Chris Piech and Mehran Sahami
CS 106A
Handout #12
May 15, 2020

Assignment #5: Strings and Word Guessing

Due: 1:30pm (Pacific Daylight Time) on Friday, May 22nd

Based on problems by Eric Roberts, Nick Parlante, and the current CS106A staff.

This assignment will give you the opportunity to get lots of practice using strings as well as start to work with dictionaries (you'll get a lot more practice with dictionaries on assignments 6 and 7 as well). You can download the starter code for this project under the "Assignments" tab on the CS106A website. The starter project will provide Python files for you to write your programs in.

As usual, the assignment is broken up into two parts. The first part of the assignment focuses on some problems to give you practice writing functions with strings and dictionaries. The second part of the assignment is a longer program that uses strings to play a word guessing game (similar to the game Hangman, if you're familiar with that game). This is a complex task, but armed with the skills you've been honing, especially using decomposition, you'll sail through.

Part 1: Strings and Dictionaries

1. String encoding

A simple means for trying to compress text with many repeated characters is to use something known as *run-length encoding*. The idea is that rather than representing every character in a string explicitly, we instead simply have each character immediately followed by a digit which denotes how many times that character should be repeated. For example, the string `'B4'` would represent the string `'BBBB'` as the character `B` is to be repeated 4 times. Similarly, the string `'m1e2t1'` would represent `'meet'`, as the character `m` is to be repeated 1 time, followed by the character `e` which is to be repeated 2 times, followed by the character `t` is to be repeated 1 time. Thus, the general format for run-length encoded strings is a series of pairs, where each pair contains a single character followed immediately by a single digit (1 through 9 only, the digit will never be 0). The digit denotes the number of consecutive occurrences of the character immediately before it in the encoded string.

Your job is to write a function named **`expand_encoded_string`** in the file **`encoded_string.py`** that takes as a parameter a called **`encoded`**, which is a string representing the run-length encoded text, and returns a string which is the expanded version of the text.

For example, if you call:

```
expand_encoded_string('B1o2k2e2p1e1r1!3')
```

your function should return the string:

```
"'Bookkeeper!!!'
```

as the result.

Doctests are provided for you to test your function. Feel free to write additional doctests if you would like practice with that aspect of Python. Also, feel free to write any additional functions that may help you solve this problem. Tests for your function are provided in the **`main`** function included in the program.

2. Totaling credit card bill by store

Realizing that keeping track of credit card expenses is a particularly good use of your computer science skills, you decide to write a program that helps you track how much money you are spending at each store in a given month. Your program should read a data file (given by the constant named **`INPUT_FILE`**), which contains your credit card bill for a given month. Each line of the credit card bill represents one transaction. Each line starts with the date of the purchase, followed by a space, then the name of the store enclosed in square brackets, like so: **`[`*`name`*`]`** (you can assume no store names contain square bracket characters and store names also do not include the dollar sign (**`$`**) character), followed by a space, and then a dollar sign, and then the amount of the purchase (as an integer, since we assume here that transactions are just rounded to a whole dollar amount). A sample data file (**`bill1.txt`**) representing a credit card bill looks as follows:

File: `bill1.txt`

```
9/2/19 [Target] $12
9/21/19 [Stanford Bookstore] $102
9/30/19 [Jamba Juice] $5
10/7/19 [Target] $17
10/22/19 [Jamba Juice] $8
10/28/19 [Target] $45
```

In the file **`credit_card_total.py`**, you should write a program reads such a credit card bill file and prints to the screen the ***total*** amount that was purchased at ***each*** store on the bill. For example, given the input file **`bill1.txt`** (above), your program should print the output:

```
Target: $74
Stanford Bookstore: $102
Jamba Juice: $13
```

The output printed by your program should match the sample output as closely as possible, but you don't need to worry about the order in which the stores are printed. You can assume that the input file is properly formatted (as described above). Two sample input files (**`bill1.txt`** and **`bill2.txt`**) are provided in the assignment project folder to help you test your program. The output of your program using the file **`bill2.txt`** as input should be (again, you don't need to worry about the ordering of the stores in the output):

```
Shake Shack: $16
Grocery Hut: $293
Ace Hardware: $14
Joan's Fabric: $18
Nom Nom Nom: $12
```

Hint: using dictionaries would be a really good way to solve this problem!

