To practice using dictionaries, we shall tackle a simple counting task — specifically, counting the words that appear in a text, as a simple case of lexical analysis.

Download a set of text files from slack which will serve as our data. You should store these files in a sensible location on your drive. Your own code should sit in the same folder. Open the file mobydick.txt, which contains the text of Melville's classic novel Moby Dick.

To simplify later tasks, I have preprocessed the text, converting it to lowercase and removing punctuation. The other files are similarly preprocessed.

Setting up your code files: For this lab, you will need to define a number of functions. Write these definitions in a file "countwordsFunctions.py". Put your code that tests these functions into a separate code file "testingCountwords.py" (which therefore needs to import from the first file).

Task 1: Counting words

Our first task is to define a function countWords that will read through a file, and count the words within it into a dictionary. Your function definition should therefore proceed as follows:

1. Create an empty dictionary.
2. Open the file for reading — the name of the file should be an 'input parameter', i.e. specified in the function call, as in (e.g.): countWords('mobydick.txt')
3. Use a for-loop to iterate through the file, reading in the lines of text, one at a time.
4. Count each word in the line into the dictionary — we'll come back to this step in a moment.
5. When counting is complete, use a return statement to return the dictionary of counts.

To get the words from a line of text, we can use the ".split()" method, which divides up a string at the places where spaces appear, returning the sub-strings as a list.

Hence, in our function, as we read through the file, we can call the .split() method on each line, and use an embedded for-loop to iterate through the words returned, counting each into the dictionary. The difficulty in coding this task is avoiding errors from trying to look up keys (words) not already present in the dictionary. Hence, we must first check if a word is present (as shown earlier). If it is, we add one to its existing score. If it is not, we simply assign it a count of 1.

Start from small:
Test your function definition by applying it to file mobypara.txt (containing a short extract from Moby Dick). Print out the dictionary of counts it returns, to check that the results look okay.

Task 2: Sorting and ranking

Next, define a function printTop20, which is given a dictionary of counts, and prints out the 20 words with the highest frequencies, e.g. so a call printTop20(counts) would print out the 20 words with the highest counts, in descending order of frequency, each along with its count (e.g. in the form “word = count”, one word per line). Use file mobypara.txt for testing, and refer to the slides on “Sorting Dictionaries by Value” for help. When your definition works, apply it to the file mobydick.txt to determine the most common words in this large English text.

Task 3: Stopwords

If your code works, you’ll find the most common terms to be the, of, and, a, to, in, etc., which are common in all English texts. Such words are of little use for discriminating between texts on different topics (e.g. sport vs. politics), a fact addressed in language processing applications (e.g. information retrieval: the technology behind web browsers) by putting them into a list of so-called stopwords — words that are ignored during the general counting of words in texts.

The file stopwords.txt, contains a list of stopwords for English. Write a function readStopWords which reads in the words, and returns them as a list of strings. The function might be called (e.g.) as : stops = readStopWords(’stopwords.txt’) Warning: although the file has only one word per line, the lines of text that you read from it are not the correct strings for the words, because they include a final line break character that needs to be stripped off (e.g. using the “.strip()” method, as in: “word = line.strip()”).

Having defined the readStopWords function, modify your definition of countWords so that it takes a second parameter — a list of stopwords — and then only counts words from the text that are not stopwords. (You can check that an item I is not in a list L with the test “I not in L”.)

Apply your modified definition of countWords to the full Moby Dick text, whilst supplying it with a list of stopwords, and print out the top-ranked words of the text by frequency again. Compare this to the set of words produced without the stopword list (which you should be able to reproduce now by supplying your new function definition with an empty stopword list). The two sets should look very different. Which do you think better captures the ‘topic’ of the text?

Task 4: Similarity (optional/challenge)

The data (set of text files) includes files george01.txt . . . george04.txt, which are news articles (that have again been preprocessed), which each mention a George: two concerning

the death of the famous footballer George Best; the other two another George. A key question is whether we can detect that two files share the same topic based on the words that they share.

We can measure similarity by computing a simple metric of lexical overlap that ignores the counts of the words in the texts, and instead simply asks what proportion of the distinct words found in either file are shared. To count the number of words shared, we can simply iterate through one dictionary (i.e. using a for-loop), and test each word's presence in the second dictionary (running a count of the words found in both). We can find out how many words there are in a single dictionary by using the len function (e.g. so "len(d)" returns the number of keys in dictionary d). If we simply add together the sizes of the two dictionaries, then we end up counting those words that appear in both dictionaries twice. However, we can correct for this by subtracting our count of the words in the overlap. Thus, if our dictionaries d1 and d2 share N words, then our similarity score would be "N / (len(d1) + len(d2) - N)" (taking care to use integer division).

Define a function similarity, which takes two dictionaries of word counts as arguments, and computes the above measure of similarity. Apply your function to compute similarity scores for each pair of texts from the 'George' collection. Do the scores help identify the documents that share a topic? Are the scores better for this purpose when you do/do not use a list of stopwords?