Assignment 5: Clever Hangman

CompSci 101 Fall 2021

Due: November 16, 2021

Overview

You will adapt your Hangman program to play Clever Hangman. You will create a **`CleverHangman.py`** module using your **`Hangman.py`** module and the **`lowerwords.txt`** file provided in the Hangman assignment.

Clever Hangman is in some ways the same as Hangman -- and you should start with your program that plays hangman that you wrote for the Hangman assignment. ***Do not import*** **`Hangman.py`**; instead, **make a new project, copy your Hangman module into this project, and rename it to** **`CleverHangman.py`**.

If you don't have a working Hangman program, we have released a form you can fill out to request a working Hangman assignment. NOTE: If you request this program, then you cannot get credit for the Assignment 4 program ***unless you've submitted the code*** that will be graded before you request a working Hangman module. Also we will not release the code until after the grace period for Assignment 4. It would be much better for you to go to office/consulting hours and get Hangman working before then. The form can be found here.

Completing the Assignment

There is a **required reading quiz** for this assignment that can be found on Sakai.

You must create a module **`CleverHangman.py`** that allows the user to play Hangman and guess a secret word using the clever/cheating method outlined below. See the sample runs and the requirements for details. See the class lecture about the clever hangman assignment for details. You will modify two of the functions from your **`Hangman.py`** module and create four new functions for this assignment.

For full credit, you should have no global variables.

There are a few main ideas to use in creating the clever version of Hangman:

1. Divide the list of possible secret words into groups, each corresponding to a word template, and choose the largest group as the source for the secret word that the user is trying to guess from.
2. Develop the program using a smaller word list at first to be sure that your code is working properly.

Requirements

There is a **required reading quiz** for this assignment that can be found on Sakai. This quiz is to help ensure you understand what this assignment is asking you to do and some of the key concepts in this assignment. You have unlimited tries and it will count towards your reading quiz grade. **Note this quiz is due the day the assignment is due, and does not have a grace day.** The intent of these assignment reading quizzes are that you **do them early to check your understanding** of the assignment. You can try it right after reading through this assignment!

There are also two sets of requirements for game play: those carried over from the Hangman assignment, and those new to this assignment.

Requirements from Hangman

1. The user must be given a choice of (e)asy or (h)ard which are 12 and 8 misses until hung, respectively. You ***do not need*** to check for bad user-input. For example, you can assume the user will enter an 'h' or an 'e'. See the example runs below where the default is 12 for easy and the user can enter 'h' for hard.
2. The user must be told how many misses are left on each turn when the user enters a letter. When a letter in the word is correctly guessed the number of misses left doesn't change, but an incorrectly-guessed letter changes the number of misses left.
3. Guessing a letter that was already guessed should not count as a miss and should not count as a guess when reporting summary game statistics.
4. When the user ***has guessed the word***, or the number of ***misses allowed is reached***, the game should be over and *individual game summary statistics* about the number of guesses and misses should be printed. See the example runs below.
5. On each turn, the secret word should be printed with underscores for unknown letters and correctly-guessed letters in place as warranted. A single space should appear between each underscore and letter.
6. After each Clever Hangman game, the user should be prompted to play again. The user can play as many games as they want. Once they decide to quit, *summary statistics for the entire "session" (multiple games)* should be printed, showing the total win-loss count.
    a. You should implement this functionality in the main program block.

Requirements New to Clever Hangman

1. The user must be asked how many letters in the word to be guessed and then a word from all words of that length ***must be chosen at random*** from these words as the secret word. ***This is different from the Hangman program*** where the number of letters in the secret word was chosen at random. ***You do not need to*** guard against bad user input.
2. The user must be ***shown all the letters that have not yet been guessed***, and these letters must appear in ***sorted order*** (see the example runs below). ***This is different***

from the Hangman program where the user was shown letters already guessed, rather than letters not yet guessed.

