Chapter 1

Introduction to Natural Language Processing and Python

1.1 The Language Challenge

Today, people from all walks of life — including professionals, students, and the general population — are confronted by unprecedented volumes of information, the vast bulk of which is stored as unstructured text. In 2003, it was estimated that the annual production of books amounted to 8 Terabytes. (A Terabyte is 1,000 Gigabytes, i.e., equivalent to 1,000 pickup trucks filled with books.) It would take a human being about five years to read the new scientific material that is produced every 24 hours. Although these estimates are based on printed materials, increasingly the information is also available electronically. Indeed, there has been an explosion of text and multimedia content on the World Wide Web. For many people, a large and growing fraction of work and leisure time is spent navigating and accessing this universe of information.

The presence of so much text in electronic form is a huge challenge to NLP. Arguably, the only way for humans to cope with the information explosion is to exploit computational techniques that can sift through huge bodies of text.

Although existing search engines have been crucial to the growth and popularity of the Web, humans require 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 expert critics 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 is a realistic long-term goal, but would involve 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.

1.1.1 The Richness of Language

Language is the chief manifestation of human intelligence. Through language we express basic needs and lofty aspirations, technical know-how and flights of fantasy. Ideas are shared over great separations of distance and time. The following samples from English illustrate the richness of language:

(1) a. Overhead the day drives level and grey, hiding the sun by a flight of grey spears. (William Faulkner, *As I Lay Dying*, 1935)

b. When using the toaster please ensure that the exhaust fan is turned on. (sign in dormitory kitchen)

c. Amiodarone weakly inhibited CYP2C9, CYP2D6, and CYP3A4-mediated activities with Ki values of 45.1-271.6 μM (Medline, PMID: 10718780)

d. Iraqi Head Seeks Arms (spoof news headline)

e. The earnest prayer of a righteous man has great power and wonderful results. (James 5:16b)

f. Twas brillig, and the slithy toves did gyre and gimble in the wabe (Lewis Carroll, *Jabberwocky*, 1872)

g. There are two ways to do this, AFAIK :smile: (internet discussion archive)

Thanks to this richness, the study of language is part of many disciplines outside of linguistics, including translation, literary criticism, philosophy, anthropology and psychology. Many less obvious disciplines investigate language use, such as law, hermeneutics, forensics, telephony, pedagogy, archaeology, cryptanalysis and speech pathology. Each applies distinct methodologies to gather observations, develop theories and test hypotheses. Yet all serve to deepen our understanding of language and of the intellect that is manifested in language.

The importance of language to science and the arts is matched in significance by the cultural treasure embodied in language. Each of the world's ~7,000 human languages is rich in unique respects, in its oral histories and creation legends, down to its grammatical constructions and its very words and their nuances of meaning. Threatened remnant cultures have words to distinguish plant subspecies according to therapeutic uses that are unknown to science. Languages evolve over time as they come into contact with each other and they provide a unique window onto human pre-history. Technological change gives rise to new words like *blog* and new morphemes like *e-* and *cyber-*. In many parts of the world, small linguistic variations from one town to the next add up to a completely different language in the space of a half-hour drive. For its breathtaking complexity and diversity, human language is as a colorful tapestry stretching through time and space.

1.1.2 The Promise of NLP

As we have seen, NLP is important for scientific, economic, social, and cultural reasons. NLP is experiencing rapid growth as its theories and methods are deployed in a variety of new language technologies. For this reason it is important for a wide range of people to have a working knowledge of NLP. Within industry, it includes people in *human-computer interaction*, *business information analysis*, and *Web software development*. Within academia, this includes people in areas from *humanities computing* and *corpus linguistics* through to *computer science* and *artificial intelligence*. We hope that you, a member of this diverse audience reading these materials, will come to appreciate the workings of this rapidly growing field of NLP and will apply its techniques in the solution of real-world problems.

This book presents a carefully-balanced selection of theoretical foundations and practical applications, and equips readers to work with large datasets, to create robust models of linguistic phenomena, and to deploy them in working language technologies. By integrating all of this into the Natural Language Toolkit (NLTK), we hope this book opens up the exciting endeavor of practical natural language processing to a broader audience than ever before.

The rest of this chapter provides a non-technical overview of Python and will cover the basic programming knowledge needed for the rest of the chapters in Part 1. It contains many examples and exercises; there is no better way to learn to program than to dive in and try these yourself. You should then feel confident in adapting the example for your own purposes. Before you know it you will be programming!

1.2 Computing with Language

As we will see, 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? Here we will treat the text as data for the programs we write, programs that manipulate and analyze it in a variety of interesting ways. The first step is to get started with the Python interpreter.

1.2.1 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 run 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 `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):

```
Python 2.5 (r25:51918, Sep 19 2006, 08:49:13)
[GCC 4.0.1 (Apple Computer, Inc. build 5341)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

> **Note**
>
> 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. Let's begin by using the Python prompt 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. One strange thing you might come across is that division doesn't always behave how you expect:

```
>>> 3/3
1
>>> 1/3
0
>>>
```

The second case is surprising because we would expect the answer to be 0.333333. We will come back to why that is the case later on in this chapter. For now, let's simply observe that these examples demonstrate how you can work interactively with the interpreter, allowing you to experiment and explore. Also, as you will see later, your intuitions about numerical expressions will be useful for manipulating other kinds of data in Python.

You should also try nonsensical expressions to see how the interpreter handles it:

```
>>> 1 +
Traceback (most recent call last):
  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.

1.2.2 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.text import *`. After printing a welcome message, it loads the text of several books, including *Moby Dick*. We can ask the interpreter to give us some information about it, such as title and word length, by typing `text1`, and `len(text1)`:

```
>>> from nltk.book import *
>>> text1
<Text: Moby Dick by Herman Melville 1851>
```

We can examine the contents of the book in a variety of ways. A concordance view shows us a given word in its context. Here we look up the word *monstrous*. Try seaching for other words; you can use the up-arrow key to access the previous command and modify the word being searched.

```
>>> 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
```

You can now try concordance searches on some of the other texts we have included. For example, to search *Sense and Sensibility* by Jane Austen, for the word *affection*, use: `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*. We've also included `text5`, the *NPS Chat Corpus*: search this for unconventional words like *im*, *ur*, *lol*.

Once you've spent some time 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. We produce a dispersion plot, where each bar represents an instance of a word and each row represents the entire text. In Figure 1.1 we see characteristically different roles played by the male and female protagonists in *Sense and Sensibility*. In Figure 1.2 we see some striking patterns of word usage over the last 220 years. You can produce these plots as shown below. You might like to try different words, and different texts.

```
>>> text2.dispersion_plot(['Elinor', 'Marianne', 'Edward', 'Willoughby'])
>>> text4.dispersion_plot(['citizens', 'democracy', 'freedom', 'duties', 'America'])
```

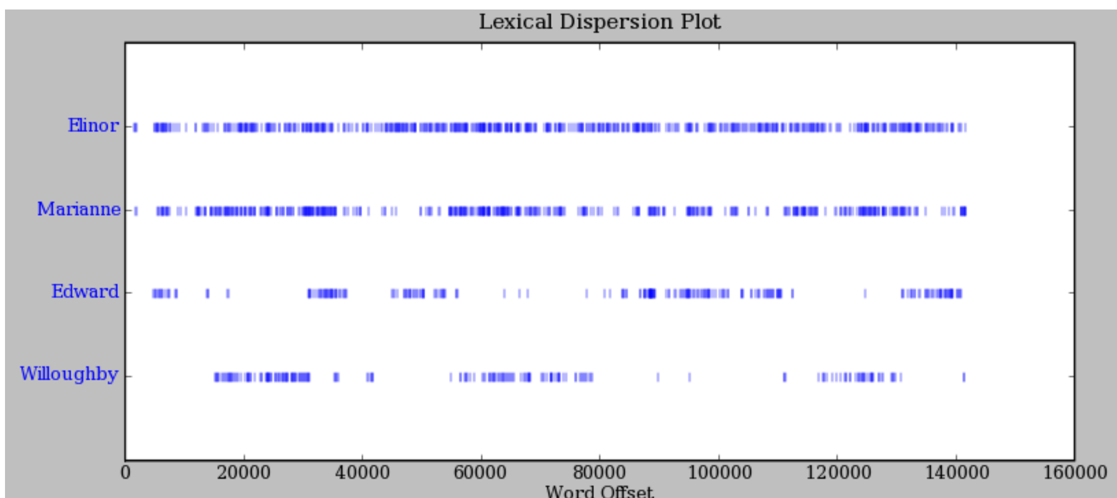

Figure 1.1: Lexical Dispersion Plot for Main Protagonists in *Sense and Sensibility*

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.

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, e.g. `text3.generate()`:

```
>>> 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
```

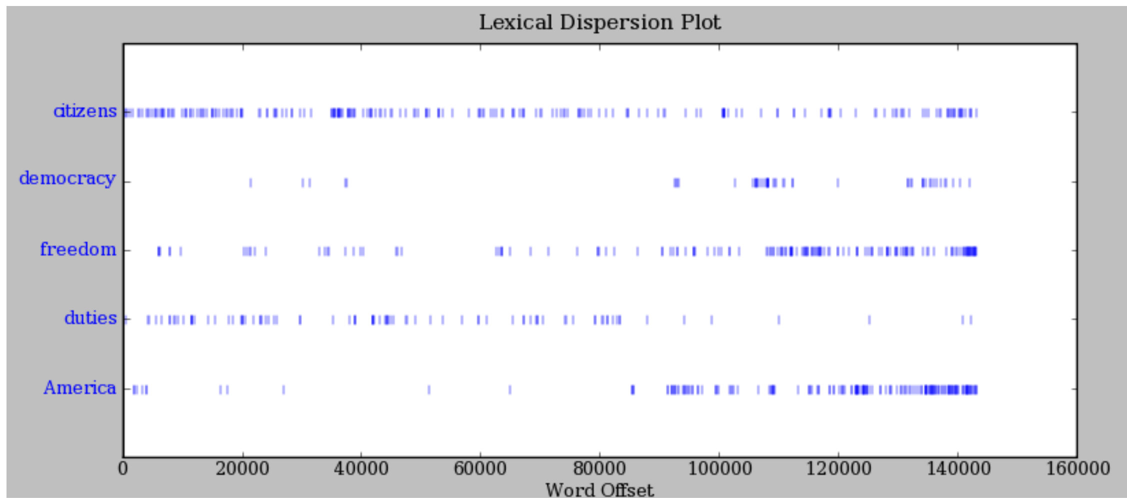

Figure 1.2: Lexical Dispersion Plot for Words in Presidential Inaugural Addresses

```
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.

Note

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 we will learn more about it in Chapter 2.

1.2.3 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.

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. Let's look at the text of *Moby Dick*:

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

That's a quarter of a million words long! 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))
16
```

Thus we can see a sorted list of vocabulary items beginning with various punctuation symbols. We can find out the size of the vocabulary by asking for the length of the set. Finally, we can calculate a measure of the lexical richness of the text and learn that each word is used 16 times on average.

