In [1]:

```
import textblob
from textblob import TextBlob
text = 'Today is a beautiful day. Tomorrow looks like bad weather.'
blob = TextBlob(text)
blob
```

Out[1]:

```
TextBlob("Today is a beautiful day. Tomorrow looks like bad weather.")
```

In [2]:

```
# Stemming removes a prefix or suffix from a word leaving only a stem, which may or may
# not be a real word. Lemmatization is similar, but factors in the word's part of speech
# meaning and results in a real word.
# Stemming and lemmatization are normalization operations, in which you prepare
# words for analysis. For example, before calculating statistics on words in a body of t
# you might convert all words to lowercase so that capitalized and lowercase words are n
# treated differently. Sometimes, you might want to use a word's root to represent the w
# many forms. For example, in a given application, you might want to treat all of the fo
# words as "program": program, programs, programmer, programming and programmed
# (and perhaps U.K. English spellings, like programmes as well).
# Words and WordLists each support stemming and lemmatization via the methods
# stem and lemmatize. Let's use both on a Word:

from textblob import Word
word = Word('varieties')
word.stem()
```

Out[2]:

```
'varieti'
```

In [3]:

```
word.lemmatize()
```

Out[3]:

```
'variety'
```

In [17]:

```
# Various techniques for detecting similarity between documents rely on word frequencies
# As you'll see here, TextBlob automatically counts word frequencies. First, let's load
# for Shakespeare's Romeo and Juliet into a TextBlob. To do so, we'll use the Path class
# from the Python Standard Library's pathlib module:

import pathlib
from pathlib import Path
from textblob import TextBlob
blob = TextBlob(Path('romeoandjuliet.txt').read_text())

# When you read a file with Path's read_text method, it closes the file immediately afte
# finishes reading the file.
```

In [18]:

```
# You can access the word frequencies through the TextBlob’s word_counts dictionary.
# Let’s get the counts of several words in the play:

blob.word_counts['juliet']
```

Out[18]:

```
303
```

In [19]:

```
blob.word_counts['romeo']
```

Out[19]:

```
445
```

In [20]:

```
blob.word_counts['thou']
```

Out[20]:

```
277
```

In [21]:

```
# If you already have tokenized a TextBlob into a WordList, you can count specific
# words in the list via the count method:

blob.words.count('joy')
```

Out[21]:

```
14
```

In [22]:

```
blob.noun_phrases.count('lady capulet')
```

Out[22]:

```
5
```

In [23]:

```
# WordNet19 is a word database created by Princeton University. The TextBlob library use
# the NLTK library’s WordNet interface, enabling you to look up word definitions, and ge
# synonyms and antonyms.
# First, let’s create a Word:

from textblob import Word
happy = Word('happy')

# The Word class’s definitions property returns a list of all the word’s definitions in
# the WordNet database:

happy.definitions

# The database does not necessarily contain every dictionary definition of a given word.
# There’s also a define method that enables you to pass a part of speech as an argument
# you can get definitions matching only that part of speech.
```

Out[23]:

```
['enjoying or showing or marked by joy or pleasure',
 'marked by good fortune',
 'eagerly disposed to act or to be of service',
 'well expressed and to the point']
```

In [24]:

```
# You can get a Word’s synsets—that is, its sets of synonyms—via the synsets property. 
# result is a list of Synset objects:

happy.synsets
```

Out[24]:

```
[Synset('happy.a.01'),
 Synset('felicitous.s.02'),
 Synset('glad.s.02'),
 Synset('happy.s.04')]
```

In [25]:

```
# There’s also a get_synsets method that enables you to pass a part of speech as an argu
# so you can get Synsets matching only that part of speech.
# You can iterate through the synsets list to find the original word’s synonyms. Each
# Synset has a lemmas method that returns a list of Lemma objects representing the synon
# A Lemma’s name method returns the synonymous word as a string. In the following
# code, for each Synset in the synsets list, the nested for loop iterates through that S
# Lemmas (if any). Then we add the synonym to the set named synonyms. We used a
# set collection because it automatically eliminates any duplicates we add to it:

synonyms = set()
```

In [26]:

```
for synset in happy.synsets:
    for lemma in synset.lemmas():
        synonyms.add(lemma.name())
```

In [27]:

```
synonyms
```

Out[27]:

```
{'felicitous', 'glad', 'happy', 'well-chosen'}
```

In [28]:

```
# If the word represented by a Lemma has antonyms in the WordNet database, invoking the
# Lemma’s antonyms method returns a list of Lemmas representing the antonyms (or an empt
# list if there are no antonyms in the database).First, let’s get the Lemmas for the Syn
# at index 0 of the synsets list:

lemmas = happy.synsets[0].lemmas()
lemmas
```

Out[28]:

```
[Lemma('happy.a.01.happy')]
```

In [29]:

```
# In this case, lemmas returned a list of one Lemma element. We can now check whether th
# database has any corresponding antonyms for that Lemma:

lemmas[0].antonyms()
```

Out[29]:

```
[Lemma('unhappy.a.01.unhappy')]
```

In [30]:

```
# The result is list of Lemmas representing the antonym(s). Here, we see that the one an
# for 'happy' in the database is 'unhappy'.
```