Part 2: Word Guessing

For the second part of this assignment, your mission is to write a program that plays a word guessing game. This game is also some times called "Hangman", but we'll just call it "WordGuess" here. The WordGuess program is designed to give you some practice working with strings and files.

The WordGuess game

When the user plays WordGuess, the computer first selects a secret word at random from a list built into the program. The program then prints out a row of dashes—one for each letter in the secret word and asks the user to guess a letter. If the user guesses a letter that is in the word, the word is redisplayed with all instances of that letter shown in the correct positions, along with any letters correctly guessed on previous turns. If the letter does not appear in the word, the user is charged with an incorrect guess. The user keeps guessing letters until either (1) the user has correctly guessed all the letters in the word or (2) the user has made eight incorrect guesses. Two sample runs of the game are shown below (user input is shown in ***italics***).

Sample Run 1:

```
The word now looks like this: -----
You have 8 guesses left
Type a single letter here, then press enter: a
That guess is correct.
The word now looks like this: -A---
You have 8 guesses left
Type a single letter here, then press enter: q
There are no Q's in the word
The word now looks like this: -A---
You have 7 guesses left
Type a single letter here, then press enter: p
That guess is correct.
The word now looks like this: -APP-
You have 7 guesses left
Type a single letter here, then press enter: C
There are no C's in the word
The word now looks like this: -APP-
You have 6 guesses left
Type a single letter here, then press enter: H
That guess is correct.
The word now looks like this: HAPP-
You have 6 guesses left
Type a single letter here, then press enter: y
That guess is correct.
Congratulations, the word is: HAPPY
```

Sample Run 2:

```
The word now looks like this: ------
You have 8 guesses left
Type a single letter here, then press enter: a
There are no A's in the word
The word now looks like this: ------
You have 7 guesses left
Type a single letter here, then press enter: p
That guess is correct.
The word now looks like this: P-----
You have 7 guesses left
Type a single letter here, then press enter: H
That guess is correct.
The word now looks like this: P--H--
You have 7 guesses left
Type a single letter here, then press enter: g
There are no G's in the word
The word now looks like this: P--H--
You have 6 guesses left
Type a single letter here, then press enter: m
There are no M's in the word
The word now looks like this: P--H--
You have 5 guesses left
Type a single letter here, then press enter: q
There are no Q's in the word
The word now looks like this: P--H--
You have 4 guesses left
Type a single letter here, then press enter: d
There are no D's in the word
The word now looks like this: P--H--
You have 3 guesses left
Type a single letter here, then press enter: L
There are no L's in the word
The word now looks like this: P--H--
You have 2 guesses left
Type a single letter here, then press enter: E
There are no E's in the word
The word now looks like this: P--H--
You have 1 guesses left
Type a single letter here, then press enter: r
There are no R's in the word
Sorry, you lost. The secret word was: PYTHON
```

In order to write the program that plays WordGuess, you should design and test your program in two parts. The first part consists of getting the interactive part of the game working with a fixed set of secret words (that are initially provided for you). The second part consists of replacing the supplied version of the secret word list with one that reads words from a file. The rest of this handout describes these two parts in more detail.

Note that your program only needs to be able to play the WordGuess game once through (i.e., the player guessing one word), but it should be pretty easy to extend your program to allow the player to play multiple rounds (i.e., guessing a word multiple times).

WordGuess Game: Part I—Playing the game

In the first part of this assignment, your job is to write a program that handles the user interaction component of the game. To solve the problem, your program must be able to:

- Choose a random word to use as the secret word. That word is chosen from a word list, as described below. (An initial implementation of this is provided for you.)
- Keep track of the user's partially guessed word, which begins as a series of dashes and then gets updated as correct letters are guessed.
- Implement the basic control structure and manage the details (ask the user to guess a letter, keep track of the number of guesses remaining, print out the various messages, detect the end of the game, and so forth).

For this part of the assignment, you will simply make use of a function that we've given you called **`get_word`** that returns a word (string) randomly chosen from a small list of words that will allow you to test your program. The initial code you are provided for the **`get_word`** function is only a temporary expedient to make it possible to code this part of the assignment. In Part II, you will reimplement the **`get_word`** function we've provided with one that reads a list of words from a data file in order to select the random word from a much larger set of possibilities.

