README.md

Assignment 4 - The Guessing Game

BACKGROUND

This assignment will explore the connections between strings and character lists. Both strings and lists are types of sequences in Python. However, the two types have important differences. One major difference is that lists are mutable (meaning that they can be changed) while strings are not mutable.

In this assignment, you will implement The Guessing Game. This game asks a user for a series of guesses to try to identify a randomly chosen target word. At the beginning of the game, the user only knows how many characters are in the word. The user has a limited number of guesses (5, by default) for characters that are not in the target word. An example of one round of the game is shown below.

```
Welcome to The Guessing Game!

==============================

Word:   _ _ _ _ _ _ _ _ _

Misses:

Guess:  a

==============================

Word:   a _ _ _ _ _ _ _ _

Misses:

Guess:  o

==============================

Word:   a _ _ _ _ _ _ _ _

Misses: o

Guess:  r

==============================

Word:   a _ _ _ r _ _ _ _

Misses: o

Guess:  m

==============================

Word:   a _ _ _ r _ _ _ _

Misses: om

Guess:  e
```

```
==============================

Word:   a _ _ e r _ _ _ e

Misses: om

Guess:  n

==============================

Word:   a _ _ e r _ _ _ e

Misses: omn

Guess:  s

==============================

Word:   a s s e r _ _ _ e

Misses: omn

Guess:  t

==============================

Word:   a s s e r t _ _ e

Misses: omn

Guess:  v

==============================

Word:   a s s e r t _ v e

Misses: omn

Guess:  i

YOU GOT IT!

==============================

Word:   a s s e r t i v e

Misses: omn

Play again (Y/N)? N

Goodbye.
```

This example demonstrates the case where the user successfully guessed the target word. After each guess, the state of the game is updated by revealing where the guess exists in the target word when the guess is successful or adding the guess to the set of misses when the guess is unsuccessful. A message indicating the success or failure to guess the word in a given round is given when the round ends. Detailed instructions for how to implement this program are below.

Clone this repository through PyCharm to complete the program on your local computer.

Let's get started.

TABLE OF CONTENTS

STEP 0 - READ THE README

Before writing a single line of Python code for this assignment, you are **strongly** encouraged to read this document in its entirety. The document includes key details about the expected implementation of your program. Diving into the implementation based solely on the description of the program above is likely to result in following a path towards your program's implementation that fails to meet the requirements of this assignment.

If you would like to consume this document in an alternative format, the README has also been converted to a PDF which is available in this repository.

DECOMPOSITION

This program requires implementing 13 functions. This may seem like an intimidating number at first. However, the function definitions are mostly short. The large number of functions is the result of a program design with a heavy emphasis on functional decomposition. When the described functions are implemented, the composition of the full program will be more manageable.

INSTRUCTIONS

Restrictions: No Python container datatypes (dict, set, etc) other than list and tuple are allowed in the implementation of this program

Your program will maintain two variables that are updated throughout the round: `display_chars` and `misses`. Both variables are lists of characters. Which variable is updated during the course of a round depends on the guess provided by the user. Between rounds, these variables will need to be re-initialized so that the lists represent the state of the current round.

Step A - Replace characters in target word with blanks

Functions: `blank_chars(word)`

```
def blank_chars(word):
    """Returns a list of underscore characters with the same length as word.
```

```
    :param word: target word as a string
    :return: a list of underscore characters ('_')

    >>> blank_chars("happiness")
    ['_', '_', '_', '_', '_', '_', '_', '_', '_']
    """
```

In **Step A**, the goal is to implement `blank_chars()`. `blank_chars()` has 1 parameter. The function creates a list of underscore characters (`_`) that is the same length as `word`. This implies that when the empty string (`''`) is passed to this function as an argument value, the empty list (`[]`) will be returned. The list created in `blank_chars()` is returned by the function. Creation of this list from the target word will occur at the beginning of **each round** after a new target word has been chosen.

```
>>> blank_chars("happiness")
['_', '_', '_', '_', '_', '_', '_', '_', '_']
```

Once `blank_chars()` has been implemented, try submitting your `guessing_game.py` file to Gradescope to see if your implementation passes the **Step A** tests. Once your program passes the **Step A** tests, proceed to **Step B**.