3. You must make the program "clever" in that after each letter guessed, the list of possible secret words is as large as possible. This is described in detail below. As part of making the program clever, ***you must write and call two functions*** described below.
4. Inside **`runGame()`**, you must include a local debug variable. When debug is set to True, the game's console output should provide information about the list of possible secret words and how dictionary keys/values are used in creating a new list of possible secret words after each letter guessed. ***The user should be prompted*** as to whether the debug value is True (for mode) or False (for play mode).
    a. The debug variable will be initialized in **`runGame()`** using the return value of the new function **`handleUserInputDebugMode()`**. The debug variable will also be passed as a parameter to the new function **`getNewWordList()`**.
    b. For more on debug mode, see the Note on debug mode, handleUserInputDebugMode(), and Making the Program Clever and Hard to Win.

Overview of Changes to Functions

The following Hangman functions are no longer needed and can be deleted:

- **`getWord`**
- **`updateHangmanWord`**
- **`processUserGuess`**

The following functions should be kept as-is from Hangman:

- **`handleUserInputDifficulty`**
- **`handleUserInputLetterGuess`**

The following functions from Hangman will be modified:

- **`createDisplayString`**
- **`runGame`**

The following new functions will be implemented:

- **`handleUserInputDebugMode`**
- **`handleUserInputWordLength`**
- **`createTemplate`**
- **`getNewWordList`**
- **`processUserGuessClever`**

Hangman Functions To Modify

`createDisplayString(lettersGuessed, missesLeft, hangmanWord)`

Instead of returning the letters the user has guessed, this function will now return the letters the user has NOT guessed. This is done by printing all the letters of the alphabet (in alphabetical sorted order) and replacing the letters that the user has guessed with an empty space (" ").

For example:
Assume the user has guessed the letters “b”, “e”, and “u”.
`letters not yet guessed: a cd fghijklmnopqrst vwxyz`
Note that there is one space between the “:” and the letter “a”.

`runGame(filename)`

This function needs to be modified to call the new functions that you will be creating for this Clever Hangman assignment. This function should also print (or cause to be printed) additional information to the console when the local variable debug is set to True. More information on debug mode is explained in the Note on debug mode, **`handleUserInputDebugMode()`**, and Making the Program Clever and Hard to Win.

New Functions to Implement

`handleUserInputDebugMode()`

This function will prompt the user if they wish to play in debug mode. True is returned if the user enters the letter “d”, indicating debug mode was chosen; False is returned otherwise.

Parameters:
No parameters

Returns:
Type: bool
True is returned if the user wishes to play in debug mode, false otherwise.

Example:

```
Which mode do you want: (d)ebug or (p)lay: d
```

`handleUserInputWordLength()`

The length of secretWord is no longer randomized, instead the user will be asked how long secretWord should be. You do not need to check for bad user input.

Parameters:
No parameters

Returns:
Type: int
The length specified by the user.

Assumptions:
The user will input a value between 5 to 10 (inclusive).

Example:

```
How many letters in the word you'll guess: 8
```

`createTemplate(currTemplate, letterGuess, word)`

This function will create a new template for the secret word that the user will see. For example, `"_ _ a _"` is the currentTemplate for the word "play" with the "a" guessed, but without spaces between the underscores and the letters. The spaces were added for readability and ***should not*** be included in your Clever Hangman. This function will modify currentTemplate to reflect letterGuess. This new template should be consistent with the currentTemplate and word.

Parameters:

- currTemplate (type: string) - the current displayed hangman template in which each character in the string represents a letter in the secret word. It is either actually a letter, or "_", representing that the user has not yet guessed that letter.
- letterGuess (type: string) - The user's current guess, represented as a string of length 1, i.e. "a".
- word (type: string) - the current secret word.

Returns:

Type: String
The new template is returned, which is consistent with the previous template and word.

Example:

`createTemplate('tr__', 'u', 'true')` returns `'tru_'`

`getNewWordList(currTemplate, letterGuess, wordList, debug)`

This function constructs a dictionary of strings as the key to lists as the value. It does this by calling **`createTemplate`** on every word in **`wordList`** with **`currTemplate`** and **`letterGuess`**. The string **`createTemplate`** returns is a key into the dictionary and the current word from **`wordList`** is added to the list that maps to that key.

