Python for Everybody

Exploring Data Using Python 3

Charles R. Severance

10.6 The most common words

Coming back to our running example of the text from *Romeo and Juliet* Act 2, Scene 2, we can augment our program to use this technique to print the ten most common words in the text as follows:

```
import string
fhand = open('romeo-full.txt')
counts = dict()
for line in fhand:
    line = line.translate(string.punctuation)
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

# Sort the dictionary by value
lst = list()
for key, val in list(counts.items()):
    lst.append((val, key))

lst.sort(reverse=True)

for key, val in lst[:10]:
    print(key, val)

# Code: http://www.pythonlearn.com/code3/count3.py
```

The first part of the program which reads the file and computes the dictionary that maps each word to the count of words in the document is unchanged. But instead of simply printing out `counts` and ending the program, we construct a list of `(val, key)` tuples and then sort the list in reverse order.

Since the value is first, it will be used for the comparisons. If there is more than one tuple with the same value, it will look at the second element (the key), so tuples where the value is the same will be further sorted by the alphabetical order of the key.

At the end we write a nice `for` loop which does a multiple assignment iteration and prints out the ten most common words by iterating through a slice of the list (`lst[:10]`).

So now the output finally looks like what we want for our word frequency analysis.

```
61 i
42 and
40 romeo
34 to
34 the
```

```
32 thou
32 juliet
30 that
29 my
24 thee
```

The fact that this complex data parsing and analysis can be done with an easy-to-understand 19-line Python program is one reason why Python is a good choice as a language for exploring information.

10.7 Using tuples as keys in dictionaries

Because tuples are *hashable* and lists are not, if we want to create a *composite* key to use in a dictionary we must use a tuple as the key.

We would encounter a composite key if we wanted to create a telephone directory that maps from last-name, first-name pairs to telephone numbers. Assuming that we have defined the variables `last`, `first`, and `number`, we could write a dictionary assignment statement as follows:

```
directory[last,first] = number
```

The expression in brackets is a tuple. We could use tuple assignment in a `for` loop to traverse this dictionary.

```
for last, first in directory:
    print(first, last, directory[last,first])
```

This loop traverses the keys in `directory`, which are tuples. It assigns the elements of each tuple to `last` and `first`, then prints the name and corresponding telephone number.

10.8 Sequences: strings, lists, and tuples - Oh My!

I have focused on lists of tuples, but almost all of the examples in this chapter also work with lists of lists, tuples of tuples, and tuples of lists. To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences.

In many contexts, the different kinds of sequences (strings, lists, and tuples) can be used interchangeably. So how and why do you choose one over the others?

To start with the obvious, strings are more limited than other sequences because the elements have to be characters. They are also immutable. If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

1. In some contexts, like a `return` statement, it is syntactically simpler to create a tuple than a list. In other contexts, you might prefer a list.

2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.

3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

Because tuples are immutable, they don't provide methods like `sort` and `reverse`, which modify existing lists. However Python provides the built-in functions `sorted` and `reversed`, which take any sequence as a parameter and return a new sequence with the same elements in a different order.

10.9 Debugging

Lists, dictionaries and tuples are known generically as *data structures*; in this chapter we are starting to see compound data structures, like lists of tuples, and dictionaries that contain tuples as keys and lists as values. Compound data structures are useful, but they are prone to what I call *shape errors*; that is, errors caused when a data structure has the wrong type, size, or composition, or perhaps you write some code and forget the shape of your data and introduce an error.

For example, if you are expecting a list with one integer and I give you a plain old integer (not in a list), it won't work.

When you are debugging a program, and especially if you are working on a hard bug, there are four things to try:

reading Examine your code, read it back to yourself, and check that it says what you meant to say.

running Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to spend some time to build scaffolding.

ruminating Take some time to think! What kind of error is it: syntax, runtime, semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?

retreating At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works and that you understand. Then you can start rebuilding.

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode.

For example, reading your code might help if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand

what your program does, you can read it 100 times and never see the error, because the error is in your head.

Running experiments can help, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, you might fall into a pattern I call "random walk programming", which is the process of making random changes until the program does the right thing. Needless to say, random walk programming can take a long time.

You have to take time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

Taking a break helps with the thinking. So does talking. If you explain the problem to someone else (or even to yourself), you will sometimes find the answer before you finish asking the question.

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get to something that works and that you understand.

Beginning programmers are often reluctant to retreat because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can paste the pieces back in a little bit at a time.

Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.

10.10 Glossary

comparable A type where one value can be checked to see if it is greater than, less than, or equal to another value of the same type. Types which are comparable can be put in a list and sorted.

data structure A collection of related values, often organized in lists, dictionaries, tuples, etc.

DSU Abbreviation of "decorate-sort-undecorate", a pattern that involves building a list of tuples, sorting, and extracting part of the result.

gather The operation of assembling a variable-length argument tuple.

hashable A type that has a hash function. Immutable types like integers, floats, and strings are hashable; mutable types like lists and dictionaries are not.

scatter The operation of treating a sequence as a list of arguments.

shape (of a data structure) A summary of the type, size, and composition of a data structure.

singleton A list (or other sequence) with a single element.

tuple An immutable sequence of elements.

tuple assignment An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left.

10.11 Exercises

Exercise 1: Revise a previous program as follows: Read and parse the "From" lines and pull out the addresses from the line. Count the number of messages from each person using a dictionary.

After all the data has been read, print the person with the most commits by creating a list of (count, email) tuples from the dictionary. Then sort the list in reverse order and print out the person who has the most commits.

```
Sample Line:
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008

Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

Exercise 2: This program counts the distribution of the hour of the day for each of the messages. You can pull the hour from the "From" line by finding the time string and then splitting that string into parts using the colon character. Once you have accumulated the counts for each hour, print out the counts, one per line, sorted by hour as shown below.

Sample Execution:

```
python timeofday.py
Enter a file name: mbox-short.txt
04 3
06 1
07 1
09 2
```

```
10 3
11 6
14 1
15 2
16 4
17 2
18 1
19 1
```

Exercise 3: Write a program that reads a file and prints the *letters* in decreasing order of frequency. Your program should convert all the input to lower case and only count the letters a-z. Your program should not count spaces, digits, punctuation, or anything other than the letters a-z. Find text samples from several different languages and see how letter frequency varies between languages. Compare your results with the tables at wikipedia.org/wiki/Letter_frequencies.

Chapter 11

Regular expressions

So far we have been reading through files, looking for patterns and extracting various bits of lines that we find interesting. We have been

using string methods like `split` and `find` and using lists and string slicing to extract portions of the lines.

This task of searching and extracting is so common that Python has a very powerful library called *regular expressions* that handles many of these tasks quite elegantly. The reason we have not introduced regular expressions earlier in the book is because while they are very powerful, they are a little complicated and their syntax takes some getting used to.

Regular expressions are almost their own little programming language for searching and parsing strings. As a matter of fact, entire books have been written on the topic of regular expressions. In this chapter, we will only cover the basics of regular expressions. For more detail on regular expressions, see:

http://en.wikipedia.org/wiki/Regular_expression

https://docs.python.org/2/library/re.html

The regular expression library `re` must be imported into your program before you can use it. The simplest use of the regular expression library is the `search()` function. The following program demonstrates a trivial use of the search function.

```
# Search for lines that contain 'From'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line):
        print(line)

# Code: http://www.pythonlearn.com/code3/re01.py
```

We open the file, loop through each line, and use the regular expression `search()` to only print out lines that contain the string "From:". This program does not

use the real power of regular expressions, since we could have just as easily used `line.find()` to accomplish the same result.

The power of the regular expressions comes when we add special characters to the search string that allow us to more precisely control which lines match the string. Adding these special characters to our regular expression allow us to do sophisticated matching and extraction while writing very little code.

For example, the caret character is used in regular expressions to match "the beginning" of a line. We could change our program to only match lines where "From:" was at the beginning of the line as follows:

```
# Search for lines that start with 'From'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:', line):
        print(line)

# Code: http://www.pythonlearn.com/code3/re02.py
```

Now we will only match lines that *start with* the string "From:". This is still a very simple example that we could have done equivalently with the `startswith()` method from the string library. But it serves to introduce the notion that regular expressions contain special action characters that give us more control as to what will match the regular expression.

11.1 Character matching in regular expressions

There are a number of other special characters that let us build even more powerful regular expressions. The most commonly used special character is the period or full stop, which matches any character.

In the following example, the regular expression "F..m:" would match any of the strings "From:", "Fxxm:", "F12m:", or "F!@m:" since the period characters in the regular expression match any character.

```
# Search for lines that start with 'F', followed by
# 2 characters, followed by 'm:'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^F..m:', line):
        print(line)

# Code: http://www.pythonlearn.com/code3/re03.py
```

This is particularly powerful when combined with the ability to indicate that a character can be repeated any number of times using the "*" or "+" characters in your regular expression. These special characters mean that instead of matching a single character in the search string, they match zero-or-more characters (in the case of the asterisk) or one-or-more of the characters (in the case of the plus sign).

We can further narrow down the lines that we match using a repeated *wild card* character in the following example:

```
# Search for lines that start with From and have an at sign
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:.+@', line):
        print(line)

# Code: http://www.pythonlearn.com/code3/re04.py
```

The search string "^From:.+@" will successfully match lines that start with "From:", followed by one or more characters (".+"), followed by an at-sign. So this will match the following line:

From: `uct.ac.za`

You can think of the ".+" wildcard as expanding to match all the characters between the colon character and the at-sign.

From:

It is good to think of the plus and asterisk characters as "pushy". For example, the following string would match the last at-sign in the string as the ".+" pushes outwards, as shown below:

From: `iupui.edu`

It is possible to tell an asterisk or plus sign not to be so "greedy" by adding another character. See the detailed documentation for information on turning off the greedy behavior.

11.2 Extracting data using regular expressions

If we want to extract data from a string in Python we can use the `findall()` method to extract all of the substrings which match a regular expression. Let's use the example of wanting to extract anything that looks like an email address from any line regardless of format. For example, we want to pull the email addresses from each of the following lines:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
          for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
```

We don't want to write code for each of the types of lines, splitting and slicing differently for each line. This following program uses `findall()` to find the lines with email addresses in them and extract one or more addresses from each of those lines.

```
import re
s = 'A message from csev@umich.edu to cwen@iupui.edu about meeting @2PM'
lst = re.findall('\S+@\S+', s)
print(lst)

# Code: http://www.pythonlearn.com/code3/re05.py
```

The `findall()` method searches the string in the second argument and returns a list of all of the strings that look like email addresses. We are using a two-character sequence that matches a non-whitespace character (\\S).

The output of the program would be:

```
['csev@umich.edu', 'cwen@iupui.edu']
```

Translating the regular expression, we are looking for substrings that have at least one non-whitespace character, followed by an at-sign, followed by at least one more non-whitespace character. The "\\S+" matches as many non-whitespace characters as possible.

The regular expression would match twice (csev@umich.edu and cwen@iupui.edu), but it would not match the string "@2PM" because there are no non-blank characters *before* the at-sign. We can use this regular expression in a program to read all the lines in a file and print out anything that looks like an email address as follows:

```
# Search for lines that have an at sign between characters
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('\S+@\S+', line)
    if len(x) > 0:
        print(x)

# Code: http://www.pythonlearn.com/code3/re06.py
```

We read each line and then extract all the substrings that match our regular expression. Since `findall()` returns a list, we simply check if the number of elements in our returned list is more than zero to print only lines where we found at least one substring that looks like an email address.

If we run the program on `mbox.txt` we get the following output:

```
['wagnermr@iupui.edu']
['cwen@iupui.edu']
```

```
['<postmaster@collab.sakaiproject.org>']
['<200801032122.m03LMFo4005148@nakamura.uits.iupui.edu>']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost)']
['source@collab.sakaiproject.org;']
```

Some of our email addresses have incorrect characters like "<" or ";" at the beginning or end. Let's declare that we are only interested in the portion of the string that starts and ends with a letter or a number.

To do this, we use another feature of regular expressions. Square brackets are used to indicate a set of multiple acceptable characters we are willing to consider matching. In a sense, the "\\S" is asking to match the set of "non-whitespace characters". Now we will be a little more explicit in terms of the characters we will match.

Here is our new regular expression:

```
[a-zA-Z0-9]\S*@\S*[a-zA-Z]
```

This is getting a little complicated and you can begin to see why regular expressions are their own little language unto themselves. Translating this regular expression, we are looking for substrings that start with a *single* lowercase letter, uppercase letter, or number "[a-zA-Z0-9]", followed by zero or more non-blank characters ("\\S*"), followed by an at-sign, followed by zero or more non-blank characters ("\\S*"), followed by an uppercase or lowercase letter. Note that we switched from "+" to "*" to indicate zero or more non-blank characters since "[a-zA-Z0-9]" is already one non-blank character. Remember that the "*" or "+" applies to the single character immediately to the left of the plus or asterisk.

If we use this expression in our program, our data is much cleaner:

```
# Search for lines that have an at sign between characters
# The characters must be a letter or number
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('[a-zA-Z0-9]\S+@\S+[a-zA-Z]', line)
    if len(x) > 0:
        print(x)

# Code: http://www.pythonlearn.com/code3/re07.py
```

```
...
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['postmaster@collab.sakaiproject.org']
['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']
['source@collab.sakaiproject.org']
```

```
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['apache@localhost']
```

Notice that on the "source@collab.sakaiproject.org" lines, our regular expression eliminated two letters at the end of the string (">;"). This is because when we append "[a-zA-Z]" to the end of our regular expression, we are demanding that whatever string the regular expression parser finds must end with a letter. So when it sees the ">" after "sakaiproject.org>;" it simply stops at the last "matching" letter it found (i.e., the "g" was the last good match).