Step B - Add spaces for displaying the revealed characters

Functions: `space_chars(chars)`

```
def space_chars(chars):
    """Returns a string with the characters in chars list separated by spaces.

    :param chars: a list of characters
    :return: a string containing characters in chars with intervening spaces

    >>> space_chars(['h', '_', 'p', 'p', '_', 'n', '_', '_', '_'])
    'h _ p p _ n _ _ _'
    """
```

Step B requires defining the `space_chars()` function. This function should be defined with 1 list parameter named `chars`. The return value of this function is a string containing the same characters in `chars` but with spaces inserted between each of the characters in `chars`. This implies that given an empty list (`[]`) as a parameter argument, `space_chars()` will return the empty string (`''`). In addition, `chars` list with one character (e.g., `['s']`) would return a single-character string (e.g., `s`) with no spaces included. An example `space_chars()` function call follows:

```
>>> space_chars(['h', '_', 'p', 'p', '_', 'n', '_', '_', '_'])
'h _ p p _ n _ _ _'
```

A method call to the `str` method `join()` will be useful in the function definition of `space_chars()`. `space_chars()` is used to output the current state of the `display_chars` list at the beginning of each round and after each guess in The Guessing Game. The full state of the game will be output by a function that you will implement in a later portion of this assignment.

Once `space_chars()` has been implemented, try submitting your `guessing_game.py` file to Gradescope to see if your implementation passes the **Step B** tests. Once your program passes the **Step B** tests, proceed to **Step C**.

Step C - Prompt user for a valid guess

Function: `get_guess()`

```
def get_guess():
    """Prompts the user for a guess to check for the game's current word. When the user
```

```
    enters input other than a single character, the function prompts the user again
    for a guess. Only when the user enters a single character will the prompt for
    a guess stop being displayed. The function returns the single-character guess
    entered by the user.

    :return guess: a single character guessed by user
    """
```

Each time the user makes a guess that guess is checked to see if the letter is present in the word. For this step, implement the following:

1. Write a function named `get_guess()`. This function should be defined without parameters. The purpose of this function is to prompt the user for a guess by displaying `Guess:` followed by a tab character `\t`.
2. If the user's guess is not a **single alphabetic character**, prompt the user for a guess again until a single alphabetic character is input by the user. The `str` class method `isalpha()` is useful for determining if a single-character string is an alphabetic letter.
3. When the user enters a valid guess, the guess will be returned (as a lowercase single-character string) by the `get_guess()` function call.

```
>>> get_guess()
Guess:  ABC
Guess:  hello
Guess:  *
Guess:  F
'f'
```

Notice that the function only returns a valid value when a single alphabetic character is input. The return value of the function call is a lowercase letter. No blank line should be output after the `Guess:` prompt.

Once `get_guess()` has been implemented as described above, all **Step C** tests should pass. Try submitting your `guessing_game.py` file to Gradescope to see if your implementation passes the **Step C** tests. Once your program passes the **Step C** tests, proceed to **Step D**.

Step D - Check the target word for user's guess

Function: `check_guess(word, guess)`

```
def check_guess(word, guess):
    """Returns a list of positions where guess is present in word.
    An empty list should be returned when guess is not a single
    character or when guess is not present in word.


    :param word: target word as a string
    :param guess: a single character guessed by user
    :return positions: list of integer positions
    """
```

Next, define a function named `check_guess()`. This function is defined with the parameters `word` and `guess` as shown above. `word` is a string representing the target word that the user is attempting to identify in the current round. `guess` is the character that the user has guessed in the current round. This function should return a list of the indices in `word` where the single-character string `guess` is found. An example will help to clarify what is required.

```
>>> target_word = "happiness"
>>> guess = 'p'
>>> check_guess(target_word, guess)
```

```
[2, 3]
```

The single-character string `p` is located at indices `2` and `3` in the word `happiness`. Therefore, the return value from the function call shown in the example is the list `[2,3]`. When `word` does not contain `guess` at any index, `check_guess()` should return the empty list (`[]`).

```
>>> target_word = "happiness"
>>> guess = 'q'
>>> check_guess(target_word, guess)
[]
```

An empty list should also be returned in the case that `guess` is not a single character.

```
>>> target_word = "happiness"
>>> guess = "pp"
>>> check_guess(target_word, guess)
[]
```

```
>>> target_word = "happiness"
>>> guess = ''
>>> check_guess(target_word, guess)
[]
```

Once `check_guess()` has been implemented as described above, all **Step D** tests should pass. Try submitting your `guessing_game.py` file to Gradescope to see if your implementation passes the **Step D** tests. Once your program passes the **Step D** tests, proceed to **Step E**.

