Homework 5

As usual, turn in a file named `hw5.py`.

1. Import the `gutenberg` corpus and get the text of Jane Austen's *Emma* as a list of tokens. Store the result in the variable `emma`. For example:

```
>>> len(emma)
192427
```

2. The time-consuming part of importing the texts from `nltk.book` is actually creating the `Text` objects from the lists of tokens. For most purposes, the token-list is a perfectly fine representation of the text.

a. Write a function called `get_texts` that takes a corpus as argument. It should go through the files of the corpus, and get the token-list for each. The return value is the list of token-lists. For example:

```
>>> from nltk.corpus import gutenberg
>>> texts = get_texts(gutenberg)
>>> texts[0] == emma
True
```

b. Nonetheless, having the names for texts is certainly convenient. NLTK actually gets the name from the token-list. The Gutenberg texts all begin with the token `'['`, followed by the name, then the token `']'`. Write a function `textname` that takes a token-list as input and returns the name, as a string, as output. If the token-list does not begin with `'['`, it should just take the first five tokens. You may assume that a token-list that starts with `'['` always contains a matching `']'`, and a token-list that does not start with `'['` is at least five tokens long. For example:

```
>>> textname(['[', 'Hi', 'there', ']', 'this', 'is', 'a', 'test'])
'Hi there'
>>> textname(['A, 'test', '1, '2, '3, 'foo', 'bar'])
'A test 1 2 3'
>>> textname(emma)
'Emma by Jane Austen 1816'
```

(Tip: you will want to use the list method `index`.)

3. Unigrams

a. Lowercase the tokens in `emma` and create a frequency distribution from them. (Do not throw away punctuation.) Store the result in `fd1`.

b. Set `A3b` to the count of the word `'town'` in `fd1`.

c. Set `A3c` to the relative frequency (probability) of the word `'town'` in `ud`.

d. Set `A3d` to the number of hapaxes in the distribution `fd1`.

4. When one formats floating-point numbers, one can specify the number of digits after the decimal point as follows:

```
>>> '{:.4}'.format(1/7)
'0.1429'
```

Write a function `print_uni` that takes a FreqDist as input and prints a table with three columns: a word, its count, and its relative frequency. It should print the words in alphabetic order. The first column should be 10 characters wide. If a word is more than 10 characters long, truncate it to 10 characters. The second column should be five characters wide, and the relative frequency should be printed with four decimal positions. Columns should be separated by single spaces. For example:

```
>>> x = FreqDist(['a', 'a', 'a', 'b', 'b', 'c', 'antidisestablishmentarianism'])
>>> print_uni(x)
a              3 0.4286
antidisest     1 0.1429
b              2 0.2857
c              1 0.1429
```

5. Implement `bigrams` as discussed in HO 6, #3. It should return a generator, and should include the wrap-around bigram.

a. Set `cfd1` to the conditional frequency distribution of *Emma.* As before, downcase the words but do not throw away punctuation. Use your new implementation of `bigrams`. You should have:

```
>>> fd1.N()
192427
>>> cfd1.N()
192427
```

b. Set `A5` to be the expected count of the bigram *miss woodhouse* in *Emma.* The Python expression you write should refer *only* to `fd1` and `cfd1`. It should *not* refer back to the original text or its bigrams.

c. Let us generalize what we did in part (a). Define a **model** for a text to be a pair `(fd, cfd)`, where `fd` is the word unigram distribution of the text, represented as a `FreqDist`, and `cfd` is the word conditional distribution, represented as a `ConditionalFreqDist`. Write a function `text_model` that takes a token-list as input and produces a model as output. Downcase words, but do not throw away punctuation. Use the wrap-around definition of `bigrams`. For example:

```
>>> emma_model = text_model(emma)
>>> emma_model[0] == fd1
True
>>> emma_model[1] == cfd1
True
```

d. The following list pairs function names with mathematical expressions. In each case, you should define a function with the given name. The function should take three arguments, *m, x, y*, where m is a model and x and y are words (that is, strings). The function should return the value of the given mathematical expression.

`condprob`	$p(y\|x)$
`biprob`	$p(x,y)$
`bicount`	$c(x,y)$
`estbiprob`	$\hat{p}(x,y)$
`estbicount`	$\hat{c}(x,y)$
`pmi`	$\mathrm{pmi}(x,y)$

For example:

```
>>> condprob(emma_model, 'miss', 'woodhouse')
0.2888146911519199
>>> bicount(emma_model, 'miss', 'woodhouse')
173.0
>>> pmi(emma_model, 'miss', 'woodhouse')
2.2493409523102224
```

6. Write a function called `collocs` that takes two arguments: a model and a minimum count. It should produce a list of triples (v,x,y), where (x,y) is one of the bigrams that occurred in the original text, and v is its value for `pmi`. The output list should be sorted from highest `pmi` value to lowest, and it should only include pairs whose count is greater or equal to the specified minimum count. (Tip: loop over the keys of the unigram distribution to get possible first words x, then loop over the distribution `cfd[x]` to get possible second words y. But don't bother looping over `cfd[x]` if x itself occurs fewer times than the minimum count.) For example:

```
>>> cs = collocs(emma_model, 5)
>>> cs[0]
(4.205084763069562, 'brunswick', 'square')
```

If you do it right, and use a minimum count of 5 or more, the function should return within a second or two. If you do the loops wrong, it could take a *very* long time to finish computing.

7. Write a function called `print_collocs` that takes one argument, a list of collocations. It should print out the first 20 collocations in the list. (If there are fewer than 20, it should print them all.) Print the first word in the first column, the second word in the second column, and the pmi value in the third column. The first and second columns should be 10 characters wide, and the pmi value should be printed with 4 digits after the decimal. There should be a single space between columns. For example:

```
>>> print_collocs([(2.4, 'hi', 'there'), (1.8, 'foo', 'bar')])
hi         there      2.4000
foo        bar        1.8000
```

Discussion

8. Play around with the threshold that you give to `collocs`. Does it have an effect on the quality of the collocations you get?