## Black Problem 3: Markov Text Generation! [35 points; individual or pair]

**Copied from: https://www.cs.hmc.edu/twiki/bin/view/CS5/MarkovMaterialBlack on 3/22/2017**

Submit this program as `hw10pr3.py` on the usual submission site...

Your goal in this section is to write a program that is capable of "learning" English from examples of text and then generating new "meaningful" English random text all by itself! The quotes in that sentence are important, because - as you will see - the learning and meaning are questionable at best. However, the algorithm can, at times, produce surprising results!

In any case, you will accomplish this goal by writing a Markov text-generation algorithm.

## The basic idea:

English is a language with a lot of structure. Words have a tendency (indeed, an obligation) to appear only in certain sequences. Grammatical rules specify legal combinations of different parts of speech. E.g., the phrase "The cat climbs the stairs" obeys a legal word sequence. "Stairs the the climbs cat", does not. Additionally, semantics (the meaning of a word or sentence) further limits possible word combinations. "The stairs climb the cat" is a perfectly legal sentence, but it doesn't make any sense and you are very unlikely to encounter this word ordering in practice.

Even without knowing the formal rules of English, or the meaning of English words, we can get an idea of what word combinations are legal simply by looking at well-formed English text and noting the combinations of words that tend to occur in practice. Then, based on our observations, we could generate *new* sentences by randomly selecting words according to commonly occurring sequences of these words. For example, consider the following text:

"I love roses and carnations. I hope I get roses for my birthday."

If we start by selecting the word "I", we notice that "I" may be followed by "love", "hope" and "get" with equal probability in this text. We randomly

select one of these words to add to our sentence, e.g. "I get". We can repeat this process with the word "get", necessarily selecting the word "roses" as the next word. Continuing this process could yield the phrase "I get roses and carnations". Note that this is a valid English sentence, but not one that we have seen before. Other novel sentences we might have generated include "I love roses for my birthday," and "I get roses for my birthday".

More formally, the process we use to generate these sentences is called a *kth-order Markov process*. A *kth-order Markov process* is a process where the next word depends only on the previous *k* words. A very special case is a first-order Markov process (when *k = 1*) in which the next word depends only on the previous one.

## Functions to write

First, here are two functions your program should have:

- `def markov_model(text, k)`

  The `markov_model` function should accept a single (potentially large) string of text (`text`) and should return a `k`-th order Markov model based on that text. Specifically, that Markov model should be a Python dictionary. For many more details, see the *Details and hints* section below. For now, we will show three examples of `markov_model` in action. Remember that the order of elements in Python dictionaries does not matter. (We provide a function below that can compare two dictionaries for equivalence.)

```
>>> text = "A B C. A B B C B. A B C C C D B B. B B C C D D B C."
>>> mm1 = markov_model(text, 1) # first-order Markov model
>>> mm1
{('C',): ['B.', 'C', 'C', 'D', 'C', 'D'],
 ('$',): ['A', 'A', 'A', 'B'],    # things that are first or follow a period
 ('A',): ['B', 'B', 'B'],
 ('B',): ['C.', 'B', 'C', 'C', 'B.', 'B', 'C', 'C.'],
 ('D',): ['B', 'D', 'B']}

>>> mm2 = markov_model(text, 2) # second-order Markov model
>>> mm2
{('B', 'C'): ['B.', 'C', 'C'],
 ('$', '$'): ['A', 'A', 'A', 'B'],
 ('D', 'D'): ['B'],
 ('$', 'A'): ['B', 'B', 'B'],
```

```
 ('C', 'C'): ['C', 'D', 'D'],
 ('$', 'B'): ['B'],
 ('A', 'B'): ['C.', 'B', 'C'],
 ('D', 'B'): ['B.', 'C.'],
 ('C', 'D'): ['B', 'D'],
 ('B', 'B'): ['C', 'C']}

>>> mm3 = markov_model(text, 3) # third-order Markov model
>>> mm3
{('$', '$', 'B'): ['B'],
 ('A', 'B', 'B'): ['C'],
 ('C', 'C', 'D'): ['B', 'D'],
 ('B', 'B', 'C'): ['B.', 'C'],
 ('C', 'D', 'D'): ['B'],
 ('$', 'B', 'B'): ['C'],
 ('$', '$', '$'): ['A', 'A', 'A', 'B'],
 ('C', 'C', 'C'): ['D'],
 ('B', 'C', 'C'): ['C', 'D'],
 ('$', 'A', 'B'): ['C.', 'B', 'C'],
 ('C', 'D', 'B'): ['B.'],
 ('$', '$', 'A'): ['B', 'B', 'B'],
 ('A', 'B', 'C'): ['C'],
 ('D', 'D', 'B'): ['C.']}
```

Second, you should have a `gen_from_model` function that generates words from a model. In this case, we will print the words, rather than return them... .