Step E - Update the state of the game based on user's guess

Functions: `update_chars(chars, guess, positions)`, `add_to_misses(misses, guess)`, and `update_state(chars, misses, guess, positions)`

```
def update_chars(chars, guess, positions):
    """Updates the list of characters, chars, so that the characters
    at the index values in the positions list are updated to the
    character guess.


    :param chars: a list of characters
    :param guess: a single character guessed by user
    :param positions: list of integer positions
    :return: None
    """
```

```
def add_to_misses(misses, guess):
    """Adds the character guess to the misses list.

    :param misses: list of guesses not present in target word
    :param guess: a single character guessed by user
    :return: None
    """
```

```
def update_state(chars, misses, guess, positions):
    """Updates the state of the game based on user's guess. Calls the function update_chars() when
    the positions list is not empty to reveal the indices where the character guess is present. Calls the
    function add_to_misses() when the positions list is empty to add guess to the misses list.

    :param chars: a list of characters
    :param misses: list of guesses not present in target word
    :param guess: a single character guessed by user
    :param positions: list of integer positions
    :return: None
    """
```

With the completion of `check_guess()` in **Step D**, we can now determine whether or not the user's guess is present in the target word. Two possible outcomes are possible:

1. the `list` returned by `check_guess()` contains the positions where `guess` occurs in the target word
2. the `list` returned by `check_guess()` is empty and, therefore, `guess` was a miss

Step E focuses on functions that deal with these two possibilities. Implement `update_chars()` according to the function definition displayed above. This function has 3 parameters. `chars` is the list of display characters for the current round. `guess` is the single character most recently entered by the user. `positions` is a list of position values where `guess` occurs in `chars`. This function updates `chars` so that each index value in the `positions` list is updated to the single-character string referenced by `guess`. After an `update_chars()` function call is complete the character at each index value in `positions` will be set to `guess` in the list `chars`. This function does not return a value. The following example is illustrative of the functionality of `update_chars()`.

```
>>> display_chars = ['_', '_', '_', '_', '_']
>>> guess = 'l'
>>> positions = [2, 3]
>>> update_chars(display_chars, guess, positions)
>>> display_chars
['_', '_', 'l', 'l', '_']
```

When the `positions` list returned by `check_guess()` is empty, the `add_to_misses()` function will be called. This function has 2 parameters. `misses` is a list of characters guessed by the user that are not present in the current round's target word. `guess` references the single-character string most recently entered by the user. After an `add_to_misses()` function call is complete, the `misses` list will contain one extra character than it did prior to the function call. This extra character will be the single-character string referenced by `guess`. This function does not return a value.

```
>>> misses = ['a','b','c']
>>> guess = 'd'
>>> add_to_misses(misses, guess)
>>> misses
['a', 'b', 'c', 'd']
```

`update_chars()` and `add_to_misses()` will be called by the `update_state()` function. This function has 4 parameters. `chars` is the list of display characters for the current round. `guess` is the single character guessed most recently by the user. `positions` is a list of index values where `guess` occurs in `chars`. `misses` is a list of characters guessed by the user that are not present in the current round's target word. A function call to `update_state()` will either perform a function call to `update_chars()` **or** `add_to_misses()` depending on the value of `positions`. When `positions` is not empty, `update_chars()` is called. An example function call is demonstrated below:

```
>>> display_chars = ['a', 'p', 'p', '_', 'e']
>>> guess = 'l'
>>> positions = [3]
```

```
>>> misses = ['d', 'x', 'r', 'y']
>>> update_state(display_chars, misses, guess, positions)
>>> display_chars
['a', 'p', 'p', 'l', 'e']
>>> misses
['d', 'x', 'r', 'y']
```

