CHAPTER 1

Language Processing and Python

It is easy to get our hands on large quantities of text. What can we do with it, assuming we can write some simple programs? In this chapter we'll tackle the following questions:

1. what can we achieve by combining simple programming techniques with large quantities of text?
2. how can we automatically extract representative words and phrases from a large text?
3. is the Python programming language suitable for such work?
4. what are some of the interesting challenges of natural language processing?

This chapter is divided into sections that skip between two quite different styles. In the "computing with language" sections we will take on some linguistically-motivated programming tasks without necessarily understanding how they work. In the "closer look at Python" sections we will systematically review key programming concepts. We'll flag the two styles in the section titles, but later chapters will mix both styles without being so up-front about it. We hope this style of introduction gives you an authentic taste of what will come later, while covering a range of elementary concepts in linguistics and computer science. If you have basic familiarity with both areas you can skip to Section 1.6; we will repeat any important points in later chapters.

1.1 Computing with Language: Texts and Words

We're all very familiar with text, since we read and write it every day. But here we will treat text as raw data for the programs we write, programs that manipulate and analyze it in a variety of interesting ways. Before we can do this, we have to get started with the Python interpreter.

Getting Started

One of the friendly things about Python is that it allows you to type directly into the interactive interpreter — the program that will be running your Python programs. You

can access the Python interpreter using a simple graphical interface called the Interactive DeveLopment Environment (IDLE). On a Mac you can find this under *Applications→MacPython*, and on Windows under *All Programs→Python*. Under Unix you can run Python from the shell by typing `idle` (if this is not installed, try typing `python`). The interpreter will print a blurb about your Python version; simply check that you are running Python 2.4 or greater (here it is 2.5.1):

```
Python 2.5.1 (r251:54863, Apr 15 2008, 22:57:26)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If you are unable to run the Python interpreter, you probably don't have Python installed correctly. Please visit http://nltk.org/ for detailed instructions.

The >>> prompt indicates that the Python interpreter is now waiting for input. When copying examples from this book be sure not to type in the >>> prompt yourself. Now, let's begin by using Python as a calculator:

```
>>> 1 + 5 * 2 - 3
8
>>>
```

Once the interpreter has finished calculating the answer and displaying it, the prompt reappears. This means the Python interpreter is waiting for another instruction.

Try a few more expressions of your own. You can use asterisk (*) for multiplication and slash (/) for division, and parentheses for bracketing expressions. Note that division doesn't always behave as you might expect — it does integer division or floating point division depending on how you specify the inputs:

```
>>> 3/3
1
>>> 1/3
0
>>> 1.0/3.0
0.33333333333333331
>>>
```

These examples demonstrate how you can work interactively with the interpreter, allowing you to experiment and explore. Now let's try a nonsensical expression to see how the interpreter handles it:

```
>>> 1 +
  File "<stdin>", line 1
    1 +
      ^
SyntaxError: invalid syntax
>>>
```

Here we have produced a syntax error. It doesn't make sense to end an instruction with a plus sign. The Python interpreter indicates the line where the problem occurred (line 1 of "standard input").

The chapter contains many examples and exercises; there is no better way to learn to NLP than to dive in and try these yourself. However, before continuing you need to install NLTK and its associated data, downloadable for free from http://nltk.org/.

Searching Text

Now that we can use the Python interpreter, let's see how we can harness its power to process text. The first step is to type a line of magic at the Python prompt, telling the interpreter to load some texts for us to explore: `from nltk.book import *` (i.e. import all names from NLTK's book module). After printing a welcome message, it loads the text of several books, including *Moby Dick*. Type the following, taking care to get spelling and punctuation exactly right:

```
>>> from nltk.book import *
>>> text1
<Text: Moby Dick by Herman Melville 1851>
>>> text2
<Text: Sense and Sensibility by Jane Austen 1811>
>>>
```

We can examine the contents of a text in a variety of ways. A concordance view shows us every occurrence of a given word, together with some context. Here we look up the word *monstrous*.

```
>>> text1.concordance("monstrous")
mong the former , one was of a most monstrous size . ... This came towards us , o
ION OF THE PSALMS . " Touching that monstrous bulk of the whale or ork we have re
all over with a heathenish array of monstrous clubs and spears . Some were thickl
ed as you gazed , and wondered what monstrous cannibal and savage could ever have
 that has survived the flood ; most monstrous and most mountainous ! That Himmale
 they might scout at Moby Dick as a monstrous fable , or still worse and more det
ath of Radney .'" CHAPTER 55 Of the monstrous Pictures of Whales . I shall ere lo
ling Scenes . In connexion with the monstrous pictures of whales , I am strongly
>>>
```

Try seaching for other words; you can use the up-arrow key to access the previous command and modify the word being searched. You can also try searches on some of the other texts we have included. For example, search *Sense and Sensibility* for the word *affection*, using `text2.concordance("affection")`. Search the book of Genesis to find out how long some people lived, using: `text3.concordance("lived")`. You could look at `text4`, the *US Presidential Inaugural Addresses* to see examples of English dating back to 1789, and search for words like *nation*, *terror*, *god* to see how these words have been used differently over time. We've also included `text5`, the *NPS Chat Corpus*: search this for unconventional words like *im*, *ur*, *lol*. (Note that this corpus is uncensored!)

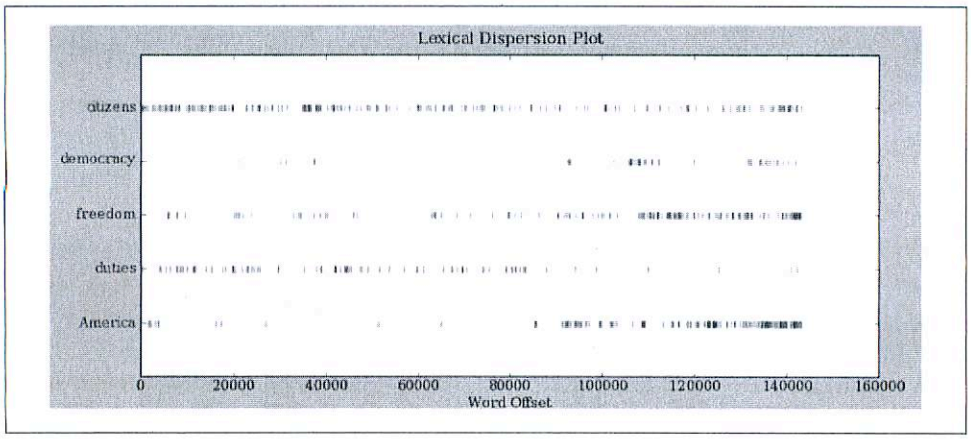

Figure 1-1. Lexical Dispersion Plot for Words in Presidential Inaugural Addresses

Once you've spent a few minutes examining these texts, we hope you have a new sense of the richness and diversity of language. In the next chapter you will learn how to access a broader range of text, including text in languages other than English.

If we can find words in a text, we can also take note of their *position* within the text, and plot word positions using a so-called dispersion plot. Each bar represents an instance of a word and each row represents the entire text. In Figure 1.1 we see some striking patterns of word usage over the last 220 years. You can produce this plot as shown below (so long as you have Numpy and Pylab installed). You might like to try different words, and different texts. Can you predict the dispersion of a word before you view it? As before, take care to get the quotes, commas, brackets and parentheses exactly right.

