Strings + Markov

Due 11:59 PM, Thursday, February 28, 2018

Introduction

Text analysis presents an interesting but complex problem—paragraphs that are easy for humans to understand contain implicit linguistic structures that can subtly change the meaning of text and therefore are difficult to algorithmically parse by computers. Statistical models can also be developed to quantify how similar one piece of text is to another, such as for plagiarism detection or searching for similar articles to an interesting one you found in an academic journal.

Markstrings (or Strings + Markov) uses several Python string parsing exercises to allow you to become familiar with built-in Python methods and culminates in the creation of a simple, automatic text-generation algorithm. You will use the skills you develop with this project in your next text-processing project, called Modeling.

Installation and Handin

Project setup. For each project, there may be support files that you will need to complete the assignment. These can be copied to your home directory by using the `cs4_install` command in a CIT Terminal window. For this project, type the command:

```
cs4_install markstrings
```

There should now be a `markstrings` folder within your `projects` directory. Using Terminal, you can move into the folders with the cd command:

```
cd ~/course/cs0040/projects/markstrings
```

Project hand-in. When you're ready to submit your project files, run:

```
cs4_handin markstrings
```

from a CIT Terminal window from your `~/course/cs0040/projects/markstrings` directory, and the entire contents of the directory will be handed in.

You can resubmit this assignment using the `cs4_handin` command at any time, but only your most recent submission with be graded.

Specification

In lecture, we've seen how every data value in Python is really an object, and that objects have functions inside them that are called *methods*. In this portion of the project, you will first use some of the string methods that we discussed in lecture to complete several exercises designed to develop your knowledge of Python string parsing.

After that, you will write a program that is capable of generating meaningful text all by itself by implementing what is known as a Markov text-generation algorithm.

Checklist

To help you organize your project workflow, we have provided a checklist of the required functions you need to write (and you're welcome to write other helper functions for code cleanliness and readability).

Note that this checklist is intentionally underspecified in some places to keep it concise—after the checklist, we've provided more detailed specifications to follow in your implementation.

Section A: Strings

Your answers for this section go in the `strings.py` file. Implement the following functions:

- ❏ **`count_ignore_case(s, sub)`:** takes in a string `s` and substring `sub` and return the number of occurrences of `sub` in `s` (ignoring cases of the letters)
- ❏ **`middle_name(fullname)`:** takes a string fullname that represents a person's full name, returns a string representing the person's middle name.
- ❏ Tests for `count_ignore_case` and `middle_name`

Section B: Markov

Your answers for this section go in the `markov.py` file. Implement the following functions:

- ❏ **`create_dictionary(filename)`:** takes a string representing the name of a text file, and that returns a dictionary of key-value pairs in which:
  - Each key is a word encountered in the text file
  - The corresponding value is a list of words that follow the key word in the text file
- ❏ **`generate_text(word_dict, num_words)`:** takes in a dictionary of word transitions (generated by the create_dictionary function) and a positive integer num_words to generate a string of num_words words
- ❏ Tests for `create_dictionary` and `generate_text`

Section A: Strings

Put your answers for this problem in `strings.py`.

count_ignore_case(s, sub)

Write a function `count_ignore_case(s, sub)` that takes a string `s` and a substring `sub`, and that *uses one or more Python string methods* to compute and return the number of occurrences of `sub` in `s`, but ignores the cases of the letters involved. For example:

```
>>> count_ignore_case('Yes, yes, YES!', 'yes')
3
>>> count_ignore_case('Yes, yes, YES!', 'YES')
3
>>> count_ignore_case('Yes, yes, YES!', 'yEs')
3
>>> count_ignore_case('Yes, yes, YES!', 'no')
0
```

Notes:

- The cases of the letters in *both* strings does *not* matter when finding the count.
- Your function *must* make use of one or more string methods to determine the count.
- You should *not* need to use a loop or recursion.
- Ensure that your function *returns* the appropriate integer, rather than printing it.

middle_name(fullname)

Write a function `middle_name(fullname)` that takes a string fullname that represents a person's full name, and that *uses one or more Python string methods* to extract and return a string representing the person's middle name. For example:

```
>>> middle_name('Martin Luther King')
'Luther'
>>> middle_name('Sarah Jessica Parker')
'Jessica'
```

If the full name has *three or more* components, you should return the *second* component. Do this even if there are more than three components. For example:

```
>>> middle_name('Jose Antonio Dominguez Banderas')
'Antonio'
```

If the full name has *fewer than three* components, you should return an empty string. For example:

```
>>> middle_name('Abraham Lincoln')
''
>>> middle_name('Madonna')
''
```

Notes:

- Your function *must* make use of one or more string methods to determine the middle name. You should *not* need to use a loop or recursion.
- Make sure that your function *returns* the appropriate string, rather than printing it.

Section B: Markov

Put your answers for this problem in the file named `markov.py`.

Algorithm Overview

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. For example, 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 phrase) further limits possible word combinations. “The stairs climb the cat” is a nearly legal sentence, but it doesn’t make 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 which word combinations are allowed simply by looking at large amounts of well-formed English text (training data) and noting the combinations of words that tend to occur in practice. Based on our observations, we can generate new sentences by randomly selecting words according to commonly occurring sequences within the training data. 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: *“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 used to generate these sentences is called a *first-order Markov process*. A first-order Markov process is a process in which the state at time t + 1 (i.e., the next word) depends only on the state at time t (i.e., the previous word). In a second-order Markov process, the next word would depend on the two previous words, and so on. Our example above was a first-order process because the choice of the next word depended only on the current word.