When `positions` is empty, `add_to_misses()` is called. An example function call is demonstrated below:

```
>>> display_chars = ['a', 'p', 'p', '_', 'e']
>>> guess = 's'
>>> positions = []
>>> misses = ['d', 'x', 'r', 'y']
>>> update_state(display_chars, misses, guess, positions)
>>> display_chars
['a', 'p', 'p', '_', 'e']
>>> misses
['d', 'x', 'r', 'y', 's']
```

Once `update_chars()`, `add_to_misses()`, and `update_state()` have been implemented as described above, all **Step E** tests should pass. Try submitting your `guessing_game.py` file to Gradescope to see if your implementation passes the **Step E** tests. Once your program passes the **Step E** tests, proceed to **Step F**.

Step F - Determine if the round has ended

Function: `is_round_complete(chars, misses)`

```
def is_round_complete(chars, misses):
    """Indicates whether or not a round has ended. This function returns True
    when the user has successfully guessed the target word or exceeds the
    number of allowed misses. Otherwise, the function returns False,
    indicating that the round is not complete. A message revealing the
    user's success or failure guessing the target word is output by this
    function when the round is complete.


    :param chars: a list of characters
    :param misses: list of guesses not present in target word
    :return status: True when round is finished, False otherwise
    """
```

Your program now includes functionality for accepting a guess from a user, checking the guess against a target word, updating the displayed characters, and registering a missed guess. In **Step F**, you will implement `is_round_complete()` which determines when a round of The Guessing Game is complete. A round ends in two scenarios:

1. the user successfully guesses all of the characters in the target word
2. the user exceeds the maximum number of guesses in the round

A constant variable `MAX_MISSES` is defined at the top of the starter code. **Do not delete this value**. The `is_round_complete()` function will return `True` in two situations:

1. the number of characters in `misses` **exceeds** `MAX_MISSES`
2. no underscore characters (`_`) remain in `chars` list

In all other cases, `is_round_complete()` returns `False`. Let's explore some examples.

```
>>> display_chars = ['_', '_', '_', '_', '_']
>>> misses = ['b', 'c', 'x', 'd']
>>> MAX_MISSES
4
>>> is_round_complete(display_chars, misses)
False
```

The `is_round_complete()` function call returned `False` because, in this case, `MAX_MISSES` is 4 but only 4 misses have occurred so far **and** there are blanks (`_`) remaining in `display_chars`. However, if on the next round the user guesses `z`, the function call to `is_round_complete()` outputs `SORRY! NO GUESSES LEFT.` and returns `True` because the number of guesses exceeds `MAX_MISSES`. Therefore, the round **is** complete.

```
>>> display_chars = ['_', '_', '_', '_', '_']
>>> misses = ['b', 'c', 'x', 'd', 'z']
>>> MAX_MISSES
4
>>> status = is_round_complete(display_chars, misses)
<BLANKLINE>
SORRY! NO GUESSES LEFT.
>>> print(status)
True
```

Notice that a blank line is output before the `SORRY! NO GUESSES LEFT.` message is output. In the case, that the user has guessed all of the characters in the target word and not exceeded `MAX_MISSES`, the round is complete as well. In this case, `YOU GOT IT!` is output and the `is_game_complete()` function call returns `True`.

```
>>> display_chars = ['a', 'p', 'p', 'l', 'e']
>>> misses = ['b', 'c', 'x', 'd']
>>> MAX_MISSES
4
>>> status = is_round_complete(display_chars, misses)
<BLANKLINE>
YOU GOT IT!
>>> print(status)
True
```

Again, notice that a blank line is output before the `YOU GOT IT!` message is output.

Once `is_round_complete()` has been implemented as described above, all **Step F** tests should pass. Try submitting your `guessing_game.py` file to Gradescope to see if your implementation passes the **Step F** tests. Once your program passes the **Step F** tests, proceed to **Step G**.