```
>>> text4.dispersion_plot(["citizens", "democracy", "freedom", "duties", "America"])
>>>
```

A concordance permits us to see words in context, e.g. we saw that *monstrous* appeared in the context *the monstrous pictures*. What other words appear in the same contexts that *monstrous* appears in? We can find out as follows:

```
>>> text1.similar("monstrous")
subtly impalpable curious abundant perilous trustworthy untoward
singular imperial few maddens loving mystifying christian exasperate
puzzled fearless uncommon domineering candid
>>> text2.similar("monstrous")
great very so good vast a exceedingly heartily amazingly as sweet
remarkably extremely
>>>
```

Observe that we get different results for different books. Melville and Austen use this word quite differently.

Now, just for fun, let's try generating some random text in the various styles we have just seen. To do this, we type the name of the text followed by the "generate" function:

```
>>> text3.generate()
In the beginning of his brother is a hairy man , whose top may reach
unto heaven ; and ye shall sow the land of Egypt there was no bread in
all that he was taken out of the month , upon the earth . So shall thy
wages be ? And they made their father ; and Isaac was old , and kissed
him : and Laban with his cattle in the midst of the hands of Esau thy
first born , and Phichol the chief butler unto his son Isaac , she
>>>
```

Note that first time you run this, it is slow because it gathers statistics about word sequences. Each time you run it, you will get different output text. Now try generating random text in the style of an inaugural address or an internet chat room. Although the text is random, it re-uses common words and phrases from the source text and gives us a sense of its style and content.

When text is printed, punctuation has been split off from the previous word. Although this is not correct formatting for English text, we do this to make it clear that punctuation does not belong to the word. This is called "tokenization", and you will learn about it in Chapter 3.

Counting Vocabulary

The most obvious fact about texts that emerges from the previous section is that they differ in the vocabulary they use. In this section we will see how to use the computer to count the words in a text, in a variety of useful ways. As before you will jump right in and experiment with the Python interpreter, even though you may not have studied Python systematically yet. Test your understanding by modifying the examples, and trying the exercises at the end of the chapter.

Let's begin by finding out the length of a text from start to finish, in terms of the words and punctuation symbols that appear. We'll use the text of *Moby Dick* again:

```
>>> len(text1)
260819
>>>
```

That's a quarter of a million words long! But how many distinct words does this text contain? To work this out in Python we have to pose the question slightly differently. The vocabulary of a text is just the *set* of words that it uses, and in Python we can list the vocabulary of `text3` with the command: `set(text3)`. This will produce many screens of words. Now try the following:

```
>>> sorted(set(text3))
['!', "'", '(', ')', ',', ',)', '.', '.)', ':', ';', ';)', '?', '?)',
'A', 'Abel', 'Abelmizraim', 'Abidah', 'Abide', 'Abimael', 'Abimelech',
'Abr', 'Abrah', 'Abraham', 'Abram', 'Accad', 'Achbor', 'Adah', ...]
>>> len(set(text3))
2789
>>> len(text3) / len(set(text3))
```

Be more explicit that capitalized words precede lower case ones.

```
16
>>>
```

Here we can see a sorted list of vocabulary items, beginning with various punctuation symbols and continuing with words starting with *A*. Words starting with *a* will appear much later, after the last "Z" word, *Zoroaster*. We discover the size of the vocabulary indirectly, by asking for the length of the set. There are fewer than 3,000 distinct words in this book. Finally, we can calculate a measure of the lexical richness of the text and learn that each word is used 16 times on average.

Next, let's focus on particular words. We can count how often a word occurs in a text, and compute what percentage of the text is taken up by a specific word:

```
>>> text3.count("smote")
5
>>> 100.0 * text4.count('a') / len(text4)
1.4587672822333748
>>>
```

I think this needs a bit more hand-holding

You might like to repeat such calculations on several texts, but it is tedious to keep retyping it for different texts. Instead, we can come up with our own name for a task, e.g. "score", and define a function that can be re-used as often as we like:

```
>>> def score(text):
...     return len(text) / len(set(text))
...
>>> score(text3)
16
>>> score(text4)
4
>>>
```

The Python interpreter changes the prompt from >>> to ... after encountering the colon at the end of the first line. The ... prompt indicates that Python expects an indented code block to appear next. It is up to you to do the indentation, by typing four spaces. To finish the indented block just enter a blank line.

Notice that we used the `score` function by typing its name, followed by an open parenthesis, the name of the text, then a close parenthesis. This is just what we did for the `len` and `set` functions earlier. These parentheses will show up often: their role is to separate the name of a task — such as `score` — from the data that the task is to be performed on — such as `text3`. Functions are an advanced concept in programming and we only mention them at the outset to give newcomers a sense of the power and creativity of programming. Later we'll see how to use such functions when tabulating data, like Table 1.1. Each row of the table will involve the same computation but with different data, and we'll do this repetitive work using functions.

P 7-10 intentionally omitted from here (no comments on them)

```
>>> mySent = ['The', 'family', 'of', 'Dashwood', 'had', 'long',
...           'been', 'settled', 'in', 'Sussex', '.']
>>> noun_phrase = mySent[:4]
>>> noun_phrase
['The', 'family', 'of', 'Dashwood']
>>> wOrDs = sorted(noun_phrase)
>>> wOrDs
['Dashwood', 'The', 'family', 'of']
>>>
```

It is good to choose meaningful variable names to help you — and anyone who reads your Python code — to understand what your code is meant to do. Python does not try to make sense of the names; it blindly follows your instructions, and does not object if you do something confusing, such as `one = 'two'` or `two = 3`. A variable name cannot be any of Python's reserved words, such as `if`, `not`, and `import`. If you use a reserved word, Python will produce a syntax error:

```
>>> not = 'Dashwood'
File "<stdin>", line 1
    not = wOrDs[0]
        ^
SyntaxError: invalid syntax
>>>
```

We can use variables to hold intermediate steps of a computation. This may make the Python code easier to follow. Thus `len(set(text1))` could also be written:

```
>>> vocab = set(text1)
>>> vocab_size = len(vocab)
>>> vocab_size
19317
>>>
```

You need to be a little bit careful in your choice of names (or identifiers) for Python variables. First, you should start the name with a letter, optionally followed by digits (`0` to `9`) or letters. Thus, `abc23` is fine, but `23abc` will cause a syntax error. You can use underscores anywhere in a name, but not a hyphen, since this gets interpreted as a minus sign.

1.3 Computing with Language: Simple Statistics

Let's return to our exploration of the ways we can bring our computational resources to bear on large quantities of text. We began this discussion in Section 1.1, and saw how to search for words in context, how to compile the vocabulary of a text, how to generate random text in the same style, and so on.

In this section we pick up the question of what makes a text distinct, and use automatic methods to find characteristic words and expressions of a text. As in Section 1.1, you will try new features of the Python language by copying them into the interpreter, and you'll learn about these features systematically in the following section.

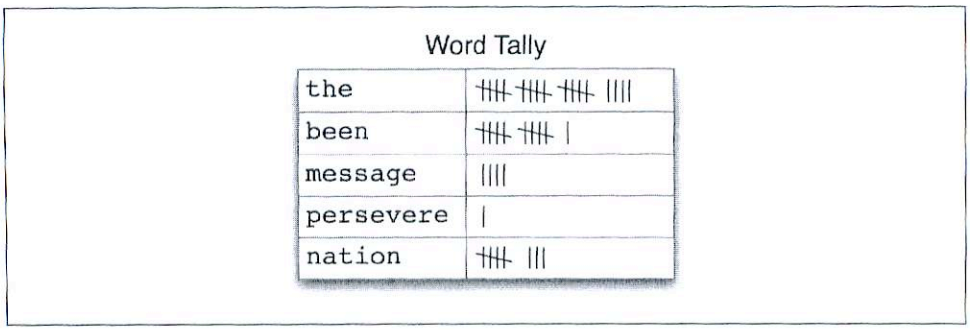

Figure 1-2. Counting Words Appearing in a Text (a frequency distribution)

Before continuing with this section, check your understanding of the previous section by predicting the output of the following code, and using the interpreter to check if you got it right. If you found it difficult to do this task, it would be a good idea to review the previous section before continuing further.

```
>>> saying = ['After', 'all', 'is', 'said', 'and', 'done', ',',
...           'more', 'is', 'said', 'than', 'done', '.']
>>> words = set(saying)
>>> words = sorted(words)
>>> words[:2]
```

Frequency Distributions

How could we automatically identify the words of a text that are most informative about the topic and genre of the text? Let's begin by finding the most frequent words of the text. Imagine how you might go about finding the 50 most frequent words of a book. One method would be to keep a tally for each vocabulary item, like that shown in Figure 1.2. We would need thousands of counters and it would be a laborious process, so laborious that we would rather assign the task to a machine.

The table in Figure 1.2 is known as a frequency distribution, and it tells us the frequency of each vocabulary item in the text. It is a "distribution" since it tells us how the the total number of words in the text — 260,819 in the case of *Moby Dick* — are distributed across the vocabulary items. Since we often need frequency distributions in language processing, NLTK provides built-in support for them. Let's use a `FreqDist` to find the 50 most frequent words of *Moby Dick*. Be sure to try this for yourself, taking care to use the correct parentheses and uppercase letters. (This code assumes that you have already done `from nltk.book import *` during your Python session.)

