Stemming, **Lemmatization, Regular Expressions and Tokenization**

Stemming and Lemmatization

We will use the emma text from the Gutenberg Corpus as we did before. Note that emmatokens has words with regular capitalization and emmawords has lower-case words with no capitalization.

NLTK has two stemmers, Porter and Lancaster, described in section 3.6 of the NLTK book. To use these stemmers, you first create them.

```
>>>porter = nltk.PorterStemmer()
>>>ancaster = nltk.LancasterStemmer()
```

Then we'll compare how the stemmers work using both the regular-cased text and the lower-cased text.

```
>>>emmaregstem = [porter.stem(t) for t in emmatokens]
>>>emmaregstem[1:100]
>>>emmalowerstem = [porter.stem(t) for t in emmawords]
>>>emmalowerstem[1:100]
```

Try the same examples with the Lancaster stemmer.

NLTK suggests that we try building our own simple stemmer by making a list of suffixes to take off.

```
def stem(word):
  for suffix in ['ing','ly','ed','ious','ies','ive','es','s']:
    if word.endswith(suffix):
      return word[:-len(suffix)]
  return word

#try the above stemmer with 'friends'
stemmedword=stem('friends')
stemmedword
```

NLTK has a lemmatizer that uses the WordNet on-line thesaurus as a dictionary to look up roots and find the word.

```
wnl = nltk.WordNetLemmatizer()
emmalemma=[wnl.lemmatize(t) for t in emmawords]
emmalemma[1:100]
```

Regular Expressions and Tokenization

So far, we have depended on the NLTK wordpunct tokenizer for our tokenization. Not only does the NLTK have other tokenizers, but we can custom-build our own tokenizer using regular expressions.

Text as Strings

We'll read text from file emma in the Gutenberg Corpus as before, but just leave it as raw text.

```
>>> import nltk
>>> from nltk import *
>>> file0 = nltk.corpus.gutenberg.fileids( ) [0]
>>> emmatext = nltk.corpus.gutenberg.raw(file0)
```

Remember that this text is a Python string. We'll quickly review some of the operations and functions that are used with strings. (See also section 3.2 in the NLTK book, http://www.nltk.org/book.)

Strings can be treated as lists of characters. So if we get the length of a string, it is the number of characters, and if we use indexing with square brackets [ ], we get substrings of characters.

```
>>> type(emmatext)              # type is string
>>> len(emmatext)               # number of characters in the book
>>> shorttext = emmatext[:150] # get the first 150 chars into the variable shorttext
```

If we wanted to do something to every character in the string, we can even loop over the characters; this will print the first 10 characters in the variable shorttext.

```
>>> for char in shorttext[:10]
        print char
```

We use the operator '+' to catenate strings together.

```
>>> string1 = 'Monty Python'
>>> string2 = 'Holy Grail'
>>> string1 + string2
>>> string1 + ' and the ' + string2
```

Also check out Table 3.2 to see other string functions. For example, we could use the function replace to replace all the new characters '\n' with a space ' '.

```
>>> newemmatext = emmatext.replace('\n', ' ')
>>> shorttext = newemmatext[:150]
>>> shorttext
```

We have redefined the variable shorttext to be the first 150 characters without newlines.

Regular Expressions for Tokenizing Text

Let's start our investigation of using regular expressions to tokenize text by looking at some simple patterns first. We'll start with a pattern for words that just finds alphabetic characters. There are several functions in the module re for finding how patterns match text, e.g. re.match finds any match at the beginning of a string, re.search finds a match anywhere in the string, and re.findall will find the substrings that matched anywhere in the string.

```
>>> import re
>>> pword = re.compile('\w+')
>>> re.findall(pword, shorttext)
```

This does fine on the alphabetic words of this simple text, noting that it ignores the number 1816, so let's get an example of text with some special characters to illustrate some other situations.

```
>>> specialtext = 'U.S.A. poster-print costs $12.40, with 10% off.'
>>> re.findall(pword, specialtext)
```