Step G - Process words file and select target word

Functions: `read_words(filepath)` and `get_word(words)`

```
def read_words(filepath):
    """Opens a file of word located at filepath, reads the file of words line by line,
    and adds each word from the file to a list. The list is returned by the
    function

    :param filepath: path to input file of words (one per line)
    :return word_list: list of strings contained in input file
    """
```

```
def get_word(words):
    """Selects a single word randomly from words list and returns it.

    :param words: list of strings
    :return word: string from words list
    """
```

The group of functions required for implementing the full Guessing Game program are almost complete. One crucial aspect of the program yet to be implemented is the generation of the target word to be used in each round. The `read_words()` function has one parameter named `filepath`. `filepath` is a string representing the location of a file of words (one word per line). `read_words()` opens the file located at `filepath` and reads each word into a list. `read_words()` returns this list when the function call completes. `read_words()` must also close the file after reading the words from the file.

```
$ cat small_word_file.txt
tycoon
small
assertive
courage
dirty
```

As an example, the file `small_word_file.txt` contains 5 words as displayed above. Passing this file's name as a string to a `read_words()` function call, results in the `read_words()` function call returning a list consisting of each word from the file.

```
>>> filepath = 'small_word_file.txt'
>>> read_words(filepath)
['tycoon', 'small', 'assertive', 'courage', 'dirty']
```

Be sure to remove any newline characters (`\n`) from the lines of the file when reading in the file. The list of words will need to be assigned to a variable as this list will be used in each round of The Guessing Game.

`get_word()` returns a single, randomly chosen word from the list of words returned by `read_words()`. This function should use the `randrange()` function from the `random` module to select an index number between 0 and the integer that is one less than the number of words in the list. The word located at the randomly-chosen index value in the list is the one that `get_word()` will return via a `get_word()` function call.

```
>>> words = ['mist', 'ideal', 'bait', 'sheet', 'supply']
>>> get_word(words)
'mist'
>>> get_word(words)
'sheet'
>>> get_word(words)
'bait'
```

The `get_word()` function calls above demonstrate the random selection of words from the list provided as the function's argument.

After implementing `read_words()` and `get_word()` properly, all **Step G** tests should pass. Try submitting your `guessing_game.py` file to Gradescope to see if your implementation passes the **Step G** tests. Once your program passes the **Step G** tests, proceed to **Step H**.

Step H - Determine if user want's to play again

Function: `is_game_complete()`

```
def is_game_complete():
    """Prompts the user with "Play again (Y/N)?". The question is repeated
    until the user enters a valid response (one of Y/y/N/n). The function
    returns False if the user enters 'Y' or 'y' and returns True if the user
    enters 'N' or 'n'.

    :return response: boolean representing game completion status
    """
```

Playing The Guessing Game happens in rounds. However, the user determines how many rounds of a single game will be played. Therefore, at the completion of a round, the program needs to determine whether to start a new round or end the game. The `is_game_complete()` function has no parameters and returns `True` when the user indicates the desire to stop playing. `is_game_complete()` returns `False` when the user indicates the desire to play again. This function uses the prompt `Play again (Y/N)?` to determine if the user wants to continue playing or end the game. The user must enter `Y` or `y` to play again. The user enters `N` or `n` to end the game. Any other input from the user causes the prompt `Play again (Y/N)?` to be displayed again until a valid input values is provided by the user.

```
>>> is_game_complete()
Play again (Y/N)? x
Play again (Y/N)? 8
Play again (Y/N)? y
False
```

Take note that no blank lines are included after the prompt is displayed. The example above demonstrates that while the user is prompted to enter `Y` to play again, `y` is also accepted. When the user enters `n` or `N` the function will return `True`.

```
>>> is_game_complete()
Play again (Y/N)? n
True
```

After implementing the `is_game_complete()` function, all **Step H** tests should pass. Try submitting your `guessing_game.py` file to Gradescope to see if your implementation passes the **Step H** tests. Once your program passes the **Step H** tests, proceed to **Step I**.