```
>>> fdist1 = FreqDist(text1)
>>> fdist1
<FreqDist with 260819 samples>
>>> vocabulary1 = fdist1.keys()
>>> vocabulary1[:50]
[',', 'the', '.', 'of', 'and', 'a', 'to', ';', 'in', 'that', "'", '-',
'his', 'it', 'I', 's', 'is', 'he', 'with', 'was', 'as', '"', 'all', 'for',
'this', '!', 'at', 'by', 'but', 'not', '--', 'him', 'from', 'be', 'on',
```

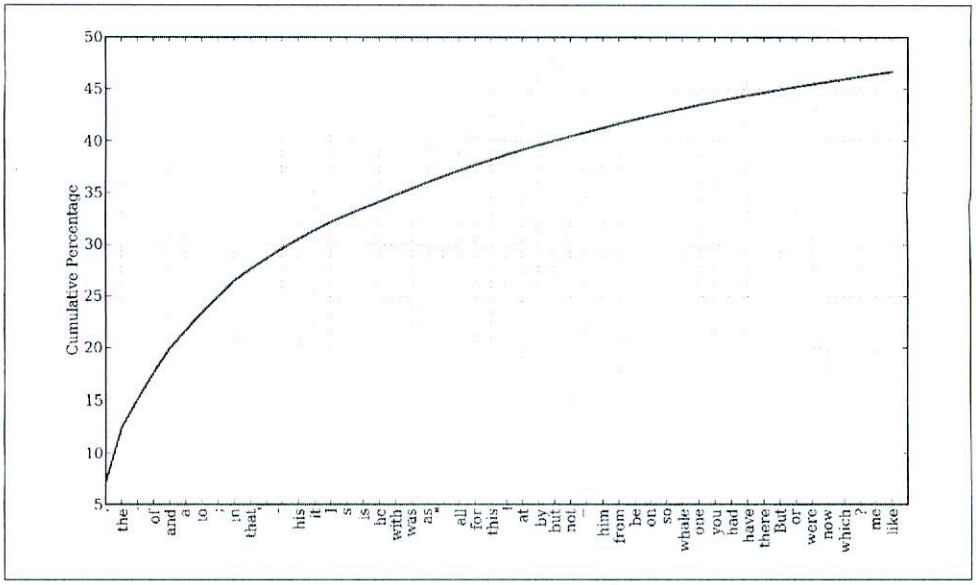

Figure 1-3. Cumulative Frequency Plot for 50 Most Frequent Words in Moby Dick

```
'so', 'whale', 'one', 'you', 'had', 'have', 'there', 'But', 'or', 'were',
'now', 'which', '?', 'me', 'like']
>>> fdist1['whale']
906
>>>
```

Do any words in the above list help us grasp the topic or genre of this text? Only one word, *whale*, is slightly informative! It occurs over 900 times. The rest of the words tell us nothing about the text; they're just English "plumbing." What proportion of English text is taken up with such words? We can generate a cumulative frequency plot for these words, using `fdist1.plot(cumulative=True)`, to produce the graph in Figure 1.3. These 50 words account for nearly half the book!

If the frequent words don't help us, how about the words that occur once only, the so-called hapaxes. See them using `fdist1.hapaxes()`. This list contains *lexicographer*, *cetological*, *contraband*, *expostulations*, and about 9,000 others. It seems that there's too many rare words, and without seeing the context we probably can't guess what half of them mean in any case.

transition is abrupt. Maybe conclude the top section by saying that neither frequent or infrequent words help, so need to try something else.

Working with Subsets of Words

Next let's look at the *long* words of a text; perhaps these will be more characteristic and informative. For this we adapt some notation from set theory. We would like to find the words from the vocabulary of the text that are more than than 15 characters long. Let's call this property P, so that $P(w)$ is true if and only if w is more than 15 characters long. Now we can express the words of interest using mathematical set notation as

shown in (1a). This means "the set of all w such that w is an element of V (the vocabulary) and w has property P".

(1) a. $\{w \mid w \in V \ \& \ P(w)\}$

b. `[w for w in V if p(w)]`

The equivalent Python expression is given in (1b). Notice how similar the two notations are. Let's go one more step and write executable Python code:

```
>>> V = set(text1)
>>> sorted(w for w in V if len(w) > 15)
['apprehensiveness', 'comprehensiveness', 'indiscriminately',
'superstitiousness', 'circumnavigating', 'simultaneousness',
'physiognomically', 'circumnavigation', 'hermaphroditical',
'subterraneousness', 'uninterpenetratingly', 'irresistibleness',
'responsibilities', 'uncompromisedness', 'uncomfortableness',
'supernaturalness', 'characteristically', 'cannibalistically',
'circumnavigations', 'indispensableness', 'preternaturalness',
'CIRCUMNAVIGATION', 'undiscriminating', 'Physiognomically']
>>>
```

The expression `w for w in V` could have equally been written `word for word in vocab`, and means "give me all words, where each word is an element of the vocabulary set". For each such word, we check that its length is greater than 15; all other words will be ignored. We will discuss this more carefully later. For now you should simply try out the above statements in the Python interpreter, and try changing the text, and changing the length condition.

Let's return to our task of finding words that characterize a text. Notice that the long words in `text4` reflect its national focus: *constitutionally*, *transcontinental*, while those in `text5` reflect its informal content: *boooooooooooglyyyyyy* and *yuuuuuuuuuuuummmmmmmmmmmm*. Have we succeeded in automatically extracting words that typify a text? Well, these very long words are often hapaxes (i.e. unique) and perhaps it would be better to find *frequently occurring* long words. This seems promising since it eliminates frequent short words (e.g. *the*) and infrequent long words like (*antiphilosophists*). Here are all words from the chat corpus that are longer than 5 characters, that occur more than 5 times:

```
>>> fdist5 = FreqDist(text5)
>>> sorted(w for w in set(text5) if len(w) > 5 and fdist5[w] > 5)
['#14-19teens', '<empty>', 'ACTION', 'anybody', 'anyone', 'around',
'cute.-ass', 'everybody', 'everyone', 'female', 'listening', 'minutes',
'people', 'played', 'player', 'really', 'seconds', 'should', 'something',
'watching']
>>>
```

Notice how we have used two conditions: `len(w) > 5` ensures that the words are longer than 5 letters, and `fdist5[w] > 5` ensures that these words occur more than five times. At last we have managed to automatically identify the frequently-occuring content-

Table 1-3. Numerical Comparison Operators

Operator	Relationship
<	less than
<=	less than or equal to
==	equal to (note this is two not one = sign)
!=	not equal to
>	greater than
>=	greater than or equal to

We can use these to select different words from a sentence of news text. Here are some examples — only the operator is changed from one line to the next.