Also note that the output of the program is a Python list that has a string as the single element in the list.

11.3 Combining searching and extracting

If we want to find numbers on lines that start with the string "X-" such as:

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

we don't just want any floating-point numbers from any lines. We only want to extract numbers from lines that have the above syntax.

We can construct the following regular expression to select the lines:

```
^X-.*: [0-9.]+
```

Translating this, we are saying, we want lines that start with "X-", followed by zero or more characters (".*"), followed by a colon (":") and then a space. After the space we are looking for one or more characters that are either a digit (0-9) or a period "[0-9.]+". Note that inside the square brackets, the period matches an actual period (i.e., it is not a wildcard between the square brackets).

This is a very tight expression that will pretty much match only the lines we are interested in as follows:

```
# Search for lines that start with 'X' followed by any non
# whitespace characters and ':'
# followed by a space and any number.
# The number can include a decimal.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X\S*: [0-9.]+', line):
        print(line)

# Code: http://www.pythonlearn.com/code3/re10.py
```

When we run the program, we see the data nicely filtered to show only the lines we are looking for.

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
```

But now we have to solve the problem of extracting the numbers. While it would be simple enough to use `split`, we can use another feature of regular expressions to both search and parse the line at the same time.

Parentheses are another special character in regular expressions. When you add parentheses to a regular expression, they are ignored when matching the string. But when you are using `findall()`, parentheses indicate that while you want the whole expression to match, you only are interested in extracting a portion of the substring that matches the regular expression.

So we make the following change to our program:

```
# Search for lines that start with 'X' followed by any
# non whitespace characters and ':' followed by a space
# and any number. The number can include a decimal.
# Then print the number if it is greater than zero.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^X\S*: ([0-9.]+)', line)
    if len(x) > 0:
        print(x)

# Code: http://www.pythonlearn.com/code3/re11.py
```

Instead of calling `search()`, we add parentheses around the part of the regular expression that represents the floating-point number to indicate we only want `findall()` to give us back the floating-point number portion of the matching string.

The output from this program is as follows:

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
..
```

The numbers are still in a list and need to be converted from strings to floating point, but we have used the power of regular expressions to both search and extract the information we found interesting.

As another example of this technique, if you look at the file there are a number of lines of the form:

```
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

If we wanted to extract all of the revision numbers (the integer number at the end of these lines) using the same technique as above, we could write the following program:

```
# Search for lines that start with 'Details: rev='
# followed by numbers and '.'
# Then print the number if it is greater than zero
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^Details:.*rev=([0-9.]+)', line)
    if len(x) > 0:
        print(x)

# Code: http://www.pythonlearn.com/code3/re12.py
```

Translating our regular expression, we are looking for lines that start with "Details:", followed by any number of characters (".*"), followed by "rev=", and then by one or more digits. We want to find lines that match the entire expression but we only want to extract the integer number at the end of the line, so we surround "[0-9]+" with parentheses.

When we run the program, we get the following output:

```
['39772']
['39771']
['39770']
['39769']
...
```

Remember that the "[0-9]+" is "greedy" and it tries to make as large a string of digits as possible before extracting those digits. This "greedy" behavior is why we get all five digits for each number. The regular expression library expands in both directions until it encounters a non-digit, or the beginning or the end of a line.

Now we can use regular expressions to redo an exercise from earlier in the book where we were interested in the time of day of each mail message. We looked for lines of the form:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

and wanted to extract the hour of the day for each line. Previously we did this with two calls to `split`. First the line was split into words and then we pulled out the fifth word and split it again on the colon character to pull out the two characters we were interested in.

While this worked, it actually results in pretty brittle code that is assuming the lines are nicely formatted. If you were to add enough error checking (or a big try/except block) to insure that your program never failed when presented with incorrectly formatted lines, the code would balloon to 10-15 lines of code that was pretty hard to read.

We can do this in a far simpler way with the following regular expression:

```
^From .* [0-9][0-9]:
```

The translation of this regular expression is that we are looking for lines that start with "From" (note the space), followed by any number of characters (".*"), followed by a space, followed by two digits "[0-9][0-9]", followed by a colon character. This is the definition of the kinds of lines we are looking for.

In order to pull out only the hour using `findall()`, we add parentheses around the two digits as follows:

```
^From .* ([0-9][0-9]):
```

This results in the following program:

```
# Search for lines that start with From and a character
# followed by a two digit number between 00 and 99 followed by ':'
# Then print the number if it is greater than zero
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^From .* ([0-9][0-9]):', line)
    if len(x) > 0: print(x)

# Code: http://www.pythonlearn.com/code3/re13.py
```

When the program runs, it produces the following output:

```
['09']
['18']
['16']
['15']
...
```

11.4 Escape character

Since we use special characters in regular expressions to match the beginning or end of a line or specify wild cards, we need a way to indicate that these characters are "normal" and we want to match the actual character such as a dollar sign or caret.

We can indicate that we want to simply match a character by prefixing that character with a backslash. For example, we can find money amounts with the following regular expression.

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

Since we prefix the dollar sign with a backslash, it actually matches the dollar sign in the input string instead of matching the "end of line", and the rest of the regular expression matches one or more digits or the period character. *Note:* Inside square brackets, characters are not "special". So when we say "[0-9.]", it really means digits or a period. Outside of square brackets, a period is the "wildcard" character and matches any character. Inside square brackets, the period is a period.

11.5 Summary

While this only scratched the surface of regular expressions, we have learned a bit about the language of regular expressions. They are search strings with special characters in them that communicate your wishes to the regular expression system as to what defines "matching" and what is extracted from the matched strings. Here are some of those special characters and character sequences:

ˆ Matches the beginning of the line.

$ Matches the end of the line.

. Matches any character (a wildcard).

\\s Matches a whitespace character.

\\S Matches a non-whitespace character (opposite of \\s).

* Applies to the immediately preceding character and indicates to match zero or more of the preceding character(s).

*? Applies to the immediately preceding character and indicates to match zero or more of the preceding character(s) in "non-greedy mode".

- Applies to the immediately preceding character and indicates to match one or more of the preceding character(s).

+? Applies to the immediately preceding character and indicates to match one or more of the preceding character(s) in "non-greedy mode".

[aeiou] Matches a single character as long as that character is in the specified set. In this example, it would match "a", "e", "i", "o", or "u", but no other characters.

[a-z0-9] You can specify ranges of characters using the minus sign. This example is a single character that must be a lowercase letter or a digit.

[ˆA-Za-z] When the first character in the set notation is a caret, it inverts the logic. This example matches a single character that is anything *other than* an uppercase or lowercase letter.

( ) When parentheses are added to a regular expression, they are ignored for the purpose of matching, but allow you to extract a particular subset of the matched string rather than the whole string when using `findall()`.

\\b Matches the empty string, but only at the start or end of a word.

\\B Matches the empty string, but not at the start or end of a word.

\\d Matches any decimal digit; equivalent to the set [0-9].

\\D Matches any non-digit character; equivalent to the set [^0-9].

11.6 Bonus section for Unix / Linux users

Support for searching files using regular expressions was built into the Unix operating system since the 1960s and it is available in nearly all programming languages in one form or another.

As a matter of fact, there is a command-line program built into Unix called *grep* (Generalized Regular Expression Parser) that does pretty much the same as the `search()` examples in this chapter. So if you have a Macintosh or Linux system, you can try the following commands in your command-line window.

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

This tells `grep` to show you lines that start with the string "From:" in the file `mbox-short.txt`. If you experiment with the `grep` command a bit and read the documentation for `grep`, you will find some subtle differences between the regular expression support in Python and the regular expression support in `grep`. As an example, `grep` does not support the non-blank character "\\S" so you will need to use the slightly more complex set notation "[^]", which simply means match a character that is anything other than a space.

11.7 Debugging

Python has some simple and rudimentary built-in documentation that can be quite helpful if you need a quick refresher to trigger your memory about the exact name of a particular method. This documentation can be viewed in the Python interpreter in interactive mode.

You can bring up an interactive help system using `help()`.

```
>>> help()

help> modules
```

If you know what module you want to use, you can use the `dir()` command to find the methods in the module as follows:

```
>>> import re
>>> dir(re)
[.. 'compile', 'copy_reg', 'error', 'escape', 'findall',
'finditer', 'match', 'purge', 'search', 'split', 'sre_compile',
'sre_parse', 'sub', 'subn', 'sys', 'template']
```

You can also get a small amount of documentation on a particular method using the dir command.

```
>>> help (re.search)
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a match object, or None if no match was found.
>>>
```

The built-in documentation is not very extensive, but it can be helpful when you are in a hurry or don't have access to a web browser or search engine.

11.8 Glossary

brittle code Code that works when the input data is in a particular format but is prone to breakage if there is some deviation from the correct format. We call this "brittle code" because it is easily broken.

greedy matching The notion that the "+" and "*" characters in a regular expression expand outward to match the largest possible string.

grep A command available in most Unix systems that searches through text files looking for lines that match regular expressions. The command name stands for "Generalized Regular Expression Parser".

regular expression A language for expressing more complex search strings. A regular expression may contain special characters that indicate that a search only matches at the beginning or end of a line or many other similar capabilities.

wild card A special character that matches any character. In regular expressions the wild-card character is the period.

11.9 Exercises

Exercise 1: Write a simple program to simulate the operation of the `grep` command on Unix. Ask the user to enter a regular expression and count the number of lines that matched the regular expression:

```
$ python grep.py
Enter a regular expression: ^Author
mbox.txt had 1798 lines that matched ^Author

$ python grep.py
Enter a regular expression: ^X-
mbox.txt had 14368 lines that matched ^X-

$ python grep.py
Enter a regular expression: java$
mbox.txt had 4218 lines that matched java$
```

Exercise 2: Write a program to look for lines of the form

```
`New Revision: 39772`
```

and extract the number from each of the lines using a regular expression and the `findall()` method. Compute the average of the numbers and print out the average.

```
Enter file:mbox.txt
38549.7949721

Enter file:mbox-short.txt
39756.9259259
```

Chapter 12

Networked programs

While many of the examples in this book have focused on reading files and looking for data in those files, there are many different sources of

information when one considers the Internet.

In this chapter we will pretend to be a web browser and retrieve web pages using the HyperText Transport Protocol (HTTP). Then we will read through the web page data and parse it.

12.1 HyperText Transport Protocol - HTTP

The network protocol that powers the web is actually quite simple and there is built-in support in Python called `sockets` which makes it very easy to make network connections and retrieve data over those sockets in a Python program.

A *socket* is much like a file, except that a single socket provides a two-way connection between two programs. You can both read from and write to the same socket. If you write something to a socket, it is sent to the application at the other end of the socket. If you read from the socket, you are given the data which the other application has sent.

But if you try to read a socket when the program on the other end of the socket has not sent any data, you just sit and wait. If the programs on both ends of the socket simply wait for some data without sending anything, they will wait for a very long time.

So an important part of programs that communicate over the Internet is to have some sort of protocol. A protocol is a set of precise rules that determine who is to go first, what they are to do, and then what the responses are to that message, and who sends next, and so on. In a sense the two applications at either end of the socket are doing a dance and making sure not to step on each other's toes.

There are many documents which describe these network protocols. The HyperText Transport Protocol is described in the following document:

http://www.w3.org/Protocols/rfc2616/rfc2616.txt

This is a long and complex 176-page document with a lot of detail. If you find it interesting, feel free to read it all. But if you take a look around page 36 of RFC2616 you will find the syntax for the GET request. To request a document from a web server, we make a connection to the `www.pythonlearn.com` server on port 80, and then send a line of the form

`GET http://data.pr4e.org/romeo.txt HTTP/1.0`

where the second parameter is the web page we are requesting, and then we also send a blank line. The web server will respond with some header information about the document and a blank line followed by the document content.

12.2 The World's Simplest Web Browser

Perhaps the easiest way to show how the HTTP protocol works is to write a very simple Python program that makes a connection to a web server and follows the rules of the HTTP protocol to requests a document and display what the server sends back.

