Python Practice 7: Python Dictionaries and Sorting — *Lexical analysis*

This lab provides an opportunity practice using Python *dictionaries* — a useful data structure, that provides a convenient basis for storing and handling data in many different circumstances.

1 Python Dictionaries

The essential purpose of a dictionary is very simple: to associate *keys* and *values*. Keys can be strings (e.g. `"bill"`), or numbers (e.g. `55`), or even tuples (e.g. the tuple '`("bill","bryson")`'), but **not** ordinary lists. The stored values may be any Python value; strings and numbers are common cases. A key can be paired with *at most one* value, i.e. if we assign a new value to a key, its previous value is lost. Refer to the slides of Lecture 6 for a refresher on Python dictionaries.

Getting started

Start by trying out some simple operations with the Python interpreter. You can create an *empty* dictionary by assigning the value '`{ }`'. This is illustrated in the example on the right, which also shows cases of assigning a value to a new key, and of assigning a new value to an existing key. Study the example thoroughly, and be sure you understand each step. When you've done so, try creating a dictionary of your own, to store phone numbers for a few people you know.

```
>>> salary = {}
>>> salary['al'] = 20000
>>> salary['bo'] = 50000
>>> salary
{'bo': 50000, 'al': 20000}
>>> salary['bo']
50000
>>> salary['bo'] = 55000
>>> salary['al'] += 2000
>>> salary
{'bo': 55000, 'al': 22000}
```

Next explore what happens if we do a *simple iteration* over a dictionary, e.g. a loop of the form "`for VAR in DICT:`". Try such a loop with your dictionary, and print the values assigned to `VAR` as the loop runs, so you can see what is assigned, i.e. does it print keys, values or key:value pairs? In the light of this, modify your loop to print the dictionary pairs as statements of the form "`key = value`", e.g.:

```
al = 20000
bo = 50000
ced = 1500
```

Now try doing look-up for a key that is *not in the dictionary*. This gives an error, as shown on the right. Avoiding such errors (which will crash your code) is a major issue in using dictionaries.

```
>>> salary['dave']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    salary['dave']
KeyError: 'dave'
```

We can check if a key is present in a dictionary by using a test of the form "`KEY in DICT`", which returns `True` or `False`, as shown on the right. Thus, we can avoid key look-up errors by couching the look-up step within a conditional that has such a test.

```
>>> 'dave' in salary
False
>>> 'bo' in salary
True
```

2 Exercises: lexical analysis

To practice using dictionaries, we tackle a simple *counting task* — specifically, counting the words that appear in a text, as a simple case of *lexical analysis*. Download the following files, for use in the lab: `mobydick.txt`, `mobypara.txt`, `george01.txt`, `george02.txt`, `george03.txt`, `george04.txt`, and `stopwords.txt`. These are mostly plain text files, for use as data. Store them in a sensible location on your U 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 "`lab7_countwords.py`". Put your code that tests these functions into a separate code file, "`lab7_testing.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 over 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, e.g. as on the right:

```
>>> s = 'this is a line'
>>> s.split()
['this', 'is', 'a', 'line']
```

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 over 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.

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`, provided with the other lab data files, 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 *linebreak* 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* a 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

The lab data 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 *shared* words, we can simply iterate over one dictionary (i.e. using a `for`-loop), and for each word test its 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)`".

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?