CSC108H Assignment 2

Due Date: Tuesday 14 March 2017 before 9:00pm

Poetry Annotator

Poetry forms like limericks, sonnets and haiku, follow pattern rules that prescribe the structure of the poem. For instance, these pattern rules may determine the number of lines and the rhyme scheme of the poem, as well as the number of syllables and the pronunciation stress patterns in each line. In this assignment, you will write code that could be used to help poets confirm that their work meets pronunciation stress pattern rules and rhyme scheme rules.

Two examples

A limerick has the following features:

- There are 5 lines of poetry.
- The first, second, and fifth lines each have 7 to 10 syllables.
- The third and fourth lines each have 5 to 7 syllables.
- The rhyme scheme is `A A B B A`. This means that lines 1, 2, and 5 rhyme (the "A" lines), and lines 3 and 4 rhyme (the "B" lines).

Here is a stupendous work of limerick art, written by Dan Zingaro at UTM.

> **Limerick**
> by Dan Zingaro
>
> I wish I had thought of a rhyme
> Before I ran all out of time!
> I'll sit here instead,
> A cloud on my head
> That rains 'til I'm covered with slime.

Petrarch, an Italian poet who lived in the 1300's, is widely credited with popularizing sonnets. A Petrarchan sonnet has the following features:

- There are 14 lines of poetry.
- The rhymes at the ends of the first 8 lines have this pattern: `A B B A A B B A`. This means that lines 1, 4, 5, and 8 rhyme (the "A" lines), and lines 2, 3, 6, and 7 rhyme (the "B" lines).
- The last 6 lines can have various rhyme schemes, including `C D C D C D` and `C D D E C E`. There are several more variations.

Here is a Petrarchan sonnet written by Christina Rossetti in 1849.

Remember
by Christina Rossetti

Remember me when I am gone away,
Gone far away into the silent land;
When you can no more hold me by the hand,
Nor I half turn to go yet turning stay.
Remember me when no more day by day
You tell me of our future that you planned:
Only remember me; you understand
It will be late to counsel then or pray.
Yet if you should forget me for a while
And afterwards remember, do not grieve:
For if the darkness and corruption leave
A vestige of the thoughts that once I had,
Better by far you should forget and smile
Than that you should remember and be sad.

Pronunciation stress patterns

An important aspect of English pronunciation is stress, or the relative emphasis used to pronounce vowel sounds. For example, the "o" in the word *program*, is stressed, while the "a" is unstressed. In other words, the "o" sound is emphasized more than the "a" sound. There is a standard way to annotate stress above a word: the symbol "/" indicates a stressed vowel, and the symbol "x" indicates an unstressed vowel. To indicate the stress in a word, we will write:

```
/ x
program
```

Note that the symbols do not line up over the vowels. The first symbol is above the first character in the word, and the second symbol follows the first after a separating space. The first symbol comes from the "o" in *program*, while the second comes from the "a".

The pronunciation of some vowel sounds is somewhere in-between stressed and unstressed. These vowel sounds are said to have *secondary stress*. In the word *declaration*, the primary stress is on the third vowel, but the first vowel also has a bit of stress. We indicate secondary stress with the symbol "\". We write:

```
\ x / x
declaration
```

Again, note that the symbols do not line up over the vowels. They appear one at a time, starting above the first character in the word, each separated by a single space.

You may be wondering how your program is going to determine the pronunciation stress patterns and rhyme schemes for a poem. Your program will make careful use of the CMU Pronouncing Dictionary!

The CMU Pronouncing Dictionary

The Carnegie Mellon University Pronouncing Dictionary describes how to pronounce words. Head there now and look up a couple of words; try searching for words like "Daniel", "is", and "goofy", and see if you can interpret the results. Try entering contractions like "I'll" (short for "I will") and "we'll" (short for "we will") to see what is reported. Can the pronouncing dictionary handle British spellings, such as "neighbour"? (The answer is `some but not all'. Try "saviour".)

Now click the "Show Lexical Stress" checkbox and see how this choice changes the results. (You should observe that some digits have been inserted into the response.)

Here is the output for "Daniel" (with "Show Lexical Stress" turned on): `D AE1 N Y AH0 L`. The separate pieces are called *phonemes* and each phoneme describes a sound. The sounds are either vowel sounds or consonant sounds. We will refer to phonemes that describe vowel sounds as *vowel phonemes*, and phonemes that describe consonant sounds as *consonant phonemes*. The phonemes that are used were defined in a project called Arpabet, created by the Advanced Research Projects Agency (ARPA) back in the 1970's.