Step I - Perform one round of the game

Function: `run_guessing_game(words_filepath)`

```
def run_guessing_game(words_filepath):
    """Controls running The Guessing Game. This includes parsing
    the words file and executing multiple rounds of the game.

    :param words_filepath: the location of the file of words for the game
    :return: None
    """
```

Congratulations on reaching this point in the development of The Guessing Game program! You now have all of the components necessary for implementing the program. The flow of the program will be controlled by the `run_guessing_game()` function. This function requires one parameter: `words_filepath`. `words_filepath` is the location of the words file that will be used in the program. The value that you should use for this parameter is available in the program through the `str` variable `filepath` which has been defined for you in the `main()` function of the starter code file `guessing_game.py`.

Step I focuses on execution of a single round of The Guessing Game. A single round of The Guessing Game includes the following:

1. Using command-line arguments, the program will be provided the location of a text file with a single word on each line. The code

for storing this file's location in your program is provided for you in the `main()` function.

2. The words in the file are parsed from the file and stored in a list **once** at the beginning of `run_guessing_game()` function.
3. A random word is selected from the list of words which becomes the target word for the round.
4. The list of revealed characters is initialized to a list of underscore characters of the same length as the target word.
5. The misses list is initialized to an empty list.
6. The state of the game (with no characters revealed and no misses listed) should be displayed to the user (using the provided function `display_game_state()`) before getting the first guess.
7. After displaying the game's state, the user is prompted for a guess.
8. For each guess:
    i. The guess is checked to see if the guess is in the word.
    ii. The state (revealed characters and misses) of the game is updated based on whether the guess is in the word.
    iii. The state of the game is displayed to the user.
    iv. The state of the game is checked to see if the round is complete
    v. When the round is not complete, the user is prompted for a new guess.
9. When the round is complete, a message revealing the outcome is displayed. This message is either (`YOU GOT IT!` or `SORRY! NO GUESSES LEFT.`). This outcome message is followed by outputting the final state of the game with the target word fully revealed and the final contents of the misses list displayed as a string.

Prior to getting input from the user, your program will **always** display the current state of the game. This means displaying the characters in the target word that have been guessed correctly as well as the misses. The `display_game_state()` function requires two parameters `chars` and `misses`. `chars` is the list of display characters for the current round. `misses` is the list of characters guessed by the user that are not present in the current round's target word.

`display_game_state()` will output these lists as seen in the example below:

```
>>> display_chars = ['s', 'c', '_', '_', '_', 'l']
>>> misses = ['t', 'z', 'p']
>>> display_game_state(display_chars, misses)

==========================

Word:   s c _ _ _ l

Misses: tzp
```

This function has been implemented for you so that you only need to provide the appropriate arguments for a `display_game_state()` function call. It is your responsibility to properly make use of `display_game_state()` function calls to output the state of the game at the correct points in your program.

Example output from the implementation of `run_guessing_game()` at this step is shown below. Both possible outcomes are shown. The first shows the user guessing the target word (`assertive`) correctly.

```


============================

Word:   _ _ _ _ _ _ _ _ _

Misses:

Guess:  a

============================
```

```
Word:   a _ _ _ _ _ _ _ _

Misses:

Guess:  o

==============================

Word:   a _ _ _ _ _ _ _ _

Misses: o

Guess:  r

==============================

Word:   a _ _ _ r _ _ _ _

Misses: o

Guess:  m

==============================

Word:   a _ _ _ r _ _ _ _

Misses: om

Guess:  e

==============================

Word:   a _ _ e r _ _ _ e

Misses: om

Guess:  n

==============================

Word:   a     e r       e

Misses: omn

Guess:  s

==============================

Word:   a s s e r _ _ _ e

Misses: omn

Guess:  t

==============================

Word:   a s s e r t _ _ e

Misses: omn

Guess:  v
```

```
==============================

Word:   a s s e r t _ v e

Misses: omn

Guess:  i

YOU GOT IT!

==============================

Word:   a s s e r t i v e

Misses: omn
```