After creating the dictionary, this function returns the (key, value) pair with the longest list. ***If there is a tie, it should ensure that the template returned is the one with the most underscores in it*** and therefore the user cannot win on that turn.

Also see Writing Function getNewWordList and Making the Program Clever and Hard to Win.

Parameters:

- currTemplate (type: string) - the current displayed hangman in which each character in the string represents a letter in the secret word and it is either an actually letter or "_", representing that the user has not yet guessed that letter.
- letterGuess (type: string) - The user's current guess, represented as a string of length 1
- wordList (type: list of strings) - list of possible secret words for Clever Hangman
- debug (type: boolean) - whether or not the game is running in debug mode. See note on debug mode here.

Returns:

Type: (string, list)
A 2-tuple is returned in which:

- The first element is the template corresponding to the largest group of words
- The second element is the list of words that match this template

`processUserGuessClever(guessedLetter, hangmanWord, missesLeft)`

Takes the user's guess, the user's current progress on the word, and the number of misses left; updates the number of misses left and indicates whether the user missed.

Parameters:

- guessedLetter (type: string) - the user's current guess, represented as a string of length 1, the return value of handleUserInputLetterGuess
- hangmanWord (type: list of strings) - represents the currently displayed hangman word, where each element in the list represents a letter in the secret word and it is either the actual letter or "_" representing that the user has not yet guessed that letter
- missesLeft (type: int) - the number of misses the user has left

Returns:

Type: list
A list with the following at each index:

- Index 0: (type: int) an updated value for missesLeft based on the user's guess in guessedLetter
- Index 1: (type: bool) indication of whether the user made a correct guess, where True means the user guessed a letter in the word and False means the user missed

Assumptions:

guessedLetter has the value returned from **`handleUserInputLetterGuess`**, which means that it is guaranteed to be a letter that the user has not already guessed. **`hangmanWord`** is the list version of the template that was returned by **`getNewWordList`**.

Understanding How Functions Fit Together

As in Hangman, it is up to you to determine how the functions (both old and new) should fit together into a working game. Although some functions are unchanged from regular Hangman, how/when they should be called in **`runGame`** may change in the context of Clever Hangman.

For example, before calling **`processUserGuessClever`** on each turn, your Clever Hangman code should already have called **`getNewWordList`** and used its return value to select a new template. This new template will be passed to **`processUserGuessClever`** as a list, and your code should determine if the guess was a miss based on the guessed letter's presence in your *updated* template.

Note on Debug Mode

Remember that the local variable debug will be initialized in **`runGame`** using the return value of

the function **handleUserInputDebugMode**. The variable debug will also be passed as a parameter to the function **getNewWordList**.

As described above, Clever Hangman will print extra information to the console if the user plays in debug mode. The extra information can be seen in Example Clever Hangman Runs. Inside **runGame** and **getNewWordList**, you will use if statements to decide whether certain information should be printed, i.e.:

```
if debug:
    print("# words left is", len(wordList))
```

Making the Program Clever and Hard to Win

When the user guesses a letter in the Clever Hangman program, the computer will change the secret word it's thinking of if that will make the game harder for the human player. ***However, any changes must be consistent with all previous guesses.***

The user/player isn't aware that the computer is changing the secret word (unless debug mode has been employed), because each time the player guesses, the computer will change the secret word in a consistent way that's not apparent to the player except perhaps via a frustration factor as the word seems harder and harder to guess.

You can see how the computer does this from the debugging output shown in the clever hangman demo partially reproduced below. To cause the game to print the output that shows the current word and number of possible words, modify the return value of **createDisplayString** in **runGame**.

Note that this output joins a game in which the player has already guessed five letters that don't appear in the word: 'a', 'i', 's', 'e', and 'o'. The debugging output shows that the computer's secret word is "*truly*" chosen from among 87 words that could have been the secret word. The user has no letters correct and then guesses a 'u'.

