Cheating Hangman Design Documentation

Author: Maggie Bickerstaffe
24 March 2023

Overview of Problem

In its essence the Hangman game consists of one player selecting a word but does not tell the other players what it is. Instead, the only information the guessing players receive is the length of the "secret" word. Players proceed to guess letters and the player who knows what the word is will reveal the placement of correct letters once they are guessed. If the players manage to guess/unveil the word before they run out of time/guesses they win. If not, the executioner, or player who knows the secret word, wins. The game includes a trust component because the guessing players must trust that the executioner is not changing the word when the players guess a letter correctly. This is where Cheating Hangman comes into play. The computer, who generates the secret word, changes it based on the player's guesses. In this version, the computer, or executioner, doesn't choose a word in the beginning but continuously iterates through a predetermined dictionary of words to decide on the best possible course of action to ensure the player loses. For example, if the player guesses the letter "a", the computer will run through its dictionary and determine in which spot do the most words have the letter "a" in or if more words have no "a". The computer then decides whether or if to reveal the location of "a". The only way the user can win is if they can get the dictionary down to one word.

Design Plan

The basis of this game is a game loop. While the player has not won or lost, the loop will keep executing with the same steps until the game is over.

The only component not in the game loops is the dictionary creation. The first thing the program does is run through the text file and read each line into the dictionary list. Therefore, every word is an item in the dictionary. Since the dictionary only needs to be created once in the beginning, it exists outside of the loops.

The entire game exists within a stillPlaying loop. The stillPlaying variable is a global boolean that is initialized as True. Once the user finishes a game, win or loss, the computer prompts the user if they wish to continue. If they respond "N", that stillPlaying is switched to False and the stillPlaying loop, and the game itself, stops executing.

def stillPlaying( ):

- Prompt the user if they want to keep playing. If yes, return True and continue the game. If no, return False and end the game.

ONBOARDING:
The game begins with a series of onboarding questions. The first two onboarding questions are within while loops. These while loops depend on the global booleans, applicable and guesses, which are both initialized as False. The first loop states that while not applicable, keep prompting the user for a word length. Once the user enters a word length, the program will run through the dictionary and if there is a word in the dictionary whose length matches the user's input, applicable is then switched to True and the loop ends. If no words match the length, the user will keep getting prompted for a word length until one does.

The second loop in the onboarding stage states that while not guesses, prompt the user for a number of guesses. If the user responds with a number above 0, guesses is set to True and the program continues. If the user doesn't enter a number above 0, the user is prompted again and again until they do.

The final question in the onboarding process is if the user wants to see a running total of words still available to play. Remember, the user can only win if they reduce this number down to one. Unlike the other two questions, this question results from a function that returns "Y" or "N" based on the user's answer and assigns that string to the displayWords variable.

def runningTotal( ):

- Prompts the user if they "want to see a running total of the number of words remaining", and assigns that input to the seeList variable. Then the function returns seeList.

Once the user has completed the onboarding questions "START GAME" is printed along with the number of guesses left and the blank word is printed. For example if the user entered a length of 5, the string "_ _ _ _ _" would appear. The remaining letters left to guess, which at this point is all of them, are printed to the screen. Additionally, the words that are the correct length are added to the wordToPlay list.

def eliminateWordByLength( ):

- Loop through all the words in the dictionary, if they match the length of word chosen by the user, then add that word to the wordsToPlay list. This is the list that will be updated throughout the game.

MAIN GAME LOOP:
Once the onboarding process is complete, the rest of the game executes in the main game loop. These functions will keep repeating until the player has either won or lost. This loop is the "while not gameOver".

Firstly, the loop takes the displayWords variable which was set as the runningTotal input, and uses that to determine if the running count of words in the wordsToPlay list should be printed. If displayWords == "Y", then displayWordsLeft is run.

def displayWordsLeft( ):

- Prints to the user interface “Display words left: “, plus the length of wordsToPlay.

Next, guessedLetter is assigned as the output of the guess( ) function. However, once guessedLetter is assigned, it checks to see if the guessLetter has been guessed yet. There is a global list called alphabetList that is initialized with a-z, string items. While the count of the guessedLetter in alphabetList is 0, the guess( ) function keeps running. Therefore, only letters that have not been guessed before can be guessed.

def guess( ):

- Set userGuess to be equal to the user input of “Guess a letter: “, then return userGuess.

