Zipf's Law (part 3)

So far we wrote an efficient script to keep track of all the frequencies of the words in a given corpus. To check Zipf's Law on our corpus, there are still two steps left: sorting the words by their frequency and plotting the data.

Sorting

Python allows you to sort or rank data using the *sorted* function. You can read more about this function with *help*.

```
>>> help(sorted)
Help on built-in function sorted in module builtins

sorted(...)
   sorted(iterable, key=None, reverse=False) --> new sorted list
(END)
```

sorted takes a list (or list-like structure) as an argument and returns a sorted version. It also takes two optional arguments, *key* and *reverse*. (You can read more about various kinds of arguments in the tutorial).

Try out *sorted* at the Python prompt. Check out what happens on the following lists:

```
[5, 2, 7, 0]
["cassowary", "rhea", "moa", "ostrich", "emu"]
```

How does this relate to operations like *2 < 5* or *"rhea" < "moa"*?

Now try again, but this time, use:

```
>>> sorted([5, 2, 7, 0], reverse=True)
```

In the function definition, *reverse=False* means that if you do not specify a *reverse* argument, it defaults to *False*; if you do specify one, it is used that instead.

The *key* argument is a little more complicated. To see why we need it, try adding these lines to the end of your script that creates the dictionary with *word -> count* on our corpus. Note that you might not have used the same variable name as me for your dictionary (I used *counts*):

```
sortedCounts = sorted(counts, reverse=True)
print(sortedCounts[:10])
```

What do you think this will print? Try it out. Were you correct?

We need both the words and their counts, so it's probably a good idea to use *counts.items()*. But this doesn't work either. What happens?

```
sortedCounts = sorted(counts.items(), reverse=True)
print(sortedCounts[:10])
```

We get:

```
[('zigzag,', 1), ('zealand', 1), ("youth,'", 3), ('youth,', 3),
("yourself.'", 2), ("yourself,'", 1), ('yourself,', 1),
("yourself!'", 1), ('yourself', 5), ('yours:', 1)]
```

This is because Python orders lists and list-like structures (like the ordered pair *('zealand', 1)*) lexicographically: it compares the first element first, and only uses the second as a tie-breaker. The *key* argument changes this behavior. *key* is a function which gets run on every element of the list; the elements are then sorted by their *key* value. For instance:

```
def thirdLetter(string):
    return string[2]

print(sorted(["cassowary", "rhea", "moa", "ostrich", "emu"]))

print([thirdLetter(word) for word in
      ["cassowary", "rhea", "moa", "ostrich", "emu"]])

print(sorted(["cassowary", "rhea", "moa", "ostrich", "emu"],
             key=thirdLetter))
```

What do each of these lines print? Why?

Write a function called *getSecond* that returns the second element of an ordered pair. Use it to sort the dictionary and print the 10 words with the largest counts. What are these words?

To write the *getSecond* function, think about the following. What does it take as argument? What does it return?

```
def getSecond(pair):
    return pair[1]
```

Plotting with Python

Now that we've sorted everything, we can go ahead and create the Zipf plot. We'd like to make a scatterplot on a log-log scale. To get a better idea of what we want to do, take a look at the graph in the “related laws” section of the Zipf’s law Wikipedia page.

We want to create a graph with logarithmic scales on both axes. The x-axis will be the ranks of the words and the y-axis will be the frequencies of the words. We'll take the logarithms using the *math* module, and create the graphic itself using the matplotlib library. Remember that we need to import the modules.

Thus to make the plot, we need the log of the rank (or index) and the log frequency for each word. Note that we'll need to convert the indices from 0-based to 1-based, since *log(0)* is not defined. We are going to store these logs in two lists: one for the x-axis and one for the y-axis, say *logInds* and *logCounts* respectively.

Given our sorted list of (word, count) pairs, how do we get access to the indices and the pairs?