The computer now divides its 87 words into groups: the groups are based on the existing word template which is five underscores, and the letter guessed: 'u'. There are three groups: one with 'u' as the third letter, one with 'u' as the second letter, and one with no occurrences of 'u'. These groups contain 32, 47, and 8 letters, respectively, as shown in the debugging output below. ***Be sure to sort the templates alphabetically before printing in debug mode!***

```
letters not yet guessed:  bcd fgh jklmn pqr tuvwxyz
misses remaining = 3

_ _ _ _ _
(word is truly)
# possible words: 87
```

```
letter> u
_____ : 8
__u__ : 32
_u___ : 47
# keys = 3
```

Remember that the words the computer chooses from, as well as each template corresponding to a group, must be consistent with all guesses made so far. This means none of the words can contain an 'a', 'i', 's', 'e', or 'o'. In particular, the group with no occurrences of 'u' includes words like "dryly" and "crypt", but does *not* include "psych" because that has an 's' in it and the player has been told there are no occurrences of 's'.

The numbers shown are the size of each group, so 'u' as the third letter includes 32 words such as bluff and blurb, but not trust because there is no 's'. The group with 'u' as the second letter has 47 words in it including "buddy", "bunch", and "furry", but not "rusty" because there is no 's'. ***You must write code to create the groups and count the words in each group as part of the function `getNewWordList`.***

The computer chooses the group with the most words, where 'u' is the second letter. It then continues to play.

```
letters not yet guessed:  bcd fgh jklmn pqr t vwxyz
misses remaining = 3
_ u _ _ _
(word is duchy)
# possible words: 47
letter> b
bu___ : 14
_u___ : 33
# keys = 2
you missed: b not in word

letters not yet guessed:   cd fgh jklmn pqr t vwxyz
misses remaining = 2
_ u _ _ _
(word is puppy)
# possible words: 33
```

Note that after guessing a 'u' correctly, the computer has chosen "*duchy*" as its secret word, but in fact it has chosen the entire group/list of words with second letter 'u' and no occurrences of 'a', 'i', 's', 'e', or 'o' as the secret word -- a group consisting of 47 words that are returned by the function `getNewWordList`.

The player guesses 'b' and the computer finds all possible templates for each word in the existing list of 47 words. There are two such templates: those with 'b' as the first letter and those with no occurrences of 'b'. There are no words with 'b' in them other than in the first position because words like numbs and cuban aren't in the list of 47 words being considered.

Again the computer chooses the largest list/group, the size of which is 33, and continues to play. Notice that the number of possible words continues to decrease. When the player guesses 'c' there are four templates:

```
letter> c
cu___ : 2
_uc__ : 2
_u_c_ : 8
_u___ : 21
# keys = 4
you missed: c not in word

letters not yet guessed:    d fgh jklmn pqr t vwxyz
misses remaining = 1
_ u _ _ _
(word is furry)
# possible words: 21
```

In summary, the important step in writing Clever Hangman is to divide the list of possible words into disjoint or non-overlapping groups and choose the largest group of words to continue to play. On each turn, the group must be consistent with previous guesses, but that's pretty simple to do: Note that every time you call **`getNewWordList`** with a wordList parameter, the function will return a smaller list of possible words. On the next turn, you will pass this smaller list to **`getNewWordList`** and further reduce the list of possible words. ***In this way, your code will make sure that the list of possible secret words is always consistent with all guesses up to that point in the game.***

Writing Function **`getNewWordList`**

This function has a list of possible secret words, the current template for the secret word (a string) the current single-letter guess, and a debug boolean as parameters. The function returns a 2-tuple. The first element of the return tuple is a new template (string), and the second element of the return tuple is the new list of possible secret words, each of which matches the template. The computer will "choose" one of these words as the secret-word; it doesn't matter which, since all words matching the template are equivalent. You'll use a dictionary in creating the tuple to return.