```
import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('data.pr4e.org', 80))
cmd = 'GET http://data.pr4e.org/romeo.txt HTTP/1.0\n\n'.encode()
mysock.send(cmd)

while True:
    data = mysock.recv(512)
    if (len(data) < 1):
        break
    print(data.decode())
mysock.close()

# Code: http://www.pythonlearn.com/code3/socket1.py
```

First the program makes a connection to port 80 on the server www.pythonlearn.com. Since our program is playing the role of the "web browser", the HTTP protocol says we must send the GET command followed by a blank line.

Once we send that blank line, we write a loop that receives data in 512-character chunks from the socket and prints the data out until there is no more data to read (i.e., the recv() returns an empty string).

The program produces the following output:

```
HTTP/1.1 200 OK
Date: Sun, 14 Mar 2010 23:52:41 GMT
Server: Apache
Last-Modified: Tue, 29 Dec 2009 01:31:22 GMT
ETag: "143c1b33-a7-4b395bea"
Accept-Ranges: bytes
```

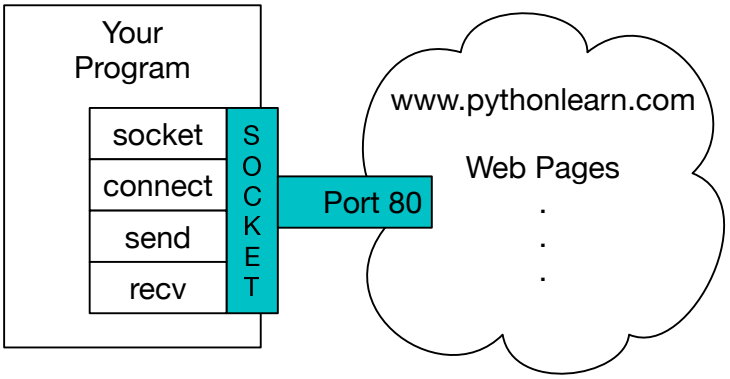

Figure 12.1: A Socket Connection

```
Content-Length: 167
Connection: close
Content-Type: text/plain

But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

The output starts with headers which the web server sends to describe the document. For example, the `Content-Type` header indicates that the document is a plain text document (`text/plain`).

After the server sends us the headers, it adds a blank line to indicate the end of the headers, and then sends the actual data of the file `romeo.txt`.

This example shows how to make a low-level network connection with sockets. Sockets can be used to communicate with a web server or with a mail server or many other kinds of servers. All that is needed is to find the document which describes the protocol and write the code to send and receive the data according to the protocol.

However, since the protocol that we use most commonly is the HTTP web protocol, Python has a special library specifically designed to support the HTTP protocol for the retrieval of documents and data over the web.

12.3 Retrieving an image over HTTP

In the above example, we retrieved a plain text file which had newlines in the file and we simply copied the data to the screen as the program ran. We can use a similar program to retrieve an image across using HTTP. Instead of copying the data to the screen as the program runs, we accumulate the data in a string, trim off the headers, and then save the image data to a file as follows:

```
import socket
import time
```

```
HOST = 'data.pr4e.org'
PORT = 80
mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect((HOST, PORT))
mysock.sendall(b'GET http://data.pr4e.org/cover.jpg HTTP/1.0\n\n')
count = 0
picture = b""

while True:
    data = mysock.recv(5120)
    if (len(data) < 1): break
    time.sleep(0.25)
    count = count + len(data)
    print(len(data), count)
    picture = picture + data

mysock.close()

# Look for the end of the header (2 CRLF)
pos = picture.find(b"\r\n\r\n")
print('Header length', pos)
print(picture[:pos].decode())

# Skip past the header and save the picture data
picture = picture[pos+4:]
fhand = open("stuff.jpg", "wb")
fhand.write(picture)
fhand.close()

# Code: http://www.pythonlearn.com/code3/urljpeg.py
```

When the program runs it produces the following output:

```
$ python urljpeg.py
2920 2920
1460 4380
1460 5840
1460 7300
...
1460 62780
1460 64240
2920 67160
1460 68620
1681 70301
Header length 240
HTTP/1.1 200 OK
Date: Sat, 02 Nov 2013 02:15:07 GMT
Server: Apache
Last-Modified: Sat, 02 Nov 2013 02:01:26 GMT
ETag: "19c141-111a9-4ea280f8354b8"
Accept-Ranges: bytes
```

```
Content-Length: 70057
Connection: close
Content-Type: image/jpeg
```

You can see that for this url, the `Content-Type` header indicates that body of the document is an image (`image/jpeg`). Once the program completes, you can view the image data by opening the file `stuff.jpg` in an image viewer.

As the program runs, you can see that we don't get 5120 characters each time we call the `recv()` method. We get as many characters as have been transferred across the network to us by the web server at the moment we call `recv()`. In this example, we either get 1460 or 2920 characters each time we request up to 5120 characters of data.

Your results may be different depending on your network speed. Also note that on the last call to `recv()` we get 1681 bytes, which is the end of the stream, and in the next call to `recv()` we get a zero-length string that tells us that the server has called `close()` on its end of the socket and there is no more data forthcoming.

We can slow down our successive `recv()` calls by uncommenting the call to `time.sleep()`. This way, we wait a quarter of a second after each call so that the server can "get ahead" of us and send more data to us before we call `recv()` again. With the delay, in place the program executes as follows:

```
$ python urljpeg.py
1460 1460
5120 6580
5120 11700
...
5120 62900
5120 68020
2281 70301
Header length 240
HTTP/1.1 200 OK
Date: Sat, 02 Nov 2013 02:22:04 GMT
Server: Apache
Last-Modified: Sat, 02 Nov 2013 02:01:26 GMT
ETag: "19c141-111a9-4ea280f8354b8"
Accept-Ranges: bytes
Content-Length: 70057
Connection: close
Content-Type: image/jpeg
```

Now other than the first and last calls to `recv()`, we now get 5120 characters each time we ask for new data.

There is a buffer between the server making `send()` requests and our application making `recv()` requests. When we run the program with the delay in place, at some point the server might fill up the buffer in the socket and be forced to pause until our program starts to empty the buffer. The pausing of either the sending application or the receiving application is called "flow control".

12.4 Retrieving web pages with `urllib`

While we can manually send and receive data over HTTP using the socket library, there is a much simpler way to perform this common task in Python by using the `urllib` library.

Using `urllib`, you can treat a web page much like a file. You simply indicate which web page you would like to retrieve and `urllib` handles all of the HTTP protocol and header details.

The equivalent code to read the `romeo.txt` file from the web using `urllib` is as follows:

```
import urllib.request, urllib.parse, urllib.error

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
for line in fhand:
    print(line.decode().strip())

# Code: http://www.pythonlearn.com/code3/urllib1.py
```

Once the web page has been opened with `urllib.urlopen`, we can treat it like a file and read through it using a `for` loop.

When the program runs, we only see the output of the contents of the file. The headers are still sent, but the `urllib` code consumes the headers and only returns the data to us.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

As an example, we can write a program to retrieve the data for `romeo.txt` and compute the frequency of each word in the file as follows:

```
import urllib.request, urllib.parse, urllib.error

counts = dict()
fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
for line in fhand:
    words = line.decode().split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1
print(counts)

# Code: http://www.pythonlearn.com/code3/urlwords.py
```

Again, once we have opened the web page, we can read it like a local file.

12.5 Parsing HTML and scraping the web

One of the common uses of the `urllib` capability in Python is to *scrape* the web. Web scraping is when we write a program that pretends to be a web browser and retrieves pages, then examines the data in those pages looking for patterns.

As an example, a search engine such as Google will look at the source of one web page and extract the links to other pages and retrieve those pages, extracting links, and so on. Using this technique, Google *spiders* its way through nearly all of the pages on the web.

Google also uses the frequency of links from pages it finds to a particular page as one measure of how "important" a page is and how high the page should appear in its search results.

12.6 Parsing HTML using regular expressions

One simple way to parse HTML is to use regular expressions to repeatedly search for and extract substrings that match a particular pattern.

Here is a simple web page:

```
<h1>The First Page</h1>
<p>
If you like, you can switch to the
<a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>.
</p>
```

We can construct a well-formed regular expression to match and extract the link values from the above text as follows:

```
href="http://.+?"
```

Our regular expression looks for strings that start with "href="http://", followed by one or more characters (".+¿'), followed by another double quote. The question mark added to the".+?" indicates that the match is to be done in a "non-greedy" fashion instead of a "greedy" fashion. A non-greedy match tries to find the *smallest* possible matching string and a greedy match tries to find the *largest* possible matching string.

We add parentheses to our regular expression to indicate which part of our matched string we would like to extract, and produce the following program:

```
# Search for lines that start with From and have an at sign
import urllib.request, urllib.parse, urllib.error
import re

url = input('Enter - ')
html = urllib.request.urlopen(url).read()
```

```
links = re.findall(b'href="(http://.*?)"', html)
for link in links:
    print(link.decode())

# Code: http://www.pythonlearn.com/code3/urlregex.py
```

The `findall` regular expression method will give us a list of all of the strings that match our regular expression, returning only the link text between the double quotes.

When we run the program, we get the following output:

```
python urlregex.py
Enter - http://www.dr-chuck.com/page1.htm
http://www.dr-chuck.com/page2.htm

python urlregex.py
Enter - http://www.pythonlearn.com/book.htm
http://www.greenteapress.com/thinkpython/thinkpython.html
http://allendowney.com/
http://www.pythonlearn.com/code
http://www.lib.umich.edu/espresso-book-machine
http://www.pythonlearn.com/py4inf-slides.zip
```

Regular expressions work very nicely when your HTML is well formatted and predictable. But since there are a lot of "broken" HTML pages out there, a solution only using regular expressions might either miss some valid links or end up with bad data.

This can be solved by using a robust HTML parsing library.

12.7 Parsing HTML using BeautifulSoup

There are a number of Python libraries which can help you parse HTML and extract data from the pages. Each of the libraries has its strengths and weaknesses and you can pick one based on your needs.

As an example, we will simply parse some HTML input and extract links using the *BeautifulSoup* library. You can download and install the BeautifulSoup code from:

http://www.crummy.com/software/

You can download and "install" BeautifulSoup or you can simply place the `BeautifulSoup.py` file in the same folder as your application.

Even though HTML looks like XML[1]i and some pages are carefully constructed to be XML, most HTML is generally broken in ways that cause an XML parser to reject the entire page of HTML as improperly formed. BeautifulSoup tolerates highly flawed HTML and still lets you easily extract the data you need.

We will use `urllib` to read the page and then use `BeautifulSoup` to extract the `href` attributes from the anchor (`a`) tags.

[1] The XML format is described in the next chapter.

```
# To run this, you can install BeautifulSoup
# https://pypi.python.org/pypi/beautifulsoup4

# Or download the file
# http://www.pythonlearn.com/code3/bs4.zip
# and unzip it in the same directory as this file

import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup

url = input('Enter - ')
html = urllib.request.urlopen(url).read()
soup = BeautifulSoup(html, 'html.parser')

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

# Code: http://www.pythonlearn.com/code3/urllinks.py
```

The program prompts for a web address, then opens the web page, reads the data and passes the data to the BeautifulSoup parser, and then retrieves all of the anchor tags and prints out the `href` attribute for each tag.

When the program runs it looks as follows:

```
python urllinks.py
Enter - http://www.dr-chuck.com/page1.htm
http://www.dr-chuck.com/page2.htm

python urllinks.py
Enter - http://www.pythonlearn.com/book.htm
http://www.greenteapress.com/thinkpython/thinkpython.html
http://allendowney.com/
http://www.si502.com/
http://www.lib.umich.edu/espresso-book-machine
http://www.pythonlearn.com/code
http://www.pythonlearn.com/
```

You can use BeautifulSoup to pull out various parts of each tag as follows:

```
# To run this, you can install BeautifulSoup
# https://pypi.python.org/pypi/beautifulsoup4

# Or download the file
# http://www.pythonlearn.com/code3/bs4.zip
# and unzip it in the same directory as this file

from urllib.request import urlopen
from bs4 import BeautifulSoup
```

```
url = input('Enter - ')
html = urlopen(url).read()

# html.parser is the HTML parser included in the standard Python 3 library.
# information on other HTML parsers is here:
# http://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser
soup = BeautifulSoup(html, "html.parser")

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    # Look at the parts of a tag
    print('TAG:', tag)
    print('URL:', tag.get('href', None))
    print('Contents:', tag.contents[0])
    print('Attrs:', tag.attrs)

# Code: http://www.pythonlearn.com/code3/urllink2.py
```

```
python urllink2.py
Enter - http://www.dr-chuck.com/page1.htm
TAG: <a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>
URL: http://www.dr-chuck.com/page2.htm
Content: ['\nSecond Page']
Attrs: [('href', 'http://www.dr-chuck.com/page2.htm')]
```

These examples only begin to show the power of BeautifulSoup when it comes to parsing HTML.

12.8 Reading binary files using urllib

Sometimes you want to retrieve a non-text (or binary) file such as an image or video file. The data in these files is generally not useful to print out, but you can easily make a copy of a URL to a local file on your hard disk using `urllib`.

The pattern is to open the URL and use `read` to download the entire contents of the document into a string variable (`img`) then write that information to a local file as follows:

```
import urllib.request, urllib.parse, urllib.error

img = urllib.request.urlopen('http://data.pr4e.org/cover.jpg').read()
fhand = open('cover.jpg', 'wb')
fhand.write(img)
fhand.close()

# Code: http://www.pythonlearn.com/code3/curl1.py
```

This program reads all of the data in at once across the network and stores it in the variable `img` in the main memory of your computer, then opens the file `cover.jpg` and writes the data out to your disk. This will work if the size of the file is less than the size of the memory of your computer.

However if this is a large audio or video file, this program may crash or at least run extremely slowly when your computer runs out of memory. In order to avoid running out of memory, we retrieve the data in blocks (or buffers) and then write each block to your disk before retrieving the next block. This way the program can read any size file without using up all of the memory you have in your computer.

```
import urllib.request, urllib.parse, urllib.error

img = urllib.request.urlopen('http://data.pr4e.org/cover.jpg')
fhand = open('cover.jpg', 'wb')
size = 0
while True:
    info = img.read(100000)
    if len(info) < 1: break
    size = size + len(info)
    fhand.write(info)

print(size, 'characters copied.')
fhand.close()

# Code: http://www.pythonlearn.com/code3/curl2.py
```

In this example, we read only 100,000 characters at a time and then write those characters to the `cover.jpg` file before retrieving the next 100,000 characters of data from the web.