- `def gen_from_model(mmodel, numwords)`

  The `gen_from_model` function should accept a Markov model named `mmodel`, generated by the `markov_model` function above, and an integer, `numwords`, which is the number of words for it to print from that model. Then `gen_from_model` should determine the *order* of the Markov model (`mmodel`) and generate `numwords` words from it, starting with the all-`'$'` tuple (see below). This function does not need to return anything. Here is an example with the simple text used above. Note that the generated "sentences" can have as many `C`s in a row as the random-number generation allows, but it can never have more than two `B`s or two `D`s in a row. A sentence can never include an `A` after the first letter, and it's impossible for a `B` to follow a `C`, unless it ends the "sentence." These are some of the characteristics that we'll look for when running your code!
- `>>> text = "A B C. A B B C B. A B C C C D B B. B B C C D D B C."`

```
>>> d = markov_model(text, 2)
>>> gen_from_model(d, 100)
A B C B. B B C C D B B. A B B C B. B B C C C D B B. B B C C D D B B.
B B C C D B B. B B C B. A B B C B. A B C. A B C B. A B C C D D B B. A
B B C B. A B C C D B C. A B B C B. A B B C B. B B C C D B B. B B C B.
B B C B.
```

## Details, hints, and other requirements...

- Using the "main" trick, your program will start automatically.
- Your program will begin by asking the user how many training files they will provide. (A training file is just a file containing some sample text.)
- The program then queries the user for the names of those training files, which will be read in by the program.
- Next, the user is asked to specify the value of *k*, the order of the Markov process.
- You should read in the contents of each file as a string and then create a very large string to pass into `markov_model` as the input named `text`.
- Within the `markov_model` function, note that you can turn a string into the list of words in that string this way: If `myString` is a string, then `myString.split()` returns a list of the words in that string. Cool! `split()` is a method of the `str` (string) class!
- The program builds a dictionary of sequences of length *k* that appear in the text. For each sequence of length *k*, the dictionary remembers the list of all words that can follow that sequence. For example, assume that we had just one training file with the text `I like spam and I like chocolate. I like spam a lot!` If we were using *k=2* then the sequences of *k=2* consecutive words would be `I like`, `like spam`, `spam and`, `and I`, etc. The sequence `I like` would be a key in the dictionary and would be associated with the words that follow it, namely `spam`, `chocolate`, and `spam`. Notice that `spam` appears twice there. That's good! That indicates that `spam` is twice as likely as `chocolate` to appear after the pattern `I like`.
- Every Markov model built for this assignment should have a "starting" `k`-tuple consisting only of `'$'`, e.g., `('$',)` for `k = 1`, and `('$', '$') for k = 2`, and `('$', '$', '$')` for `k == 3`, and so on... .
- As words are handled, they will first become the final (highest-index) element of the following `k`-tuple. Then they'll "move" to the left with each new addition to the tuple. Slicing can help here!
- The program now generates the `N` words!

- As wods are generated, we proceed as follows. We start with a sequence of *k* words which are simply this: `('$', '$', ..., '$')` (with `k` dollar signs. Then this *k*-word sequence is used to select a random next word using the Markov dictionary of valid next words. The next word should be selected at random but weighted by its probability of occurring as a next word (e.g. in the example above, `spam` would be twice as likely as `chocolate` to appear after `I like`).
- If your program ever encounters a *k*-word sequence that never had a word after it, it should simply "back up" the tuple (adding a `'$'` on the left and removing the rightmost element). This can continue for as long as necessary. Since the initial tuple is `('$',)*k`, this process will always *eventually* yield a valid key in the Markov model.
- For this problem, ***leave the punctuation at the ends of the words***. These signal the end of a sentence and, thus, the start of a new sentence. As a result of this, the words `spam` and `spam.` in the original files will be handled quite differently by the text generation!
- After a sentence-ending word has been generated, if you can't find a following word you should reset the process (simply make the next key the original one: `('$',)*k`) so that you start over cleanly.
- Each time you see a punctuation symbol (period, exclamation point, or question mark) generated, that ends one sentence. Thus, the sentence "I like spam." ending in a period would look like it comprises the words `I`, `like`, and `spam.` where the last word is spam with a period at the end. There might be another word `spam` as well, but these would be treated differently!
- To use randomness in your program, you can simply `import random` at the top of your file. This module provides many functions. For example, if `L` is a list then `random.choice(L)` will return a randomly selected element of that list. For more documentation on `random`, see this page.

## Your magnum opus: the 500-word essay, Markov-style!

Finally, create a Markov model with input text and an order (`k`) of your choice, and then generate at least 500 words from your model.

Describe the input texts and the approach you took to building your model (i.e., the order of your model) in a comment at the bottom of your `hw10pr3.py` file.

Then, paste the results of your "writing" into a comment underneath that explanation! If you have any favorite excerpts, call them out in the introduction!

Have fun with the Markov text generation!

The text you want to use is up to you. Dr. Seuss or Nursery Rhymes? Song lyrics? Or more esoteric things, such as IRS documents like this one (http://www.irs.gov/uac/IRS-Says-Offshore-Effort-Tops-$5-Billion,-Announces-New-Details-on-the-Voluntary-Disclosure-Program-and-Closing-of-Offshore-Loophole may be good. Random scenes from Shakespeare might also strike your fancy (http://www-tech.mit.edu/Shakespeare/). Please play with your program and try using some of your own text. Be sure to share your results, as noted above!

Submit your code as `hw10pr3.py` to the usual place...