In the CMU Pronouncing Dictionary, all vowel phonemes end in a `0`, `1`, or `2`, with the digit indicating a level of stress. Consonant phonemes do not end in a digit. The number of syllables in a word is the same as the number of vowel sounds in the word, so you can determine the number of syllables in a word by counting the number of phonemes that end in a digit.

As an example, in the word "secondary" (`S EH1 K AH0 N D EH2 R IY0`), there are four vowel phonemes, and therefore four syllables. The vowel phonemes are `EH1`, `AH0`, `EH2`, and `IY0`. The digit `0` indicates that a vowel phoneme is unstressed, the digit `1` indicates a primary stress, and the digit `2` indicates a secondary stress. Try saying the word "secondary" out loud so that you can hear for yourself which syllables have stress and which do not.

Your program will need to distinguish between the levels of syllabic stress. It will display the stress for each of the words in a given poem.

In this assignment, you are **not** to use the original CMU Pronouncing Dictionary, but rather a version that we have prepared based on the CMU Pronouncing Dictionary. Our version differs from the CMU version: we have removed alternate pronunciations for words and words that do not start and end with alphanumeric characters (like `#HASH-MARK`, `#POUND-SIGN` and `#SHARP-SIGN`). After downloading the necessary files (see below for details), take a look at the file `our_dictionary.txt` to see the format; note that any line beginning with `;;;` is a comment and is not part of the dictionary.

The words in `our_dictionary.txt` are all uppercase and do not contain surrounding punctuation. When your program looks up the pronunciation of a word, use the uppercase form, with no leading or trailing punctuation. The function `prepare_word` in the starter code file `stress_and_rhyme_functions.py` will be helpful here.

Data representation of the pronouncing dictionary

The code in the file `annotate_poetry.py` that we provide reads the contents of the CMU Pronouncing Dictionary into a list of `str` and passes it as an argument to the function `make_pronouncing_table` that you are to write, in order to build a data type that we call a *pronouncing table*.

A *pronouncing table* is a two item list, where both of the items are parallel lists. The first item in a pronouncing table is a list of words and the second item in a pronouncing table is a list of pronunciations. Each word is a `str` and all of the letters in the word are uppercase. Each pronunciation is a list of phonemes, and so is a `list` of `str`. Here is an example:

```
SMALL_TABLE = [['A', 'BOX', 'CONSISTENT', 'DON\'T', 'FOX', 'IN', 'SOCKS'],
               [['AH0'],
                ['B', 'AA1', 'K', 'S'],
                ['K', 'AH0', 'N', 'S', 'IH1', 'S', 'T', 'AH0', 'N', 'T'],
                ['D', 'OW1', 'N', 'T'],
                ['F', 'AA1', 'K', 'S'],
                ['IH0', 'N'],
                ['S', 'AA1', 'K', 'S']]]
```

The pronunciation for the word at index `i` in the words list is at index `i` in the pronunciations list. In the example above, the word `'BOX'` is at index 1 in the words list, and its pronunciation is the list `['B', 'AA1', 'K', 'S']`, which is at index 1 in the pronunciations list.

The argument to the function `make_pronouncing_table` is a list of `str`. We call each item in the list a *pronouncing line*. A pronouncing line is a line from the CMU Pronouncing Dictionary of the form: a word followed by the phonemes that describe how to pronounce the word. An example is `'BOX B AA1 K S'`. In this assignment, you are to write code that takes an example like this and separates the word and pronunciation information into `'BOX'` and `['B', 'AA1', 'K', 'S']`.

Detecting rhyme schemes

We say that two *words* rhyme with each other if the final vowel phonemes and all subsequent consonant phoneme(s) after the final vowel phonemes match (i.e., are the same and are in the same order). For example, in `SMALL_TABLE` above, we can see that `'BOX'`, `'FOX'`, and `'SOCKS'` all rhyme with each other, because their pronunciations all end with `'AA1'`, `'K'`, `'S'`. (Linguists use a slightly different definition for rhyme. They note that the stress on the final vowel of two words does not always have to be the same for the words to rhyme. For example, `'FOX'` (pronounced `'F AA1 K S'`) rhymes with `'XEROX'` (pronounced `'Z IH1 R AA0 K S'`). **However in this assignment we will use the definition for *rhyme* given in the first sentence of this paragraph.**)