```
>>> [w for w in sent7 if len(w) < 4]
[',', '61', 'old', ',', 'the', 'as', 'a', '29', '.']
>>> [w for w in sent7 if len(w) <= 4]
[',', '61', 'old', ',', 'will', 'join', 'the', 'as', 'a', 'Nov.', '29', '.']
>>> [w for w in sent7 if len(w) == 4]
['will', 'join', 'Nov.']
>>> [w for w in sent7 if len(w) != 4]
['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'the', 'board',
'as', 'a', 'nonexecutive', 'director', '29', '.']
>>>
```

Notice the pattern in all of these examples: `[w for w in text if` *condition* `]`. In these cases the condition was always a numerical comparison. However, we can also test various properties of words, using the functions listed in Table 1.4.

Table 1-4. Some Word Comparison Operators

Function	Meaning
`s.startswith(t)`	s starts with t
`s.endswith(t)`	s ends with t
`t in s`	t is contained inside s
`s.islower()`	all cased characters in s are lowercase
`s.isupper()`	all cased characters in s are uppercase
`s.isalpha()`	all characters in s are alphabetic
`s.isalnum()`	all characters in s are alphanumeric
`s.isdigit()`	all characters in s are digits
`s.istitle()`	s is titlecased

Here are some examples of these operators being used to select words from our texts. The first example finds all words ending with *ableness*. The second example finds words containing the substring *gnt*. Can you explain what the last example does?

An if statement is known as a control structure because it controls whether the code in the indented block will be run. Another control structure is the for loop. Don't forget the colon and the four spaces:

```
>>> for word in ['Call', 'me', 'Ishmael', '.']:
...     print word
...
Call
me
Ishmael
.
>>>
```

This is called a loop because Python executes the code in circular fashion. It starts by doing `word = 'Call'`, effectively using the `word` variable to name the first item of the list. Then it displays the value of `word` to the user. Next, it goes back to the `for` statement, and does `word = 'me'`, before displaying this new value to the user, and so on. It continues in this fashion until every item of the list has been processed.

Looping with Conditions

Now we can combine the `if` and `for` statements. We will loop over every item of the list, and only print the item if it ends with the letter "l". We'll pick another name for the variable to demonstrate that Python doesn't try to make sense of variable names.

```
>>> sent1 = ['Call', 'me', 'Ishmael', '.']
>>> for xyzzy in sent1:
...     if xyzzy.endswith('l'):
...         print xyzzy
...
Call
Ishmael
>>>
```

You will notice that `if` and `for` statements have a colon at the end of the line, before the indentation begins. In fact, all Python control structures end with a colon. The colon indicates that the current statement relates to the indented block that follows.

We can also specify an action to be taken if the condition of the `if` statement is not met. Here we see the `elif` "else if" statement, and the `else` statement. Notice that these also have colons before the indented code.

```
>>> for token in sent1:
...     if token.islower():
...         print 'lowercase word'
...     elif token.istitle():
...         print 'titlecase word'
...     else:
...         print 'punctuation'
...
titlecase word
lowercase word
titlecase word
```

```
punctuation
>>>
```

As you can see, even with this small amount of Python knowledge, you can start to build multi-line Python programs. It's important to develop such programs in pieces, testing that each piece does what you expect before combining them into a program. This is why the Python interactive interpreter is so invaluable, and why you should get comfortable using it.

Finally, let's combine the idioms we've been exploring. First we create a list of *cie* and *cei* words, then we loop over each item and print it. Notice the comma at the end of the print statement, which tells Python to produce its output on a single line.

```
>>> confusing = sorted(w for w in set(text2) if 'cie' in w or 'cei' in w)
>>> for word in confusing:
...     print word,
ancient ceiling conceit conceited conceive conscience
conscientious conscientiously deceitful deceive ...
>>>
```

1.5 Computing with Language: Semantic Networks (NOTES)

Texts are our primary source of written language data. However, we have an important secondary source, namely lexical resources, compilations of words and their meanings such as dictionaries and thesauri. Dictionaries are organized to support efficient lookup of particular word forms. Thesauri, on the other hand, support efficient lookup in terms of meanings. The most general electronic lexical resource is known as a wordnet, or a semantic network, and we can easily access it by form or meaning. NLTK includes the English WordNet, and the 117,798 nouns are accessible to you in the variable `N`. We can look up individual words as follows:

```
>>> len(N)
117798
>>> dish = N['dish']
>>> len(dish)
6
>>> dish[0].gloss
'a piece of dishware normally used as a container for holding or serving food;
"we gave them a set of dishes for a wedding present"'
>>> dish[1].gloss
'a particular item of prepared food; "she prepared a special dish for dinner"'
>>> dish[4].gloss
'directional antenna consisting of a parabolic reflector for microwave or
radio frequency radiation'
>>>
```

We see that *dish* has six senses, and the definitions for three of them are shown. Let's look up a verb:

```
>>> len(V)
11529
>>> serve = V['serve']
```

```
>>> len(serve)
15
>>> serve[1].gloss
'do duty or hold offices; serve in a specific function; "He served as head of the department for three y
>>> serve[4].gloss
'help to some food; help with food or drink; "I served him three times, and after that he helped himself
>>> serve[14].gloss
'put the ball into play; "It was Agassi\'s turn to serve"'
>>>
```

There are many fewer verbs than nouns, but they often have more senses. Here we see three senses of the verb *to serve*. We can access semantically related forms. The hypernyms of a word sense are more general terms...

