Zipf's Law

In this unit, we'll be verifying the pattern of word frequencies known as Zipf's Law. Zipf's idea was pretty simple. Imagine taking a natural language corpus and making a list of all the words ranked by frequency. For instance, 'the' might be the most common word, and we'd give it rank 1. 'of' might be the second, which gets rank 2. And 'cassowary' might bring up the rear with rank 1082 or whatever. The "Law" is that the frequency of the word with rank *n* is proportional to *1/n*. (In other words, 'the' is around twice as common as *of*, and a thousand times more common than 'cassowary'.)

The easiest way to check Zipf's Law for a particular corpus is to plot the frequencies of the words in rank order on a log-log graph. To the extent that the Law is true for that dataset, the points should appear in a straight line. So we'll start by writing a program that does this. And we will use as our corpus "Alice in Wonderland" ("alice.txt" on the Carmen website for the class).

Software design

This preliminary discussion gives us our first ideas about what the program we're writing needs to do.

- Compute the frequency of each word in the corpus
- Rank the words by frequency
- Make a plot of the data

So our program will probably have three parts, one for each operation. Let's focus on the first one for now. To compute the frequency of each word, we'll need to:

- Open the corpus file
- Read in each utterance
- Tokenize it into words
- Check the identity of each word and add 1 to the appropriate counter

The first three steps of this process are pretty straightforward from the last assignment. But the fourth step might take a bit more work.

You might imagine doing it with separate counter variables, the way we kept track of the total of words spoken by men and by women in the first assignment:

```
#------------------initialization of tracking variables-------------
countThe = 0
countOf = 0
...                    #lots of variables
countCassowary = 0

#------------------processing--------------------
if word == "the":
  countThe += 1
elif word == "of":
  countOf += 1
...                    #lots of 'if's
elif word == "cassowary":
  countCassowary += 1
```

But of course this isn't really practical.

Instead, we're going to need some kind of **data structure**. This is a variable, or a group of variables, that store some complex information in a way that allows us to get to specific pieces of it when we need them.

What data structure you should use for a given problem is partly a matter of clarity (you want something that makes sense and is easy to use) and partly of efficiency (you want your code to run quickly).

Before going on to the next section, think about how you'd want to represent the counts of words.

Parallel lists

Counting words with lists

The first structure we'll look at is a pair of *parallel lists*. We'll store all the words in the first list (in the order they first occur in the text). And in the second list, we'll store the counts of those words. So, for instance, the beginning of the lists will be:

```
LIST A: ["alice's", 'adventures', 'in', 'wonderland', 'lewis', 'carroll', 'the', 'millennium', 'fulcrum']
LIST B: [11, 4, 351, 2, 1, 1, 1605, 1, 1]
```

Our data structure is meant to represent a mapping from *word -> count*. In an abstract sense, a list is a mapping from *index -> data*; that is, we can ask for list[0] and get back the appropriate value. So what we've done with the parallel lists is build the following:

```
LIST A: index -> word
LIST B: index -> count
```

To derive *word -> count*, we need to find the *index* which maps to *word* and then check the corresponding *count*.

Python provides several builtin functions which can make this task easier, but for now, we should do things from the bottom up to make sure we understand how they work.

So we will first start to find the correct *index*. Build a simple example program which defines a list and outputs the index in this list of a given word. For example, we will defined the following list and search for the index of the word "adventures". For the word “adventures”, the index should be 1.

```
lexicon = ["alice's", 'adventures', 'in', 'wonderland', 'lewis', 'carroll', 'the', 'millennium', 'fulcrum', 'edition']
searchFor = "adventures"              #we can change this word around

#your code goes here

print("the index corresponding to", searchFor, "is", wordIndex)
```

Like some of our previous programs, this one has an initializer block that runs once, and then a processing loop. Since we're looking for an index, we'll need a way to count up the numbers from 0 to some stopping point. Python offers two ways to do this: the *range* function takes an integer *n* and returns the list *[0, 1... n - 1]*. The *enumerate* function takes a list *[a0, a1, ...an]* and returns the list *[(0, a0), (1, a1)..., (n, an)]*. For instance, you can try out the following:

```
for (ind, item) in enumerate(["alice's", "adventures", "in"]):
    print("item", ind, "is", item)
```

Our code will look like this:

```
lexicon = ["alice's", 'adventures', 'in', 'wonderland', 'lewis', 'carroll', 'the', 'millennium', 'fulcrum', 'edition']
searchFor = "adventures"

wordIndex = None

for (ind, item) in enumerate(lexicon):
  if(item == searchFor):
    wordIndex = ind

print("the index corresponding to", searchFor, "is", wordIndex)
```

Software engineering interlude: functions

The index-finding program is a little sub-element of our Zipf's Law program; we've written and tested it independently, and we're eventually going to end up using it several times. We could copy and paste it into all those places, but this makes our code longer and harder to read--- and if we need to modify the index-finder, we may end up having to make the same change three or four times. Instead, we can **abstract** it into a function. Functions that you write are just like Python's builtin functions, like *len* and *print*. They take *arguments* and *return* a *value*. To create your own function, you use the **def** keyword:

```
def findIndex(searchFor, lexicon):
    [statements]
```

The *def* block *defines* a new function with the name you specify (here, *findIndex*). All the statements in the indented block under *def* belong to the function. Somewhere inside the function, you can use the keyword *return*, followed by an expression. If the function hits that line, it stops executing, and whatever was returned is the *value* of the function.

