Containers

The goal of this assignment is to give you a chance to practice using lists, tuples, dictionaries, and sets.

There are three parts to this assignment. None of the individual functions require much code, but writing concise, elegant solutions may take a little thought.

Part 1: Anagrams

Two strings are anagrams if the characters in the first word can be rearranged to make the second word, e.g. 'cinema' and 'iceman' are anagrams. "dormitory" and "dirtyroom" are also anagrams.

You may assume for this assignment that both strings must have the same length to be anagrams.

Part 1.1: Anagrams using only strings and lists

Implement (including automated unit tests) a function *anagram(str1, str2)* that returns True if str1 and str2 are anagrams ***using only lists***.

Hint: Your anagram() function needs only one line

You'll find more hints in Canvas but see if you can come up with a solution before checking the hint.

Note: A student was recently asked to solve this problem at a job interview with Microsoft.

Part 1.2: Anagrams using defaultdict

Part 1.1 was fun but didn't require much programming so we'll look at another solution to determining if two strings are anagrams using defaultdicts. Write a function anagram_dd(str1, str2) that also returns True or False, but ***uses the following strategy***:

1. Create a defaultdict of integers, dd
2. Go through str1, adding each character to dd as the key and incrementing the value by 1. E.g. say str1 == "dormitory". After this step, dd == {'d': 1, 'o': 2, 'r': 2, 'm': 1, 'i': 1, 't': 1, 'y': 1}
3. Go through each character, c, in str2. if the character is a key in dd, then decrement the value of dd[c] by 1. if c is not a key in dd, then you know that str1 and str2 are not anagrams because str2 has a character c that is not in str1.

4. If you successfully processed all of the characters in str2, then iterate through each of the values in dd to insure that every value == 0. This must be true if the two strings are anagrams. In our example above, dd == {'d': 0, 'o': 0, 'r': 0, 'm': 0, 'i': 0, 't': 0, 'y': 0} if str2 == "dirtyroom". If any value is not 0, then return False, else return True.

You might find Python's **any(iterable)** statement helpful depending upon your solution. **any(iterable)** evaluates to True if any of the elements in the iterable are True. E.g. any(0, 0) == False,

any(0, 1, 2, 3) == True

Part 1.3: Anagrams using Counters

Determine if two strings are anagrams ***using only collections.counters***.

Hint: My solution has only one line using the most_common() method. Careful though, the order of the elements in most_common() may be different if two elements occur the same number of times.

Part 2:

Write a function covers_alphabet(sentence) that returns True if sentence includes at least one instance of every character in the alphabet or False ***using only Python sets***. E.g.

covers_alphabet("AbCdefghiJklomnopqrStuvwxyz") is True
covers_alphabet("We promptly judged antique ivory buckles for the next prize") is True
covers_alphabet("xyz") is False
Hint: Be sure to convert the input string to lower case before comparing.
Hint: Here's a list of all characters that you can copy/paste rather than typing it yourself
abcdefghijklomnopqrstuvwxyz
Hint: Here's a few strings that **do** cover all the characters in the alphabet:

- abcdefghijklomnopqrstuvwxyz
- aabbcdefghijklomnopqrstuvwxyzzabc
- The quick brown fox jumps over the lazy dog
- We promptly judged antique ivory buckles for the next prize

Hint: The string tested against the alphabet must include at least all of the characters in the alphabet, but may also contain others, E.g. the sentence, "The quick, brown, fox; jumps over the lazy dog!" is not grammatically correct but it does cover the alphabet.

Part 3: Book index

A book index lists all of the words in the book along with a unique list of pages where that word occurs anywhere in the book.

Write a function ***book_index(words)***, and automated tests, to create a book index. The input to ***book_index()*** is a list of the form

```
[('word1', 1), ('word2', 2), ('word1', 1), ('word1', 3)]
```

where each item in the list is a tuple with a word and the page where the word appears. Note that a word may appear on many different pages and a word may appear multiple times on a single page. Your ***book_index()*** function should return a list of the form

```
[['word1', [1, 3]], ['word2', [2]]
```

where each word appears once in ***alphabetical order*** along with a ***sorted list of distinct pages*** where the word occurs. The items in the result should be ***sorted by words and the pages should be sorted in ascending order.***

For example,

```
# sample input

woodchucks = [('how', 3), ('much', 3), ('wood', 3), ('would', 2), ('a', 1),
              ('woodchuck', 1), ('chuck', 3), ('if', 1), ('a', 1), ('woodchuck', 2),
              ('could', 2), ('chuck', 1), ('wood', 1)]
book_index(woodchucks) should return
    [['a', [1]], ['chuck', [1, 3]], ['could', [2]], ['how', [3]], ['if', [1]], ['much', [3]],
     ['wood', [1, 3]], ['woodchuck', [1, 2]], ['would', [2]]]
```

Note that the words and pages are in ascending order.
Hint: My implementation of the book_index() function has only 4 lines of code and uses defaultdict(set), and list comprehensions. You may choose any solution you like. You may create your own input to your book_index() function or you may use my example.