```
>>> dish[1]['hypernym']
[{noun: nutriment, nourishment, nutrition, sustenance, aliment, alimentation, victuals}]
>>> serve[4]['hypernym']
[{verb: provide, supply, ply, cater}]
>>>
```

Uses: topic detection, retrieval, disambiguation, ...

1.6 Automatic Natural Language Understanding

We have been exploring language bottom-up, with the help of texts, dictionaries, and a programming language. However, we're also interested in exploiting our knowledge of language and computation by building useful language technologies.

At a purely practical level, we all need help to navigate the universe of information locked up in unstructured text on the Web. Search engines have been crucial to the growth and popularity of the Web, but have some shortcomings. It takes skill, knowledge, and some luck, to extract answers to such questions as *What tourist sites can I visit between Philadelphia and Pittsburgh on a limited budget? What do experts say about digital SLR cameras? What predictions about the steel market were made by credible commentators in the past week?* Getting a computer to answer them automatically involves a range of language processing tasks, including information extraction, inference, and summarization, and would need to be carried out on a scale and with a level of robustness that is still beyond our current capabilities.

On a more philosophical level, a long-standing challenge within artificial intelligence has been to build intelligent machines, and a major part of intelligent behaviour is understanding language. For many years this goal has been seen as too difficult. However, as NLP technologies become more mature, and robust methods for analysing unrestricted text become more widespread, the prospect of natural language understanding has re-emerged as a plausible goal.

In this section we describe some language processing components and systems, to give you a sense the interesting challenges that are waiting for you.

Word Sense Disambiguation

In word sense disambiguation we want to work out which sense of a word was intended in a given context. Consider the ambiguous words *serve* and *dish*:

(2) a. *serve*: help with food or drink; hold an office; put ball into play
b. *dish*: plate; course of a meal; communications device

Now, in a sentence containing the phrase: *he served the dish*, you can detect that both *serve* and *dish* are being used with their food meanings. Its unlikely that the topic of discussion shifted from sports to crockery in the space of three words — this would force you to invent bizarre images, like a tennis pro taking out her frustrations on a china tea-set laid out beside the court. In other words, we automatically disambiguate words using context, exploiting the simple fact that nearby words have closely related meanings. As another example of this contextual effect, consider the word *by*, which has three meanings: *the book by Chesterton* (agentive); *the cup by the stove* (locative); and *submit by Friday* (temporal). Observe in (3c) that the meaning of the italicized word helps us interpret the meaning of *by*.

(3) a. The lost children were found by the *searchers* (agentive)
b. The lost children were found by the *mountain* (locative)
c. The lost children were found by the *afternoon* (temporal)

Pronoun Resolution

A deeper kind of language understanding is to work out who did what to whom — i.e. to detect the subjects and objects of verbs. You learnt to do this in elementary school, but its harder than you might think. In the sentence *the thieves stole the paintings* it is easy to tell who performed the stealing action. Consider three possible following sentences in (4c), and try to determine what was sold, caught, and found (one case is ambiguous).

(4) a. The thieves stole the paintings. They were subsequently *sold*.
b. The thieves stole the paintings. They were subsequently *caught*.
c. The thieves stole the paintings. They were subsequently *found*.

Answering this question involves finding the antecedent of the pronoun *they* (the theives or the paintings). Computational techniques for solving this problem fall under the heading of anaphora resolution and semantic role labeling.

Generating Language

If we can automatically solve such problems, we will have understood enough of the text to perform some important language generation tasks, such as question answering

and machine translation. In the first case, a machine should be able to answer a user's questions relating to collection of texts:

(5) a. *Text:* ... The thieves stole the paintings. They were subsequently sold. ...
b. *Human:* Who or what was sold?
c. *Machine:* The paintings.

The machine's answer demonstrates that it has correctly worked out that *they* refers to paintings and not to theives. In the second case, the machine should be able to translate the text into another language, accurately conveying the meaning of the original text. In translating the above text into French, we are forced to choose the gender of the pronoun in the second sentence: *ils* (masculine) if the thieves are sold, and *elles* (feminine) if the paintings are sold. Correct translation actually depends on correct understanding of the pronoun.

(6) a. The thieves stole the paintings. They were subsequently sold.
b. Les voleurs ont volé les peintures. *Ils* ont été *vendus* plus tard. (the thieves)
c. Les voleurs ont volé les peintures. *Elles* ont été *vendues* plus tard. (the paintings)

In all of the above examples — working out the sense of a word, the subject of a verb, the antecedent of a pronoun — are steps in establishing the meaning of a sentence, things we would expect a language understanding system to be able to do. We'll come back to some of these topics later in the book.

Spoken Dialog Systems

In the history of artificial intelligence, the chief measure of intelligence has been a linguistic one, namely the Turing Test: can a dialogue system, responding to a user's text input, perform so naturally that we cannot distinguish it from a human-generated response? In contrast, today's commercial dialogue systems are very limited, but still perform useful functions in narrowly-defined domains, as we see below:

S: How may I help you?
U: When is Saving Private Ryan playing?
S: For what theater?
U: The Paramount theater.
S: Saving Private Ryan is not playing at the Paramount theater, but it's playing at the Madison theater at 3:00, 5:30, 8:00, and 10:30.

You could not ask this system to provide driving instructions or details of nearby restaurants unless the required information had already been stored and suitable question-answer pairs had been incorporated into the language processing system.

Observe that the above system seems to understand the user's goals: the user asks when a movie is showing and the system correctly determines from this that the user wants

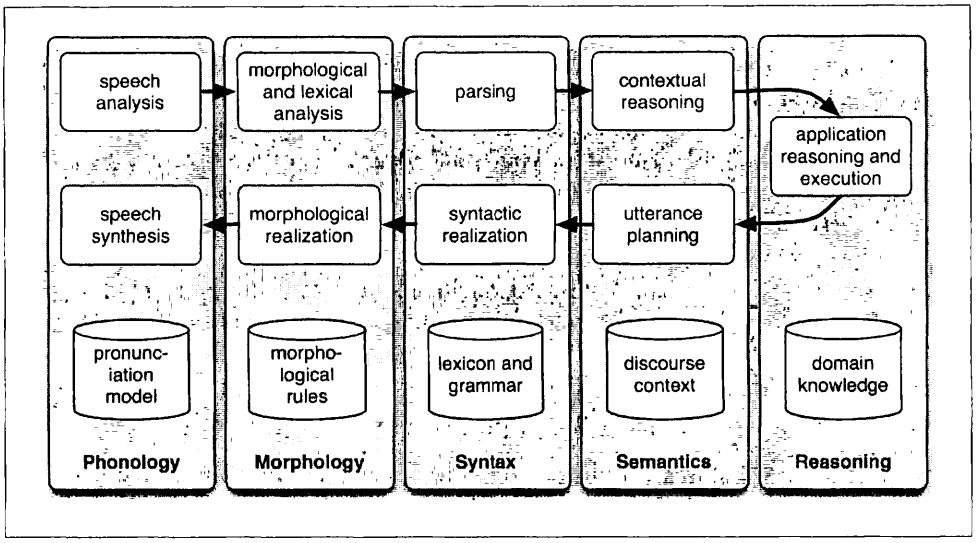

Figure 1-4. Simple Pipeline Architecture for a Spoken Dialogue System

to see the movie. This inference seems so obvious that you probably didn't notice it was made, yet a natural language system needs to be endowed with this capability in order to interact naturally. Without it, when asked *Do you know when Saving Private Ryan is playing*, a system might unhelpfully respond with a cold *Yes*. However, the developers of commercial dialogue systems use contextual assumptions and business logic to ensure that the different ways in which a user might express requests or provide information are handled in a way that makes sense for the particular application. So, if you type *When is ...*, or *I want to know when ...*, or *Can you tell me when ...*, simple rules will always yield screening times. This is enough for the system to provide a useful service.

Dialogue systems give us an opportunity to mention the complete processing pipeline for NLP. Figure 1.4 shows the architecture of a simple dialogue system.

Along the top of the diagram, moving from left to right, is a "pipeline" of some language understanding components. These map from speech input via syntactic parsing to some kind of meaning representation. Along the middle, moving from right to left, is the reverse pipeline of components for converting concepts to speech. These components make up the dynamic aspects of the system. At the bottom of the diagram are some representative bodies of static information: the repositories of language-related data that the processing components draw on to do their work.

Textual Entailment

The challenge of language understanding has been brought into focus in recent years by a public "shared task" called Recognizing Textual Entailment (RTE). The basic scenario is simple. Suppose you want to find find evidence to support the hypothesis:

Sandra Goudie was defeated by Max Purnell, and that you have another short text that seems to be relevant, for example, *Sandra Goudie was first elected to Parliament in the 2002 elections, narrowly winning the seat of Coromandel by defeating Labour candidate Max Purnell and pushing incumbent Green MP Jeanette Fitzsimons into third place*. Does the text provide enough evidence for you to accept the hypothesis? In this particular case, the answer will be No. You can draw this conclusion easily, but it is very hard to come up with automated methods for making the right decision. The RTE Challenges provide data which allow competitors to develop their systems, but not enough data to brute-force approaches using standard machine learning techniques. Consequently, some linguistic analysis is crucial. In the above example, it is important for the system to note that *Sandra Goudie* names the person being defeated in the hypothesis, not the person doing the defeating in the text. As another illustration of the difficulty of the task, consider the following text/hypothesis pair:

(7) a. David Golinkin is the editor or author of eighteen books, and over 150 responsa, articles, sermons and books

b. Golinkin has written eighteen books

In order to determine whether or not the hypothesis is supported by the text, the system needs the following background knowledge: (i) if someone is an author of a book, then he/she has written that book; (ii) if someone is an editor of a book, then he/she has not written that book; (iii) if someone is editor or author of eighteen books, then he/she is not author of eighteen books.

Limitations of NLP

Despite the research-led advances in tasks like RTE, natural language systems that have been deployed for real-world applications still cannot perform common-sense reasoning or draw on world knowledge in a general and robust manner. We can wait for these difficult artificial intelligence problems to be solved, but in the meantime it is necessary to live with some severe limitations on the reasoning and knowledge capabilities of natural language systems. Accordingly, right from the beginning, an important goal of NLP research has been to make progress on the holy grail of natural language understanding *without* recourse to this unrestricted knowledge and reasoning capability.

This is one of the goals of this book, and we hope to equip you with the knowledge and skills to build useful NLP systems, and to contribute to the long-term vision of building intelligent machines.

1.7 Summary

- Texts are represented in Python using lists: `['Monty', 'Python']`. We can use indexing, slicing and the `len()` function on lists.
- We get the vocabulary of a text `t` using `sorted(set(t))`.

- We operate on each item of a text using `[f(x) for x in text]`.
- We process each word in a text using a `for` statement such as `for w in t:` or `for word in text:`. This must be followed by the colon character and an indented block of code, to be executed each time through the loop.
- We test a condition using an `if` statement: `if len(word) < 5:`. This must be followed by the colon character and an indented block of code, to be executed only if the condition is true.
- A frequency distribution is a collection of items along with their frequency counts (e.g. the words of a text and their frequency of appearance).
- WordNet is a semantically-oriented dictionary of English, consisting of synonym sets — or synsets — and organized into a hierarchical network.