We might like to repeat the last of these calculations on several texts, but it is tedious to keep retyping this line for different texts. Instead, we can come up with our own name for this 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
```

Note

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`.

Now that we've had an initial sample of language processing in Python, we will continue with a systematic introduction to the language.

1.3 Python Basics: Strings and Variables

1.3.1 Representing text

We can't simply type text directly into the interpreter because it would try to interpret the text as part of the Python language:

```
>>> Hello World
Traceback (most recent call last):
  File "<stdin>", line 1
    Hello World
              ^
SyntaxError: invalid syntax
>>>
```

Here we see an error message. Note that the interpreter is confused about the position of the error, and points to the end of the string rather than the start.

Python represents a piece of text using a **string**. Strings are **delimited** — or separated from the rest of the program — by quotation marks:

```
>>> 'Hello World'
'Hello World'
>>> "Hello World"
'Hello World'
>>>
```

We can use either single or double quotation marks, as long as we use the same ones on either end of the string.

Now we can perform calculator-like operations on strings. For example, adding two strings together seems intuitive enough that you could guess the result:

```
>>> 'Hello' + 'World'
'HelloWorld'
>>>
```

When applied to strings, the + operation is called **concatenation**. It produces a new string that is a copy of the two original strings pasted together end-to-end. Notice that concatenation doesn't do anything clever like insert a space between the words. The Python interpreter has no way of knowing that you want a space; it does *exactly* what it is told. Given the example of +, you might be able guess what multiplication will do:

```
>>> 'Hi' + 'Hi' + 'Hi'
'HiHiHi'
>>> 'Hi' * 3
'HiHiHi'
>>>
```

The point to take from this (apart from learning about strings) is that in Python, intuition about what should work gets you a long way, so it is worth just trying things to see what happens. You are very unlikely to break anything, so just give it a go.

1.3.2 Storing and Reusing Values

After a while, it can get quite tiresome to keep retyping Python statements over and over again. It would be nice to be able to store the **value** of an expression like `'Hi' + 'Hi' + 'Hi'` so that we can use it again. We do this by saving results to a location in the computer's memory, and giving the location a name. Such a named place is called a **variable**. In Python we create variables by **assignment**, which involves putting a value into the variable:

```
>>> msg = 'Hello World'          ①
>>> msg                          ②
'Hello World'                    ③
>>>
```

In line ① we have created a variable called `msg` (short for 'message') and set it to have the string value `'Hello World'`. We used the = operation, which **assigns** the value of the expression on the right to the variable on the left. Notice the Python interpreter does not print any output; it only prints

output when the statement returns a value, and an assignment statement returns no value. In line ② we inspect the contents of the variable by naming it on the command line: that is, we use the name msg. The interpreter prints out the contents of the variable in line ③.

Variables stand in for values, so instead of writing 'Hi' * 3 we could assign variable msg the value 'Hi', and num the value 3, then perform the multiplication using the variable names:

```
>>> msg = 'Hi'
>>> num = 3
>>> msg * num
'HiHiHi'
>>>
```

The names we choose for the variables are up to us. Instead of msg and num, we could have used any names we like:

```
>>> marta = 'Hi'
>>> foo123 = 3
>>> marta * foo123
'HiHiHi'
>>>
```

Thus, the reason for choosing meaningful variable names is to help you — and anyone who reads your code — to understand what it 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 potentially confusing such as assigning a variable two the value 3, with the assignment statement: two = 3.

Note that we can also assign a new value to a variable just by using assignment again:

```
>>> msg = msg * num
>>> msg
'HiHiHi'
>>>
```

Here we have taken the value of msg, multiplied it by 3 and then stored that new string (HiHiHi) back into the variable msg.

1.3.3 Printing and Inspecting Strings

So far, when we have wanted to look at the contents of a variable or see the result of a calculation, we have just typed the variable name into the interpreter. We can also see the contents of msg using print msg:

```
>>> msg = 'Hello World'
>>> msg
'Hello World'
>>> print msg
Hello World
>>>
```

On close inspection, you will see that the quotation marks that indicate that Hello World is a string are missing in the second case. That is because inspecting a variable, by typing its name into the interactive interpreter, prints out the Python **representation** of a value. In contrast, the print statement only prints out the value itself, which in this case is just the text contained in the string.

In fact, you can use a sequence of comma-separated expressions in a print statement:

```
>>> msg2 = 'Goodbye'
>>> print msg, msg2
Hello World Goodbye
>>>
```

> **Note**
>
> If you have created some variable `v` and want to find out about it, then type `help (v)` to read the help entry for this kind of object. Type `dir(v)` to see a list of operations that are defined on the object.

You need to be a little bit careful in your choice of names (or **identifiers**) for Python variables. Some of the things you might try will cause an error. 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 (both within and at the start of the variable name), but not a hyphen, since this gets interpreted as an arithmetic operator. A second problem is shown in the following snippet.

```
>>> not = "don't do this"
  File "<stdin>", line 1
    not = "don't do this"
    ^
SyntaxError: invalid syntax
```

Why is there an error here? Because `not` is reserved as one of Python's 30 odd **keywords**. These are special identifiers that are used in specific syntactic contexts, and cannot be used as variables. It is easy to tell which words are keywords if you use IDLE, since they are helpfully highlighted in orange.

1.3.4 Creating Programs with a Text Editor

The Python interative interpreter performs your instructions as soon as you type them. Often, it is better to compose a multi-line program using a text editor, then ask Python to run the whole program at once. Using IDLE, you can do this by going to the `File` menu and opening a new window. Try this now, and enter the following one-line program:

```
msg = 'Hello World'
```

Save this program in a file called `test.py`, then go to the `Run` menu, and select the command `Run Module`. The result in the main IDLE window should look like this:

```
>>> ================================ RESTART ================================
>>>
>>>
```

Now, where is the output showing the value of `msg`? The answer is that the program in `test.py` will show a value only if you explicitly tell it to, using the `print` command. So add another line to `test.py` so that it looks as follows:

```
msg = 'Hello World'
print msg
```

Select `Run Module` again, and this time you should get output that looks like this:

```
>>> ================================ RESTART ================================
>>>
Hello World
>>>
```

From now on, you have a choice of using the interactive interpreter or a text editor to create your programs. It is often convenient to test your ideas using the interpreter, revising a line of code until it does what you expect, and consulting the interactive help facility. Once you're ready, you can paste the code (minus any >>> prompts) into the text editor, continue to expand it, and finally save the program in a file so that you don't have to retype it in again later.

1.3.5 Exercises

1. ☼ Start up the Python interpreter (e.g. by running IDLE). Try the examples in section 1.2, then experiment with using Python as a calculator.

2. ☼ Try the examples in this section, then try the following.

   a) Create a variable called `msg` and put a message of your own in this variable. Remember that strings need to be quoted, so you will need to type something like:

   ```
   >>> msg = "I like NLP!"
   ```

   b) Now print the contents of this variable in two ways, first by simply typing the variable name and pressing enter, then by using the `print` command.

   c) Try various arithmetic expressions using this string, e.g. `msg + msg`, and `5 * msg`.

   d) Define a new string `hello`, and then try `hello + msg`. Change the `hello` string so that it ends with a space character, and then try `hello + msg` again.

3. ☺ Discuss the steps you would go through to find the ten most frequent words in a two-page document.

1.4 Slicing and Dicing

Strings are so important that we will spend some more time on them. Here we will learn how to access the individual **characters** that make up a string, how to pull out arbitrary **substrings**, and how to reverse strings.

1.4.1 Accessing Individual Characters

The positions within a string are numbered, starting from zero. To access a position within a string, we specify the position inside square brackets:

```
>>> msg = 'Hello World'
>>> msg[0]
'H'
>>> msg[3]
'l'
```

```
>>> msg[5]
' '
>>>
```

This is called **indexing** or **subscripting** the string. The position we specify inside the square brackets is called the **index**. We can retrieve not only letters but any character, such as the space at index 5.

Note

Be careful to distinguish between the string `' '`, which is a single whitespace character, and `''`, which is the empty string.

The fact that strings are indexed from zero may seem counter-intuitive. You might just want to think of indexes as giving you the position in a string immediately *before* a character, as indicated in Figure 1.3.

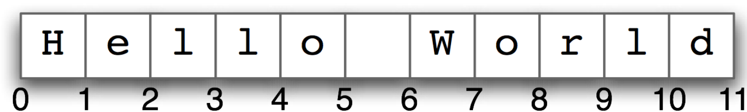

Figure 1.3: String Indexing

Now, what happens when we try to access an index that is outside of the string?

```
>>> msg[11]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
>>>
```

The index of `11` is outside of the range of valid indices (i.e., 0 to 10) for the string `'Hello World'`. This results in an error message. This time it is not a syntax error; the program fragment is syntactically correct. Instead, the error occurred while the program was running. The `Traceback` message indicates which line the error occurred on (line 1 of "standard input"). It is followed by the name of the error, `IndexError`, and a brief explanation.

In general, how do we know what we can index up to? If we know the length of the string is n, the highest valid index will be $n - 1$. We can get access to the length of the string using the built-in `len( )` function.