Remember the *enumerate* method? Using it, we will get what we want. We can simply iterate over the output of the *enumerate* function on our sorted list, and populate our lists *logInds* and *logCounts*.

```
for (ind, (word, count)) in enumerate(sortedCounts):
        [compute the logs of the rank and the frequency,
        and add them to our logInds and logCounts lists]
```

Next, use web search to find out how to use the scatterplot commands from Matplotlib and use them to create your plot.

What shows up? My plot isn't exactly a straight line... but over much of its length, the trend is roughly linear.

Plotting with R

We can also just use R to plot. R works easily with CSV (comma-separated values) files (where each line has the same number of fields, separated by a comma). So our script will output a CSV file with the necessary information. We will have a field for rank, another one for word, and a last one for frequency. So for instance the first few lines of the CSV file for "alice.txt" will be:

```
1,the,1605
2,and,766
3,to,706
4,a,614
...
```

To write a CSV file, we will use the CSV module. This will allow us to define a writer object, in which we can add lines using the writerow method. This method takes a list as argument. Each element in the list corresponds to a field in the CSV file:

```
import csv

writer = csv.writer(file(output_filename, 'w'), delimiter=',')
writer.writerow([1, "the", 1605])
```

Here below is some R code to help you with plotting (You can find the testZipf.csv file on Carmen). The first bit of code uses the ggplot library:

```
# read the data (csv format) assuming it has a header row
data = read.csv("testZipf.csv")
# call the relevant library for plotting
library(ggplot2)
# plot the data (Rank on x-axis, Frequency on y-axis)
qplot(log(data$Rank), log(data$Frequency))
```

You can also just use the plot method in R:

```
# read the data (csv format) assuming it has a header row
data = read.csv("testZipf.csv")
# plot the data (Rank on x-axis, Frequency on y-axis)
plot(log(data$Rank), log(data$Frequency))
```

Zipf for bigrams

A *bigram* is a pair of adjacent words in a text; for instance, if our string is:

```
alice's adventures in wonderland
```

The bigrams are:

```
alice's adventures
        adventures in
                   in wonderland
```

(We might imagine adding dummy *START* and *END* tokens, so that each real word was a member of two bigrams. But for our purposes here, this isn't necessary.)

Bigrams are a very cheap approximation of some local syntactic and semantic processes; if a word occurs in many bigrams, we might imagine it is more productive, while if it occurs only in a few, it might be very contextually dependent. Since single words follow a Zipf-like law, we might expect bigrams to do the same... but how similar will the plots be? Are rare *bigrams* relatively more common than rare single words? And do common pairs of words dominate the distribution more or less than common single words?

We'll find out by adjusting our code to count bigrams. First, write code that prints out all the bigrams in a list of words like *["alice's", "adventures", "in", "wonderland"]* as shown in the example above. Each bigram contains a word and the following word.

There are two ways to do this--- one which uses a numerical index, and another using list slices (*lst[begin:end]* notation). Either is fine.

Now copy your word counting program and modify it using this code to count bigrams. One way to do this is to make the keys in the dictionary tuples which are written *(word1, word2)* using parentheses rather than square brackets.

Python *tuples* are just like *lists*, except you can't assign to them using the *tpl[1] = 2* notation. Lists aren't allowed as dictionary keys because this assignment operation might cause odd behavior...

```
mydict = {}
keyOne = [1, 2]
mydict[keyOne] = 1 #will actually throw an error, but pretend it works
keyTwo = [1, 3]
mydict[keyTwo] = 2
keyTwo[1] = 2      #dictionary now has two entries with key [1, 2]
print(mydict[ [1, 2] ]) #what should this do?
```

So Python just doesn't let you do this. Instead you have to use tuples:

```
mydict = {}
keyOne = (1, 2)
mydict[keyOne] = 1 #this is fine
keyTwo = (1, 3)
mydict[keyTwo] = 2
keyTwo[1] = 2      #but you aren't allowed to do this!
```

Use your program to make the plot for bigrams. What does it look like?