In [51]:
```
from nltk.corpus import wordnet as wn
from nltk.wsd import lesk
from nltk.corpus import sentiwordnet as swn
```

WordNet is a library that primarily stores and outputs the various parts of speech and their synonyms for any particular word called synsets. The WordNet library also contains word to word relationships called synset relationships and the root form of a word called morphy.

In [6]:
```
synsets_list = wn.synsets('response')
synsets_list
```

Out[6]:
```
[Synset('response.n.01'),
 Synset('reaction.n.03'),
 Synset('answer.n.01'),
 Synset('reception.n.01'),
 Synset('response.n.05'),
 Synset('reply.n.02'),
 Synset('response.n.07')]
```

In [9]:
```
wn.synset('reception.n.01').definition()
```

Out[9]:
```
'the manner in which something is greeted'
```

In [11]:
```
wn.synset('reception.n.01').examples()
```

Out[11]:
```
['she did not expect the cold reception she received from her superiors']
```

In [12]:
```
wn.synset('reception.n.01').lemmas()
```

Out[12]:
```
[Lemma('reception.n.01.reception'), Lemma('reception.n.01.response')]
```

In [17]:
```
word = wn.synset('reception.n.01')
traverse_word = wn.synset('reception.n.01')
while len(traverse_word.hypernyms()) > 0:
    print(traverse_word.hypernyms()[0])
    traverse_word = traverse_word.hypernyms()[0]
```

```
Synset('greeting.n.01')
Synset('acknowledgment.n.03')
Synset('message.n.02')
Synset('communication.n.02')
Synset('abstraction.n.06')
Synset('entity.n.01')
```

The WordNet noun relationship structure is organized such that an entity is always at the top of the hierarchy. This may be because the most abstract form of any (physical or non-physical) object's description is an entity. As we traverse the hierarchy downwards, the specificity of the object becomes more apparent. For example, WordNet moves from abstraction, which could quite literally be any idea, to communication which has to do with the transfer of information. At the bottom, it is greeting, communicating some form of initial contact, which is the most closely specific idea/description of our original word.

In [19]:
```
verb_word = wn.synsets('decorate')
verb_word
```

Out[19]:
```
[Synset('decorate.v.01'),
 Synset('deck.v.01'),
 Synset('decorate.v.03'),
 Synset('dress.v.09')]
```

In [20]:
```
verb_word = verb_word[3]
verb_word.definition()
```

Out[20]:
```
'provide with decoration'
```

verb_word.examples()

In [22]:
```
verb_word.lemmas()
```

Out[22]:
```
[Lemma('dress.v.09.dress'), Lemma('dress.v.09.decorate')]
```

In [23]:
```
traverse_word = wn.synset('dress.v.09')
while len(traverse_word.hypernyms()) > 0:
    print(traverse_word.hypernyms()[0])
    traverse_word = traverse_word.hypernyms()[0]
```

```
Synset('arrange.v.01')
Synset('put.v.01')
Synset('move.v.02')
```

The WordNet verb relationship structure is organized such that the most abstract form of a verb word is shown -- which is move. Move describes any particular action such that it creates a flux in the physical world which is, as we previously stated, the most abstraction verb word. For verbs, it follows the same structure as nouns. As we can see, as we move down the specificity of the words becomes closer and closer to our original word.

In [31]:
```
wn.morphy('friendly')
```

Out[31]:
```
'friendly'
```

In [33]:
```
wn.morphy('dressing', wn.VERB)
```

Out[33]:
```
'dress'
```

In [34]:
```
wn.morphy('dressed', wn.VERB)
```

Out[34]:
```
'dress'
```

In [37]:
```
wn.wup_similaritt('')
```

Out[37]:
```
'dress'
```

In [39]:
```
wn.synsets('bottle')
```

Out[39]:
```
[Synset('bottle.n.01'),
 Synset('bottle.n.02'),
 Synset('bottle.n.03'),
 Synset('bottle.v.01'),
 Synset('bottle.v.02')]
```

In [41]:
```
wn.synsets('can')
```

Out[41]:
```
[Synset('can.n.01'),
 Synset('can.n.02'),
 Synset('can.n.03'),
 Synset('buttocks.n.01'),
 Synset('toilet.n.02'),
 Synset('toilet.n.01'),
 Synset('can.v.01'),
 Synset('displace.v.03')]
```

In [46]:
```
word1 = wn.synsets('bottle')[0]
word2 = wn.synsets('can')[0]
print('Wu-Palmer similarity:', wn.wup_similarity(word1, word2))
sentence1 = "I drank a lot of soda from my Coca Cola bottle".split()
sentence2 = "The can holds a lot of water".split()
word1_lesk = lesk(sentence1, 'bottle')
print(word1_lesk, word1_lesk.definition())
word2_lesk = lesk(sentence2, 'can')
print(word2_lesk, word2_lesk.definition())
```

```
Wu-Palmer similarity: 0.8235294117647058
Synset('bottle.n.02') the quantity contained in a bottle
Synset('can.v.01') preserve in a can or tin
```