This program runs as follows:

```
python curl2.py
568248 characters copied.
```

If you have a Unix or Macintosh computer, you probably have a command built in to your operating system that performs this operation as follows:

```
curl -O http://www.pythonlearn.com/cover.jpg
```

The command `curl` is short for "copy URL" and so these two examples are cleverly named `curl1.py` and `curl2.py` on www.pythonlearn.com/code as they implement similar functionality to the `curl` command. There is also a `curl3.py` sample program that does this task a little more effectively, in case you actually want to use this pattern in a program you are writing.

12.9 Glossary

BeautifulSoup A Python library for parsing HTML documents and extracting data from HTML documents that compensates for most of the imperfections

in the HTML that browsers generally ignore. You can download the BeautifulSoup code from www.crummy.com.

port A number that generally indicates which application you are contacting when you make a socket connection to a server. As an example, web traffic usually uses port 80 while email traffic uses port 25.

scrape When a program pretends to be a web browser and retrieves a web page, then looks at the web page content. Often programs are following the links in one page to find the next page so they can traverse a network of pages or a social network.

socket A network connection between two applications where the applications can send and receive data in either direction.

spider The act of a web search engine retrieving a page and then all the pages linked from a page and so on until they have nearly all of the pages on the Internet which they use to build their search index.

12.10 Exercises

Exercise 1: Change the socket program `socket1.py` to prompt the user for the URL so it can read any web page. You can use `split('/')` to break the URL into its component parts so you can extract the host name for the socket `connect` call. Add error checking using `try` and `except` to handle the condition where the user enters an improperly formatted or non-existent URL.

Exercise 2: Change your socket program so that it counts the number of characters it has received and stops displaying any text after it has shown 3000 characters. The program should retrieve the entire document and count the total number of characters and display the count of the number of characters at the end of the document.

Exercise 3: Use `urllib` to replicate the previous exercise of (1) retrieving the document from a URL, (2) displaying up to 3000 characters, and (3) counting the overall number of characters in the document. Don't worry about the headers for this exercise, simply show the first 3000 characters of the document contents.

Exercise 4: Change the `urllinks.py` program to extract and count paragraph (p) tags from the retrieved HTML document and display the count of the paragraphs as the output of your program. Do not display the paragraph text, only count them. Test your program on several small web pages as well as some larger web pages.

Exercise 5: (Advanced) Change the socket program so that it only shows data after the headers and a blank line have been received. Remember that `recv` is receiving characters (newlines and all), not lines.

Chapter 13

Using Web Services

Once it became easy to retrieve documents and parse documents over HTTP using programs, it did not take long to develop an approach where we

started producing documents that were specifically designed to be consumed by other programs (i.e., not HTML to be displayed in a browser).

There are two common formats that we use when exchanging data across the web. The "eXtensible Markup Language" or XML has been in use for a very long time and is best suited for exchanging document-style data. When programs just want to exchange dictionaries, lists, or other internal information with each other, they use JavaScript Object Notation or JSON (see www.json.org). We will look at both formats.

13.1 eXtensible Markup Language - XML

XML looks very similar to HTML, but XML is more structured than HTML. Here is a sample of an XML document:

```
<person>
  <name>Chuck</name>
  <phone type="intl">
     +1 734 303 4456
   </phone>
   <email hide="yes"/>
</person>
```

Often it is helpful to think of an XML document as a tree structure where there is a top tag `person` and other tags such as `phone` are drawn as *children* of their parent nodes.

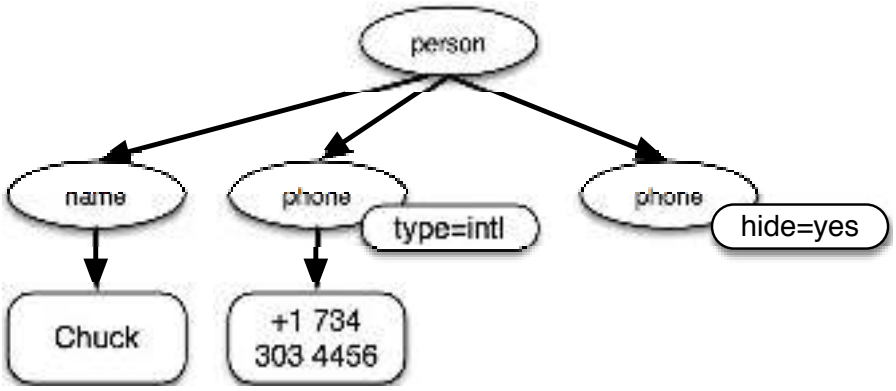

Figure 13.1: A Tree Representation of XML

13.2 Parsing XML

Here is a simple application that parses some XML and extracts some data elements from the XML:

```
import xml.etree.ElementTree as ET

data = '''
<person>
  <name>Chuck</name>
  <phone type="intl">
     +1 734 303 4456
   </phone>
   <email hide="yes"/>
</person>'''

tree = ET.fromstring(data)
print('Name:', tree.find('name').text)
print('Attr:', tree.find('email').get('hide'))

# Code: http://www.pythonlearn.com/code3/xml1.py
```

Calling `fromstring` converts the string representation of the XML into a "tree" of XML nodes. When the XML is in a tree, we have a series of methods we can call to extract portions of data from the XML.

The `find` function searches through the XML tree and retrieves a *node* that matches the specified tag. Each node can have some text, some attributes (like hide), and some "child" nodes. Each node can be the top of a tree of nodes.

```
Name: Chuck
Attr: yes
```

Using an XML parser such as `ElementTree` has the advantage that while the XML in this example is quite simple, it turns out there are many rules regarding

valid XML and using `ElementTree` allows us to extract data from XML without worrying about the rules of XML syntax.

13.3 Looping through nodes

Often the XML has multiple nodes and we need to write a loop to process all of the nodes. In the following program, we loop through all of the `user` nodes:

```
import xml.etree.ElementTree as ET

input = '''
<stuff>
    <users>
        <user x="2">
            <id>001</id>
            <name>Chuck</name>
        </user>
        <user x="7">
            <id>009</id>
            <name>Brent</name>
            </user>
        </users>
</stuff>'''

stuff = ET.fromstring(input)
lst = stuff.findall('users/user')
print('User count:', len(lst))

for item in lst:
    print('Name', item.find('name').text)
    print('Id', item.find('id').text)
    print('Attribute', item.get("x"))

# Code: http://www.pythonlearn.com/code3/xml2.py
```

The `findall` method retrieves a Python list of subtrees that represent the `user` structures in the XML tree. Then we can write a `for` loop that looks at each of the user nodes, and prints the `name` and `id` text elements as well as the `x` attribute from the `user` node.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

13.4 JavaScript Object Notation - JSON

The JSON format was inspired by the object and array format used in the JavaScript language. But since Python was invented before JavaScript, Python's syntax for dictionaries and lists influenced the syntax of JSON. So the format of JSON is nearly identical to a combination of Python lists and dictionaries.

Here is a JSON encoding that is roughly equivalent to the simple XML from above:

```
{
  "name" : "Chuck",
  "phone" : {
    "type" : "intl",
    "number" : "+1 734 303 4456"
   },
   "email" : {
     "hide" : "yes"
   }
}
```

You will notice some differences. First, in XML, we can add attributes like "intl" to the "phone" tag. In JSON, we simply have key-value pairs. Also the XML "person" tag is gone, replaced by a set of outer curly braces.

In general, JSON structures are simpler than XML because JSON has fewer capabilities than XML. But JSON has the advantage that it maps *directly* to some combination of dictionaries and lists. And since nearly all programming languages have something equivalent to Python's dictionaries and lists, JSON is a very natural format to have two cooperating programs exchange data.

JSON is quickly becoming the format of choice for nearly all data exchange between applications because of its relative simplicity compared to XML.

13.5 Parsing JSON

We construct our JSON by nesting dictionaries (objects) and lists as needed. In this example, we represent a list of users where each user is a set of key-value pairs (i.e., a dictionary). So we have a list of dictionaries.

In the following program, we use the built-in *json* library to parse the JSON and read through the data. Compare this closely to the equivalent XML data and code above. The JSON has less detail, so we must know in advance that we are getting a list and that the list is of users and each user is a set of key-value pairs. The JSON is more succinct (an advantage) but also is less self-describing (a disadvantage).

```
import json

input = '''
[
  { "id" : "001",
```

```
    "x" : "2",
    "name" : "Chuck"
  } ,
  { "id" : "009",
    "x" : "7",
    "name" : "Chuck"
  }
]'''

info = json.loads(input)
print('User count:', len(info))

for item in info:
    print('Name', item['name'])
    print('Id', item['id'])
    print('Attribute', item['x'])

# Code: http://www.pythonlearn.com/code3/json2.py
```

If you compare the code to extract data from the parsed JSON and XML you will see that what we get from *json.loads()* is a Python list which we traverse with a `for` loop, and each item within that list is a Python dictionary. Once the JSON has been parsed, we can use the Python index operator to extract the various bits of data for each user. We don't have to use the JSON library to dig through the parsed JSON, since the returned data is simply native Python structures.

The output of this program is exactly the same as the XML version above.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

In general, there is an industry trend away from XML and towards JSON for web services. Because the JSON is simpler and more directly maps to native data structures we already have in programming languages, the parsing and data extraction code is usually simpler and more direct when using JSON. But XML is more self-descriptive than JSON and so there are some applications where XML retains an advantage. For example, most word processors store documents internally using XML rather than JSON.

13.6 Application Programming Interfaces

We now have the ability to exchange data between applications using HyperText Transport Protocol (HTTP) and a way to represent complex data that we are sending back and forth between these applications using eXtensible Markup Language (XML) or JavaScript Object Notation (JSON).

The next step is to begin to define and document "contracts" between applications using these techniques. The general name for these application-to-application contracts is *Application Program Interfaces* or APIs. When we use an API, generally one program makes a set of *services* available for use by other applications and publishes the APIs (i.e., the "rules") that must be followed to access the services provided by the program.

When we begin to build our programs where the functionality of our program includes access to services provided by other programs, we call the approach a *Service-Oriented Architecture* or SOA. A SOA approach is one where our overall application makes use of the services of other applications. A non-SOA approach is where the application is a single standalone application which contains all of the code necessary to implement the application.

We see many examples of SOA when we use the web. We can go to a single web site and book air travel, hotels, and automobiles all from a single site. The data for hotels is not stored on the airline computers. Instead, the airline computers contact the services on the hotel computers and retrieve the hotel data and present it to the user. When the user agrees to make a hotel reservation using the airline site, the airline site uses another web service on the hotel systems to actually make the reservation. And when it comes time to charge your credit card for the whole transaction, still other computers become involved in the process.

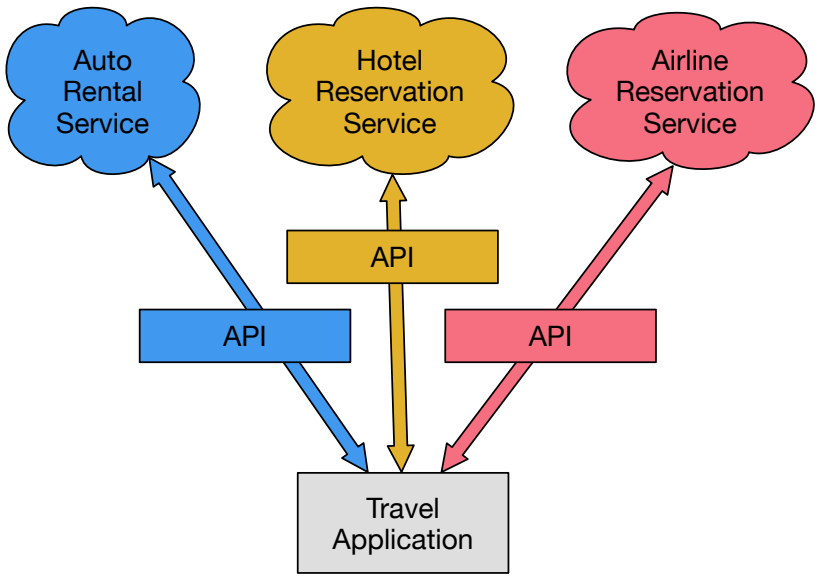

Figure 13.2: Service Oriented Architecture

A Service-Oriented Architecture has many advantages including: (1) we always maintain only one copy of data (this is particularly important for things like hotel reservations where we do not want to over-commit) and (2) the owners of the data can set the rules about the use of their data. With these advantages, an SOA system must be carefully designed to have good performance and meet the user's needs.

When an application makes a set of services in its API available over the web, we call these *web services*.

13.7 Google geocoding web service

Google has an excellent web service that allows us to make use of their large database of geographic information. We can submit a geographical search string like "Ann Arbor, MI" to their geocoding API and have Google return its best guess as to where on a map we might find our search string and tell us about the landmarks nearby.

The geocoding service is free but rate limited so you cannot make unlimited use of the API in a commercial application. But if you have some survey data where an end user has entered a location in a free-format input box, you can use this API to clean up your data quite nicely.

When you are using a free API like Google's geocoding API, you need to be respectful in your use of these resources. If too many people abuse the service, Google might drop or significantly curtail its free service.