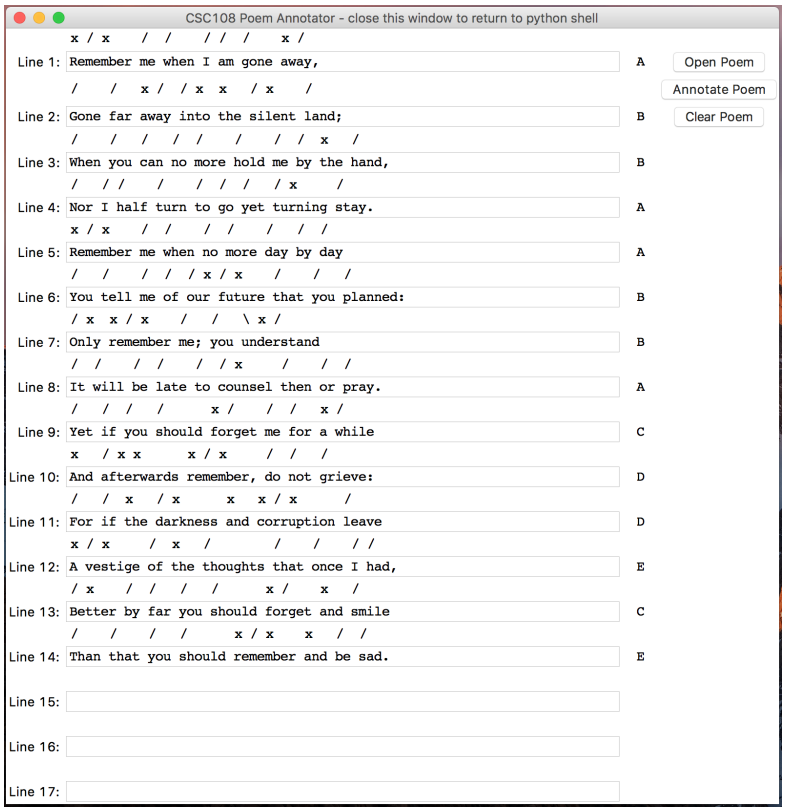
We say that two *lines* in a poem rhyme with each other if the last word in the lines rhyme. When two (or more) lines rhyme, they are annotated with the same uppercase letter. When annotating the rhyme scheme in a poem, consecutive uppercase letters are used for each different set of rhyming lines, starting with the letters `A`, `B`, `C`, etc. See the annotated poem in the next section for an example.

The main program file `annotate_poetry.py`

After completing the required functions (see below for details), you can run the `annotate_poetry.py` code and annotate a poem with the correct pronunciation stress patterns and rhyme schemes! After starting `annotate_poetry.py`, a `CSC108 Poem Annotator' window will appear. You will need to click in the window before you can annotate poems. To enter a poem in the window, you can either click on the `Open Poem`

button and select a text file on your computer, or you can type your poem into the boxes. You can even revise a poem that you have read in from a file by typing in a box that already contains words. If the code that you have written is correct, clicking on the `Annotate Poem` button will cause the poem to be annotated with the pronunciation stress patterns and the rhyme scheme. Clicking on the `Clear Poem` button will clear the window, and you can start over with a new poem. The poem window needs to be closed before you can return to your Python shell.

If your code has been written correctly, you will be able to recreate a window that looks like the following image. It contains an annotation of the poem Remember.



Rules about poems

Poems can be displayed in the poem window by reading them from a file. Blank lines that are before the start of a poem are removed by the program. When multiple blank lines are between stanzas in a poem, all but one of the blank lines are removed. Blank lines that appear after the end of a poem are also removed. Finally, spaces that appear at the start or end of a line in a poem are removed from the poem by the program.

Poems can also be entered by typing into the poem window. Any extra blank lines and spaces are removed from the poem when the `Annotate Poem` button is clicked.

There are some limits on the size of the displayed poem. The number of lines displayed in the poem window is at most `MAX_LINES_IN_POEM`, a constant that is defined in the `annotate_poetry.py` file. In addition, each line in a poem will be shortened to have at most `MAX_CHAR_IN_POEM_LINE` characters. These limits are enforced by the code in the `annotate_poetry.py` file. The code that you write, in particular the function `convert_to_lines`, does not need to use these constants.