```
>>> len(msg)
11
>>>
```

Informally, a **function** is a named snippet of code that provides a service to our program when we **call** or execute it by name. We call the `len()` function by putting parentheses after the name and giving it the string `msg` we want to know the length of. Because `len()` is built into the Python interpreter, IDLE colors it purple.

We have seen what happens when the index is too large. What about when it is too small? Let's see what happens when we use values less than zero:

```
>>> msg[-1]
'd'
>>>
```

This does not generate an error. Instead, negative indices work from the *end* of the string, so `-1` indexes the last character, which is `'d'`.

```
>>> msg[-3]
'r'
>>> msg[-6]
' '
>>>
```

Now the computer works out the location in memory relative to the string's address plus its length, subtracting the index, e.g. `3136 + 11 - 1 = 3146`. We can also visualize negative indices as shown in Figure 1.4.

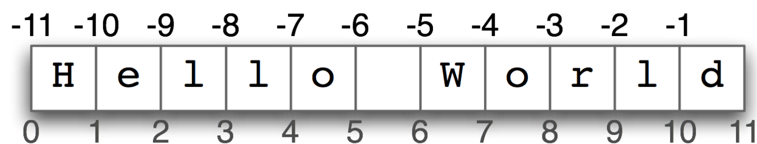

Figure 1.4: Negative Indices

Thus we have two ways to access the characters in a string, from the start or the end. For example, we can access the space in the middle of `Hello` and `World` with either `msg[5]` or `msg[-6]`; these refer to the same location, because `5 = len(msg) - 6`.

1.4.2 Accessing Substrings

In NLP we usually want to access more than one character at a time. This is also pretty simple; we just need to specify a start and end index. For example, the following code accesses the substring starting at index `1`, up to (but not including) index `4`:

```
>>> msg[1:4]
'ell'
>>>
```

The notation `:4` is known as a **slice**. Here we see the characters are `'e'`, `'l'` and `'l'` which correspond to `msg[1]`, `msg[2]` and `msg[3]`, but not `msg[4]`. This is because a slice *starts* at the first index but finishes *one before* the end index. This is consistent with indexing: indexing also starts from zero and goes up to *one before* the length of the string. We can see this by slicing with the value of `len()`:

```
>>> len(msg)
11
>>> msg[0:11]
'Hello World'
>>>
```

We can also slice with negative indices — the same basic rule of starting from the start index and stopping one before the end index applies; here we stop before the space character:

```
>>> msg[0:-6]
'Hello'
>>>
```

Python provides two shortcuts for commonly used slice values. If the start index is `0` then you can leave it out, and if the end index is the length of the string then you can leave it out:

```
>>> msg[:3]
'Hel'
>>> msg[6:]
'World'
>>>
```

The first example above selects the first three characters from the string, and the second example selects from the character with index 6, namely `'W'`, to the end of the string.

1.4.3 Exercises

1. ☼ Define a string `s = 'colorless'`. Write a Python statement that changes this to "colourless" using only the slice and concatenation operations.

2. ☼ Try the slice examples from this section using the interactive interpreter. Then try some more of your own. Guess what the result will be before executing the command.

3. ☼ We can use the slice notation to remove morphological endings on words. For example, `'dogs'[:-1]` removes the last character of `dogs`, leaving `dog`. Use slice notation to remove the affixes from these words (we've inserted a hyphen to indicate the affix boundary, but omit this from your strings): `dish-es`, `run-ning`, `nation-ality`, `un-do`, `pre-heat`.

4. ☼ We saw how we can generate an `IndexError` by indexing beyond the end of a string. Is it possible to construct an index that goes too far to the left, before the start of the string?

5. ☼ We can also specify a "step" size for the slice. The following returns every second character within the slice, in a forward or reverse direction:

   ```
   >>> msg[6:11:2]
   'Wrd'
   >>> msg[10:5:-2]
   'drW'
   >>>
   ```

   Experiment with different step values.

6. ☼ What happens if you ask the interpreter to evaluate `msg[::-1]`? Explain why this is a reasonable result.

1.5 Strings, Sequences, and Sentences

We have seen how words like *Hello* can be stored as a string `'Hello'`. Whole sentences can also be stored in strings, and manipulated as before, as we can see here for Chomsky's famous nonsense sentence:

```
>>> sent = 'colorless green ideas sleep furiously'
>>> sent[16:21]
'ideas'
>>> len(sent)
37
>>>
```

However, it turns out to be a bad idea to treat a sentence as a sequence of its characters, because this makes it too inconvenient to access the words. Instead, we would prefer to represent a sentence as a sequence of its *words*; as a result, indexing a sentence accesses the words, rather than characters. We will see how to do this now.

1.5.1 Lists

A **list** is designed to store a sequence of values. A list is similar to a string in many ways except that individual items don't have to be just characters; they can be arbitrary strings, integers or even other lists.

A Python list is represented as a sequence of comma-separated items, delimited by square brackets. Here are some lists:

```
>>> squares = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> shopping_list = ['juice', 'muffins', 'bleach', 'shampoo']
```

We can also store sentences and phrases using lists. Let's create part of Chomsky's sentence as a list and put it in a variable `cgi`:

```
>>> cgi = ['colorless', 'green', 'ideas']
>>> cgi
['colorless', 'green', 'ideas']
>>>
```

Because lists and strings are both kinds of sequence, they can be processed in similar ways; just as strings support `len()`, indexing and slicing, so do lists. The following example applies these familiar operations to the list `cgi`:

```
>>> len(cgi)
3
>>> cgi[0]
'colorless'
>>> cgi[-1]
'ideas'
>>> cgi[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>>
```

Here, `cgi[-5]` generates an error, because the fifth-last item in a three item list would occur before the list started, i.e., it is undefined. We can also slice lists in exactly the same way as strings:

```
>>> cgi[1:3]
['green', 'ideas']
>>> cgi[-2:]
['green', 'ideas']
>>>
```

Lists can be concatenated just like strings. Here we will put the resulting list into a new variable `chomsky`. The original variable `cgi` is not changed in the process:

```
>>> chomsky = cgi + ['sleep', 'furiously']
>>> chomsky
['colorless', 'green', 'ideas', 'sleep', 'furiously']
>>> cgi
['colorless', 'green', 'ideas']
>>>
```

Now, lists and strings do not have exactly the same functionality. Lists have the added power that you can change their elements. Let's imagine that we want to change the 0th element of `cgi` to `'colorful'`, we can do that by assigning the new value to the index `cgi[0]`:

```
>>> cgi[0] = 'colorful'
>>> cgi
['colorful', 'green', 'ideas']
>>>
```

On the other hand if we try to do that with a *string* — changing the 0th character in `msg` to `'J'` — we get:

```
>>> msg[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>>
```

This is because strings are **immutable** — you can't change a string once you have created it. However, lists are **mutable**, and their contents can be modified at any time. As a result, lists support a number of operations, or **methods**, that modify the original value rather than returning a new value. A method is a function that is associated with a particular object. A method is called on the object by giving the object's name, then a period, then the name of the method, and finally the parentheses containing any arguments. For example, in the following code we use the `sort()` and `reverse()` methods:

```
>>> chomsky.sort()
>>> chomsky.reverse()
>>> chomsky
['sleep', 'ideas', 'green', 'furiously', 'colorless']
>>>
```

As you will see, the prompt reappears immediately on the line after `chomsky.sort()` and `chomsky.reverse()`. That is because these methods do not produce a new list, but instead modify the original list stored in the variable `chomsky`.

Lists also have an `append()` method for adding items to the end of the list and an `index()` method for finding the index of particular items in the list:

```
>>> chomsky.append('said')
>>> chomsky.append('Chomsky')
>>> chomsky
['sleep', 'ideas', 'green', 'furiously', 'colorless', 'said', 'Chomsky']
>>> chomsky.index('green')
2
>>>
```

Finally, just as a reminder, you can create lists of any values you like. As you can see in the following example for a lexical entry, the values in a list do not even have to have the same type (though this is usually not a good idea, as we will explain in Section 5.2).

```
>>> bat = ['bat', [[1, 'n', 'flying mammal'], [2, 'n', 'striking instrument']]]
>>>
```

1.5.2 Working on Sequences One Item at a Time

We have shown you how to create lists, and how to index and manipulate them in various ways. Often it is useful to step through a list and process each item in some way. We do this using a `for` loop. This is our first example of a **control structure** in Python, a statement that *controls* how other statements are run:

```
>>> for num in [1, 2, 3]:
...     print 'The number is', num
...
The number is 1
The number is 2
The number is 3
```

The `for` loop has the general form: `for` *variable* `in` *sequence* followed by a colon, then an indented block of code. The first time through the loop, the variable is assigned to the first item in the sequence, i.e. `num` has the value `1`. This program runs the statement `print 'The number is', num` for this value of `num`, before returning to the top of the loop and assigning the second item to the variable. Once all items in the sequence have been processed, the loop finishes.

Now let's try the same idea with a list of words:

```
>>> chomsky = ['colorless', 'green', 'ideas', 'sleep', 'furiously']
>>> for word in chomsky:
...     print len(word), word[-1], word
...
9 s colorless
5 n green
5 s ideas
5 p sleep
9 y furiously
```

The first time through this loop, the variable is assigned the value `'colorless'`. This program runs the statement `print len(word), word[-1], word` for this value, to produce the output line: `9 s colorless`. This process is known as **iteration**. Each iteration of the `for` loop starts by assigning the next item of the list `chomsky` to the **loop variable** `word`. Then the indented **body** of the loop is run. Here the body consists of a single command, but in general the body can contain as many lines of code as you want, so long as they are all indented by the same amount. (We recommend that you always use exactly 4 spaces for indentation, and that you never use tabs.)

We can run another `for` loop over the Chomsky nonsense sentence, and calculate the average word length. As you will see, this program uses the `len()` function in two ways: to count the number of characters in a word, and to count the number of words in a phrase. Note that `x += y` is shorthand for `x = x + y`; this idiom allows us to **increment** the `total` variable each time the loop is run.