The second example shows the user exeeding the maximum number of incorrect guesses when the target word is `courage`. Here `MAX_MISSES` is 5.

```
==============================

Word:   _ _ _ _ _ _ _

Misses:

Guess:  c

==============================

Word:   c _ _ _ _ _ _

Misses:

Guess:  m

==============================

Word:   c _ _ _ _ _ _

Misses: m

Guess:  l

==============================

Word:   c _ _ _ _ _ _

Misses: ml

Guess:  p

==============================

Word:   c _ _ _ _ _ _

Misses: mlp
```

```
Guess:  r

==============================

Word:   c _ _ r _ _ _

Misses: mlp

Guess:  n

==============================

Word:   c _ _ r _ _ _

Misses: mlpn

Guess:  k

==============================

Word:   c     r

Misses: mlpnk

Guess:  h

SORRY! NO GUESSES LEFT.

==============================

Word:   c o u r a g e

Misses: mlpnkh
```

When `run_guessing_game()` is able to perform a single round of The Guessing Game as demonstrated in this step, all **Step I** tests should pass. Try submitting your `guessing_game.py` file to Gradescope to see if your implementation passes the **Step I** tests. Once your program passes the **Step I** tests, proceed to **Step J**.

Step J - Allow multiple rounds to be played

Function to update: `run_guessing_game(words_filepath)`

Yes. That's right. We are not quite done with the `run_guessing_game()` function. At this point, your program can execute one round of The Guessing Game before the program ends. However, the final program requires that the user can play as many games as the user desires. After completing **Step J**, your program will have this functionality. Specifically, your program will pass the tests for this step when the following occurs:

At the start of the program, `Welcome to The Guessing Game!` should be output once by a `run_guessing_game()` function call.

When the program ends, `Goodbye.` should be the final message output by a `run_guessing_game()` function call. A blank line should be output before the `Goodbye.` message such that the last lines of the program will appear as follows when the user chooses to end the program:

```
Play again (Y/N)? n

Goodbye.
```

Update your definition of `run_guessing_game()`, such that the following requirements are met:

1. At least one round of The Guessing Game must be run each time your program is executed.
2. Each round of the game selects a random word from the list of words included in the words file provided when the program is executed. This word becomes the target word for the round.
3. The list of misses is reset to an empty list before the start of each round so that no misses are recorded when a new round begins.
4. After completing a round, the program asks the user whether or not to play again using the prompt `Play again (Y/N)?`. Use `is_game_complete()` for displaying this prompt.
5. When answering this question affirmatively (by entering `Y` or `y`), a new round of the game starts.
6. When the user indicates a desire to end the game (by entering `N` or `n`), the program displays **one blank line** followed by `Goodbye.` before ending.

Below is an example run of the program demonstrating a game with two rounds. The user indicates not wanting to continue the game after the second round.

```
$ python guessing_game.py word_file.txt
Welcome to The Guessing Game!

==============================

Word:   _ _ _ _ _ _ _

Misses:

Guess:  a

==============================

Word:   _ _ _ _ a _ _

Misses:

Guess:  b

==============================

Word:   _ _ _ _ a _ _

Misses: b

Guess:  g

==============================

Word:   _ _ _ _ a g _

Misses: b

Guess:  p

==============================

Word:   _ _ _ _ a g _

Misses: bp

Guess:  d
```

```
==============================

Word:   _ _ _ _ a g _

Misses: bpd

Guess:  c

==============================

Word:   c _ _ _ a g _

Misses: bpd

Guess:  o

==============================

Word:   c o _ _ a g _

Misses: bpd

Guess:  r

==============================

Word:   c o _ r a g _

Misses: bpd

Guess:  u

==============================

Word:   c o u r a g _

Misses: bpd

Guess:  e

YOU GOT IT!

==============================

Word:   c o u r a g e

Misses: bpd

Play again (Y/N)? y

==============================

Word:   _ _ _ _ _

Misses:

Guess:  p

==============================

Word:   _ _ _ _ _
```

```
Misses: p

Guess:  i

==============================

Word:   _ i _ _ _

Misses: p

Guess:  t

==============================

Word:   _ i _ t _

Misses: p

Guess:  n

==============================

Word:   _ i _ t _

Misses: pn

Guess:  w

==============================

Word:   _ i _ t _

Misses: pnw

Guess:  s

==============================

Word:   _ i _ t _

Misses: pnws

Guess:  l

==============================

Word:   _ i _ t _

Misses: pnwsl

Guess:  y

==============================

Word:   _ i _ t y

Misses: pnwsl

Guess:  o

SORRY! NO GUESSES LEFT.
```

```
==============================

Word:   d i r t y

Misses: pnwslo

Play again (Y/N)? n

Goodbye.
```

Make sure to call `run_guessing_game()` (with the location of the words file passed as an argument) from within the program's `main()` function.