not yet introduced

1.8 Further Reading

Natural Language Processing

Several websites have useful information about NLP, including conferences, resources, and special-interest groups, e.g. `www.lt-world.org`, `www.aclweb.org`, `www.elsnet.org`. The website of the *Association for Computational Linguistics*, at `www.aclweb.org`, contains an overview of computational linguistics, including copies of introductory chapters from recent textbooks. Wikipedia has entries for NLP and its subfields (but don't confuse natural language processing with the other NLP: neuro-linguistic programming.) The new, second edition of *Speech and Language Processing*, is a more advanced textbook that builds on the material presented here. Three books provide comprehensive surveys of the field: [Cole, 1997], [Dale, Moisl, & Somers, 2000], [Mitkov, 2002]. Several NLP systems have online interfaces that you might like to experiment with, e.g.:

- WordNet: `http://wordnet.princeton.edu/`
- Translation: `http://world.altavista.com/`
- ChatterBots: `http://www.loebner.net/Prizef/loebner-prize.html`
- Question Answering: `http://www.answerbus.com/`
- Summarization: `http://newsblaster.cs.columbia.edu/`

there should be renewed

Python

[Rossum & Drake, 2006] is a Python tutorial by Guido van Rossum, the inventor of Python and Fred Drake, the official editor of the Python documentation. It is available online at `http://docs.python.org/tut/tut.html`. A more detailed but still introductory text is [Lutz & Ascher, 2003], which covers the essential features of Python, and also provides an overview of the standard libraries. A more advanced text, [Rossum & Drake, 2006] is the official reference for the Python language itself, and describes the syntax of Python and its built-in datatypes in depth. It is also available online at `http://`

We need to check dates.

docs.python.org/ref/ref.html. [Beazley, 2006] is a succinct reference book; although not suitable as an introduction to Python, it is an excellent resource for intermediate and advanced programmers. Finally, it is always worth checking the official *Python Documentation* at http://docs.python.org/.

Two freely available online texts are the following:

- Josh Cogliati, *Non-Programmer's Tutorial for Python*, http://en.wikibooks.org/wiki/Non-Programmer's_Tutorial_for_Python/Contents
- Jeffrey Elkner, Allen B. Downey and Chris Meyers, *How to Think Like a Computer Scientist: Learning with Python* (Second Edition), http://openbookproject.net/thinkCSpy/

1.9 Exercises

1. ☼ How many words are there in text2? How many distinct words are there?
2. ☼ Compare the lexical diversity scores for humor and romance fiction in Table 1.1. Which genre is more lexically diverse?
3. ☼ Produce a dispersion plot of the four main protagonists in *Sense and Sensibility*: Elinor, Marianne, Edward, Willoughby. What can you observe about the different roles played by the males and females in this novel? Can you identify the couples?
4. ☼ According to Strunk and White's *Elements of Style*, the word *however*, used at the start of a sentence, means "in whatever way" or "to whatever extent", and not "nevertheless". They give this example of correct usage: *However you advise him, he will probably do as he thinks best.* (http://www.bartleby.com/141/strunk3.html) Use the concordance tool to study actual usage of this word in the various texts we have been considering.
5. ◑ Consider the following Python expression: len(set(text4)). State the purpose of this expression. Describe the two steps involved in performing this computation.
6. ◑ How many times does the word *lol* appear in text5? How much is this as a percentage of the total number of words in this text?
7. ◑ Pick a pair of texts and study the differences between them, in terms of vocabulary, vocabulary richness, genre, etc. Can you find pairs of words which have quite different meanings across the two texts, such as *monstrous* in *Moby Dick* and in *Sense and Sensibility*?
8. ◑ Compare the frequency of use of the modal verbs *will* and *could* in text2 (romance fiction) and text7 (news). Which modal verb is more common in which genre?
9. ☼ Create a variable phrase containing a list of words. Experiment with the operations described above, including addition, multiplication, indexing, slicing, and sorting.

10. ☼ The index of *the* in `sent3` is 1, because `sent3[1]` gives us `'the'`. What are the indexes of the two other occurrences of this word in `sent3`?
11. ☼ Our artificial sentence had 20 elements. What does the interpreter do when you enter `sent[20]`? Why?
12. ◑ Use `text6.index(??)` to find the index of the word *sunset*. By a process of trial and error, find the slice for the complete sentence that contains this word.
13. ◑ Use the addition, set, and sorted operations to compute the vocabulary of the sentences defined above (`sent1` ...).
14. ◑ What is the difference between `sorted(set(w.lower() for w in text1))` and `sorted(w.lower() for w in set(text1))`? Which one will usually give a larger value?
15. ◑ Write the slice expression to produces the last two words of `text2`.
16. ◑ Read the BBC News article: *UK's Vicky Pollards 'left behind'* `http://news.bbc.co.uk/1/hi/education/6173441.stm`. The article gives the following statistic about teen language: "the top 20 words used, including yeah, no, but and like, account for around a third of all words." How many word types account for a third of all word tokens, for a variety of text sources? What do you conclude about this statistic? Read more about this on *LanguageLog*, at `http://itre.cis.upenn.edu/~myl/languagelog/archives/003993.html`.
17. ◑ Assign a new value to `sent`, namely the sentence `['she', 'sells', 'sea', 'shells', 'by', 'the', 'sea', 'shore']`, then write code to perform the following tasks:
    a. Print all words beginning with `'sh'`:
    b. Print all words longer than 4 characters.

1. ◑ What does the following Python do? `sum(len(w) for w in text1)` Can you use it to work out the average word length of a text?
2. ◑ What is the difference between the test `w.isupper()` and `not w.islower()`?