You can read the online documentation for this service, but it is quite simple and you can even test it using a browser by typing the following URL into your browser:

http://maps.googleapis.com/maps/api/geocode/json?address=Ann+Arbor%2C+MI

Make sure to unwrap the URL and remove any spaces from the URL before pasting it into your browser.

The following is a simple application to prompt the user for a search string, call the Google geocoding API, and extract information from the returned JSON.

```
import urllib.request, urllib.parse, urllib.error
import json

serviceurl = 'http://maps.googleapis.com/maps/api/geocode/json?'

while True:
    address = input('Enter location: ')
    if len(address) < 1: break

    url = serviceurl + urllib.parse.urlencode(
        {'sensor': 'false', 'address': address})

    print('Retrieving', url)
    uh = urllib.request.urlopen(url)
    data = uh.read().decode()
    print('Retrieved', len(data), 'characters')

    try:
        js = json.loads(data)
    except:
        js = None

    if not js or 'status' not in js or js['status'] != 'OK':
        print('==== Failure To Retrieve ====')
        print(data)
```

```
        continue

    print(json.dumps(js, indent=4))

    lat = js["results"][0]["geometry"]["location"]["lat"]
    lng = js["results"][0]["geometry"]["location"]["lng"]
    print('lat', lat, 'lng', lng)
    location = js['results'][0]['formatted_address']
    print(location)

# Code: http://www.pythonlearn.com/code3/geojson.py
```

The program takes the search string and constructs a URL with the search string as a properly encoded parameter and then uses *urllib* to retrieve the text from the Google geocoding API. Unlike a fixed web page, the data we get depends on the parameters we send and the geographical data stored in Google's servers.

Once we retrieve the JSON data, we parse it with the *json* library and do a few checks to make sure that we received good data, then extract the information that we are looking for.

The output of the program is as follows (some of the returned JSON has been removed):

```
$ python geojson.py
Enter location: Ann Arbor, MI
Retrieving http://maps.googleapis.com/maps/api/
  geocode/json?sensor=false&address=Ann+Arbor%2C+MI
Retrieved 1669 characters

{
    "status": "OK",
    "results": [
        {
            "geometry": {
                "location_type": "APPROXIMATE",
                "location": {
                    "lat": 42.2808256,
                    "lng": -83.7430378
                }
            },
            "address_components": [
                {
                    "long_name": "Ann Arbor",
                    "types": [
                        "locality",
                        "political"
                    ],
                    "short_name": "Ann Arbor"
                }
            ],
```

```
            "formatted_address": "Ann Arbor, MI, USA",
            "types": [
                "locality",
                "political"
            ]
        }
    ]
}
lat 42.2808256 lng -83.7430378
Ann Arbor, MI, USA

Enter location:
```

You can download www.pythonlearn.com/code3/geoxml.py to explore the XML variant of the Google geocoding API.

13.8 Security and API usage

It is quite common that you need some kind of "API key" to make use of a vendor's API. The general idea is that they want to know who is using their services and how much each user is using. Perhaps they have free and pay tiers of their services or have a policy that limits the number of requests that a single individual can make during a particular time period.

Sometimes once you get your API key, you simply include the key as part of POST data or perhaps as a parameter on the URL when calling the API.

Other times, the vendor wants increased assurance of the source of the requests and so they add expect you to send cryptographically signed messages using shared keys and secrets. A very common technology that is used to sign requests over the Internet is called *OAuth*. You can read more about the OAuth protocol at www.oauth.net.

As the Twitter API became increasingly valuable, Twitter went from an open and public API to an API that required the use of OAuth signatures on each API request. Thankfully there are still a number of convenient and free OAuth libraries so you can avoid writing an OAuth implementation from scratch by reading the specification. These libraries are of varying complexity and have varying degrees of richness. The OAuth web site has information about various OAuth libraries.

For this next sample program we will download the files *twurl.py*, *hidden.py*, *oauth.py*, and *twitter1.py* from www.pythonlearn.com/code and put them all in a folder on your computer.

To make use of these programs you will need to have a Twitter account, and authorize your Python code as an application, set up a key, secret, token and token secret. You will edit the file *hidden.py* and put these four strings into the appropriate variables in the file:

```
# Keep this file separate
```

```
# https://apps.twitter.com/
# Create new App


def oauth():
    return {"consumer_key": "h7Lu...Ng",
            "consumer_secret": "dNKenAC3New...mmn7Q",
            "token_key": "10185562-eibxCp9n2...P4GEQQOSGI",
            "token_secret": "H0ycCFemmC4wyf1...qoIpBo"}

# Code: http://www.pythonlearn.com/code3/hidden.py
```

The Twitter web service are accessed using a URL like this:

https://api.twitter.com/1.1/statuses/user_timeline.json

But once all of the security information has been added, the URL will look more like:

```
https://api.twitter.com/1.1/statuses/user_timeline.json?count=2
&oauth_version=1.0&oauth_token=101...SGI&screen_name=drchuck
&oauth_nonce=09239679&oauth_timestamp=1380395644
&oauth_signature=rLK...BoD&oauth_consumer_key=h7Lu...GNg
&oauth_signature_method=HMAC-SHA1
```

You can read the OAuth specification if you want to know more about the meaning of the various parameters that are added to meet the security requirements of OAuth.

For the programs we run with Twitter, we hide all the complexity in the files *oauth.py* and *twurl.py*. We simply set the secrets in *hidden.py* and then send the desired URL to the *twurl.augment()* function and the library code adds all the necessary parameters to the URL for us.

This program retrieves the timeline for a particular Twitter user and returns it to us in JSON format in a string. We simply print the first 250 characters of the string:

```
import urllib.request, urllib.parse, urllib.error
import twurl

TWITTER_URL = 'https://api.twitter.com/1.1/statuses/user_timeline.json'

while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                        {'screen_name': acct, 'count': '2'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url)
    data = connection.read().decode()
```

```
    print(data[:250])
    headers = dict(connection.getheaders())
    # print headers
    print('Remaining', headers['x-rate-limit-remaining'])

# Code: http://www.pythonlearn.com/code3/twitter1.py
```

When the program runs it produces the following output:

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 17:30:25 +0000 2013","
id":384007200990982144,"id_str":"384007200990982144",
"text":"RT @fixpert: See how the Dutch handle traffic
intersections: http:\/\/t.co\/tIiVWtEhj4\n#brilliant",
"source":"web","truncated":false,"in_rep
Remaining 178

Enter Twitter Account:fixpert
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 18:03:56 +0000 2013",
"id":384015634108919808,"id_str":"384015634108919808",
"text":"3 months after my freak bocce ball accident,
my wedding ring fits again! :)\n\nhttps:\/\/t.co\/2XmHPx7kgX",
"source":"web","truncated":false,
Remaining 177

Enter Twitter Account:
```

Along with the returned timeline data, Twitter also returns metadata about the request in the HTTP response headers. One header in particular, *x-rate-limit-remaining*, informs us how many more requests we can make before we will be shut off for a short time period. You can see that our remaining retrievals drop by one each time we make a request to the API.

In the following example, we retrieve a user's Twitter friends, parse the returned JSON, and extract some of the information about the friends. We also dump the JSON after parsing and "pretty-print" it with an indent of four characters to allow us to pore through the data when we want to extract more fields.

```
import urllib.request, urllib.parse, urllib.error
import twurl
import json

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                        {'screen_name': acct, 'count': '5'})
```

```
    print('Retrieving', url)
    connection = urllib.request.urlopen(url)
    data = connection.read().decode()
    headers = dict(connection.getheaders())
    print('Remaining', headers['x-rate-limit-remaining'])
    js = json.loads(data)
    print(json.dumps(js, indent=4))

    for u in js['users']:
        print(u['screen_name'])
        s = u['status']['text']
        print('  ', s[:50])

# Code: http://www.pythonlearn.com/code3/twitter2.py
```

Since the JSON becomes a set of nested Python lists and dictionaries, we can use a combination of the index operation and `for` loops to wander through the returned data structures with very little Python code.

The output of the program looks as follows (some of the data items are shortened to fit on the page):

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/friends ...
Remaining 14


{
    "next_cursor": 1444171224491980205,
    "users": [
        {
            "id": 662433,
            "followers_count": 28725,
            "status": {
                "text": "@jazzychad I just bought one .__.",
                "created_at": "Fri Sep 20 08:36:34 +0000 2013",
                "retweeted": false,
            },
            "location": "San Francisco, California",
            "screen_name": "leahculver",
            "name": "Leah Culver",
        },
        {
            "id": 40426722,
            "followers_count": 2635,
            "status": {
                "text": "RT @WSJ: Big employers like Google ...",
                "created_at": "Sat Sep 28 19:36:37 +0000 2013",
            },
            "location": "Victoria Canada",
            "screen_name": "_valeriei",
            "name": "Valerie Irvine",
```

```
    ],
    "next_cursor_str": "1444171224491980205"
}
```

```
leahculver
   @jazzychad I just bought one .__.
_valeriei
   RT @WSJ: Big employers like Google, AT&amp;amp;T are h
ericbollens
   RT @lukew: sneak peek: my LONG take on the good &a
halherzog
   Learning Objects is 10. We had a cake with the LO,
scweeker
   @DeviceLabDC love it! Now where so I get that "etc

Enter Twitter Account:
```

The last bit of the output is where we see the for loop reading the five most recent "friends" of the *drchuck* Twitter account and printing the most recent status for each friend. There is a great deal more data available in the returned JSON. If you look in the output of the program, you can also see that the "find the friends" of a particular account has a different rate limitation than the number of timeline queries we are allowed to run per time period.

These secure API keys allow Twitter to have solid confidence that they know who is using their API and data and at what level. The rate-limiting approach allows us to do simple, personal data retrievals but does not allow us to build a product that pulls data from their API millions of times per day.

13.9 Glossary

API Application Program Interface - A contract between applications that defines the patterns of interaction between two application components.

ElementTree A built-in Python library used to parse XML data.

JSON JavaScript Object Notation. A format that allows for the markup of structured data based on the syntax of JavaScript Objects.

SOA Service-Oriented Architecture. When an application is made of components connected across a network.

XML eXtensible Markup Language. A format that allows for the markup of structured data.

13.10 Exercises

Exercise 1: Change either the www.pythonlearn.com/code3/geojson.py or www.pythonlearn.com/code3/geoxml.py to print out the two-character country code from the retrieved data. Add error checking so your program does not traceback if the country code is not there. Once you have it working, search for "Atlantic Ocean" and make sure it can handle locations that are not in any country.

Chapter 14

Object-Oriented Programming

14.1 Managing Larger Programs

At the beginning of this book, we came up with four basic programming patterns which we use to construct programs:

- Sequential code
- Conditional code (if statements)
- Repetitive code (loops)
- Store and reuse (functions)

In later chapters, we explored simple variables as well as collection data structures like lists, tuples, and dictionaries.

As we build programs, we design data structures and write code to manipulate those data structures. There are many ways to write programs and by now, you probably have written some programs that are "not so elegant" and other programs that are "more elegant". Even though your programs may be small, you are starting to see how there is a bit of "art" and "aesthetic" to writing code.

As programs get to be millions of lines long, it becomes increasingly important to write code that is easy to understand. If you are working on a million line program, you can never keep the entire program in your mind at the same time. So we need ways to break the program into multiple smaller pieces so to solve a problem, fix a bug, or add a new feature we have less to look at.

In a way, object oriented programming is a way to arrange your code so that you can zoom into 500 lines of the code, and understand it while ignoring the other 999,500 lines of code for the moment.

14.2 Getting Started

Like many aspects of programming it is necessary to learn the concepts of object oriented programming before you can use them effectively. So approach this chapter as a way to learn some terms and concepts and work through a few simple examples to lay a foundation for future learning. Throughout the rest of the book we will be using objects in many of the programs but we won't be building our own new objects in the programs.

The key outcome of this chapter is to have a basic understanding of how objects are constructed and how they function and most importantly how we make use of the capabilities of objects that are provided to us by Python and Python libraries.

14.3 Using Objects

It turns out we have been using objects all along in this class. Python provides us with many built-in objects. Here is some simple code where the first few lines should feel very simple and natural to you.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])

print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Code: http://www.pythonlearn.com/code3/party1.py
```

But instead of focusing on what these lines accomplish, lets look at what is really happening from the point of view of object-oriented programming. Don't worry if the following paragraphs don't make any sense the first time you read them because we have not yet defined all these terms.

The first line is *constructing* an object of type *list*, the second and third lines are calling the `append()` *method*, the fourth line is calling the `sort()` *method*, and the fifth line is retrieving the item at position 0.

The sixth line is calling the `__getitem__()` method in the `stuff` list with a parameter of zero.

```
print (stuff.__getitem__(0))
```

The seventh line is an even more verbose way of retrieving the 0th item in the list.

```
print (list.__getitem__(stuff,0))
```

In this code, we care calling the `__getitem__` method in the `list` class and passing in the list (`stuff`) and the item we want retrieved from the list as parameters.

The last three lines of the program are completely equivalent, but it is more convienent to simply use the square bracket syntax to look up an item at a particular position in a list.

We can take a look into the capbilities of an object by looking at the output of the `dir()` function:

```
>>> stuff = list()
>>> dir(stuff)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

The precise definition of `dir()` is that it lists the *methods* and *attributes* of a Python object.

