Homework Turnin

Name:	Alexander A Miller
Email:	alexandermiller@email.arizona.edu
Section:	1E
Course:	CS 120 17au
Assignment:	hw3
Receipt ID:	e73f8d9a2f3afa6e4e710ccef3720955

Turnin Successful!

The following file(s) were received:

rhymes.py (9461 bytes)

```
"""
File: rhymes.py
Author: Alexander Miller
Purpose: program will perform the following:
* read in a pronunciation dictionary from input and organize the information appropriately
* use input to read in 'word' w/o prompting
* collect all words that rhyme with 'word'
* if 'word' has more than one pronunciation, then it collects all words
    * that rhyme with each pronunciation (eliminating duplicates not necessary)
* the program should be case insensitive
* the program will print out the collected words


Input-File Assumptions:
* word is given at beginning of line; rest of line is sequence of phonemes giving pronunciation
    * lines will differ in length
* file is txt doc

Ouput Format:
* print out one word per line all words that rhyme with any pronunciations of word
* case, order, and duplicates irrelevant

Programming Requirements:
* dictionary mapping word to its various pronunciations; each pronunciation will be its own element of th

"""


def main():
    word_dict, multi_dict, word_list = input_fn()
    word = word_collection(word_list)
    x = word
    primary_stress = phoneme_processing(x, multi_dict, word_dict)
    successful_word_bin = process_staging(word_dict, multi_dict, word, primary_stress)
    print_function(successful_word_bin)


def input_fn():
    """
    Description: reads in file and organizes information
    Parameters: none
    Returns: word_dict, multi_dict, word_list
    Pre-condition: input-file assumptions: doc is txt w/ text at beginning of each line and phonemes afte
    Post-Condition: word_dict will be dictionary containing phonemes logged under words
        * multi_dict will be dictionary detailing which words have multiple pronunciations and how many
        * word_list provides list of words for easy access
```

```
    """
    inputfile = input()
    openfile = open(inputfile,'r')
    word_dict = {}
    multi_dict = {}
    word_list = []
    for line in openfile:
        linelist = line.split()
        # objective: refine to avoid overwriting duplicate entries; log words in dictionary along pronunc
        if linelist[0] not in word_dict:
            word_dict[linelist[0]] = [[linelist[0]]] + [linelist[1:]] #logging key within its value for e
            word_list = word_list + [linelist[0]]
        else: # objective: create a list of lists with each pronunciation as separate sublist
            word_dict[linelist[0]] = word_dict[linelist[0]] + [linelist[1:]]
            # objective: keep track of the fact that this word has multiple occurences
            if linelist[0] not in multi_dict:
                multi_dict[linelist[0]] = 2 # if we get to this stage, then it has occured once already
            else:
                multi_dict[linelist[0]] = multi_dict[linelist[0]] + 1
    openfile.close()
    return word_dict, multi_dict, word_list


def word_collection(word_list):
    """
    Description: uses an input validation loop to ensure typed word is in dictionary; returns word (upper
    Parameters: word_list
    Returns: word
    Pre-condition: none
    Post-Condition: word will be in dictionary and upper-case
    """
    word = 'a' # placeholder
    # objective: use an input validation loop to ensure that typed word is actually in the dictionary
    while word not in word_list:
        word = input()
        word = word.upper()
    return word
```

```
def phoneme_processing(x, multi_dict, word_dict):
    """
    Description: a cross-roads to branch items with multiple pronunciations from items with single pronur
    Parameters: x, multi_dict, word_dict
    Returns: primary_stress
    Pre-condition: none
    Post-Condition: returns primary stress from subfunctions
    """
    if x not in multi_dict:
        primary_stress = single_phoneme_acquisition(word_dict, x)
    else:
        primary_stress = multi_phoneme_acquisition(word_dict, multi_dict, x)
    return primary_stress


def single_phoneme_acquisition(word_dict, x):
    """
    Description: handles acquisition of primary stress for single-pronunciation words
    Parameters: word_dict, x
    Returns: primary_stress
    Pre-condition: none
    Post-Condition:  will return primary stress, a list 1 entry, or a string indicating a failed test
        * failed tests occur when there is no primary stress
        * these will be removed from comparison (later)
    """
    phoneme_list = word_dict[x][1] #the key is saved in value pos. 0 for later access
    accum = 0
    # objective: acquire primary stress phoneme
