frequency of letters in Scrabble when choosing letters.
 - *This function requires a small amount of somewhat confusing code that involves concepts/functions not covered in the unit content, so I have **completed this function for you in in the starter file**.*
 - *You do not need to change this function definition in any way. You simply need to use the function in your code when implementing Requirement 1.*

2. The “display_letters()” function takes one parameter named “letters” (the list of 9 letters selected at the start of the game). The function should print the letters in a 3x3 grid, as shown on Page 3 of this assignment brief. The function should not return anything.
 - *This function simply involves some print statements and concatenation of individual letters from the letters list with spaces and the characters “-” and “|”.*
 - *You will need to refer to the letters in the letters list, i.e. “letters[0]” to “letters[8]”.*
 - *Task 2 of Workshop 4 involves creating a function that is quite similar to this function.*

3. The `validate_word()` function takes two parameters named `word` (the word that the user entered) and `letters` (the list of 9 letters selected at the start of the game). The function should check that the `word` is made up entirely of letters in the `letters` list. A letter cannot be used more times than it appears in the `letters` list. If the `word` is valid, the function should return `True`. If it is not valid, the function should return `False`.
- *There are various ways that you can approach this function, most of which will involve looping through each character of the `word`.*
 - *There will be a post on the Blackboard discussion board to help you with this function.*

The definitions of these functions should be at the start of the program (as they are in the starter file provided), and it should be called where needed in the program. Revise Module 4 if you are uncertain about defining and using functions, and be sure to implement them so that they receive and return values exactly as described above.

Ensure that the functions do exactly what is specified above and nothing more – it is important to adhere to the stated specifications of a function when working on a programming project.

You are welcome to write additional functions if you feel they improve your program, but be sure to consider the characteristics and ideals of functions as outlined in Lecture 4.

Optional Additions and Enhancements for “word_find.py”

Below are some suggestions for minor additions and enhancements that you can make to the program to further test and demonstrate your programming ability. They are *not required* and you can earn full marks in the assignment without implementing them.

- Make it so that after the game ends, the program asks the user whether they would like to play again. If so, start the game again (with new letters). This will involve putting most of the program into the body of a loop.
- Modify the “`select_letters()`” function so that it ensures that the list of 9 characters contains at least one vowel, and only contains a Q if it also contains a U.
- Modify the “`display_letters()`” function so that it uses [Unicode box-drawing characters](#) to print a prettier grid, such as the one pictured. The characters you’ll need are “`┌`”, “`—`”, “`┐`”, “`└`”, “`|`”, “`├`”, “`┤`”, “`┴`”, “`┬`”, “`┴`” and “`┬`”. You should be able to copy and paste them directly into strings in your code!
- When recording a log of a game, include two additional key-value pairs in the log dictionary: The user’s name (prompt them to enter it) and the date (use the “[datetime](#)” module). Make sure that these items are shown in the `log_viewer.py` program (detailed below).

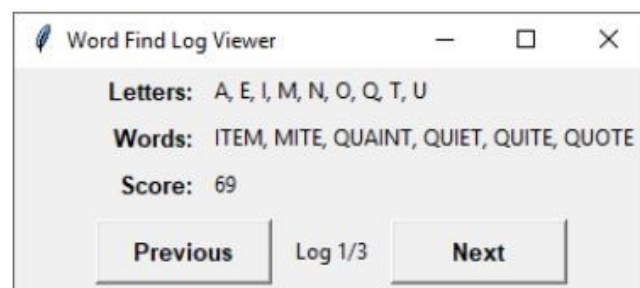
N	O	I
O	T	T
R	C	G

Overview of “log_viewer.py”

“log_viewer.py” is a program with a Graphical User Interface (GUI), as covered in Module 9. It should be coded in an Object Oriented style, as covered in Module 8. Everything you need to know in order to develop this program is covered in the first 9 modules of the unit. This program should be developed after “word_find.py”.

The entirety of this program can be implemented in under 125 lines of code (although implementing optional additions may result in a longer program). *This number is not a limit or a goal.* You must use the “tkinter” module to create the GUI, and you will also need to use the “tkinter.messagebox” and “json” modules.

This program uses the data from the “logs.txt” file. The program should load all of the data from the file *once only* - when the program begins. The program simply allows the user to view the logs created by “word_find.py”.



The only way that the user can interact with the program is by pressing the “Previous” and “Next” buttons, which change the log data that is currently displayed. If they click “Previous” when on the first log or “Next” when on the final log, a messagebox should appear, alerting them of the issue:



The following pages detail how to implement the program.