```
>>> total = 0
```

```
>>> for word in chomsky:
...     total += len(word)
...
>>> total / len(chomsky)
6
>>>
```

We can also write `for` loops to iterate over the characters in strings. This `print` statement ends with a trailing comma, which is how we tell Python not to print a newline at the end.

```
>>> sent = 'colorless green ideas sleep furiously'
>>> for char in sent:
...     print char,
...
c o l o r l e s s   g r e e n   i d e a s   s l e e p   f u r i o u s l y
>>>
```

A note of caution: we have now iterated over words and characters, using expressions like `for word in sent:` and `for char in sent:`. Remember that, to Python, `word` and `char` are meaningless variable names, and we could just as well have written `for foo123 in sent:`. The interpreter simply iterates over the items in the sequence, quite oblivious to what kind of object they represent, e.g.:

```
>>> for foo123 in 'colorless green ideas sleep furiously':
...     print foo123,
...
c o l o r l e s s   g r e e n   i d e a s   s l e e p   f u r i o u s l y
>>> for foo123 in ['colorless', 'green', 'ideas', 'sleep', 'furiously']:
...     print foo123,
...
colorless green ideas sleep furiously
>>>
```

However, you should try to choose 'sensible' names for loop variables because it will make your code more readable.

1.5.3 String Formatting

The output of a program is usually structured to make the information easily digestible by a reader. Instead of running some code and then manually inspecting the contents of a variable, we would like the code to tabulate some output. We already saw this above in the first `for` loop example that used a list of words, where each line of output was similar to `5 p sleep`, consisting of a word length, the last character of the word, then the word itself.

There are many ways we might want to format such output. For instance, we might want to place the length value in parentheses *after* the word, and print all the output on a single line:

```
>>> for word in chomsky:
...     print word, '(', len(word), '),',
colorless ( 9 ), green ( 5 ), ideas ( 5 ), sleep ( 5 ), furiously ( 9 ),
>>>
```

However, this approach has a couple of problems. First, the `print` statement intermingles variables and punctuation, making it a little difficult to read. Second, the output has spaces around every item that was printed. A cleaner way to produce structured output uses Python's **string formatting expressions**. Before diving into clever formatting tricks, however, let's look at a really simple example. We are going to use a special symbol, `%s`, as a placeholder in strings. Once we have a string containing this placeholder, we follow it with a single `%` and then a value `v`. Python then returns a new string where `v` has been slotted in to replace `%s`:

```
>>> "I want a %s right now" % "coffee"
'I want a coffee right now'
>>>
```

In fact, we can have a number of placeholders, but following the `%` operator we need to specify exactly the same number of values. Note that the parentheses are required.

```
>>> "%s wants a %s %s" % ("Lee", "sandwich", "for lunch")
'Lee wants a sandwich for lunch'
>>>
```

We can also provide the values for the placeholders indirectly. Here's an example using a `for` loop:

```
>>> menu = ['sandwich', 'spam fritter', 'pancake']
>>> for snack in menu:
...     "Lee wants a %s right now" % snack
...
'Lee wants a sandwich right now'
'Lee wants a spam fritter right now'
'Lee wants a pancake right now'
>>>
```

We oversimplified things when we said that placeholders were of the form `%s`; in fact, this is a complex object, called a **conversion specifier**. This has to start with the `%` character, and ends with conversion character such as `s`` or ``d`. The `%s` specifier tells Python that the corresponding variable is a string (or should be converted into a string), while the `%d` specifier indicates that the corresponding variable should be converted into a decimal representation. The string containing conversion specifiers is called a **format string**.

Picking up on the `print` example that we opened this section with, here's how we can use two different kinds of conversion specifier:

```
>>> for word in chomsky:
...     print "%s (%d)," % (word, len(word)),
colorless (9), green (5), ideas (5), sleep (5), furiously (9),
>>>
```

To summarize, string formatting is accomplished with a three-part object having the syntax: *`format % values`*. The *`format`* section is a string containing format specifiers such as `%s` and `%d` that Python will replace with the supplied values. The *`values`* section of a formatting string is a parenthesized list containing exactly as many items as there are format specifiers in the *`format`* section. In the case that there is just one item, the parentheses can be left out. (We will discuss Python's string-formatting expressions in more detail in Section 5.3.2).

In the above example, we used a trailing comma to suppress the printing of a newline. Suppose, on the other hand, that we want to introduce some additional newlines in our output. We can accomplish this by inserting the "special" character `\n` into the `print` string:

```
>>> for word in chomsky:
...     print "Word = %s\nIndex = %s\n*****" % (word, chomsky.index(word))
...
Word = colorless
Index = 0
*****
Word = green
Index = 1
*****
Word = ideas
Index = 2
*****
Word = sleep
Index = 3
*****
Word = furiously
Index = 4
*****
>>>
```

1.5.4 Converting Between Strings and Lists

Often we want to convert between a string containing a space-separated list of words and a list of strings. Let's first consider turning a list into a string. One way of doing this is as follows:

```
>>> s = ''
>>> for word in chomsky:
...     s += ' ' + word
...
>>> s
' colorless green ideas sleep furiously'
>>>
```

One drawback of this approach is that we have an unwanted space at the start of `s`. It is more convenient to use the `join()` method. We specify the string to be used as the "glue", followed by a period, followed by the `join()` function.

```
>>> sent = ' '.join(chomsky)
>>> sent
'colorless green ideas sleep furiously'
>>>
```

So `' '.join(chomsky)` means: take all the items in `chomsky` and concatenate them as one big string, using `' '` as a spacer between the items.

Now let's try to reverse the process: that is, we want to convert a string into a list. Again, we could start off with an empty list `[]` and `append()` to it within a `for` loop. But as before, there is a more succinct way of achieving the same goal. This time, we will *split* the new string `sent` on whitespace:

To consolidate your understanding of joining and splitting strings, let's try the same thing using a semicolon as the separator:

```
>>> sent = ';'.join(chomsky)
>>> sent
'colorless;green;ideas;sleep;furiously'
```

```
>>> sent.split(';')
['colorless', 'green', 'ideas', 'sleep', 'furiously']
>>>
```

To be honest, many people find the notation for `join()` rather unintuitive. There is another function for converting lists to strings, again called `join()` which is called directly on the list. It uses whitespace by default as the "glue". However, we need to explicitly **import** this function into our code. One way of doing this is as follows:

```
>>> import string
>>> string.join(chomsky)
'colorless green ideas sleep furiously'
>>>
```

Here, we imported something called `string`, and then called the function `string.join()`. In passing, if we want to use something other than whitespace as "glue", we just specify this as a second parameter:

```
>>> string.join(chomsky, ';')
'colorless;green;ideas;sleep;furiously'
>>>
```

We will see other examples of statements with `import` later in this chapter. In general, we use `import` statements when we want to get access to Python code that doesn't already come as part of core Python. This code will exist somewhere as one or more files. Each such file corresponds to a Python **module** — this is a way of grouping together code and data that we regard as reusable. When you write down some Python statements in a file, you are in effect creating a new Python module. And you can make your code depend on another module by using the `import` statement. In our example earlier, we imported the module `string` and then used the `join()` function from that module. By adding `string.` to the beginning of `join()`, we make it clear to the Python interpreter that the definition of `join()` is given in the `string` module. An alternative, and equally valid, approach is to use the `from` *module* `import` *identifier* statement, as shown in the next example:

```
>>> from string import join
>>> join(chomsky)
'colorless green ideas sleep furiously'
>>>
```

In this case, the name `join` is added to all the other identifier that we have defined in the body of our programme, and we can use it to call a function like any other.

> **Note**
>
> If you are creating a file to contain some of your Python code, do *not* name your file `nltk.py`: it may get imported in place of the "real" NLTK package. (When it imports modules, Python first looks in the current folder / directory.)

1.5.5 Mini-Review

Strings and lists are both kind of **sequence**. As such, they can both be indexed and sliced:

```
>>> query = 'Who knows?'
>>> beatles = ['john', 'paul', 'george', 'ringo']
>>> query[2]
'o'
>>> beatles[2]
'george'
>>> query[:2]
'Wh'
>>> beatles[:2]
['john', 'paul']
>>>
```

Similarly, strings can be concatenated and so can lists (though not with each other!):

```
>>> newstring = query + " I don't"
>>> newlist = beatles + ['brian', 'george']
```

What's the difference between strings and lists as far as NLP is concerned? As we will see in Chapter 2, when we open a file for reading into a Python program, what we get initially is a string, corresponding to the contents of the whole file. If we try to use a `for` loop to process the elements of this string, all we can pick out are the individual characters in the string — we don't get to choose the granularity. By contrast, the elements of a list can be as big or small as we like: for example, they could be paragraphs, sentence, phrases, words, characters. So lists have this huge advantage, that we can be really flexible about the elements they contain, and correspondingly flexible about what the downstream processing will act on. So one of the first things we are likely to do in a piece of NLP code is convert a string into a list (of strings). Conversely, when we want to write our results to a file, or to a terminal, we will usually convert them to a string.

1.5.6 Exercises

1. ☼ Using the Python interactive interpreter, experiment with the examples in this section. Think of a sentence and represent it as a list of strings, e.g. ['Hello', 'world']. Try the various operations for indexing, slicing and sorting the elements of your list. Extract individual items (strings), and perform some of the string operations on them.

2. ☼ Split `sent` on some other character, such as `'s'`.

3. ☼ We pointed out that when `phrase` is a list, `phrase.reverse()` returns a modified version of `phrase` rather than a new list. On the other hand, we can use the slice trick mentioned in the exercises for the previous section, `[::-1]` to create a *new* reversed list without changing `phrase`. Show how you can confirm this difference in behavior.

4. ☼ We have seen how to represent a sentence as a list of words, where each word is a sequence of characters. What does `phrase1[2][2]` do? Why? Experiment with other index values.

5. ☼ Write a `for` loop to print out the characters of a string, one per line.

6. ☼ What is the difference between calling `split` on a string with no argument or with `' '` as the argument, e.g. `sent.split()` versus `sent.split(' ')`? What happens when the string being split contains tab characters, consecutive space characters, or a sequence of tabs and spaces? (In IDLE you will need to use `'\t'` to enter a tab character.)

7. ☼ Create a variable `words` containing a list of words. Experiment with `words.sort( )` and `sorted(words)`. What is the difference?

8. ☼ Earlier, we asked you to use a text editor to create a file called `test.py`, containing the single line `msg = 'Hello World'`. If you haven't already done this (or can't find the file), go ahead and do it now. Next, start up a new session with the Python interpreter, and enter the expression `msg` at the prompt. You will get an error from the interpreter. Now, try the following (note that you have to leave off the `.py` part of the filename):