Getting only alphabetic text leaves lots of the string unmatched. Let's start making a more general regular expression to match tokens by matching words can have an internal hyphen. In this case, we need to put parentheses around the part of the pattern that can be repeated 0 or more times. Unfortunately, findall will then only report the part that matched inside those parentheses, so we'll put an extra pair of parentheses around the whole match.

```
>>> ptoken = re.compile('(\w+(-\w+)*)')
>>> re.findall(ptoken, specialtext)
```

re.findall has reported both the whole matched text and the internal matched text. We could fix this by using the re.groups function to access only the outer match, but instead we'll just ignore this for now while we're developing regular expressions. NLTK provides a more elegant way to represent the regular expressions later.

Now we try to make a pattern to match abbreviations that might have a "." inside, like U.S.A. We only allow capitalized letters.

```
>>> pabbrev = re.compile('(([A-Z]\.)+)')
>>> re.findall(pabbrev, specialtext)
```

This worked well, so let's combine it with the words pattern to match either words or abbreviations.

```
>>> ptoken = re.compile('(\w+(-\w+)*|([A-Z]\.)+)')
>>> re.findall(ptoken, specialtext)
```

Well, that didn't work because it first found the alphabetic words which found 'U', 'S' and 'A' as separate words before it could match the abbreviations. So the **order of the matching patterns**

really matters if an earlier pattern matches part of what you want to match. We can switch the order of the token patterns to match abbreviations first and then alphabetics.

```
>>> ptoken = re.compile('(([A-Z]\.)+|\w+(-\w+)*)')
>>> re.findall(ptoken, specialtext)
```

That worked much better. Now we'll add an expression to match the currency, and we'll put an 'r' in front of the string, Python's notation for a raw string. This accepts '\' as itself in a string.

```
>>> ptoken = re.compile(r'(([A-Z]\.)+|\w+(-\w+)*|\$?\d+(\.\d+)?)')
>>> re.findall(ptoken, specialtext)
```

We can keep on adding expressions, but the notation is getting awkward. We can make a prettier regular expression that is equivalent to his one by using Python's triple quotes (works for either """ or ''') that allows a string to go across multiple lines without adding a newline character. And we also use the regular expression verbose flag to allow us to put comments at the end of every line, which the re compiler will ignore.

```
ptoken = re.compile(r'''([A-Z]\.)+    # abbreviations, e.g. U.S.A.
      | \w+(-\w+)*                    # words with internal hyphens
      | \$?\d+(\.\d+)?                # currency, like $12.40
      ''', re.X)                      # verbose flag
```

But we still had to put in extra parentheses for the findall function, which messed up the output a lot. So NLTK has provided us with an even better way to write these expressions.

Regular Expression Tokenizer using NLTK Tokenizer
(From section 3.7 in the NLTK book http://www.nltk.org/book)

NLTK has built a tokenizing function that helps you write tokenizers by giving it the compiled pattern. Regular expressions can also be written down in the "verbose" version, using the (?x) flag that allows the alternatives to be on different lines with comments, and it also alleviates the need to put extra parentheses.

```
pattern = r''' (?x)           # set flag to allow verbose regexps
      ([A-Z]\.)+              # abbreviations, e.g. U.S.A
      | \w+(-\w+)*            # words with internal hyphens
      | \$?\d+(\.\d+)?%?      # currency and percentages, $12.40, 50%
      | \.\.\.                # ellipsis
      | [][.,;"'?():-_']      # separate special character tokens
      '''
```

As far as I can tell, the nltk function regexp_tokenize applies these regular expressions to text by applying each regular expression in order to get anything that matches as a token. We observed the importance of the order of expressions earlier, but also note that it is important that the expression to separate special characters as individual tokens comes last in the list, so that other expressions, such as the words with internal hyphens, can first get longer tokens that involve individual characters.