If a poem contains a word that is not in `our_dictionary.txt` an error message will be displayed in the Python shell after the `Annotate Poem` button is clicked. Special characters like nonstandard apostrophes may also cause word pronunciation lookup to fail. Change the word or character and try again. Nonstandard apostrophes can appear when you cut-and-paste from poetry websites.

Files to download

All of the files that you will download for the assignment are described in this section. These files must all be placed in the same folder.

Please download the zip file that contains the files. Save this file in the folder where you plan to do your work. Open the file to unzip it and extract the files.

- Python Code
  - The starter code for this assignment is in the file `stress_and_rhyme_functions.py`. **Complete** the functions listed in the Required Functions section (see below) in this file.
  - The main program file is `annotate_poetry.py`. It contains the code for creating and managing the poem window. It is complete and must not be changed.
  - The type checker is `a2_type_checker.py`. It is complete and must not be changed. There are more details on the type checker below.
- Pronouncing Dictionary
  - The file `our_dictionary.txt` is our version of the CMU Pronouncing Dictionary. You **must** use this file, **not** any files from the CMU website.
- Sample Poetry
  - The remaining `.txt` files contain example poems, including the poems from this handout.

Required functions

In the file `stress_and_rhyme_functions.py`, you must implement nine functions. In addition to the descriptions below, please see the docstring descriptions provided in the starter code. **Add a second example function call to each of the given docstrings** in `stress_and_rhyme_functions.py`.

Before writing the functions, you are encouraged to learn about the type `str` methods `split` and `strip`. Evaluate the statements `help(str.split)` and `help(str.strip)` in the Python shell for details.

Function name: (Parameter types) -> Return type	Full Description
`get_word: (str) -> str`	A pronouncing line is a line from our pronouncing dictionary. It is a string that contains a word followed by the phonemes that describe how to pronounce the word (the word and each phoneme is separated by whitespace). An example is `'BOX B AA1 K`

	S'. The parameter is a pronouncing line. Return the word in the pronouncing line. With the example above, 'BOX' would be returned.
get_pronunciation: (str) -> list of str	A pronouncing line is a line from our pronouncing dictionary. It is a string that contains a word followed by the phonemes that describe how to pronounce the word (the word and each phoneme is separated by whitespace). An example is 'BOX B AA1 K S'. The parameter is a pronouncing line. Return a list containing the pronunciation (list of phonemes) from the pronouncing line. With the example above, ['B', 'AA1', 'K', 'S'] would be returned.
make_pronouncing_table: (list of str) -> pronouncing table	The parameter is a list of pronouncing lines from a pronouncing dictionary. Return the pronouncing table built from those lines.
look_up_pronunciation: (str, pronouncing table) -> list of str	The first parameter is a word and the second is a pronouncing table. Return a list containing the pronunciation (list of phonemes) for the word in the pronouncing table.
is_vowel_phoneme: (str) -> bool	A string contains a **vowel phoneme** if and only if it has three characters, with the first character being one of A, E, I, O, or U, the second character is an upper case letter, and the last character is one of 0, 1, or 2. As examples, the word **CONSISTENT** (K, AH0, N, S, IH1, S, T, AH0, N, T) contains three vowel phonemes and the word **BOX** (B AA1 K S) contains one. The parameter represents a phoneme. Return whether or not it is a vowel phoneme.
last_syllable: (list of str) -> list of str	The parameter is a list of phonemes. Return the last vowel phoneme and any subsequent consonant phoneme(s) from the list. For example, given the list ['K', 'AH0', 'N', 'S', 'IH1', 'S', 'T', 'AH0', 'N', 'T'], the list ['AH0', 'N', 'T'] would be returned.
convert_to_lines: (str) -> list of str	The parameter represents a poem. Return a list of lines from the poem with whitespace characters stripped from the beginning and end of each line. Lines can be determined by splitting the poem based on the newline character. Blank lines that appear before the first nonblank line in the poem or after the last nonblank line in the poem are not to be included in the list of lines. Blank lines that separate stanzas in the poem are to be represented by a single empty string in the list of lines.
detect_rhyme_scheme: (list of str, pronouncing table) -> list of str	The first parameter is a list of lines in a poem and the second is a pronouncing table. Return the rhyme scheme for the poem. A blank line that separates stanzas should be assigned the rhyme scheme marker ' ' (a string containing a single space character).
get_stress_pattern: (str, pronouncing table) -> str	The first parameter is a word and the second is a pronouncing table. Return the stress pattern for word according to the pronouncing table. Separate each stress marker (one of /, x, or \) with a space, and pad the returned string with spaces so that it

	has the same number of characters as the word. For example, given the word `declaration`, the eleven character string `'\ x / x  '` would be returned.

a2_type_checker.py

We are providing a type-check module that can be used to test whether your functions in `stress_and_rhyme_functions.py` have the correct parameter and return types. To use the type checker, place `a2_type_checker.py` in the **same** folder (directory) as your `stress_and_rhyme_functions.py` and run it.