```
>>> from test import msg
>>> msg
```

This time, Python should return with a value. You can also try `import test`, in which case Python should be able to evaluate the expression `test.msg` at the prompt.

9. ◑ Process the list `chomsky` using a `for` loop, and store the result in a new list `lengths`. Hint: begin by assigning the empty list to `lengths`, using `lengths = []`. Then each time through the loop, use `append()` to add another length value to the list.

10. ◑ Define a variable `silly` to contain the string: `'newly formed bland ideas are inexpressible in an infuriating way'`. (This happens to be the legitimate interpretation that bilingual English-Spanish speakers can assign to Chomsky's famous phrase, according to Wikipedia). Now write code to perform the following tasks:

    a) Split `silly` into a list of strings, one per word, using Python's `split()` operation, and save this to a variable called `bland`.

    b) Extract the second letter of each word in `silly` and join them into a string, to get `'eoldrnnnna'`.

    c) Combine the words in `bland` back into a single string, using `join()`. Make sure the words in the resulting string are separated with whitespace.

    d) Print the words of `silly` in alphabetical order, one per line.

11. ◑ The `index()` function can be used to look up items in sequences. For example, `'inexpressible'.index('e')` tells us the index of the first position of the letter `e`.

    a) What happens when you look up a substring, e.g. `'inexpressible'.index('re')`?

    b) Define a variable `words` containing a list of words. Now use `words.index()` to look up the position of an individual word.

    c) Define a variable `silly` as in the exercise above. Use the `index()` function in combination with list slicing to build a list `phrase` consisting of all the words up to (but not including) `in` in `silly`.

1.6 Making Decisions

So far, our simple programs have been able to manipulate sequences of words, and perform some operation on each one. We applied this to lists consisting of a few words, but the approach works the same for lists of arbitrary size, containing thousands of items. Thus, such programs have some interesting qualities: (i) the ability to work with language, and (ii) the potential to save human effort through automation. Another useful feature of programs is their ability to *make decisions* on our behalf; this is our focus in this section.

1.6.1 Making Simple Decisions

Most programming languages permit us to execute a block of code when a **conditional expression**, or `if` statement, is satisfied. In the following program, we have created a variable called `word` containing the string value `'cat'`. The `if` statement then checks whether the condition `len(word) < 5` is true. Because the conditional expression is true, the body of the `if` statement is invoked and the `print` statement is executed.

```
>>> word = "cat"
>>> if len(word) < 5:
...     print 'word length is less than 5'
...
word length is less than 5
>>>
```

If we change the conditional expression to `len(word) >= 5`, to check that the length of `word` is greater than or equal to `5`, then the conditional expression will no longer be true, and the body of the `if` statement will not be run:

```
>>> if len(word) >= 5:
...     print 'word length is greater than or equal to 5'
...
>>>
```

The `if` statement, just like the `for` statement above is a **control structure**. An `if` statement is a control structure because it controls whether the code in the body will be run. You will notice that both `if` and `for` have a colon at the end of the line, before the indentation begins. That's because all Python control structures end with a colon.

What if we want to do something when the conditional expression is not true? The answer is to add an `else` clause to the `if` statement:

```
>>> if len(word) >= 5:
...     print 'word length is greater than or equal to 5'
... else:
...     print 'word length is less than 5'
...
word length is less than 5
>>>
```

Finally, if we want to test multiple conditions in one go, we can use an `elif` clause that acts like an `else` and an `if` combined:

```
>>> if len(word) < 3:
...   print 'word length is less than three'
... elif len(word) == 3:
...   print 'word length is equal to three'
... else:
...   print 'word length is greater than three'
...
word length is equal to three
>>>
```

It's worth noting that in the condition part of an `if` statement, a nonempty string or list is evaluated as true, while an empty string or list evaluates as false.

```
>>> mixed = ['cat', '', ['dog'], []]
>>> for element in mixed:
...     if element:
...         print element
...
cat
['dog']
```

That is, we *don't* need to say `if len(element) > 0:` in the condition.

What's the difference between using `if...elif` as opposed to using a couple of `if` statements in a row? Well, consider the following situation:

```
>>> animals = ['cat', 'dog']
>>> if 'cat' in animals:
...     print 1
... elif 'dog' in animals:
...     print 2
...
1
>>>
```

Since the `if` clause of the statement is satisfied, Python never tries to evaluate the `elif` clause, so we never get to print out `2`. By contrast, if we replaced the `elif` by an `if`, then we would print out both `1` and `2`. So an `elif` clause potentially gives us more information than a bare `if` clause; when it evaluates to true, it tells us not only that the condition is satisfied, but also that the condition of the main `if` clause was *not* satisfied.

1.6.2 Conditional Expressions

Python supports a wide range of operators like < and >= for testing the relationship between values. The full set of these **relational operators** are shown in Table 1.1.

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

Operator	Relationship
>	greater than
>=	greater than or equal to

Table 1.1:

Conditional Expressions

Normally we use conditional expressions as part of an `if` statement. However, we can test these relational operators directly at the prompt:

```
>>> 3 < 5
True
>>> 5 < 3
False
>>> not 5 < 3
True
>>>
```

Here we see that these expressions have **Boolean** values, namely `True` or `False`. `not` is a Boolean operator, and flips the truth value of Boolean statement.

Strings and lists also support conditional operators:

```
>>> word = 'sovereignty'
>>> 'sovereign' in word
True
>>> 'gnt' in word
True
>>> 'pre' not in word
True
>>> 'Hello' in ['Hello', 'World']
True
>>> 'Hell' in ['Hello', 'World']
False
>>>
```

Strings also have methods for testing what appears at the beginning and the end of a string (as opposed to just anywhere in the string:

```
>>> word.startswith('sovereign')
True
>>> word.endswith('ty')
True
>>>
```

1.6.3 Iteration, Items, and `if`

Now it is time to put some of the pieces together. We are going to take the string `'how now brown cow'` and print out all of the words ending in `'ow'`. Let's build the program up in stages. The first step is to split the string into a list of words:

```
>>> sentence = 'how now brown cow'
>>> words = sentence.split()
>>> words
['how', 'now', 'brown', 'cow']
>>>
```

Next, we need to iterate over the words in the list. Just so we don't get ahead of ourselves, let's print each word, one per line:

```
>>> for word in words:
...     print word
...
how
now
brown
cow
```

The next stage is to only print out the words if they end in the string `'ow'`. Let's check that we know how to do this first:

```
>>> 'how'.endswith('ow')
True
>>> 'brown'.endswith('ow')
False
>>>
```

Now we are ready to put an `if` statement inside the `for` loop. Here is the complete program:

```
>>> sentence = 'how now brown cow'
>>> words = sentence.split()
>>> for word in words:
...     if word.endswith('ow'):
...         print word
...
how
now
cow
>>>
```

As you can see, even with this small amount of Python knowledge it is possible to develop useful programs. The key idea is to develop the program in pieces, testing that each one does what you expect, and then combining them to produce whole programs. This is why the Python interactive interpreter is so invaluable, and why you should get comfortable using it.

1.6.4 A Taster of Data Types

Integers, strings and lists are all kinds of **data types** in Python, and have types `int`, `str` and `list` respectively. In fact, every value in Python has a type. Python's `type()` function will tell you what an object's type is:

```
>>> oddments = ['cat', 'cat'.index('a'), 'cat'.split()]
>>> for e in oddments:
...     type(e)
...
```

```
<type 'str'>
<type 'int'>
<type 'list'>
>>>
```

The type determines what operations you can perform on the data value. So, for example, we have seen that we can index strings and lists, but we can't index integers:

```
>>> one = 'cat'
>>> one[0]
'c'
>>> two = [1, 2, 3]
>>> two[1]
2
>>> three = 1234
>>> three[2]
Traceback (most recent call last):
  File "<pyshell#95>", line 1, in -toplevel-
    three[2]
TypeError: 'int' object is unsubscriptable
>>>
```

The fact that this is a problem with types is signalled by the class of error, i.e., `TypeError`; an object being "unsubscriptable" means we can't index into it.

Similarly, we can concatenate strings with strings, and lists with lists, but we cannot concatenate strings with lists:

```
>>> query = 'Who knows?'
>>> beatles = ['john', 'paul', 'george', 'ringo']
>>> query + beatles
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'list' objects
```

You may also have noticed that our analogy between operations on strings and numbers at the beginning of this chapter broke down pretty soon:

```
>>> 'Hi' * 3
'HiHiHi'
>>> 'Hi' - 'i'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
>>> 6 / 2
3
>>> 'Hi' / 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
>>>
```

These error messages are another example of Python telling us that we have got our data types in a muddle. In the first case, we are told that the operation of substraction (i.e., −) cannot apply to objects of type `str`, while in the second, we are told that division cannot take `str` and `int` as its two operands.

1.6.5 Exercises

1. ☼ Assign a new value to `sentence`, namely the string `'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.

   c) Generate a new sentence that adds the popular hedge word `'like'` before every word beginning with `'se'`. Your result should be a single string.

2. ☼ Write code to abbreviate text by removing all the vowels. Define `sentence` to hold any string you like, then initialize a new string `result` to hold the empty string `''`. Now write a `for` loop to process the string, one character at a time, and append any non-vowel characters to the result string.

3. ☼ We pointed out that when empty strings and empty lists occur in the condition part of an `if` clause, they evaluate to false. In this case, they are said to be occuring in a **Boolean context**. Experiment with different kind of non-Boolean expressions in Boolean contexts, and see whether they evaluate as true or false.

4. ☼ Review conditional expressions, such as `'row' in 'brown'` and `'row' in ['brown', 'cow']`.

   a) Define `sent` to be the string `'colorless green ideas sleep furiously'`, and use conditional expressions to test for the presence of particular words or substrings.

   b) Now define `words` to be a list of words contained in the sentence, using `sent.split()`, and use conditional expressions to test for the presence of particular words or substrings.

5. ◑ Write code to convert text into *hAck3r*, where characters are mapped according to the following table:

Input:	e	i	o	l	s	.	ate
Output:	3	1	0	|	5	5w33t!	8

Table 1.2:

1.7 Getting Organized

Strings and lists are a simple way to organize data. In particular, they **map** from integers to values. We can "look up" a character in a string using an integer, and we can look up a word in a list of words using an integer. These cases are shown in Figure 1.5.

However, we need a more flexible way to organize and access our data. Consider the examples in Figure 1.6.

In the case of a phone book, we look up an entry using a *name*, and get back a number. When we type a domain name in a web browser, the computer looks this up to get back an IP address. A word

String

0	g
1	r
2	e
3	e
4	n

List

0	colorless
1	green
2	ideas
3	sleep
4	furiously

Figure 1.5: Sequence Look-up

Phone List

Alex	x154
Dana	x642
Kim	x911
Les	x120
Sandy	x124

Domain Name Resolution

aclweb.org	128.231.23.4
amazon.com	12.118.92.43
google.com	28.31.23.124
pythonb.org	18.21.3.144
sourceforge.net	51.98.23.53

Word Frequency Table

computational	25
language	196
linguistics	17
natural	56
processing	57

Figure 1.6: Dictionary Look-up

frequency table allows us to look up a word and find its frequency in a text collection. In all these cases, we are mapping from names to numbers, rather than the other way round as with indexing into sequences. In general, we would like to be able to map between arbitrary types of information. Table 1.3 lists a variety of linguistic objects, along with what they map.

Linguistic Object	**Maps**	
	from	**to**
Document Index	Word	List of pages (where word is found)
Thesaurus	Word sense	List of synonyms
Dictionary	Headword	Entry (part of speech, sense definitions, etymology)
Comparative Wordlist	Gloss term	Cognates (list of words, one per language)
Morph Analyzer	Surface form	Morphological analysis (list of component morphemes)

Table 1.3:

Linguistic Objects as Mappings from Keys to Values

Most often, we are mapping from a string to some structured object. For example, a document index maps from a word (which we can represent as a string), to a list of pages (represented as a list of integers). In this section, we will see how to represent such mappings in Python.

1.7.1 Accessing Data with Data

Python provides a **dictionary** data type that can be used for mapping between arbitrary types.

Note

A Python dictionary is somewhat like a linguistic dictionary — they both give you a systematic means of looking things up, and so there is some potential for confusion. However, we hope that it will usually be clear from the context which kind of dictionary we are talking about.

Here we define `pos` to be an empty dictionary and then add three entries to it, specifying the part-of-speech of some words. We add entries to a dictionary using the familiar square bracket notation:

```
>>> pos = {}
```

```
>>> pos['ideas']
'n'
>>> pos['colorless']
'adj'
>>>
```

The item used for look-up is called the **key**, and the data that is returned is known as the **value**. As with indexing a list or string, we get an exception when we try to access the value of a key that does not exist:

```
>>> pos['missing']
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 'missing'
>>>
```

This raises an important question. Unlike lists and strings, where we can use `len()` to work out which integers will be legal indices, how do we work out the legal keys for a dictionary? Fortunately, we can check whether a key exists in a dictionary using the `in` operator:

```
>>> 'colorless' in pos
True
>>> 'missing' in pos
False
>>> 'missing' not in pos
True
>>>
```

Notice that we can use `not in` to check if a key is *missing*. Be careful with the `in` operator for dictionaries: it only applies to the keys and not their values. If we check for a value, e.g. `'adj' in pos`, the result is `False`, since `'adj'` is not a key. We can loop over all the entries in a dictionary using a `for` loop.

```
>>> for word in pos:
...     print "%s (%s)" % (word, pos[word])
...
colorless (adj)
furiously (adv)
ideas (n)
>>>
```

We can see what the contents of the dictionary look like by inspecting the variable `pos`. Note the presence of the colon character to separate each key from its corresponding value:

```
>>> pos
{'furiously': 'adv', 'ideas': 'n', 'colorless': 'adj'}
>>>
```

Here, the contents of the dictionary are shown as **key-value pairs**. As you can see, the order of the key-value pairs is different from the order in which they were originally entered. This is because dictionaries are not sequences but mappings. The keys in a mapping are not inherently ordered, and any ordering that we might want to impose on the keys exists independently of the mapping. As we shall see later, this gives us a lot of flexibility.

We can use the same key-value pair format to create a dictionary:

```
>>> pos = {'furiously': 'adv', 'ideas': 'n', 'colorless': 'adj'}
>>>
```

Using the dictionary methods `keys()`, `values()` and `items()`, we can access the keys and values as separate lists, and also the key-value pairs:

```
>>> pos.keys()
['colorless', 'furiously', 'ideas']
>>> pos.values()
['adj', 'adv', 'n']
>>> pos.items()
[('colorless', 'adj'), ('furiously', 'adv'), ('ideas', 'n')]
>>> for (key, val) in pos.items():
...     print "%s ==> %s" % (key, val)
...
colorless ==> adj
furiously ==> adv
ideas ==> n
>>>
```

Note that keys are forced to be unique. Suppose we try to use a dictionary to store the fact that the word *content* is both a noun and a verb:

```
>>> pos['content'] = 'n'
>>> pos['content'] = 'v'
>>> pos
{'content': 'v', 'furiously': 'adv', 'ideas': 'n', 'colorless': 'adj'}
>>>
```

Initially, `pos['content']` is given the value `'n'`, and this is immediately overwritten with the new value `'v'`. In other words, there is only one entry for `'content'`. If we wanted to store multiple values in that entry, we could use a list, e.g. `pos['content'] = ['n', 'v']`.

1.7.2 Counting with Dictionaries

The values stored in a dictionary can be any kind of object, not just a string — the values can even be dictionaries. The most common kind is actually an integer. It turns out that we can use a dictionary to store **counters** for many kinds of data. For instance, we can have a counter for all the letters of the alphabet; each time we get a certain letter we increment its corresponding counter:

```
>>> phrase = 'colorless green ideas sleep furiously'
>>> count = {}
>>> for letter in phrase:
...     if letter not in count:
...         count[letter] = 0
...     count[letter] += 1
>>> count
{'a': 1, ' ': 4, 'c': 1, 'e': 6, 'd': 1, 'g': 1, 'f': 1, 'i': 2,
 'l': 4, 'o': 3, 'n': 1, 'p': 1, 's': 5, 'r': 3, 'u': 2, 'y': 1}
>>>
```

Observe that `in` is used here in two different ways: `for letter in phrase` iterates over every letter, running the body of the `for` loop. Inside this loop, the conditional expression `if`

`letter not in count` checks whether the letter is missing from the dictionary. If it is missing, we create a new entry and set its value to zero: `count[letter] = 0`. Now we are sure that the entry exists, and it may have a zero or non-zero value. We finish the body of the `for` loop by incrementing this particular counter using the += assignment operator. Finally, we print the dictionary, to see the letters and their counts. This method of maintaining many counters will find many uses, and you will become very familiar with it. To make counting much easier, we can use `defaultdict`, a special kind of container introduced in Python 2.5. This is also included in NLTK for the benefit of readers who are using Python 2.4, and can be imported as shown below.

```
>>> phrase = 'colorless green ideas sleep furiously'
>>> from nltk import defaultdict
>>> count = defaultdict(int)
>>> for letter in phrase:
...     count[letter] += 1
>>> count
{'a': 1, ' ': 4, 'c': 1, 'e': 6, 'd': 1, 'g': 1, 'f': 1, 'i': 2,
 'l': 4, 'o': 3, 'n': 1, 'p': 1, 's': 5, 'r': 3, 'u': 2, 'y': 1}
>>>
```

Note

Calling `defaultdict(int)` creates a special kind of dictionary. When that dictionary is accessed with a non-existent key — i.e. the first time a particular letter is encountered — then `int()` is called to produce the initial value for this key (i.e. `0`). You can test this by running the above code, then typing `count['X']` and seeing that it returns a zero value (and not a `KeyError` as in the case of normal Python dictionaries). The function defaultdict is very handy and will be used in many places later on.

There are other useful ways to display the result, such as sorting alphabetically by the letter:

```
>>> sorted(count.items())
[(' ', 4), ('a', 1), ('c', 1), ('d', 1), ('e', 6), ('f', 1), ...,
...('y', 1)]
>>>
```

Note

The function `sorted()` is similar to the `sort()` method on sequences, but rather than sorting in-place, it produces a new sorted copy of its argument. Moreover, as we will see very soon, `sorted()` will work on a wider variety of data types, including dictionaries.

1.7.3 Getting Unique Entries

Sometimes, we don't want to count at all, but just want to make a record of the items that we have seen, regardless of repeats. For example, we might want to compile a vocabulary from a document. This is a sorted list of the words that appeared, regardless of frequency. At this stage we have two ways to do this. The first uses lists, while the second uses sets.