We put this pattern into NLTK's function for a regular expression tokenizer. Note the order of the arguments to this function; first give the text and then the pattern.

```
>>> nltk.regexp_tokenize(shorttext], pattern)
>>> nltk.regexp_tokenize(specialtext, pattern)
```

Note that if there are any characters not matched by one of the regular expression patterns, then it is omitted as a token in the result. (For some reason, the % is being ignored, and I didn't have time to debug it.)

Next, we'll try to make a regular expression tokenizer appropriate for tweet text. Some of the patterns in this tokenizer are taken from tweetmotif, a Python regular expression tokenizer written for tweets by Brendan O'Connor (http://tweetmotif.com/about). Here is a tokenizer:

```
tweetPattern = r''' (?x)     # set flag to allow verbose regexps
      (https?://|www)\S+     # simple URLs
      | (:-\)|;-\))          # small list of emoticons
      | &(amp|lt|gt|quot);   # XML or HTML entity
      | \#\w+                # hashtags
      | @\w+                 # mentions
      | \d+:\d+              # timelike pattern
      | \d+\.\d+             # number with a decimal
      | (\d+,)+?\d{3}(?=([^,]|$))  # number with a comma
      | ([A-Z]\.)+           # simple abbreviations
      | (--+)                # multiple dashes
      | \w+(-\w+)*           # words with internal hyphens or
                               apostrophes
      | ['\".?!,:;]+         # special characters
      '''
```

We need some examples to work with:

```
tweet1 = "@natalieohayre I agree #hc09 needs reform- but not by
crooked politicians who r clueless about healthcare! #tcot #fishy
NO GOV'T TAKEOVER!"

tweet2 = "To Sen. Roland Burris: Affordable, quality health
insurance can't wait http://bit.ly/j63je #hc09 #IL #60660"

tweet3 = "RT @karoli: RT @Seriou: .@whitehouse I will stand w/
Obama on #healthcare, I trust him. #p2 #tlot"
```

Try the regexp tokenizer on these tweets, for example:
`nltk.regexp_tokenize(tweet1,tweetPattern)`

Exercises

Stemming

For documents that come from NLTK corpora, read the NLTK book sections from Chapter 2 on the different corpora. Also note that Chapter 2 discusses how to load your own text with the PlainCorpusReader, or you can just read text from files.

Example of using word frequencies to analyze text:
Nate Silver's analysis of State of the Union Speeches from 1962 to 2010.
http://www.fivethirtyeight.com/2010/01/obamas-sotu-clintonian-in-good-way.html
Here is a statement of the question that he is trying to answer by looking at word frequencies to compare the SOTU speech in 2010 with earlier speeches:
"What did President Obama focus his attention upon and how does this compare to his predecessors?"
[And you may find it interesting to note also the criticism of the technique at http://thelousylinguist.blogspot.com/2010/01/bad-linguistics-sigh.html. I am not expecting you to be experts in how you categorize the words or bigrams that you use in your analysis (and I actually think that Nate Silver was not too far off the mark in picking categories of words to answer his question.)]

Tokenization

Choose one of the following, i.e. work with either the regular pattern or the tweet pattern in the tokenizer. You may work in groups.

1. Run the regexp tokenizer with the regular pattern on the sentence "Mr. Black and Mrs. Brown attended the lecture by Dr. Gray, but Gov. White wasn't there."
   a. Design and add a line to the pattern of this tokenizer so that titles like "Mr." are tokenized as having the dot inside the token. Test and add some other titles to your list of titles.
   b. Design and add the pattern of this tokenizer so that words with a single apostrophe, such as "wasn't" are taken as a single token.

2. Run the regexp tokenizer with the tweet pattern on the three example tweets.
   a. Design and add a line to the pattern of this tokenizer so that titles like "Sen." And "Rep." are tokenized as having the dot inside the token. Test and add some other titles to your list of titles.
   b. Design and add to the pattern of this tokenizer so that words with a single apostrophe, such as "can't" are taken as a single token.
   c. Design and add to the pattern of this tokenizer so that the abbreviation "w/" is taken as a single token.

****