Constructor of the GUI Class of “log_viewer.py”

The **constructor** (the “`__init__`” method) of your GUI class must implement the following:

1. Create the main window of the program and give it a title of “Word Find Log Viewer”.
 - *You are welcome to set other main window settings to make the program look/behave as desired.*
2. Try to open the “logs.txt” file in read mode and load the JSON data from the file into an attribute named “`self.logs`”, and then close the file.
 - *If any exceptions occur (due to the file not existing, or it not containing valid JSON data), show an error messagebox with a “Missing/Invalid file” message and use the “`destroy()`” method on the main window to end the program. Include a “`return`” statement in the exception handler after destroying the main window to halt the constructor so that the program ends cleanly.*
3. Create a “`self.current_log`” attribute to keep track of which log is currently being displayed in the GUI, and set it to 0.
 - *This attribute represents an index number in the list of logs loaded from the text file (`self.logs`).*

4. Use `Frame`, `Label` and `Button` and widgets from the “`tkinter`” module to implement the GUI depicted on the previous page.
 - *You will save time if you design the GUI and determine exactly which widgets you will need and how to lay them out before you start writing the code.*
 - *You are welcome change the layout/appearance, as long as the functionality is implemented.*
 - *See Reading 9.1 for information regarding various settings that can be applied to widgets to make them appear with the desired padding, colour, size, etc.*
 - *The “`fill`” and “`anchor`” settings may be useful – See Reading 9.1 to find out how to use them!*
 - ***Do not set the text for the labels that will contain log data at this point – they will be set in the “`show_log()`” method. Only set the text of labels that do not change during the program.***

5. Lastly, the constructor should end by calling the “`show_log()`” method to display a the first log in the GUI, and then call “`tkinter.mainloop()`” to start the main loop.
 - *To call a method of the class, you need to include “`self.`” at the start, e.g. “`self.show_log()`”.*

That is all that the constructor requires. The following pages detail the methods mentioned above, and some optional additions and enhancements.

Methods in the GUI class of “log_viewer.py”

Your GUI class requires three methods to implement the functionality of the program - “show_log()”, “previous_log()” and “next_log()”. As part of the GUI class of “log_viewer.py”, you must define and use these methods.

1. The “show_log()” method is responsible for displaying the details of a log in the GUI. It displays the log at `self.current_log`'s index number in the `self.logs` list. It is called at the end of the constructor, and by the `previous_log()` and `next_log()` methods.
 - The `self.logs` attribute is a list of dictionaries. Referring to a specific index number of the list will obtain the dictionary of that log. See Page 4 for details regarding the keys of a log dictionary.
 - The dictionary contains a list of letters and a list of words, which you need to turn into comma-separated strings to display in the GUI. The “`.join()`” string method will be very useful here.
 - The “`configure()`” method (see Reading 9.1) can be used on a widget to change the text it displays. Alternatively, you can use a `StringVar` to control the text of a widget (see Lecture 9).
 - As well as changing the text of the labels displaying the letters, words and score, remember to change the text between the buttons that shows which log you are viewing, e.g. “Log 1/3”.
2. The “previous_log()” method is called when the user clicks the “Previous” button. It should subtract 1 from `self.current_log` and then call the `show_log()` method, or show a messagebox if the program is already displaying the first log.
 - The `self.current_log` attribute will be 0 if the program is displaying the first log.

3. The “`next_log()`” method is called when the user clicks the “Next” button. It should add 1 to `self.current_log` and then call the `show_log()` method, or show a messagebox if the program is already displaying the last log.
 - *Determining the index number of the last log will involve using the length of the `self.logs` list.*

These three methods are all that are required to implement the functionality of the program, but you may write/use additional methods if you feel they improve your program.

Note that it is possible (but trickier in some ways) to implement the functionality without the `previous_log()` and `next_log()` methods. You are welcome to do so if you are able!

Optional Additions and Enhancements for “log_viewer.py”

Below are some suggestions for minor additions and enhancements that you can make to the program to further test and demonstrate your programming ability. They are *not required* and you can earn full marks in the assignment without implementing them.

- Prevent the button positions from jumping about when the length of the word list causes the width of the window to change. This can be achieved by giving the main window a fixed width, and using the “wraplength” setting on the `Label` containing the word list to make it split the text across multiple lines.
- Add a “Statistics” button to the GUI and create a method that is called when it is clicked. The method should determine statistics about the logs and show them in a messagebox. Example statistics include the highest score, average score, most words and fewest words.
- Add “First” and “Last” buttons to the GUI which immediately go to the first and last log. You can use Unicode characters (copy them from [this table](#)) instead of text on the buttons.
- Add a “Delete Logs” button to the GUI. When clicked, a “`askyesno()`” messagebox should be appear to make sure that the user wants to delete the logs. If they click “Yes”, delete the “logs.txt” file, show a confirmation message, and end the program.