The total number of guesses, or guessNum, is then subtracted by 1. The next piece of the game loop is the most integral part of CheatingHangman. Here, the program takes the guessedLetter and calculates the position where that letter is most located at or if more words don’t contain that letter at all. This is the groupFamilies function and it assigns the maxPosition variable. It can return either an integer, which is equal to an index or a list which is equal to multiple indexes. This is because if you were returning the position of “y” in “yay”, the indexes would be both 0 and 2.

def groupFamilies(letter):

- The groupFamilies function takes in the guessedLetter and returns the maxPosition. First it initializes the familiesSingularPresence dictionary, the occurrence list, and the multipleOccurrences list. All are empty at this point. First we iterate through the wordsToPlay list and if the number of times guessedLetter appears in a word is 1, then add the index of guessedLetter into the occurrences list. Then for single occurrences, we iterate from -1 through the length of the word(while int i = -1 < wordLength), and at each point we loop through all the numbers in occurrences and if i == the num in occurrences count += 1. Then after occurrences have been looped through, add an entry to the familiesSingularPresence dictionary where the key is the index and the value is the number of times that guessedLetter appears at that index throughout all the words in wordsToPlay. This includes the index of -1, or when guessedLetter does not appear in the word.

- If guessedLetter appears multiple times in a word, then the index is recorded in an “indexes” list and guessedLetter is replaced with “_” until guessedLetter is no longer in the word. This ensures that all occurrences of guessedLetter are recorded and the indexes are correct. The indexes list is then added to multipleOccurences, so multipleOccurrences is a list of lists.

- Finally, we create maxPosition (= -1) and maxCount (= 0) variables and loop first through all items in familiesSingularPresence and then all items in multipleOccurences. For every

key in familiesSingularPresence if the value is greater than maxCount, maxPosition is set to the key. We then count the items in multipleOccurrences and if the count is larger than maxPosition, maxPosition is set to that list of indexes and maxCount is set to the count.

- maxPosition is then returned. This variable is either equal or a singular index or a list of indexes.

We then run the storeGuesses function. This just adds each guessed letter and its maximum position to the topLetters dictionary.

def storeGuesses(guessedLetter, maxPosition)

- Add key = guessedLetter, value = maxPosition to the global topLetters dictionary

We generate the new display word based on the new guesses by the user. We set displayWord to the output of the generateDisplayWord function and then print displayWord.

def displayWord(wordLength)

- First we run through the length of the words and create a template with dashes. If all the words are 3 letters long, word is set to "_ _ _".
- Then we run through the topLetters dictionary and if the value is an integer and it isn't -1, we splice the word from the beginning to the value, add the key, and then finish with the value to the end of the word.
- If the value isn't an integer, we do the same thing but for every integer in the list.
- Return the word

Then, we print out the updated number of guesses left as well as the updated alphabet string.

def displayAlphabet(guessedLetter):

- Remove the guessed letter from the list of the letters (alphabetLetters) then make a string with the items (letters) in alphabetString and return that string.
- If a user has only guessed "a" it would return "bcdefghijklmnopqrstuvwxyz"

Finally, we set won equal to checkForWin( ), if won, gameOver is set to True and the while notGameOver loop ends. If not won, we check if the guessNum is 0, and if it is, gameOver is also set to True and the loop ends as well.

def checkForWin( ):

- If the display word does not contain a "_" return True.

If gameOver, many of the global variables are reset, including applicable, guesses, gameOver, and the alphabetList. If won the user receives a won message and if they lost they receive a lost message plus a random word from the dictionary that is pretending to be the word all along.

Then we run stillPlaying( ) and if the user says “Y”, the stillPlaying loop keeps going and the game restarts. Otherwise the program quits.

def wonMessage( ):