The main idea is to use a dictionary in which each possible template/string is a key and the

corresponding value is a list of words that matches that template.
For example, the list of words matching

```
't _ t _ _'
```

Is this list

```
[titan, tithe, title, titus, total, tutor]
```

and words matching

```
'_ _ _ t t'
```

Is this list (not all words may be recognizable as typical English words).

```
[brett, burtt, hiatt, knott, pratt, scott, wyatt]
```

Creating the Dictionary

To create this dictionary, you iterate over every possible word in parameter **`wordList`** (***initially this is all words, but the list of possible words changes after each guess made by the player***). You should compare each possible word to the parameter **`currentTemplate`** and the parameter **`letterGuess`** by the user. This combination of word from parameter **`wordList`**, **`currentTemplate`**, and **`letterGuess`** creates a key in the dictionary -- you get this key by calling the function **`createTemplate`** once for each word in parameter **`wordList`**.

For example, suppose the player is guessing a four-letter word with no letters guessed, the current guess is 'O', and the list of possible words as shown. Since no letters are guessed you'll have these three parameters to **`getNewWordList`**:

- The string template would be **`"_ _ _ _"`** (no spaces between underscores)
- The letter guessed is **`'O'`**
- The list of words is **`[ "OBOE", "NOON", "ODOR", "ROOM", "SOLO", "TRIO", "GOTO", "OATH", "OXEN", "PICK", "FRAT", "HOOP"]`**

The (key, value) pairs in the dictionary you'll code in **`getNewWordList`** are formed as follows by iterating over the words in **`wordList`**. Since no letters have been guessed, the template is **`"_ _ _ _"`** . You'll write code to create a dictionary key from the template, the word, and the guessed letter by passing them to function **`createTemplate`** and getting the string that's the key in the dictionary as the return value.

In the table here, the left column has parameters to the function **`createTemplate`**. You see the

template (with spaces added for clarity), each word from the list of possible words, and the letter guessed. Write code using the template, the letter, and the word to return a string that's a key in the dictionary as shown below. The value in the dictionary associated with the key is a list of words that match the key.

Parameters to `createTemplate`	**Return value: key in dictionary**	**Value associated with key in dictionary of function `getNewWordList`**
`" _ _ _ _ ", "OBOE", "O"`	`"O _ O _"`	`["OBOE"]`
`" _ _ _ _ ", "NOON", "O"`	`"_ O O _"`	`["NOON"]`
`" _ _ _ _ ", "ODOR", "O"`	`"O _ O _"`	`["OBOE", "ODOR"]`
`" _ _ _ _ ", "ROOM", "O"`	`"_ O O _"`	`["NOON", "ROOM"]`
`" _ _ _ _ ", "SOLO", "O"`	`"_ O _ O"`	`["SOLO"]`
`" _ _ _ _ ", "TRIO", "O"`	`"_ _ _ O"`	`["TRIO"]`
`" _ _ _ _ ", "GOTO", "O"`	`"_ O _ O"`	`["SOLO", "GOTO"]`
`" _ _ _ _ ", "OATH", "O"`	`"O _ _ _"`	`["OATH"]`
`" _ _ _ _ ", "OXEN", "O"`	`"O _ _ _"`	`["OATH", "OXEN"]`
`"_ _ _ _", "PICK", "O"`	`"_ _ _ _"`	`["PICK"]`
`"_ _ _ _", "FRAT", "O"`	`"_ _ _ _"`	`["PICK", "FRAT"]`
`"_ _ _ _", "HOOP", "O"`	`"_ O O _"`	`["NOON, "ROOM", "HOOP"]`

This would create a dictionary with six (key,value) pairs as shown below.

Key	Value
`"O _ O _"`	`["OBOE", "ODOR"]`
`"_ O O _"`	`["NOON", "ROOM", "HOOP"]`
`" _ O _O"`	`["SOLO", "GOTO"]`
`"_ _ _ O"`	`["TRIO"]`
`"O _ _ _"`	`["OATH", "OXEN"]`
`"_ _ _ _"`	`["PICK", "FRAT"]`

The largest collection of values (most words) corresponds to key `"_ O O _"`, so the program would pick a secret word at random from the list of three words: `["NOON", "ROOM", "HOOP"]` and the template for the game would be `"_ O O _"` with three possibilities. These two values, the string template and the corresponding list of words, are returned as a 2-tuple by `getNewWordList`. Find the value (list of strings) with the maximal length, and then return the key associated with this list and the list as elements of a 2-tuple like this:

```
("_ O O _", ["NOON, "ROOM", "HOOP"] )
```

The player may think she has hit the jackpot with two O's in the word, and that may be true, but there are more words to eliminate than with any other template.