    while accum != len(phoneme_list):
        phoneme = phoneme_list[accum]
        if '1' in phoneme:
            primary_stress = [accum]
            break
        accum = accum + 1
    try: #in the very large dictionary there appear to be entries without primary stress which are wrench
        # this is a work-around
        return primary_stress
```

```
    except:
        primary_stress = 'fail'
        return primary_stress


def multi_phoneme_acquisition(word_dict, multi_dict, x):
    """
    Description: handles acquisition of primary stress for multi-pronunciation words
    Parameters: word_dict, x, multi_dict
    Returns: primary_stress
    Pre-condition: none
    Post-Condition:  will return primary stress, a list with 2+ entries, or a string indicating a failed
        * failed tests occur when there is no primary stress
        * these will be removed from comparison (later)
    """
    accum = 0
    primary_stress = [] #multiple primary stress locations to consider
    pronunciations = word_dict[x][1:] #again: index 0 contains key for future access
    # objective: acquire primary stress phonemes
    while accum != multi_dict[x]: #recall: multi_dict[word] yields number of pronunciations
        sublist = pronunciations[accum]
        accum2 = 0
        while accum2 != len(sublist):
            if '1' in sublist[accum2]:
                primary_stress = primary_stress +[accum2]
                break
            accum2 = accum2 + 1
        accum = accum + 1
    try:
        return primary_stress
    except:
        primary_stress = 'fail'
        return primary_stress


def process_staging(word_dict, multi_dict, word, primary_stress):
    """
    Description: generates and returns the collection of matches
    * facilitates comparison via word_processing sub function
    * organizes word_processing by single vs. multiple pronunciation comparisons
    Parameters: word_dict, multi_dict, word, primary_stress
    Returns: successful_word_bin
    Pre-condition: none
    Post-Condition: a collection of successful matches will be returned
    """
    # objective: iterate through all multi-pronunciation entries
    successful_word_bin = []
    base_stress = primary_stress
    accum = 1 # pushed up one to account for keyname being stored in pos 0
    accum2 = 0
    for i in multi_dict:
        length = multi_dict[i] + 1
        successful_word_bin = word_processing(successful_word_bin, length, accum, accum2, base_stress, wo
    for i in word_dict:
        if i not in multi_dict:
            length = 2 #only one pronunciation to check against
            successful_word_bin = word_processing(successful_word_bin, length, accum, accum2, base_stress
    return successful_word_bin


def word_processing(successful_word_bin, length, accum, accum2, base_stress, word_dict, i, multi_dict, wo
    """
    Description: performs the grunt work of comparison for single and multi-pronunciation words
    * called by process_staging, which organizes its focus to one group or the other
    Parameters: successful_word_bin, length, accum, accum2, base_stress, word_dict, i, multi_dict, word
    Returns: successful_word_bin
    Pre-condition: process_staging will have called it to process multi-pronunciation functions or single
    Post-Condition: returns a collection of successful matches
    """
    while accum2 != len(base_stress): #need to do it several times if there are multiple pronunciations f
        respective_stress = base_stress[accum2]
        accum3 = 1
        while accum != length: #repeat for number of pronunciations of 'word'
            x = word_dict[i][0][0] #the second zero gets me the string out of a list form
            primary_stress = phoneme_processing(x, multi_dict, word_dict)
            if primary_stress != 'fail': #trap for words w/o primary stress
```

```
                while accum3 != len(primary_stress) + 1: #repeats for comparisons with multiple pronuncia
                    if word_dict[word][accum][respective_stress:] == word_dict[i][accum3][primary_stress|
                        if word_dict[word][accum][respective_stress-1:respective_stress] != word_dict[i]|
                            successful_word_bin = successful_word_bin + [word_dict[i][0]]
                    accum3 = accum3 + 1
            accum = accum + 1
        accum2 = accum2 + 1
    return successful_word_bin


def print_function(successful_word_bin):
    """
    Description: prints the results
    Parameters: successful_word_bin
    Returns: none
    Pre-condition: none
    Post-Condition: results will be printed
    """
    accum = 0
    while accum != len(successful_word_bin):
        print(successful_word_bin[accum])
        accum = accum + 1


main()
```