Implementing a first-order Markov text generator will involve writing two functions: one to process a file and create a dictionary of legal word transitions, and another to actually generate the new text.

We will consider words to be different even if they only differ by capitalization or punctuation. For example, 'spam', 'Spam', and 'spam!' will all be considered distinct words.

create_dictionary(filename)

Write a function `create_dictionary(filename)` that takes a string representing the name of a text file, and that returns a dictionary of key-value pairs in which:

- each key is a word encountered in the text file
- the corresponding value is a list of words that follow the key word in the text file.

For example, the dictionary produced for the text *"I love roses and carnations. I hope I get roses for my birthday."* would include the following key-value pairs, among others:

```
'I': ['love', 'hope', 'get']
'love': ['roses']
'roses': ['and', 'for']
'my': ['birthday.']
# as well as others!
```

Guidelines

- You should *not* try to remove the punctuation from the words of the text file.
- The keys of the dictionary should include every word in the file *except* the *sentence-ending words*. A sentence-ending word is defined to be any word whose last character is a period ('.'), a question mark ('?'), or an exclamation point ('!'). A sentence-ending word *should* be included in the lists associated with the words that it follows (i.e., in the value parts of the appropriate key-value pairs), but it *not* appear as its own key.
- If a word `w1` is followed by another word `w2` multiple times in the text file, then `w2` should appear multiple times in the list of words associated with `w1`. This will allow you to capture the frequency with which word combinations appear.
- In addition to the words in the file, the dictionary should include the string `'$'` as a special key referred to as the *sentence-start symbol*. This symbol will be used when choosing the first word in a sentence. In the dictionary, the list of words associated with the key `'$'` should include:
    - the first word in the file
    - every word in the file that follows a sentence-ending word.
- Doing this will ensure that the list of words associated with `'$'` includes all of the words that start a sentence. For example, the dictionary for the text *"I scream. You scream. We all scream for ice cream."* would include the following entry for the sentence-start symbol:

```
'$': ['I', 'You', 'We']
```

You may find it helpful to consult the `word_frequencies` function from lecture. We will also discuss some additional strategies for `create_dictionary` in lecture.

Testing

To test your code, we have provided a file called `sample.txt`:

```
# sample.txt
A B A. A B C. B A C. C C C.
```

Try this in the command line:

```
>>> word_dict = create_dictionary('sample.txt')
>>> word_dict
{'A':['B', 'B', 'C.'], 'C':['C', 'C.'], 'B':['A.', 'C.', 'A'],
 '$':['A', 'A', 'B', 'C']}
```

The order of the keys–or of the elements within a given key's list of values–may not be the same as what you see above, but the elements of the lists should appear in the quantities shown above for each of the four keys `'A'`, `'B'`, `'C'`, and `'$'`.

Here are some additional files you can use for testing:

- `brown_vision.txt` - an edited version of Brown's vision statement
- `brave.txt` - lyrics from the song *Brave* by Sara Bareilles, and its dictionary
- `alice.txt` - the text to Alice in Wonderland by Charles Lutwidge Dodgson

Here again, the ordering that you obtain for the keys and list elements in the dictionaries may be different. In addition, we have edited the formatting of the dictionaries to make them easier to read.

We have also provided a file entitled `brave_dictionary.txt,` which contains the entire dictionary generated from our solution code run on `brave.txt`

generate_text(word_dict, num_words)

Write a function `generate_text(word_dict, num_words)` that takes as parameters a dictionary of word transitions (generated by the `create_dictionary` function) named

`word_dict` and a positive integer named `num_words`. The function should use `word_dict` to generate and print a string of `num_words` words.

Guidelines

- The first word should be chosen randomly from the words associated with the sentence-start symbol, `'$'`. The second word should be chosen randomly from the list of words associated with the first word, *etc*. When the current word ends in a period (`'.'`), question mark (`'?'`), or exclamation point (`'!'`), the function should detect this and start a new sentence by again choosing a random word from among those associated with `'$'`.
- Do not include `'$'` in the output text. It should only be used as an internal marker for your function.
- You can use the `random.choice` function to choose from a list of possible words. Don't forget to include `import random` at the top of your file.
    - For example, if `wordlist` is the list of possible words at a given point in the generated text, you can use `random.choice(wordlist)` to select a random word from `wordlist`.
- Here again, you shouldn't try to remove or change the punctuation associated with the words, and you don't need to worry if the generated text doesn't end with appropriate punctuation. The generated text won't be perfect, but most of the time it will at least be meaningful!
- If your function encounters a word that doesn't have any words associated with it in the dictionary, the function should start a new sentence. This situation can occur if the last word in the file used to create the dictionary was unique and did not end with punctuation.

Testing

Below are two examples using the same text file as above. Your output may differ because of the randomness involved in the generation.

```
>>> word_dict = create_dictionary('sample.txt')
>>> generate_text(word_dict, 20)
B C. C C C. C C C C C C C C C C. C C C. A
>>> generate_text(word_dict, 20)
A B A. C C C. B A B C. A C. B A. C C C C C C.
```

Try some other examples using longer documents containing English words, such as the works of William Shakespeare. In particular, we have provided a text file named `romeo.txt` that contains the first act of *Romeo and Juliet*, along with the files that we provided above in the examples for `create_dictionary`.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this document or have any concerns about this and any other CS4 document by posting on Piazza or filling out our anonymous feedback form.