- Print “You Won!” to the user interface

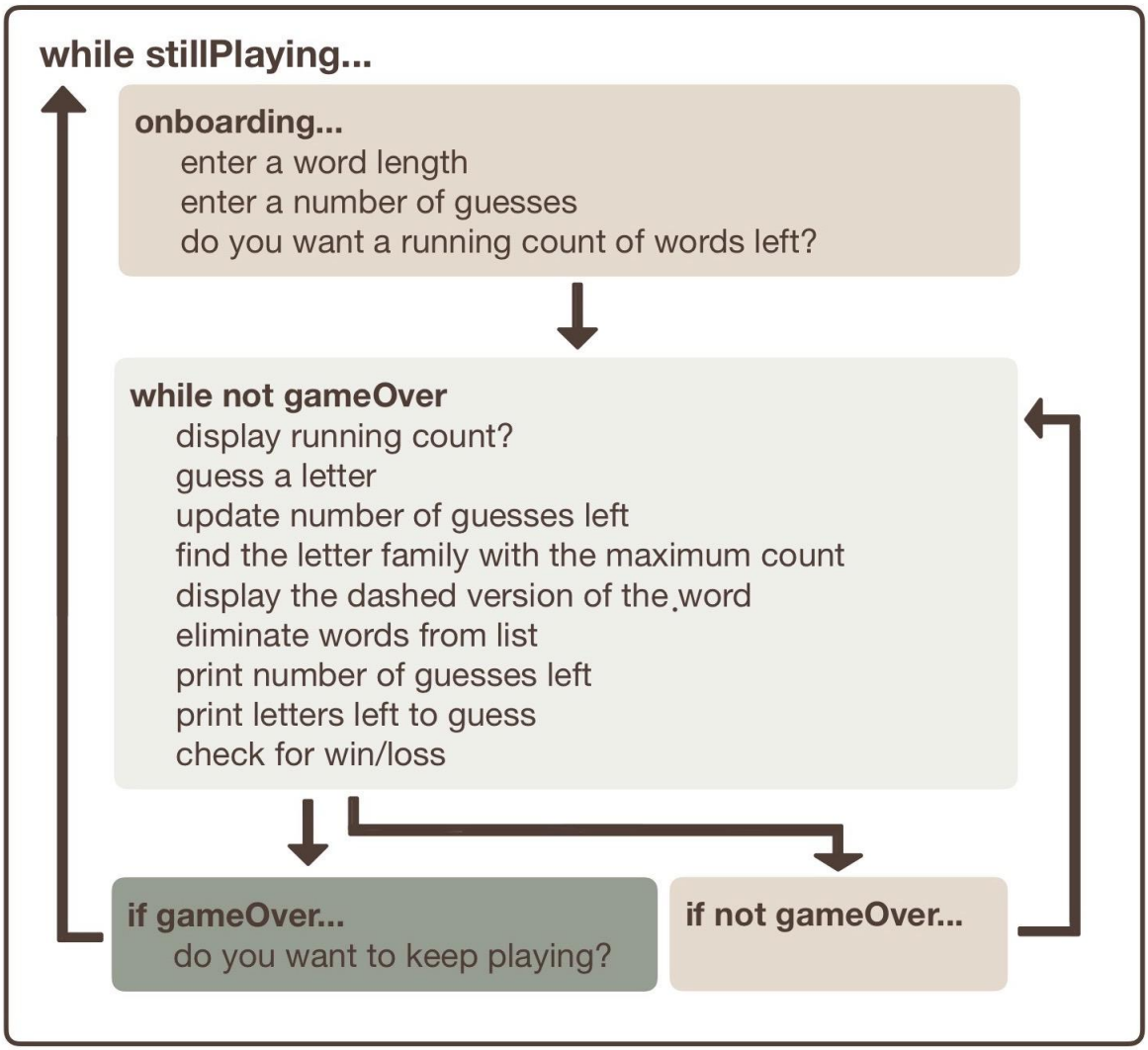
def lostMessage( ):

- Print “You Lost! The secret word was [random word from wordsToPlay].”

def stillPlaying( ):

- Prompt the user if they want to keep playing. If they respond “Y”, then return True.

Design Diagram:



Alternative Approaches and Discussion

Almost all games are coded with a game loop. This is essentially the part of the program that will loop over and over again until someone wins or loses. However, within the game loop there are many places where different logic or algorithms could have been used. I often used while loops to run until a variable was usable, such as the guessedLetter, word length, or number of guesses. However, if statements could have also been utilized, saying that if an input doesn't work, run that function again. While loops made more sense to me because if the logic of a statement is wrong, trying again is how I thought about the problem. The largest area of alternative programming would have to be in the groupFamilies function. I used three functions for this part. First a list of the index location for every word, then a dictionary that used that list to create a key of the index and values of the count of that index. Then a loop to determine which count was the highest and to return that associated index. I believe that the first step of this could have been eliminated by creating a dictionary from the beginning. Since you can have two keys of the same value, you should be able to iterate through the word and store the index and a count from the beginning. Then when you get to that index again you will override the initial dictionary entry and increase the value by 1.