The rest of this chapter will define all of the above terms so make sure to come back after you finish the chapter and re-read the above paragraphs to check your understanding.

14.4 Starting with Programs

A program in its most basic form takes some input, does some processing, and produces some output. Our elevator conversion program demonstrates a very short but complete program showing all three of these steps.

```
usf = input('Enter the US Floor Number: ')
wf = int(usf) - 1
print('Non-US Floor Number is',wf)

# Code: http://www.pythonlearn.com/code3/elev.py
```

If we think a bit more about this program, there is the "outside world" and the program. The input and output aspects are where the program interacts with the outside world. Within the program we have code and data to accomplish the task the program is designed to solve.

When we are "in" the program, we have some defined interactions with the "outside" world, but those interactions are well defined and generally not something we focus on. While we are coding we worry only about the details "inside the program".

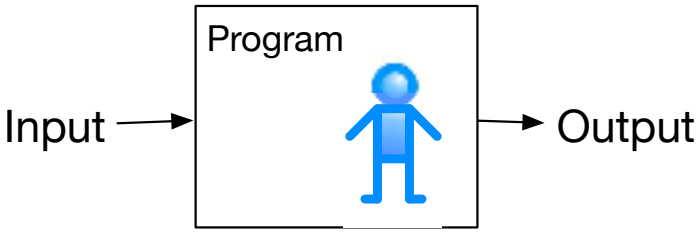

Figure 14.1: A Program

One way to think about object oriented programming is that we are separating our program into multiple "zones". Each "zone" contains some code and data (like a program) and has well defined interactions with the outside world and the other zones within the program.

If we look back at the link extraction application where we used the BeautifulSoup library, we can see a program that is constructed by connecting different objects together to accomplish a task:

```
# To run this, you can install BeautifulSoup
# https://pypi.python.org/pypi/beautifulsoup4

# Or download the file
# http://www.pythonlearn.com/code3/bs4.zip
# and unzip it in the same directory as this file

import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup

url = input('Enter - ')
html = urllib.request.urlopen(url).read()
soup = BeautifulSoup(html, 'html.parser')

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

# Code: http://www.pythonlearn.com/code3/urllinks.py
```

We read the URL into a string, and then pass that into `urllib` to retrieve the data from the web. The `urllib` library uses the `socket` library to make the actual network connection to retrieve the data. We take the string that we get back from `urllib` and hand it to BeautifulSoup for parsing. BeautifulSoup makes use of another object called `html.parser`[1] and returns an object. We call the `tags()` method in the returned object and then get a dictionary of tag objects, and loop through the tags and call the `get()` method for each tag to print out the 'href' attribute.

[1] https://docs.python.org/3/library/html.parser.html

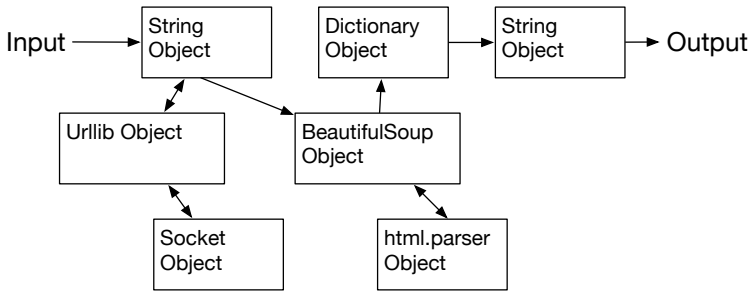

Figure 14.2: A Program as Network of Objects

We can draw a picture of this program and how the objects work together.

The key here is not to fully understand how this program works but to see how we build a network of interacting objects and orchestrate the movement of information between the objects to create a program. It is also important to note that when you looked at that program several chapters back, you could fully understand what was going on in the program without even realizing that the program was "orchestrating the movement of data between objects". Back then it was just lines of code that got the job done.

14.5 Subdividing a Problem - Encapsulation

One of the advantages of the object oriented approach is that it can hide complexity. For example, while we need to know how to use the `urllib` and BeautifulSoup code, we do not need to know how those libraries work internally. It allows us to focus on the part of the problem we need to solve and ignore the other parts of the program.

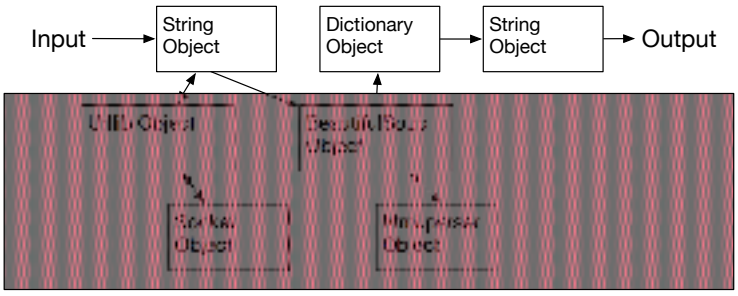

Figure 14.3: Ignoring Detail When Using an Object

This ability to focus on a part of a program that we care about and ignore the rest of the program is also helpful to the developers of the objects. For example the programmers developing BeautifulSoup do not need to know or care about how we retrieve our HTML page, what parts we want to read or what we plan to do with the data we extract from the web page.

Another word we use to capture this idea that we ignore the internal detail of objects we use is "encapsulation". This means that we can know how to use an

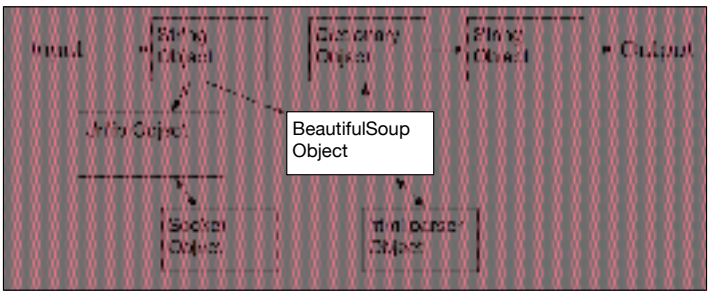

Figure 14.4: Ignoring Detail When Building an Object

object without knowing how it internally acomplishes what we need done.

14.6 Our First Python Object

At its simplest, an object is some code plus data structures that is smaller than a whole program. Defining a function allows us to store a bit of code and give it a name and then later invoke that code using the name of the function.

An object can contain a number of functions (which we call "methods") as well as data that is used by those functions. We call data items that are part of the object "attributes".

We use the `class` keyword to define the data and code that will make up each of the objects. The class keyword includes the name of the class and begins an indented block of code where we include the attributes (data) and methods (code).

```
class PartyAnimal:
   x = 0

   def party(self) :
     self.x = self.x + 1
     print("So far",self.x)

an = PartyAnimal()
an.party()
an.party()
an.party()
PartyAnimal.party(an)

# Code: http://www.pythonlearn.com/code3/party2.py
```

Each method looks like a function, starting with the `def` keyword and consisting of an indended block of code. This example has one attribute (x) and one method (party). The methods have a special first parameter that we name by convention `self`.

Much like the `def` keyword does not cause function code to be executed, the `class` keyword does not create an object. Instead, the `class` keyword defines a

template indicating what data and code will be contained in each object of type `PartyAnimal`. The class is like a cookie cutter and the objects created using the class are the cookies[2]. You don't put frosting on the cookie cutter, you put frosting on the cookies - and you can put different frosting on each cookie.

Figure 14.5: A Class and Two Objects

If you continue through the example code, we see the first executable line of code:

```
an = PartyAnimal()
```

This is where we instruct Python to construct (e.g. create) an *object* or "instance of the class named PartyAnimal". It looks like a function call to the class itself and Python constructs the object with the right data and methods and returns the object which is then assigned to the variable `an`. In a way this is quite similar to the following line which we have been using all along:

```
counts = dict()
```

Here we are instructing Python to construct an object using the `dict` template (already present in Python), return the instance of dictionary and assign it to the variable `counts`.

When the PartyAnimal class is used to construct an object, the variable `an` is used to point to that object. We use `an` to access the code and data for that particular instance of a PartyAnimal object.

Each Partyanimal object/instance contains within it a variable `x` and a method/function named `party`. We call that `party` method in this line:

```
an.party()
```

When the `party` method is called, the first parameter (which we call by convention `self`) points to the particular instance of the PartyAnimal object that `party` is called from within. Within the `party` method, we see the line:

[2] Cookie image copyright CC-BY https://www.flickr.com/photos/dinnerseries/23570475099

```
self.x = self.x + 1
```

This syntax using the 'dot' operator is saying 'the x within self'. So each time `party()` is called, the internal `x` value is incremented by 1 and the value is printed out.

To help make sense of the difference between a global function and a method within a class/object, the following line is another way to call the `party` method within the `an` object:

```
PartyAnimal.party(an)
```

In this variation, we are accessing the code from within the *class* and explicitly passing the object pointer `an` in as the first parameter (i.e. `self` within the method). You can think of `an.party()` as shorthand for the above line.

When the program executes, it produces the following output:

```
So far 1
So far 2
So far 3
So far 4
```

The object is constructed, and the `party` method is called four times, both incrementing and printing the value for `x` within the `an` object.

14.7 Classes as Types

As we have seen, in Python, all variables have a type. And we can use the built-in `dir` function to examine the capabilities of a variable. We can use `type` and `dir` with the classes that we create.

```
class PartyAnimal:
   x = 0

   def party(self) :
     self.x = self.x + 1
     print("So far",self.x)

an = PartyAnimal()
print ("Type", type(an))
print ("Dir ", dir(an))
print ("Type", type(an.x))
print ("Type", type(an.party))

# Code: http://www.pythonlearn.com/code3/party3.py
```

When this program executes, it produces the following output:

```
Type <class '__main__.PartyAnimal'>
Dir  ['__class__', '__delattr__', ...
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'party', 'x']
Type <class 'int'>
Type <class 'method'>
```

You can see that using the `class` keyword, we have created a new type. From the `dir` output, you can see both the `x` integer attribute and the `party` method are available in the object.

14.8 Object Lifecycle

In the previous examples, we are defining a class (template) and using that class to create an instance of that class (object) and then using the instance. When the program finishes, all the varaibles are discarded. Usually we don't think much about the creation and destruction of variables, but often as our objects become more complex, we need to take some action within the object to set things up as the object is being constructed and possibly clean things up as the object is being discarded.

If we want our object to be aware of these moments of construction and destruction, we add specially named methods to our object:

```
class PartyAnimal:
   x = 0

   def __init__(self):
     print('I am constructed')

   def party(self) :
     self.x = self.x + 1
     print('So far',self.x)

   def __del__(self):
     print('I am destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)

# Code: http://www.pythonlearn.com/code3/party4.py
```

When this program executes, it produces the following output:

```
I am constructed
So far 1
```

```
So far 2
I am destructed 2
an contains 42
```

As Python is constructing our object, it calls our `__init__` method to give us a chance to set up some default or initial values for the object. When Python encounters the line:

```
an = 42
```

It actually 'thows our object away' so it can reuse the `an` variable to store the value `42`. Just at the moment when our `an` object is being 'destroyed' our destructor code (`__del__`) is called. We cannot stop our variable from being destroyed, but we can do any necessary cleanup right before our object no longer exists.

When developing objects, it is quite common to add a constructor to an object to set in intial values in the object, it is relatively rare to need to need a destructor for an object.

14.9 Many Instances

So far, we have been defining a class, making a single object, using that object, and then throwing the object away. But the real power in object oriented happens when we make many instances of our class.

When we are making multiple objects from our class, we might want to set up different initial values for each of the objects. We can pass data into the constructors to give each object a different initial value:

```
class PartyAnimal:
   x = 0
   name = ''
   def __init__(self, nam):
     self.name = nam
     print(self.name,'constructed')

   def party(self) :
     self.x = self.x + 1
     print(self.name,'party count',self.x)

s = PartyAnimal('Sally')
s.party()
j = PartyAnimal('Jim')
j.party()
s.party()
```

 # Code: http://www.pythonlearn.com/code3/party5.py

The constructor has both a `self` parameter that points to the object instance and then additional parameters that are passed into the constructor as the object is being constructed:

```
s = PartyAnimal('Sally')
```

Within the constructor, the line:

```
self.name = nam
```

Copies the parameter that is passed in (`nam`) into the `name` attribute within the object instance.

The output of the program shows that each of the objects (`s` and `j`) contain their own independent copies of `x` and `nam`:

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Sally party count 2
```

14.10 Inheritance

Another powerful feature of object oriented programming is the ability to create a new class by extending an existing class. When extending a class, we call the original class the 'parent class' and the new class as the 'child class'.

For this example, we will move our `PartyAnimal` class into its own file:

```
class PartyAnimal:
   x = 0
   name = ''
   def __init__(self, nam):
     self.name = nam
     print(self.name,'constructed')

   def party(self) :
     self.x = self.x + 1
     print(self.name,'party count',self.x)

# Code: http://www.pythonlearn.com/code3/party.py
```

Then, we can 'import' the `PartyAnimal` class in a new file and extend it as follows:

```
from party import PartyAnimal

class CricketFan(PartyAnimal):
   points = 0
   def six(self):
      self.points = self.points + 6
      self.party()
      print(self.name,"points",self.points)
```

```
s = PartyAnimal("Sally")
s.party()
j = CricketFan("Jim")
j.party()
j.six()
print(dir(j))

# Code: http://www.pythonlearn.com/code3/party6.py
```

When we are defining the `CricketFan` object, we indicate that we are extending the `PartyAnimal` class. This means that all of the variables (`x`) and methods (`party`) from the `PartyAnimal` class are inherited by the `CricketFan` class.

You can see that within the `six` method in the `CricketFan` class, we can call the `party` method from the `PartyAnimal` class. The variables and methods from the parent class are *merged* into the child class.

As the program executes, we can see that the `s` and `j` are independent instances of `PartyAnimal` and `CricketFan`. The `j` object has additional capabilities beyond the `s` object.

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Jim party count 2
Jim points 6
['__class__', '__delattr__', ... '__weakref__',
'name', 'party', 'points', 'six', 'x']
```

In the `dir` output for the `j` object (instance of the `CricketFan` class) you can see that it both has the attributes and methods of the parent class as well as the attributes and methods that were added when the class was extended to create the `CricketFan` class.