When `run_guessing_game()` has been updated to enable multiple rounds of The Guessing Game as demonstrated in this step, all **Step J** tests should pass. Try submitting your `guessing_game.py` file to Gradescope to see if your implementation passes the **Step J** tests. Once your program passes the **Step J** tests, proceed to **Step K**.

Step K - Handle an invalid file location

Function to update: `run_guessing_game(words_filepath)`

Whenever files are read by a program, you should be careful to first check that the file to be read actually exists. This is especially important when the file name is provided as a command-line argument by a user of the program. When an attempt to open a file that does not exist at the location on your file system is provided to the `open()` function, Python will complain with a `FileNotFoundError`.

To avoid your program ending with this error when the user of your program provides a non-existent file path, add a `try` statement to the `run_guessing_game()` function that informs the user that the file location provided to the program does not represent a valid file. When an invalid file location string is passed to `read_words()`, your program will output

```
The provided file location is not valid. Please enter a valid path to a file.
```

A function call for such a case is demonstrated below (assuming the user entered the file location `nonexistent_file.txt`):

```
>>> run_guessing_game("nonexistent_file.txt")
The provided file location is not valid. Please enter a valid path to a file.
```

`run_guessing_game()` should return from the function call after this message is displayed to the user. The `try` statement should be executed before outputting the welcome message to the user.

In the case that a `try` statement is not included in the `run_guessing_game()` function, the same function call will result in an exception message similar to the one below:

```
>>> run_guessing_game("nonexistent_file.txt")
FileNotFoundError: [Errno 2] No such file or directory: 'nonexistent_file.txt'
```

When this final update to run_guessing_game() is made, all **Step K** tests should pass. This indicates that you program is functioning as expected. Congratulations!

EVALUATION

This assignment will use Gradescope's Autograder tool. Therefore, you will receive immediate feedback when submitting this assignment. You should strive to have your submitted code pass all of the tests run by the Gradescope Autograder by the time you have completed **Step K**.

It is just as important to write clean and readable code as it is to write correct and functional code. A portion of your grade on this assignment will be based on how well you follow certain coding conventions. Make sure to follow the standards discussed in class, and before you submit your assignment, take a minute to review your code to check for stylistic issues like those mentioned below.

Comments

To make your program easier to read, you should add comments before (docstrings) and inside your functions to make your intention clearer. Good comments give the reader a clue about what a function does and, in some cases, how it works.

Be sure to include header comments (including your name) for the submitted file.

Function design

Function names should include verbs indicating what is accomplished by the function. Most of the functions in this assignment have been named for you. However, if you add any additional functions to further decompose your solution, be sure to name these functions appropriately:

- function names should include verbs
- all words in a function name should be written in lower-case letters
- multiple words in a function name should be separated by underscores

Functions should be decomposed such that the length of each is small. This is an important part of proper decomposition. There is a bit of an art to this but you should ensure that your functions solve well-defined sub-problems. The function definitions should not, in general, have 10s of lines of codes. This is especially true for the programs written in this class.

Variable names

Variable naming conventions are as follows:

- names should be descriptive
- all characters in the name should be written in lowercase (with the exception of constant variables)
- multiple words should be separated by underscores ('_').

Spacing

Indentation should be consistent in the files that are submitted. The convention is that each block of indented code is indented using four spaces. Your program should follow this convention.

Spaces should be used between operators and operands in the expressions written in your code.

Magic Numbers

Program should not contain any constant numeric values. Constant variables are to be defined and used in place of hard-coding numeric constants into the program.

ASSIGNMENT SUBMISSION

You should submit the following file on Gradescope (do not include any files not included in the list below):

- **guessing_game.py**

The Autograder will assign points to passed tests. These results should be used as a guide for keeping you on track towards an implementation of this assignment that meets all of the requirements laid out above.

Good luck!