As an example, here is a fairly trivial function:

```
def addTwo(num):
    print("I am adding two to", num)
    return num + 2
```

You can imagine Python function definition to transform a function *call* like *mySum = addTwo(23)* into:

```
#------variable binding------------
num = 23
#------function body--------------
print("I am adding two to", num)
returnValue = num + 2
#------after the function---------
mySum = returnValue
```

Actually, this imagined view isn't quite correct... the real answer involves the concept of *variable scope*, and in particular what happens if you're already using the variable name *num* outside the function, for example *num = 10*. In this case, *num* has the value of *23* inside the function, but when the function is over, it snaps back to its previous value of *10*. At this stage, a more detailed explanation is likely to be more confusing than helpful.

Now write the *findIndex* function. (You might wonder what happens if the word you're looking for isn't *in* the lexicon. In this case, we still need to return something... a good choice is the special Python value *None*, which is Python's default way of saying something is not found or not defined.)

Write some test code to check whether it works on a few simple cases.

Our findIndex function will be as follows:

```
def findIndex(word, lexicon):
   #----check if the word is already in the lexicon
   for (ind,lexeme) in enumerate(lexicon):
      if lexeme == word:
         return ind
   return None
```

Back to word counting: updating our data structure

With the *findIndex* function, we're making progress towards our parallel list implementation. What we want to do is cycle through all the words in the *Alice* corpus, and for each one, get its index with *findIndex* and increment the corresponding count. Doing this should be straightforward (using the cycling-through-words code from the previous assignment as a model, and the *findIndex* function you just wrote)... except for the case where *findIndex* returns *None*.

We can imagine what this case is like. Your lexicon contains a list of words in the order of first appearance, and your counts contain the number of times you've seen each one so far. For instance, they may look like this:

```
lexicon = ["alice's", 'adventures', 'in',]
counts = [1, 1, 1]
```

Your next word is "wonderland", which you've never seen before. Once you've incorporated it into the data structure, what should it look like?

We'll need to tack "wonderland" on to the end of the *lexicon* list, and we'll need to add another slot for a number to the end of the *counts* list. The best way to do this is with the **append** method (this is another class function, like *split*). The first argument to *append* (before the dot) is a list, and *append* takes its second argument (inside the parentheses) and glues it on to the end.

```
>>> myList = [1, 2, 3]
>>> myList
[1, 2, 3]
>>> myList.append(6)
>>> myList
[1, 2, 3, 6]

>>> myList = ["apple", "banana", "orange"]
>>> myList.append("grape")
>>> myList
['apple', 'banana', 'orange', 'grape']
```

We should lengthen both lists only in the case where the index of the word is *None*--- otherwise, we'd end up putting words into the lexicon more than once.

Now you have all the pieces to write the word counting program with parallel lists. Go ahead and write it; your program should read in all of *alice.txt* and compute the count for each word.

Print the frequency of the following words: “Alice”, “cassowary”, “roses”, “rabbit”, “mirror”.

How long does the whole script take? You can find out using *time* at the command line.

```
$ time python3 countWords.py
real   0m1.239s
user   0m1.216s
sys    0m0.008s
```

The time you're probably interested in is the *real* or *wallclock* time, which is a straightforward measure of how long the program took to execute. (The other two explain how this time is partionned between your code and the operating system. Usually they aren't very important.)

Efficiency interlude

In the next section, we'll look at a data structure that outperforms the parallel list structure for counting the words in *alice.txt*. What's wrong with the parallel lists? Recall that, conceptually, parallel lists had this structure:

```
LIST A: index -> word
LIST B: index -> count
```

These arrows show operations which are fast. If we have an index, we can find both the corresponding word and count. However, our program goes in the opposite direction. In order to obtain *word -> count*, we had to invert the *index -> word* mapping to obtain a *word -> index*. How long does it take our program to do this? In the worst case (where the word isn't in the lexicon at all), we have to check every word in the lexicon in order to figure out the answer.

What we really need is a structure where the arrow goes the other way: *word -> index*, or even just *word -> count*. In the next section, we'll see that Python supports such a structure: the **dictionary**. We won't go into detail about how a dictionary works, but you can gain some insight by thinking about a real-life paper dictionary.

A paper dictionary has the structure *word -> definition*. If you want to look up the word *cassowary*, you can use alphabetic comparison to find it quickly. First pull out volume *C*, eliminating 25 other volumes in a single operation! Open the volume to the middle (perhaps *cobra*); *cassowary* has to be before that, so flip back to *ceilidh* and then perhaps overshoot to *caracara* before ending up on the right page, where you can search word-by-word until you find what you want. Out of the entire dictionary, you've looked at only ten or eleven entries before ending up where you want to be.

Computer *dictionary* structures use all these ideas to ensure a fast *key -> value* mapping. They separate the data into *volumes* (this strategy is called *hashing*). They sort things to enable a quick back-and-forth search (*search trees*). And when the data is narrowed down enough, they flip through it entry-by-entry (*linear search*, which is what we did for the whole dataset in the previous section). Understanding the design and implementation of these kinds of structures is an entire course all on its own. We won't go into more detail here; the thing to remember is that a Python dictionary maps quickly from *key -> value* for *any* key, just as a list maps from *index -> value*.