Printing the Dictionary Contents in Debug Mode

If debug is True, the game's console output should provide information about how the dictionary keys/values are used in creating a new list of possible secret words. For each template in the dictionary (sorted lexicographically/alphabetically), you will ***print the template, the length of the word list corresponding to that template, and the total number of keys (templates)***. An example for the dictionary in the previous section would be:

```
O_O_ : 2
O___ : 2
_OO_ : 3
_O_O : 2
___O : 1
____ : 2
# keys = 6
```

Implementing and Debugging Tips

Develop the program using a smaller word list at first to be sure that your code is working properly.

For example: Like Hangman, the completed game will draw words from **`lowerwords.txt`**. However, it may be hard to debug your program with that many possible words. One idea would be to create a smaller word file and use that instead of **`lowerwords.txt`** when debugging your game.

Example Clever Hangman Runs

Session with 2 Games

In the first game, the user loses on hard mode with debug set to True. In the second game, the user again loses on hard mode but with debug set to False.

```
Which mode do you want: (d)ebug or (p)lay: d
How many letters in the word you'll guess: 6
How many misses do you want? Hard has 8 and Easy has 12
(h)ard or (e)asy> h

letters not yet guessed: abcdefghijklmnopqrstuvwxyz
misses remaining = 8

_ _ _ _ _ _
(word is flamer)
# possible words: 6166
letter> a
______ : 3441
_____a : 80
____a_ : 233
___a__ : 316
___a_a : 11
__a___ : 549
__a__a : 19
__a_a_ : 10
__aa__ : 1
_a____ : 962
_a___a : 39
_a__a_ : 57
_a_a__ : 40
```

```
_a_a_a : 12
_a_aa_ : 3
a_____ : 273
a____a : 21
a___a_ : 30
a__a__ : 32
a__a_a : 3
a_a___ : 26
a_a__a : 7
aa____ : 1
# keys = 23
you missed: a not in word

letters not yet guessed:  bcdefghijklmnopqrstuvwxyz
misses remaining = 7

_ _ _ _ _ _
(word is mounds)
# possible words: 3441
letter> o
______ : 2105
_____o : 23
____o_ : 147
____oo : 1
___o__ : 148
___o_o : 1
___oo_ : 4
__o___ : 228
__o__o : 2
__o_o_ : 8
__oo__ : 32
_o____ : 528
_o___o : 6
_o__o_ : 41
_o_o__ : 15
_o_o_o : 1
_o_oo_ : 1
_oo___ : 77
_oo_oo : 1
o_____ : 60
o___o_ : 3
o__o__ : 8
o__oo_ : 1
# keys = 23
```

```
you missed: o not in word

letters not yet guessed:  bcdefghijklmn pqrstuvwxyz
misses remaining = 6

_ _ _ _ _ _
(word is burkes)
# possible words: 2105
letter> u
______ : 1441
_____u : 2
____u_ : 36
___u__ : 84
___u_u : 1
__u___ : 107
_u____ : 362
_u__u_ : 13
_u_u__ : 11
u_____ : 37
u___u_ : 5
u__u__ : 5
u_u___ : 1
# keys = 13
you missed: u not in word

letters not yet guessed:  bcdefghijklmn pqrst vwxyz
misses remaining = 5

_ _ _ _ _ _
(word is wilted)
# possible words: 1441
letter> i
______ : 503
_____i : 2
____i_ : 54
___i__ : 158
___i_i : 2
__i___ : 225
__i__i : 1
__i_i_ : 7
__ii__ : 2
_i____ : 355
_i__i_ : 28
_i_i__ : 56
_i_i_i : 2
```

```
i_____ : 28
i__i__ : 16
i_i___ : 2
# keys = 16
you missed: i not in word

letters not yet guessed:  bcdefgh jklmn pqrst vwxyz
misses remaining = 4

_ _ _ _ _ _
(word is served)
# possible words: 503
letter> e
______ : 2
_____e : 5
____e_ : 13
___e__ : 9
___e_e : 2
___ee_ : 5
__e___ : 42
__e__e : 12
__e_e_ : 23
__ee__ : 36
__ee_e : 9
_e____ : 13
_e___e : 13
_e__e_ : 160
_e__ee : 2