14.11 Summary

This is a very quick introduction to object-oriented programming that focuses mainly on terminology and the syntax of defining and using objects. Let's quickly review the code that we looked at in the beginning of the chapter. At this point you should fully understand what is going on.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])

print (stuff.__getitem__(0))
```

```
print (list.__getitem__(stuff,0))

# Code: http://www.pythonlearn.com/code3/party1.py
```

The first line constructs a `list` *object*. When Python creates the `list` object, it calls the *constructor* method (named `__init__`) to set up the internal data attributes that will be used to store the list data. Due to *encapsulation* we neither need to know, nor need to care about these in internal data attributes are arranged.

We are not passing any parameters to the *constructor* and when the constructor returns, we use the variable `stuff` to point to the returned instance of the `list` class.

The second and third lines are calling the `append` method with one parameter to add a new item at the end of the list by updating the attributes within `stuff`. Then in the fourth line, we call the `sort` method with no parameters to sort the data within the `stuff` object.

Then we print out the first item in the list using the square brackets which are a shortcut to calling the `__getitem__` method within the `stuff` *object*. And this is equivalent to calling the `__getitem__` method in the `list` *class* passing the `stuff` object in as the first parameter and the position we are looking for as the second parameter.

At the end of the program the `stuff` object is discarded but not before calling the *destructor* (named `__del__`) so the object can clean up any loose ends as necessary.

Those are the basics and terminology of object oriented programming. There are many additional details as to how to best use object oriented approaches when developing large applications and libraries that are beyond the scope of this chapter.[3]

14.12 Glossary

attribute A variable that is part of a class.

class A template that can be used to construct an object. Defines the attrbutes and methods that will make up the object.

child class A new class created when a parent class is extended. The child class inherits all of the attributes and methods of the parent class.

constructor An optional specially named method (`__init__`) that is called at the moment when a class is being used to construct an object. Usually this is used to set up initial values for the object.

[3] If you are curious about where the list class is defined, take a look at (hopefully the URL won't change) https://github.com/python/cpython/blob/master/Objects/listobject.c - the list class is written in a language called "C". If you take a look at that source code and find it curious you might want to explore a few Computer Science courses.

destructor An optional specially named method (`__del__`) that is called at the moment just before an object is destroyed. Destructors are rarely used.

inheritance When we create a new class (child) by extending an existing class (parent). The child class has all the attributes and methods of the parent class plus additional asttributes and methods defined by the child class.

method A function that is contained within a class and the objects that are constructed from the class. Some object-oriented patterns use 'message' instead of 'method' to describe this concept.

object A constructed instance of a class. An object contains all of the attributes and methods that were defined by the class. Some object-oriented documentation uses the term 'instance' interchangably with 'object'.

parent class The class which is being extended to create a new chld class. The parent class contributes all of its methods and attributes to the new chaild class.

Chapter 15

Using databases and SQL

15.1 What is a database?

A *database* is a file that is organized for storing data. Most databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference is that the database is on disk (or other permanent storage), so it persists after the program ends. Because a database is stored on permanent storage, it can store far more data than a dictionary, which is limited to the size of the memory in the computer.

Like a dictionary, database software is designed to keep the inserting and accessing of data very fast, even for large amounts of data. Database software maintains its performance by building *indexes* as data is added to the database to allow the computer to jump quickly to a particular entry.

There are many different database systems which are used for a wide variety of purposes including: Oracle, MySQL, Microsoft SQL Server, PostgreSQL, and SQLite. We focus on SQLite in this book because it is a very common database and is already built into Python. SQLite is designed to be *embedded* into other applications to provide database support within the application. For example, the Firefox browser also uses the SQLite database internally as do many other products.

http://sqlite.org/

SQLite is well suited to some of the data manipulation problems that we see in Informatics such as the Twitter spidering application that we describe in this chapter.

15.2 Database concepts

When you first look at a database it looks like a spreadsheet with multiple sheets. The primary data structures in a database are: *tables*, *rows*, and *columns*.

In technical descriptions of relational databases the concepts of table, row, and column are more formally referred to as *relation*, *tuple*, and *attribute*, respectively. We will use the less formal terms in this chapter.

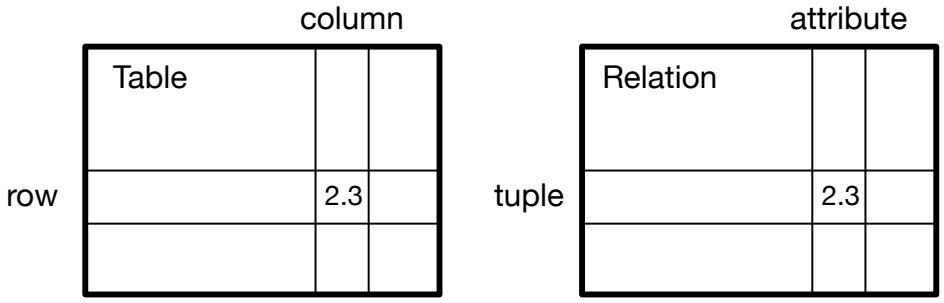

Figure 15.1: Relational Databases

15.3 Database Browser for SQLite

While this chapter will focus on using Python to work with data in SQLite database files, many operations can be done more conveniently using software called the *Database Browser for SQLite* which is freely available from:

http://sqlitebrowser.org/

Using the browser you can easily create tables, insert data, edit data, or run simple SQL queries on the data in the database.

In a sense, the database browser is similar to a text editor when working with text files. When you want to do one or very few operations on a text file, you can just open it in a text editor and make the changes you want. When you have many changes that you need to do to a text file, often you will write a simple Python program. You will find the same pattern when working with databases. You will do simple operations in the database manager and more complex operations will be most conveniently done in Python.

15.4 Creating a database table

Databases require more defined structure than Python lists or dictionaries[1].

When we create a database *table* we must tell the database in advance the names of each of the *columns* in the table and the type of data which we are planning to store in each *column*. When the database software knows the type of data in each column, it can choose the most efficient way to store and look up the data based on the type of data.

You can look at the various data types supported by SQLite at the following url:

http://www.sqlite.org/datatypes.html

Defining structure for your data up front may seem inconvenient at the beginning, but the payoff is fast access to your data even when the database contains a large amount of data.

[1] SQLite actually does allow some flexibility in the type of data stored in a column, but we will keep our data types strict in this chapter so the concepts apply equally to other database systems such as MySQL.

The code to create a database file and a table named `Tracks` with two columns in the database is as follows:

```
import sqlite3

conn = sqlite3.connect('music.sqlite')
cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Tracks')
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')

conn.close()

# Code: http://www.pythonlearn.com/code3/db1.py
```

The `connect` operation makes a "connection" to the database stored in the file `music.sqlite3` in the current directory. If the file does not exist, it will be created. The reason this is called a "connection" is that sometimes the database is stored on a separate "database server" from the server on which we are running our application. In our simple examples the database will just be a local file in the same directory as the Python code we are running.

A *cursor* is like a file handle that we can use to perform operations on the data stored in the database. Calling `cursor()` is very similar conceptually to calling `open()` when dealing with text files.

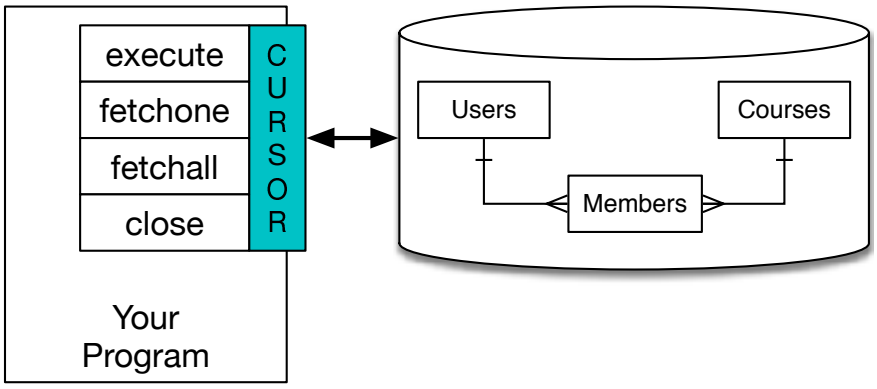

Figure 15.2: A Database Cursor

Once we have the cursor, we can begin to execute commands on the contents of the database using the `execute()` method.

Database commands are expressed in a special language that has been standardized across many different database vendors to allow us to learn a single database language. The database language is called *Structured Query Language* or *SQL* for short.

http://en.wikipedia.org/wiki/SQL

In our example, we are executing two SQL commands in our database. As a convention, we will show the SQL keywords in uppercase and the parts of the

command that we are adding (such as the table and column names) will be shown in lowercase.

The first SQL command removes the `Tracks` table from the database if it exists. This pattern is simply to allow us to run the same program to create the `Tracks` table over and over again without causing an error. Note that the `DROP TABLE` command deletes the table and all of its contents from the database (i.e., there is no "undo").

```
cur.execute('DROP TABLE IF EXISTS Tracks ')
```

The second command creates a table named `Tracks` with a text column named `title` and an integer column named `plays`.

```
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')
```

Now that we have created a table named `Tracks`, we can put some data into that table using the SQL `INSERT` operation. Again, we begin by making a connection to the database and obtaining the `cursor`. We can then execute SQL commands using the cursor.

The SQL `INSERT` command indicates which table we are using and then defines a new row by listing the fields we want to include `(title, plays)` followed by the `VALUES` we want placed in the new row. We specify the values as question marks `(?, ?)` to indicate that the actual values are passed in as a tuple `( 'My Way', 15 )` as the second parameter to the `execute()` call.

```
import sqlite3

conn = sqlite3.connect('music.sqlite')
cur = conn.cursor()

cur.execute('INSERT INTO Tracks (title, plays) VALUES (?, ?)',
    ('Thunderstruck', 20))
cur.execute('INSERT INTO Tracks (title, plays) VALUES (?, ?)',
    ('My Way', 15))
conn.commit()

print('Tracks:')
cur.execute('SELECT title, plays FROM Tracks')
for row in cur:
     print(row)

cur.execute('DELETE FROM Tracks WHERE plays < 100')

cur.close()

# Code: http://www.pythonlearn.com/code3/db2.py
```

First we `INSERT` two rows into our table and use `commit()` to force the data to be written to the database file.

Tracks

title	plays
Thunderstruck	20
My Way	15

Figure 15.3: Rows in a Table

Then we use the `SELECT` command to retrieve the rows we just inserted from the table. On the `SELECT` command, we indicate which columns we would like `(title, plays)` and indicate which table we want to retrieve the data from. After we execute the `SELECT` statement, the cursor is something we can loop through in a `for` statement. For efficiency, the cursor does not read all of the data from the database when we execute the `SELECT` statement. Instead, the data is read on demand as we loop through the rows in the `for` statement.

The output of the program is as follows:

```
Tracks:
('Thunderstruck', 20)
('My Way', 15)
```

Our `for` loop finds two rows, and each row is a Python tuple with the first value as the `title` and the second value as the number of `plays`.

Note: You may see strings starting with `u'` *in other books or on the Internet. This was an indication in Python 2 that the strings are* Unicode* strings that are capable of storing non-Latin character sets. In Python 3, all strings are unicode strings by default.*

At the very end of the program, we execute an SQL command to `DELETE` the rows we have just created so we can run the program over and over. The `DELETE` command shows the use of a `WHERE` clause that allows us to express a selection criterion so that we can ask the database to apply the command to only the rows that match the criterion. In this example the criterion happens to apply to all the rows so we empty the table out so we can run the program repeatedly. After the `DELETE` is performed, we also call `commit()` to force the data to be removed from the database.

15.5 Structured Query Language summary

So far, we have been using the Structured Query Language in our Python examples and have covered many of the basics of the SQL commands. In this section, we look at the SQL language in particular and give an overview of SQL syntax.

Since there are so many different database vendors, the Structured Query Language (SQL) was standardized so we could communicate in a portable manner to database systems from multiple vendors.

A relational database is made up of tables, rows, and columns. The columns generally have a type such as text, numeric, or date data. When we create a table, we indicate the names and types of the columns:

```
CREATE TABLE Tracks (title TEXT, plays INTEGER)
```

To insert a row into a table, we use the SQL INSERT command:

```
INSERT INTO Tracks (title, plays) VALUES ('My Way', 15)
```

The INSERT statement specifies the table name, then a list of the fields/columns that you would like to set in the new row, and then the keyword VALUES and a list of corresponding values for each of the fields.

The SQL SELECT command is used to retrieve rows and columns from a database. The SELECT statement lets you specify which columns you would like to retrieve as well as a WHERE clause to select which rows you would like to see. It also allows an optional ORDER BY clause to control the sorting of the returned rows.