The Wu-Palmer similarity score was able to associate the like-ness relationship between bottle and can generally accurately. Most likely, the ancestoral words for both of these words have to do with a medium of storage for some liquid or physical object. It's also interesting to note that the part of speech for 'can' used in the second sentence is meant to be a noun but the lesk algorithm recognizes as a verb. It seems as though both the Wu-Palmer and Lesk algorithm has significant flaws. For example, in the book, the Wu-Palmer alogrithm was not able to generally accurately describe the like-ness of hit and slap. Here, the Lesk algorithm fails to identify the appropriate parts of speech for one of the targetted words.

In [53]:
```
wn.synsets('collapse')
for word in wn.synsets('collapse'):
    print(word, word.definition())
```

```
Synset('collapse.n.01') an abrupt failure of function or complete physical exhaust
ion
Synset('collapse.n.02') a natural event caused by something suddenly falling down
or caving in
Synset('flop.n.04') the act of throwing yourself down
Synset('crash.n.03') a sudden large decline of business or the prices of stocks (e
specially one that causes additional failures)
Synset('collapse.v.01') break down, literally or metaphorically
Synset('break_down.v.08') collapse due to fatigue, an illness, or a sudden attack
Synset('collapse.v.03') fold or close up
Synset('crumble.v.01') fall apart
Synset('collapse.v.05') cause to burst
Synset('crack_up.v.01') suffer a nervous breakdown
Synset('collapse.v.07') lose significance, effectiveness, or value
```

In [55]:
```
collapse = swn.senti_synsets('collapse')
for word in collapse:
    print(word)
```

```
<collapse.n.01: PosScore=0.25 NegScore=0.5>
<collapse.n.02: PosScore=0.0 NegScore=0.375>
<flop.n.04: PosScore=0.0 NegScore=0.0>
<crash.n.03: PosScore=0.0 NegScore=0.0>
<collapse.v.01: PosScore=0.0 NegScore=0.0>
<break_down.v.08: PosScore=0.0 NegScore=0.25>
<collapse.v.03: PosScore=0.25 NegScore=0.0>
<crumble.v.01: PosScore=0.0 NegScore=0.0>
<collapse.v.05: PosScore=0.0 NegScore=0.0>
<crack_up.v.01: PosScore=0.0 NegScore=0.5>
<collapse.v.07: PosScore=0.5 NegScore=0.0>
```

In [69]:
```
senti_sentence = 'I collapsed when I heard the news'.split()
neg = 0
pos = 0
for word in senti_sentence:
    polarity_score = list(swn.senti_synsets(word))
    if polarity_score:
        score = polarity_score[3]
        print(score)
        pos += score.pos_score()
        neg += score.neg_score()
print("Negative:", neg)
print("Positive:", pos)
```

```
<one.s.01: PosScore=0.0 NegScore=0.25>
<crumble.v.01: PosScore=0.0 NegScore=0.0>
<one.s.01: PosScore=0.0 NegScore=0.25>
<hear.v.04: PosScore=0.0 NegScore=0.0>
<news.n.04: PosScore=0.125 NegScore=0.0>
Negative: 0.5
Positive: 0.125
```

The senti-synset scores provide a good overview of the sentiment of a given word or sentence; provided that the correct definition of a word is given. In this context, the polarity score of an entire sentence structure could possibly be zero if the definitions are incorrectly selected. In NLP applications, the senti-synset scores are a good base tool for measuring the

tones of a particular statement. This is important because emotions, whether positive, neutral, or negative, exist in language and identifying the intended emotional tone of a statement is key to accurately comprehending a sentence.

In [90]:
```
from nltk.book import*
text4
text4.collocations()
```

```
United States; fellow citizens; years ago; four years; Federal
Government; General Government; American people; Vice President; God
bless; Chief Justice; one another; fellow Americans; Old World;
Almighty God; Fellow citizens; Chief Magistrate; every citizen; Indian
tribes; public debt; foreign nations
```

In [97]:
```
import math
text = ' '.join(text4)
num_words = len(set(text))
years = text.count('years')/num_words
ago = text.count('ago')/num_words
years_ago = text.count('years ago')/num_words
pmi = math.log2(years_ago / (years * ago))
print(pmi)
```

```
-1.7675539139996292
```

The mutual information formula for the words 'years ago' describes the stickiness of these two words in this particular text. What I mean by stickiness is the tendency for these two words to be side by side. After computing the mutual information, the score can shows that the word 'years ago' are not as linked as previously thought in this text. Most likely either the word 'year' or 'ago' is used separate from each other more often than linked.