_e_e__ : 59
_e_e_e : 7
_e_ee_ : 3
_ee___ : 6
_ee__e : 5
_ee_e_ : 34
e____e : 1
e___e_ : 5
e___ee : 2
e__e__ : 20
e__ee_ : 2
e_e___ : 9
e_e__e : 1
e_e_e_ : 3
# keys = 29
```

```
letters not yet guessed:  bcd fgh jklmn pqrst vwxyz
misses remaining = 4
_ e _ _ e _
(word is tested)
# possible words: 160
letter> s
_e__e_ : 100
_e__es : 16
_e_se_ : 11
_e_ses : 3
_es_e_ : 13
_esse_ : 5
_esses : 1
se__e_ : 7
se__es : 2
se_se_ : 1
se_ses : 1
# keys = 11
you missed: s not in word

letters not yet guessed:  bcd fgh jklmn pqr t vwxyz
misses remaining = 3
_ e _ _ e _
(word is kepler)
# possible words: 100
letter> r
_e__e_ : 45
_e__er : 32
_e_re_ : 1
_er_e_ : 8
_er_er : 6
_erre_ : 1
_errer : 1
re__e_ : 3
re__er : 2
re_re_ : 1
# keys = 10
you missed: r not in word

letters not yet guessed:  bcd fgh jklmn pq  t vwxyz
misses remaining = 2
_ e _ _ e _
(word is wedded)
```

```
# possible words: 45
letter> d
_e__e_ : 11
_e__ed : 20
_e_de_ : 2
_e_ded : 4
_ed_e_ : 1
_ed_ed : 2
_edded : 2
de__e_ : 1
de__ed : 2
# keys = 9

letters not yet guessed:  bc  fgh jklmn pq  t vwxyz
misses remaining = 2
_ e _ _ e d
(word is belted)
# possible words: 20
letter> l
_e__ed : 10
_el_ed : 4
_elled : 5
le__ed : 1
# keys = 4
you missed: l not in word

letters not yet guessed:  bc  fgh jk mn pq  t vwxyz
misses remaining = 1
_ e _ _ e d
(word is vented)
# possible words: 10
letter> t
_e__ed : 4
_e_ted : 1
_etted : 4
te_ted : 1
# keys = 4
you missed: t not in word
you're hung!!
word was begged
you made 10 guesses with 8 misses

Do you want to play again? y or n> y
```

```
Which mode do you want: (d)ebug or (p)lay: p
How many letters in the word you'll guess: 7
How many misses do you want? Hard has 8 and Easy has 12
(h)ard or (e)asy> h

letters not yet guessed: abcdefghijklmnopqrstuvwxyz
misses remaining = 8
_ _ _ _ _ _ _
letter> e
you missed: e not in word

letters not yet guessed: abcd fghijklmnopqrstuvwxyz
misses remaining = 7
_ _ _ _ _ _ _
letter> a
you missed: a not in word

letters not yet guessed:  bcd fghijklmnopqrstuvwxyz
misses remaining = 6
_ _ _ _ _ _ _
letter> i

letters not yet guessed:  bcd fgh jklmnopqrstuvwxyz
misses remaining = 6
_ _ _ _ i _ _
letter> o
you missed: o not in word

letters not yet guessed:  bcd fgh jklmn pqrstuvwxyz
misses remaining = 5
_ _ _ _ i _ _
letter> u

letters not yet guessed:  bcd fgh jklmn pqrst vwxyz
misses remaining = 5
_ u _ _ i _ _
letter> m
you missed: m not in word

letters not yet guessed:  bcd fgh jkl n pqrst vwxyz
misses remaining = 4
_ u _ _ i _ _
letter> a
```

```
you already guessed that
letter> n

letters not yet guessed:  bcd fgh jkl   pqrst vwxyz
misses remaining = 4
_ u _ _ i n _
letter> g

letters not yet guessed:  bcd f h jkl   pqrst vwxyz
misses remaining = 4
_ u _ _ i n g
letter> c
you missed: c not in word

letters not yet guessed:  b d f h jkl   pqrst vwxyz
misses remaining = 3
_ u _ _ i n g
letter> d
you missed: d not in word

letters not yet guessed:  b   f h jkl   pqrst vwxyz
misses remaining = 2
_ u _ _ i n g
letter> r
you missed: r not in word

letters not yet guessed:  b   f h jkl   pq st vwxyz
misses remaining = 1
_ u _ _ i n g
letter> s
you missed: s not in word
you're hung!!
word was pulping
you made 12 guesses with 8 misses

Do you want to play again? y or n> n
You won 0 game(s) and lost 2
```