If the type checks pass: the output will tell you that the typechecker passed (and what it means for the typechecker to pass!). If the typechecker passes, then the parameters and return types match the assignment specification for each of the functions.

If any type checks fail: Look carefully at the message provided. One or more of your parameter or return types does not match the assignment specification. Fix your code and re-run the tests. Make sure the tests pass before submitting.

Additional requirements

- Use the defined constants that are provided in `stress_and_rhyme_functions.py` when appropriate.
- Do not change any of the existing code. Add to it as specified in the comments.
- Do not call `print`, `input`, or `open`.
- Do not use any `break` or `continue` statements. Any functions that do will receive a mark of zero. We are imposing this restriction (and we have not even taught you these statements) because they are very easy to "abuse," resulting in terrible code.
- Do not add any other `import` statements.

How to tackle this assignment

Principles:

- To avoid getting overwhelmed, deal with one function at a time. Start with functions that don't call any other functions; this will allow you to test them right away.
- For each helper function that you write, start by adding at least one example call to the docstring *before* you write the function.
- Keep in mind throughout that any function you have might be a useful helper for another function. Part of your marks will be for taking advantage of opportunities to call an existing function.
- As you write each function, begin by designing it in English, using only a few sentences. If your design is longer than that, shorten it by describing the steps at a higher level that leaves out some of the details. When you translate your design into

Python, look for steps that are described at such a high level that they don't translate directly into Python. Design a helper function for each of these high-level steps, and put a call to the helpers into your code. Don't forget to write a great docstring for each helper!

Steps:

Here is a good order in which to tackle the pieces of this assignment.

1. Read this handout thoroughly and carefully, making sure you understand everything in it.
2. Read the `stress_and_rhyme_functions.py` starter code to get an overview of what you will be writing.
3. Implement and test the required functions in `stress_and_rhyme_functions.py`, along with any helper functions.
4. Read the code provided in `annotate_poetry.py` and run it. If there are any problems with the results, try to identify which of your functions has an issue, and go back to testing that function.

Marking

These are the aspects of your work that we will focus on in the marking:

- **Correctness**: The `annotate_poetry.py` program should run as described in the "main program" section by using your functions from `stress_and_rhyme_functions.py`. We will check the correctness of each of the required functions from `stress_and_rhyme_functions.py`. Remember that the spelling of file and function names, including case, is important, and that functions must have the correct number of required parameters. The parameters must be given in the specified order. Compared to functions from the previous assignment, there are many more possible cases to test (and cases where your code could go wrong). If you want to get a great mark on the correctness of your functions, do a great job of testing your functions under all possible conditions. Then we won't be able to find any errors that you haven't already fixed!
- **Style and Design**: Your code should be well documented with docstrings as well as internal comments, and it should adhere to our Python style guidelines. Your program should be broken down into functions, both to avoid repetitive code and to make the program easier to read. If a function body is more than about 20 lines long, introduce helper functions to do some of the work -- even if they will only be called once. All helper functions should have complete docstring descriptions. Make sure that you read the Python style guidelines page for some important rules and guidelines about formatting your code. Finally, your variable names should be meaningful, and your code as simple and clear as possible.

Submitting your assignment

You must hand in your work electronically, using the MarkUs online system. Instructions for doing so are posted on the Assignments page of the course website.

The very last thing you do before submitting should be to run `a2_type_checker.py` one last time and ensure that the type checks pass. This will prevent your code from receiving a correctness grade of zero due to a small error that was made during your final changes before submission.

For this assignment, you only need to hand in the file:

- `stress_and_rhyme_functions.py`

Once you have submitted, be sure to check that you have submitted the correct version; new or missing files will not be accepted after the due date. Remember that the correct spelling of filenames, including case, is necessary. **If your files are not named exactly as above, your code will receive zero for correctness.**