The game

The two sample runs shown previously should be sufficient to illustrate the basic operation of the game, but the following points may help to clarify a few issues:

- In the **`main`** function, we call the **`get_word`** function to get a secret word for the user to guess, store it in a variable named **`secret_word`** and then pass that **`secret_word`** to a function called **`play_game`**. For this part of the assignment, you should implement the **`play_game`** function, along with any additional functions that are needed to properly decompose the program, to produce a working game.
- You should accept the user's guesses in **either lower or upper case**, even though all letters in the secret words are written in upper case.
- If the user guesses something that is more than a single character, your program should tell the user that the guess should only be a single letter and accept a new guess. It should not count the guess that was more than a single character as an incorrect guess. Here's an example of what that interaction should look like (where the user enters the guess ***AA***). Notice that the user's guess did not reduce the number of guesses they have remaining.

```
The word now looks like this: ------
You have 8 guesses left
Type a single letter here, then press enter: AA
Guess should only be a single character.
The word now looks like this: ------
You have 8 guesses left
Type a single letter here, then press enter:
```

- If the user guesses a correct letter more than once, your program should do nothing (basically, it's just another correct guess, so the number of guesses left is not reduced). Guessing an incorrect letter a second time should be counted as another wrong guess. (In each case, these interpretations are the easiest way to handle the situation, and your program will probably do the right thing even if you don't think about these cases in detail.)
- If the user guesses a character, such as a period (`.`) or exclamation point (`!`), that is not a letter (and therefore would not be in the secret word), you can just count that as an incorrect guess.
- The number of guesses the player initially starts with is determined by the constant `INITIAL_GUESSES`.

WordGuess Game: Part II—Reading a word list from a file

Part II is much easier than Part I and requires less than half a page of code. Your job in this part of the assignment is simply to re-implement the `get_word` function so that instead of being hard-coded to select from a meager list of three words, it reads a much larger word list from a file and returns a word from that list. The steps involved in this part of the assignment are as follows:

1. Reimplement the `get_word` function so that it opens the file specified by the constant `LEXICON_FILE` and reads it line by line.
2. Read the lines from the file into a list.
3. Using that list of words, *randomly* return any word in the list from the `get_word` function. You can feel free to define additional helper functions if you need to, but it's likely that all the code you need to implement this functionality will fit nicely in the `get_word` function itself.

Note that nothing in the rest of the program should have to change in response to this change in the implementation of `get_word`. Insulating parts of a program from changes in other parts is a fundamental principle of good software design. Thus, in your implementation of the `get_word`, you should not be changing the parameters of this function.

As a side note, the file `Lexicon.txt` contains a very large list of words that can used to make the game more challenging to play, and the final version of your program should use that file for input. But to help you test this part of your program, we also provide a much smaller word list in the file `TestLexicon.txt`. You can feel free to use the `TestLexicon.txt` file to help you develop/debug your reimplementation of the `get_word` function, but make sure the final version of your program works properly with the full `Lexicon.txt` file.

Hint: remember to use the `strip` function when reading the file to avoid any nasty complications arising from newline characters at the end of lines.

IMPORTANT NOTE: Make sure your output matches that in the sample runs as closely as possible.

Extension ideas

There are many ways to extend the WordGuess game to make it more fun if you want. Of course, we encourage you to some up with your own ideas, but here are a few possibilites:

- You could use graphics to spice up the display. For example, you could use graphics to have this program emulate the game "Hangman" where a person (stick figure) is displayed part by part with each incorrect guess the user makes. You could even go further and animate the pictures.
- You could expand the program to play something like Wheel of Fortune, in which the single word is replaced by a common phrase and in which you have to buy vowels.
- Use your imagination!

Submitting your work

Once you've gotten all the parts of this assignment working, congratulations! You've now also got a fun game that you can play.

Make sure to submit **all** the python files for this assignment on Paperless. You should make sure to submit the files:

```
encoded_string.py
credit_card_total.py
word_guess.py
```

Additionally, in the Assignment 5 project folder, we have provided a file called **`extension.py`** that you can use if you want to write any extensions that you might want to make based on this assignment. The file doesn't contain any useful code to begin with. So, you only need to submit the **`extension.py`** file if you've written some sort of extension in that file that you'd like us to see.