Word Wrangler

Help Center

Overview

In this mini-project we will create a simple word game. The game will take an input word and then generate all valid words that can be created using the letters in the input word. You will then play the game by guessing all of the words. The objective of this mini-project is to work with ordered lists and recursion. While it is possible to write the functions in other ways, we strongly encourage you to follow the spirit of the mini-project. The only way to become more comfortable with recursion is to write recursive functions!

We have provided a working GUI and a class to manage the state of the game. You will be responsible for writing several helper functions. These functions will be used by the provided code in order to build the list of valid words that are composed of letters from the input word.

Testing your mini-project

The provided template includes the stubs for the functions that you will need to implement for this mini-project. Remember you should be testing each function as you write it. Don't try to implement all of the functions. If you do, you will have errors that will interact in inexplicable ways making your program hard to debug.

Submit your code (with the calls to `run` commented out) to this Owltest page. Note that trying to debug your mini-project using the tests in OwlTest can be very tedious since they are slow and give limited feedback. Instead, we *strongly* suggest that you first test your program using your own test suite. Programs that pass these tests are much more likely to pass the OwlTest tests.

Remember that OwlTest uses Pylint to check that you have followed the coding style guidelines for this class. Deviations from these style guidelines will result in deductions from your final score. Please read the feedback from Pylint closely. If you have questions, feel free to consult this page and the class forums.

When you are ready to submit your code to be graded formally, submit your code to the CourseraTest page for this mini-project that is linked on the main assignment page.

Implementation

In this mini-project you should not use `set`, `sorted`, or `sort`. Don't be surprised when you lose points if you do. Again, the point of the mini-project is for you to improve your skills in working with ordered lists and recursion, not to write the most compact code possible using Python built-in functions/methods. It is important to understand the concepts involved in working with ordered data, as you will be able to apply those concepts to a much wider range of problems in the future. Further, not every language has efficient support for this type of data, so it is also important to understand how these functions might be implemented.

Important: *None* of these functions should mutate their inputs. You must leave the inputs as they are and return new lists.

Phase One

You should first write the functions `remove_duplicates(list1)` and `intersect(list1, list2)`. These functions can both be written iteratively (using loops, not recursion). All of the arguments to these

functions are lists in ascending order. Both functions should return new sorted lists.

- `remove_duplicates` should return a new sorted list with the same elements as the input, `list1`, but without any duplicate elements.
- `intersect` should return a new sorted list that contains only the elements that are in both input lists, `list1` and `list2`.

Remember that the input lists will already be sorted. You should exploit this fact to make these functions simpler to write than if you had to handle arbitrary lists.

Phase Two

You should next implement merge sorting. To do so, you will need to write two functions, `merge(list1, list2)` and `merge_sort(list1)`. The input lists to `merge` will both be lists in ascending order. The input to `merge_sort` will be an unsorted list. While you could write `merge` recursively, you should write it iteratively because it will generate too many recursive calls for reasonably sized lists. However, you must write `merge_sort` recursively.

- `merge` should return a new sorted list that contains all of the elements in either `list1` and `list2`.
- `merge_sort` should return a new sorted list that contains all of the elements in `list1` sorted in ascending order.

Again, remember that the input lists to `merge` will already be sorted. You should exploit this fact to make it simpler to write. Further, `merge_sort` should *use* merge to help sort the list!

Phase Three

Now, you should implement `gen_all_strings(word)`. This function is the heart of the word wrangler game. It takes a string as input and returns a list of strings. It should return *all* possible strings that can be constructed from the letters in the input `word`. More formally, it should return all strings of length 0 to `len(word)` that can be composed of the letters that occur in `word`, in any order. Further, each letter should be considered distinct, so if the same letter appears twice in the word, then the output will have duplicate strings.

For example, if `word` is `"aab"`, `gen_all_strings` would return `["", "b", "a", "ab", "ba", "a", "ab", "ba", "aa", "aa", "aab", "aab", "aba", "aba", "baa", "baa"]`. Notice that the string `"aa"` appears twice in the output. This is because each `a` is considered distinct, so these two strings correspond to the two different orderings of the two `a` letters in the input word. Note that it does not matter in what order the strings appear in the list. The functions you have already written will be used afterwards to sort and remove duplicates from the final lists (you should *not* do this within `gen_all_strings`).

Note that this function is similar (but *not* identical) to something that you have previously written in this course. This time, however, ordering of the output list is not important, duplicates are allowed, and you must write it recursively. The basic idea is as follows:

1. Split the input `word` into two parts: the first character (`first`) and the remaining part (`rest`).
2. Use `gen_all_strings` to generate all appropriate strings for `rest`. Call this list `rest_strings`.
3. For each string in `rest_strings`, generate new strings by inserting the initial character, `first`, in all possible positions within the string.
4. Return a list containing the strings in `rest_strings` as well as the new strings generated in step 3.

This is a rough outline of one approach that will allow you to generate all strings that can be composed

from the letters of `word`. (Remember to think about the base case!)

Final Touches

Finally, you should implement `load_words(filename)` which will load a "dictionary" (not a Python dictionary, but rather a conventional dictionary with valid English language words) from a file. The dictionary file is simply one string per line, where each string is a valid word in the game. You can see the dictionary file here. You may use either `urllib2.urlopen` (either in CodeSkulptor or locally) or `open` (only if you are running locally and have downloaded the file) to open the file. Note that this function will not be tested. You only need implement it if you wish to play the word wrangler game using the provided GUI.

Created Wed 14 May 2014 8:53 AM PDT

Last Modified Tue 18 Aug 2015 2:25 PM PDT