Submitting and Grading

1. Log in to Gradescope.
2. Under the CompSci 101 dashboard, click Assignment 5: Clever Hangman.

3. Click Submit Programming Assignment, click “Click to browse”, and select the required file. Click “Browse Files” again to select and submit more files.
4. Once the file is selected, click Upload.

Remember to complete the reading quiz on Sakai.

Autograded portion [41 points]

- [2] Functions from **`Hangman.py`** (same as before)
    - [1] handleUserInputDifficulty
        - Gives the user the option for easy or hard mode
        - Returns the correct value
    - [1] handleUserInputLetterGuess
        - Prints out the value in the displayString variable
        - Takes in user input to guess a letter
        - Returns the user's guessed letter
        - Checks if the user entered a repeated letter and prompts the user to enter a new letter until a non-repeated letter is entered
- [39] Functions new to/modified in **`CleverHangman.py`**
    - [6] createDisplayString
        - Letters not yet guessed are in sorted order
        - Returns a string that contains the letters not yet guessed, hangmanWord, misses remaining
        - Returned string is in the correct format
    - [3] handleUserInputDebugMode
        - Gives the user the option for debug or play mode
        - Returns bool representing whether game will be played debug mode
    - [3] handleUserInputWordLength
        - Prompts user for the length of the word to be guessed
        - Returns int specified by user
    - [4] createTemplate(currentTemplate, letterGuess, word)
        - Takes three parameters currentTemplate (str), letterGuess (str), and word (str)
        - Returns correct string representing the updated template
    - [11] getNewWordList(currentTemplate, letterGuess, wordList)
        - Takes three parameters currentTemplate(str), letterGuess (str), and wordList (list of str)
        - Returns correct tuple (string, list of str) representing the longest template and list of words that fit in this template, respectively
        - The templates and counts are in alphabetical order
        - Correctly breaks ties when multiple templates have identical lengths for corresponding word lists
    - [1] processUserGuessClever
        - Updates the value of missesLeft
        - Updates whether or not the user guessed wrong
        - Returns the correct values in a list

    - [6] runGame
        - Calls the other functions to run the game
        - Takes in a single parameter with the filename
    - [5] debug mode
        - Shows secret word
        - Shows number of possible words

Hand-graded portion [9 points]

- [6] Hand-Running CleverHangman.py: Single Game
    - [1] Game begins without errors
    - [1] Possible to get through a whole game without errors
    - [1] Prints correct single-game summary statistics
    - [1] Game consistently works properly
    - [2] Prints correct output in debug mode
- [1] Hand-Running CleverHangman.py: Multiple-Game Session
    - [1] Prompts user if they want to play again, follows the user's request, and prints correct multi-game summary statistics
- [2] Rest of submission
    - [2] point for name at the top and docstring comments (including for any optional helper functions)