```
>>> sentence = "she sells sea shells by the sea shore".split()
>>> words = []
```

```
>>> for word in sentence:
...     if word not in words:
...         words.append(word)
...
>>> sorted(words)
['by', 'sea', 'sells', 'she', 'shells', 'shore', 'the']
>>>
```

There is a better way to do this task using Python's **set** data type. We can convert `sentence` into a set, using `set(sentence)`:

```
>>> set(sentence)
set(['shells', 'sells', 'shore', 'she', 'sea', 'the', 'by'])
>>>
```

The order of items in a set is not significant, and they will usually appear in a different order to the one they were entered in. The main point here is that converting a list to a set removes any duplicates. We convert it back into a list, sort it, and print. Here is the complete program:

```
>>> sentence = "she sells sea shells by the sea shore".split()
>>> sorted(set(sentence))
['by', 'sea', 'sells', 'she', 'shells', 'shore', 'the']
```

Here we have seen that there is sometimes more than one way to solve a problem with a program. In this case, we used three different built-in data types, a list, a dictionary, and a set. The `set` data type mostly closely modeled our task, so it required the least amount of work.

1.7.4 Scaling Up

We can use dictionaries to count word occurrences. For example, the following code uses NLTK's corpus reader to load *Macbeth* and count the frequency of each word. Before we can use NLTK we need to tell Python to load it, using the statement `import nltk`.

```
>>> import nltk
>>> count = nltk.defaultdict(int)                          # initialize a dictionary
>>> for word in nltk.corpus.gutenberg.words('shakespeare-macbeth.txt'): # tokenize
...     word = word.lower()                                # normalize to lowercase
...     count[word] += 1                                   # increment the counter
...
>>>
```

You will learn more about accessing corpora in Section 2.2.3. For now, you just need to know that `gutenberg.words()` returns a list of words, in this case from Shakespeare's play *Macbeth*, and we are iterating over this list using a `for` loop. We convert each word to lowercase using the string method `word.lower()`, and use a dictionary to maintain a set of counters, one per word. Now we can inspect the contents of the dictionary to get counts for particular words:

```
>>> count['scotland']
12
>>> count['the']
692
>>>
```

1.7.5 Exercises

1. ☺ Review the mappings in Table 1.3. Discuss any other examples of mappings you can think of. What type of information do they map from and to?

2. ☼ Using the Python interpreter in interactive mode, experiment with the examples in this section. Create a dictionary d, and add some entries. What happens if you try to access a non-existent entry, e.g. d['xyz']?

3. ☼ Try deleting an element from a dictionary, using the syntax del d['abc']. Check that the item was deleted.

4. ☼ Create a dictionary e, to represent a single lexical entry for some word of your choice. Define keys like headword, part-of-speech, sense, and example, and assign them suitable values.

5. ☼ Create two dictionaries, d1 and d2, and add some entries to each. Now issue the command d1.update(d2). What did this do? What might it be useful for?

6. ◑ Write a program that takes a sentence expressed as a single string, splits it and counts up the words. Get it to print out each word and the word's frequency, one per line, in alphabetical order.

1.8 Regular Expressions

For a moment, imagine that you are editing a large text, and you have strong dislike of repeated occurrences of the word *very*. How could you find all such cases in the text? To be concrete, let's suppose that we assign the following text to the variable s:

```
>>> s = """Google Analytics is very very very nice (now)
... By Jason Hoffman 18 August 06
... Google Analytics, the result of Google's acquisition of the San
... Diego-based Urchin Software Corporation, really really opened its
... doors to the world a couple of days ago, and it allows you to
... track up to 10 sites within a single google account.
... """
>>>
```

Python's triple quotes """ are used here since they allow us to break a string across lines.

One approach to our task would be to convert the string into a list, and look for adjacent items that are both equal to the string 'very'. We use the range(n) function in this example to create a list of consecutive integers from 0 up to, but not including, n:

```
>>> text = s.split()
>>> for n in range(len(text)):
...     if text[n] == 'very' and text[n+1] == 'very':
...                 print n, n+1
...
3 4
4 5
>>>
```

However, such an approach is not very flexible or convenient. In this section, we will present Python's **regular expression** module `re`, which supports powerful search and substitution inside strings. As a gentle introduction, we will start out using a utility function `re_show()` to illustrate how regular expressions match against substrings. `re_show()` takes two arguments, a pattern that it is looking for, and a string in which the pattern might occur.

```
>>> import nltk
>>> nltk.re_show('very very', s)
Google Analytics is {very very} very nice (now)
...
>>>
```

(We have only displayed the first part of `s` that is returned, since the rest is irrelevant for the moment.) As you can see, `re_show` places curly braces around the first occurrence it has found of the string `'very very'`. So an important part of what `re_show` is doing is searching for any substring of `s` that **matches** the pattern in its first argument.

Now we might want to modify the example so that `re_show` highlights cases where there are two *or more* adjacent sequences of `'very'`. To do this, we need to use a **regular expression operator**, namely `'+'`. If *s* is a string, then *s*+ means: 'match one or more occurrences of *s*'. Let's first look at the case where *s* is a single character, namely the letter `'o'`:

```
>>> nltk.re_show('o+', s)
G{oo}gle Analytics is very very very nice (n{o}w)
...
>>>
```

`'o+'` is our first proper regular expression. You can think of it as matching an *infinite set* of strings, namely the set {`'o'`, `'oo'`, `'ooo'`, ...}. But we would really like to match sequences of least two `'o'`s; for this, we need the regular expression `'oo+'`, which matches any string consisting of `'o'` followed by one or more occurrences of `o`.

```
>>> nltk.re_show('oo+', s)
G{oo}gle Analytics is very very very nice (now)
...
>>>
```

Let's return to the task of identifying multiple occurrences of `'very'`. Some initially plausible candidates won't do what we want. For example, `'very+'` would match `'veryyy'` (but not `'very very'`), since the + scopes over the immediately preceding expression, in this case `'y'`. To widen the scope of +, we need to use parentheses, as in `'(very)+'`. Will this match `'very very'`? No, because we've forgotten about the whitespace between the two words; instead, it will match strings like `'veryvery'`. However, the following *does* work:

```
>>> nltk.re_show('(very\s)+', s)
Google Analytics is {very very very }nice (now)
>>>
```

Characters preceded by a \, such as `'\s'`, have a special interpretation inside regular expressions; thus, `'\s'` matches a whitespace character. We could have used `' '` in our pattern, but `'\s'` is better practice in general. One reason is that the sense of "whitespace" we are using is more general than you might have imagined; it includes not just inter-word spaces, but also tabs and newlines. If you try to inspect the variable `s`, you might initially get a shock:

```
>>> s
"Google Analytics is very very very nice (now)\nBy Jason Hoffman
18 August 06\nGoogle
...
>>>
```

You might recall that '\n' is a special character that corresponds to a newline in a string. The following example shows how newline is matched by '\s'.

```
>>> s2 = "I'm very very\nvery happy"
>>> nltk.re_show('very\s', s2)
I'm {very }{very
}{very }happy
>>>
```

Python's re.findall(*patt*, *s*) function is a useful way to find all the substrings in *s* that are matched by *patt*. Before illustrating, let's introduce two further special characters, '\d' and '\w': the first will match any digit, and the second will match any alphanumeric character. Before we can use re.findall() we have to load Python's regular expression module, using import re.

```
>>> import re
>>> re.findall('\d\d', s)
['18', '06', '10']
>>> re.findall('\s\w\w\w\s', s)
[' the ', ' the ', ' its\n', ' the ', ' and ', ' you ']
>>>
```

As you will see, the second example matches three-letter words. However, this regular expression is not quite what we want. First, the leading and trailing spaces are extraneous. Second, it will fail to match against strings such as 'the San', where two three-letter words are adjacent. To solve this problem, we can use another special character, namely '\b'. This is sometimes called a "zero-width" character; it matches against the empty string, but only at the beginning and end of words:

```
>>> re.findall(r'\b\w\w\w\b', s)
['now', 'the', 'the', 'San', 'its', 'the', 'ago', 'and', 'you']
```

Note

This example uses a Python **raw string**: r'\b\w\w\w\b'. The specific justification here is that in an ordinary string, \b is interpreted as a backspace character. Python will convert it to a backspace in a regular expression unless you use the r prefix to create a raw string as shown above. Another use for raw strings is to match strings that include backslashes. Suppose we want to match 'either\or'. In order to create a regular expression, the backslash needs to be escaped, since it is a special character; so we want to pass the pattern \\ to the regular expression interpreter. But to express this as a Python string literal, each backslash must be escaped again, yielding the string '\\\\'. However, with a raw string, this reduces down to r'\\'.

Returning to the case of repeated words, we might want to look for cases involving 'very' or 'really', and for this we use the disjunction operator |.

```
>>> nltk.re_show('((very|really)\s)+', s)
Google Analytics is {very very very }nice (now)
By Jason Hoffman 18 August 06
Google Analytics, the result of Google's acquisition of the San
Diego-based Urchin Software Corporation, {really really }opened its
doors to the world a couple of days ago, and it allows you to
track up to 10 sites within a single google account.
>>>
```

In addition to the matches just illustrated, the regular expression `'((very|really)\s)+'` will also match cases where the two disjuncts occur with each other, such as the string `'really very really '`.

Let's now look at how to perform substitutions, using the `re.sub()` function. In the first instance we replace all instances of `l` with `s`. Note that this generates a string as output, and doesn't modify the original string. Then we replace any instances of `green` with `red`.

```
>>> sent = "colorless green ideas sleep furiously"
>>> re.sub('l', 's', sent)
'cosorsess green ideas sseep furioussy'
>>> re.sub('green', 'red', sent)
'colorless red ideas sleep furiously'
>>>
```

We can also disjoin individual characters using a square bracket notation. For example, `[aeiou]` matches any of `a`, `e`, `i`, `o`, or `u`, that is, any vowel. The expression `[^aeiou]` matches any single character that is *not* a vowel. In the following example, we match sequences consisting of a non-vowel followed by a vowel.

```
>>> nltk.re_show('[^aeiou][aeiou]', sent)
{co}{lo}r{le}ss g{re}en{ i}{de}as s{le}ep {fu}{ri}ously
>>>
```

Using the same regular expression, the function `re.findall()` returns a list of all the substrings in `sent` that are matched:

```
>>> re.findall('[^aeiou][aeiou]', sent)
['co', 'lo', 'le', 're', ' i', 'de', 'le', 'fu', 'ri']
>>>
```

1.8.1 Groupings

Returning briefly to our earlier problem with unwanted whitespace around three-letter words, we note that `re.findall()` behaves slightly differently if we create **groups** in the regular expression using parentheses; it only returns strings that occur within the groups:

```
>>> re.findall('\s(\w\w\w)\s', s)
['the', 'the', 'its', 'the', 'and', 'you']
>>>
```

The same device allows us to select only the non-vowel characters that appear before a vowel:

```
>>> re.findall('([^aeiou])[aeiou]', sent)
['c', 'l', 'l', 'r', ' ', 'd', 'l', 'f', 'r']
>>>
```

By delimiting a second group in the regular expression, we can even generate pairs (or **tuples**) that we may then go on and tabulate.

```
>>> re.findall('([^aeiou])([aeiou])', sent)
[('c', 'o'), ('l', 'o'), ('l', 'e'), ('r', 'e'), (' ', 'i'),
    ('d', 'e'), ('l', 'e'), ('f', 'u'), ('r', 'i')]
>>>
```

Our next example also makes use of groups. One further special character is the so-called wildcard element, `'.'`; this has the distinction of matching any single character (except `'\n'`). Given the string `s3`, our task is to pick out login names and email domains:

```
>>> s3 = """
... <hart@vmd.cso.uiuc.edu>
... Final editing was done by Martin Ward <Martin.Ward@uk.ac.durham>
... Michael S. Hart <hart@pobox.com>
... Prepared by David Price, email <ccx074@coventry.ac.uk>"""
```

The task is made much easier by the fact that all the email addresses in the example are delimited by angle brackets, and we can exploit this feature in our regular expression:

```
>>> re.findall(r'<(.+)@(.+)>', s3)
[('hart', 'vmd.cso.uiuc.edu'), ('Martin.Ward', 'uk.ac.durham'),
('hart', 'pobox.com'), ('ccx074', 'coventry.ac.uk')]
>>>
```

Since `'.'` matches any single character, `'.+'` will match any non-empty *string* of characters, including punctuation symbols such as the period.

One question that might occur to you is how do we specify a match against a period? The answer is that we have to place a `'\'` immediately before the `'.'` in order to escape its special interpretation.

```
>>> re.findall(r'(\w+\.)', s3)
['vmd.', 'cso.', 'uiuc.', 'Martin.', 'uk.', 'ac.', 'S.',
'pobox.', 'coventry.', 'ac.']
>>>
```

Now, let's suppose that we wanted to match occurrences of both `'Google'` and `'google'` in our sample text. If you have been following up till now, you would reasonably expect that this regular expression with a disjunction would do the trick: `'(G|g)oogle'`. But look what happens when we try this with `re.findall()`:

```
>>> re.findall('(G|g)oogle', s)
['G', 'G', 'G', 'g']
>>>
```

What is going wrong? We innocently used the parentheses to indicate the scope of the operator `'|'`, but `re.findall()` has interpreted them as marking a group. In order to tell `re.findall()` "don't try to do anything special with these parentheses", we need an extra piece of notation:

```
>>> re.findall('(?:G|g)oogle', s)
['Google', 'Google', 'Google', 'google']
>>>
```

Placing `'?:'` immediately after the opening parenthesis makes it explicit that the parentheses are just being used for scoping.

1.8.2 Practice Makes Perfect

Regular expressions are very flexible and very powerful. However, they often don't do what you expect. For this reason, you are strongly encouraged to try out a variety of tasks using `re_show()` and `re.findall()` in order to develop your intuitions further; the exercises below should help get you started. We suggest that you build up a regular expression in small pieces, rather than trying to get it completely right first time. Here are some operators and sequences that are commonly used in natural language processing.

Commonly-used Operators and Sequences	
`*`	Zero or more, e.g. `a*`, `[a-z]*`
`+`	One or more, e.g. `a+`, `[a-z]+`
`?`	Zero or one (i.e. optional), e.g. `a?`, `[a-z]?`
`[..]`	A set or range of characters, e.g. `[aeiou]`, `[a-z0-9]`
`(..)`	Grouping parentheses, e.g. `(the|a|an)`
`\b`	Word boundary (zero width)
`\d`	Any decimal digit (`\D` is any non-digit)
`\s`	Any whitespace character (`\S` is any non-whitespace character)
`\w`	Any alphanumeric character (`\W` is any non-alphanumeric character
`\t`	The tab character
`\n`	The newline character

Table 1.4:

1.8.3 Exercises

1. ☼ Describe the class of strings matched by the following regular expressions. Note that `'*'` means: match zero or more occurrences of the preceding regular expression.

    a) `[a-zA-Z]+`

    b) `[A-Z][a-z]*`

    c) `\d+(\.\d+)?`

    d) `([bcdfghjklmnpqrstvwxyz][aeiou][bcdfghjklmnpqrstvwxyz])*`

    e) `\w+|[^\w\s]+`

    Test your answers using `re_show()`.

2. ☼ Write regular expressions to match the following classes of strings:

    a) A single determiner (assume that *a*, *an*, and *the* are the only determiners).

    b) An arithmetic expression using integers, addition, and multiplication, such as `2*3+8`.

3. ◑ The above example of extracting (name, domain) pairs from text does not work when there is more than one email address on a line, because the + operator is "greedy" and consumes too much of the input.

   a) Experiment with input text containing more than one email address per line, such as that shown below. What happens?

   b) Using `re.findall()`, write another regular expression to extract email addresses, replacing the period character with a range or negated range, such as `[a-z]+` or `[^ >]+`.

   c) Now try to match email addresses by changing the regular expression `.+` to its "non-greedy" counterpart, `.+?`

```
>>> s = """
... austen-emma.txt:hart@vmd.cso.uiuc.edu  (internet)  hart@uiucvmd (bitnet)
... austen-emma.txt:Internet (72600.2026@compuserve.com); TEL: (212-254-5093)
... austen-persuasion.txt:Editing by Martin Ward (Martin.Ward@uk.ac.durham)
... blake-songs.txt:Prepared by David Price, email ccx074@coventry.ac.uk
... """
```

4. ◑ Write code to convert text into Pig Latin. This involves two steps: move any consonant (or consonant cluster) that appears at the start of the word to the end, then append *ay*, e.g. *string* → *ingstray*, *idle* → *idleay*. `http://en.wikipedia.org/wiki/Pig_Latin`

5. ◑ Write code to convert text into *hAck3r* again, this time using regular expressions and substitution, where `e` → `3`, `i` → `1`, `o` → `0`, `l` → `|`, `s` → `5`, `.` → `5w33t!`, `ate` → `8`. Normalize the text to lowercase before converting it. Add more substitutions of your own. Now try to map `s` to two different values: `$` for word-initial `s`, and `5` for word-internal `s`.

6. ★ Read the Wikipedia entry on *Soundex*. Implement this algorithm in Python.

1.9 Summary

- Text is represented in Python using strings, and we type these with single or double quotes: `'Hello'`, `"World"`.
- The characters of a string are accessed using indexes, counting from zero: `'Hello World'[1]` gives the value `e`. The length of a string is found using `len()`.
- Substrings are accessed using slice notation: `'Hello World'[1:5]` gives the value `ello`. If the start index is omitted, the substring begins at the start of the string; if the end index is omitted, the slice continues to the end of the string.
- Sequences of words are represented in Python using lists of strings: `['colorless', 'green', 'ideas']`. We can use indexing, slicing and the `len()` function on lists.
- Strings can be split into lists: `'Hello World'.split()` gives `['Hello', 'World']`. Lists can be joined into strings: `'/'.join(['Hello', 'World'])` gives `'Hello/World'`.

- Lists can be sorted in-place: `words.sort()`. To produce a separate, sorted copy, use: `sorted(words)`.
- We process each item in a string or list using a `for` statement: `for word in phrase`. 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 dictionary is used to map between arbitrary types of information, such as a string and a number: `freq['cat'] = 12`. We create dictionaries using the brace notation: `pos = {}`, `pos = {'furiously': 'adv', 'ideas': 'n', 'colorless': 'adj'}`.
- Some functions are not available by default, but must be accessed using Python's `import` statement.
- Regular expressions are a powerful and flexible method of specifying patterns. Once we have imported the `re` module, we can use `re.findall()` to find all substrings in a string that match a pattern, and we can use `re.sub()` to replace substrings of one sort with another.

1.10 Further Reading

1.10.1 Python

Two freely available online texts are the following:

- Josh Cogliati, *Non-Programmer's Tutorial for Python*, http://en.wikibooks.org/wiki/Non-Programmer's_Tut
- Allen B. Downey, Jeffrey Elkner and Chris Meyers, *How to Think Like a Computer Scientist: Learning with Python*, http://www.ibiblio.org/obp/thinkCSpy/

An Introduction to Python [Rossum & Jr., 2006] is a Python tutorial by Guido van Rossum, the inventor of Python and Fred L. Drake, Jr., 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.

[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/.

1.10.2 Regular Expressions

There are many references for regular expressions, both practical and theoretical. [Friedl, 2002] is a comprehensive and detailed manual in using regular expressions, covering their syntax in most major programming languages, including Python.

For an introductory tutorial to using regular expressions in Python with the `re` module, see A. M. Kuchling, *Regular Expression HOWTO*, http://www.amk.ca/python/howto/regex/.

Chapter 3 of [Mertz, 2003] provides a more extended tutorial on Python's facilities for text processing with regular expressions.

http://www.regular-expressions.info/ is a useful online resource, providing a tutorial and references to tools and other sources of information.

About this document...

This chapter is a draft from *Natural Language Processing* [http://nltk.org/book.html], by Steven Bird, Ewan Klein and Edward Loper, Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [http://nltk.org/], Version 0.9.5, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [http://creativecommons.org/licenses/by-nc-nd/3.0/].

This document is