```
SELECT * FROM Tracks WHERE title = 'My Way'
```

Using * indicates that you want the database to return all of the columns for each row that matches the WHERE clause.

Note, unlike in Python, in a SQL WHERE clause we use a single equal sign to indicate a test for equality rather than a double equal sign. Other logical operations allowed in a WHERE clause include <, >, <=, >=, !=, as well as AND and OR and parentheses to build your logical expressions.

You can request that the returned rows be sorted by one of the fields as follows:

```
SELECT title,plays FROM Tracks ORDER BY title
```

To remove a row, you need a WHERE clause on an SQL DELETE statement. The WHERE clause determines which rows are to be deleted:

```
DELETE FROM Tracks WHERE title = 'My Way'
```

It is possible to UPDATE a column or columns within one or more rows in a table using the SQL UPDATE statement as follows:

```
UPDATE Tracks SET plays = 16 WHERE title = 'My Way'
```

The UPDATE statement specifies a table and then a list of fields and values to change after the SET keyword and then an optional WHERE clause to select the rows that are to be updated. A single UPDATE statement will change all of the rows that match the WHERE clause. If a WHERE clause is not specified, it performs the UPDATE on all of the rows in the table.

These four basic SQL commands (INSERT, SELECT, UPDATE, and DELETE) allow the four basic operations needed to create and maintain data.

15.6 Spidering Twitter using a database

In this section, we will create a simple spidering program that will go through Twitter accounts and build a database of them. *Note: Be very careful when running this program. You do not want to pull too much data or run the program for too long and end up having your Twitter access shut off.*

One of the problems of any kind of spidering program is that it needs to be able to be stopped and restarted many times and you do not want to lose the data that you have retrieved so far. You don't want to always restart your data retrieval at the very beginning so we want to store data as we retrieve it so our program can start back up and pick up where it left off.

We will start by retrieving one person's Twitter friends and their statuses, looping through the list of friends, and adding each of the friends to a database to be retrieved in the future. After we process one person's Twitter friends, we check in our database and retrieve one of the friends of the friend. We do this over and over, picking an "unvisited" person, retrieving their friend list, and adding friends we have not seen to our list for a future visit.

We also track how many times we have seen a particular friend in the database to get some sense of their "popularity".

By storing our list of known accounts and whether we have retrieved the account or not, and how popular the account is in a database on the disk of the computer, we can stop and restart our program as many times as we like.

This program is a bit complex. It is based on the code from the exercise earlier in the book that uses the Twitter API.

Here is the source code for our Twitter spidering application:

```
from urllib.request import urlopen
import urllib.error
import twurl
import json
import sqlite3

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()

cur.execute('''
            CREATE TABLE IF NOT EXISTS Twitter
            (name TEXT, retrieved INTEGER, friends INTEGER)''')

while True:
    acct = input('Enter a Twitter account, or quit: ')
    if (acct == 'quit'): break
    if (len(acct) < 1):
        cur.execute('SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
        try:
```

```
            acct = cur.fetchone()[0]
        except:
            print('No unretrieved Twitter accounts found')
            continue

    url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '5'})
    print('Retrieving', url)
    connection = urlopen(url)
    data = connection.read().decode()
    headers = dict(connection.getheaders())

    print('Remaining', headers['x-rate-limit-remaining'])
    js = json.loads(data)
    # Debugging
    # print json.dumps(js, indent=4)

    cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?', (acct, ))

    countnew = 0
    countold = 0
    for u in js['users']:
        friend = u['screen_name']
        print(friend)
        cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
                    (friend, ))
        try:
            count = cur.fetchone()[0]
            cur.execute('UPDATE Twitter SET friends = ? WHERE name = ?',
                        (count+1, friend))
            countold = countold + 1
        except:
            cur.execute('''INSERT INTO Twitter (name, retrieved, friends)
                        VALUES (?, 0, 1)''', (friend, ))
            countnew = countnew + 1
    print('New accounts=', countnew, ' revisited=', countold)
    conn.commit()

cur.close()

# Code: http://www.pythonlearn.com/code3/twspider.py
```

Our database is stored in the file `spider.sqlite3` and it has one table named `Twitter`. Each row in the `Twitter` table has a column for the account name, whether we have retrieved the friends of this account, and how many times this account has been "friended".

In the main loop of the program, we prompt the user for a Twitter account name or "quit" to exit the program. If the user enters a Twitter account, we retrieve the list of friends and statuses for that user and add each friend to the database if not already in the database. If the friend is already in the list, we add 1 to the `friends` field in the row in the database.

If the user presses enter, we look in the database for the next Twitter account that we have not yet retrieved, retrieve the friends and statuses for that account, add them to the database or update them, and increase their `friends` count.

Once we retrieve the list of friends and statuses, we loop through all of the `user` items in the returned JSON and retrieve the `screen_name` for each user. Then we use the `SELECT` statement to see if we already have stored this particular `screen_name` in the database and retrieve the friend count (`friends`) if the record exists.

```
countnew = 0
countold = 0
for u in js['users'] :
    friend = u['screen_name']
    print friend
    cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
        (friend, ) )
    try:
        count = cur.fetchone()[0]
        cur.execute('UPDATE Twitter SET friends = ? WHERE name = ?',
            (count+1, friend) )
        countold = countold + 1
    except:
        cur.execute('''INSERT INTO Twitter (name, retrieved, friends)
            VALUES ( ?, 0, 1 )''', ( friend, ) )
        countnew = countnew + 1
print 'New accounts=',countnew,' revisited=',countold
conn.commit()
```

Once the cursor executes the `SELECT` statement, we must retrieve the rows. We could do this with a `for` statement, but since we are only retrieving one row (`LIMIT 1`), we can use the `fetchone()` method to fetch the first (and only) row that is the result of the `SELECT` operation. Since `fetchone()` returns the row as a *tuple* (even though there is only one field), we take the first value from the tuple using to get the current friend count into the variable `count`.

If this retrieval is successful, we use the SQL `UPDATE` statement with a `WHERE` clause to add 1 to the `friends` column for the row that matches the friend's account. Notice that there are two placeholders (i.e., question marks) in the SQL, and the second parameter to the `execute()` is a two-element tuple that holds the values to be substituted into the SQL in place of the question marks.

If the code in the `try` block fails, it is probably because no record matched the `WHERE name = ?` clause on the SELECT statement. So in the `except` block, we use the SQL `INSERT` statement to add the friend's `screen_name` to the table with an indication that we have not yet retrieved the `screen_name` and set the friend count to zero.

So the first time the program runs and we enter a Twitter account, the program runs as follows:

```
Enter a Twitter account, or quit: drchuck
```

```
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit: quit
```

Since this is the first time we have run the program, the database is empty and we create the database in the file `spider.sqlite3` and add a table named `Twitter` to the database. Then we retrieve some friends and add them all to the database since the database is empty.

At this point, we might want to write a simple database dumper to take a look at what is in our `spider.sqlite3` file:

```
import sqlite3

conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()
cur.execute('SELECT * FROM Twitter')
count = 0
for row in cur:
    print(row)
    count = count + 1
print(count, 'rows.')
cur.close()

# Code: http://www.pythonlearn.com/code3/twdump.py
```

This program simply opens the database and selects all of the columns of all of the rows in the table `Twitter`, then loops through the rows and prints out each row.

If we run this program after the first execution of our Twitter spider above, its output will be as follows:

```
('opencontent', 0, 1)
('lhawthorn', 0, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
20 rows.
```

We see one row for each `screen_name`, that we have not retrieved the data for that `screen_name`, and everyone in the database has one friend.

Now our database reflects the retrieval of the friends of our first Twitter account (*drchuck*). We can run the program again and tell it to retrieve the friends of the next "unprocessed" account by simply pressing enter instead of a Twitter account as follows:

```
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 18  revisited= 2
Enter a Twitter account, or quit:
```

```
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

Since we pressed enter (i.e., we did not specify a Twitter account), the following code is executed:

```
if ( len(acct) < 1 ) :
    cur.execute('SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
    try:
        acct = cur.fetchone()[0]
    except:
        print 'No unretrieved twitter accounts found'
        continue
```

We use the SQL `SELECT` statement to retrieve the name of the first (`LIMIT 1`) user who still has their "have we retrieved this user" value set to zero. We also use the `fetchone()[0]` pattern within a try/except block to either extract a `screen_name` from the retrieved data or put out an error message and loop back up.

If we successfully retrieved an unprocessed `screen_name`, we retrieve their data as follows:

~~ {.python{ url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '20'}) print 'Retrieving', url connection = urllib.urlopen(url) data = connection.read() js = json.loads(data)

cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?', (acct,)) ~~

Once we retrieve the data successfully, we use the `UPDATE` statement to set the `retrieved` column to 1 to indicate that we have completed the retrieval of the friends of this account. This keeps us from retrieving the same data over and over and keeps us progressing forward through the network of Twitter friends.

If we run the friend program and press enter twice to retrieve the next unvisited friend's friends, then run the dumping program, it will give us the following output:

```
('opencontent', 1, 1)
('lhawthorn', 1, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
('cnxorg', 0, 2)
('knoop', 0, 1)
('kthanos', 0, 2)
('LectureTools', 0, 1)
...
55 rows.
```

We can see that we have properly recorded that we have visited `lhawthorn` and `opencontent`. Also the accounts `cnxorg` and `kthanos` already have two followers. Since we now have retrieved the friends of three people (`drchuck`, `opencontent`, and `lhawthorn`) our table has 55 rows of friends to retrieve.

Each time we run the program and press enter it will pick the next unvisited account (e.g., the next account will be `steve_coppin`), retrieve their friends, mark them as retrieved, and for each of the friends of `steve_coppin` either add them to the end of the database or update their friend count if they are already in the database.

Since the program's data is all stored on disk in a database, the spidering activity can be suspended and resumed as many times as you like with no loss of data.

15.7 Basic data modeling

The real power of a relational database is when we create multiple tables and make links between those tables. The act of deciding how to break up your application data into multiple tables and establishing the relationships between the tables is called *data modeling*. The design document that shows the tables and their relationships is called a *data model*.

Data modeling is a relatively sophisticated skill and we will only introduce the most basic concepts of relational data modeling in this section. For more detail on data modeling you can start with:

http://en.wikipedia.org/wiki/Relational_model

Let's say for our Twitter spider application, instead of just counting a person's friends, we wanted to keep a list of all of the incoming relationships so we could find a list of everyone who is following a particular account.

Since everyone will potentially have many accounts that follow them, we cannot simply add a single column to our `Twitter` table. So we create a new table that keeps track of pairs of friends. The following is a simple way of making such a table:

```
CREATE TABLE Pals (from_friend TEXT, to_friend TEXT)
```

Each time we encounter a person who `drchuck` is following, we would insert a row of the form:

```
INSERT INTO Pals (from_friend,to_friend) VALUES ('drchuck', 'lhawthorn')
```

As we are processing the 20 friends from the `drchuck` Twitter feed, we will insert 20 records with "drchuck" as the first parameter so we will end up duplicating the string many times in the database.

This duplication of string data violates one of the best practices for *database normalization* which basically states that we should never put the same string data in the database more than once. If we need the data more than once, we create a numeric *key* for the data and reference the actual data using this key.

In practical terms, a string takes up a lot more space than an integer on the disk and in the memory of our computer, and takes more processor time to compare and sort. If we only have a few hundred entries, the storage and processor time hardly matters. But if we have a million people in our database and a possibility

of 100 million friend links, it is important to be able to scan data as quickly as possible.

We will store our Twitter accounts in a table named `People` instead of the `Twitter` table used in the previous example. The `People` table has an additional column to store the numeric key associated with the row for this Twitter user. SQLite has a feature that automatically adds the key value for any row we insert into a table using a special type of data column (`INTEGER PRIMARY KEY`).

We can create the `People` table with this additional `id` column as follows:

```
CREATE TABLE People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)
```

Notice that we are no longer maintaining a friend count in each row of the `People` table. When we select `INTEGER PRIMARY KEY` as the type of our `id` column, we are indicating that we would like SQLite to manage this column and assign a unique numeric key to each row we insert automatically. We also add the keyword `UNIQUE` to indicate that we will not allow SQLite to insert two rows with the same value for `name`.

Now instead of creating the table `Pals` above, we create a table called `Follows` with two integer columns `from_id` and `to_id` and a constraint on the table that the *combination* of `from_id` and `to_id` must be unique in this table (i.e., we cannot insert duplicate rows) in our database.

```
CREATE TABLE Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id) )
```

When we add `UNIQUE` clauses to our tables, we are communicating a set of rules that we are asking the database to enforce when we attempt to insert records. We are creating these rules as a convenience in our programs, as we will see in a moment. The rules both keep us from making mistakes and make it simpler to write some of our code.

In essence, in creating this `Follows` table, we are modelling a "relationship" where one person "follows" someone else and representing it with a pair of numbers indicating that (a) the people are connected and (b) the direction of the relationship.

15.8 Programming with multiple tables

We will now redo the Twitter spider program using two tables, the primary keys, and the key references as described above. Here is the code for the new version of the program:

```
import urllib.request, urllib.parse, urllib.error
import twurl
import json
import sqlite3
```

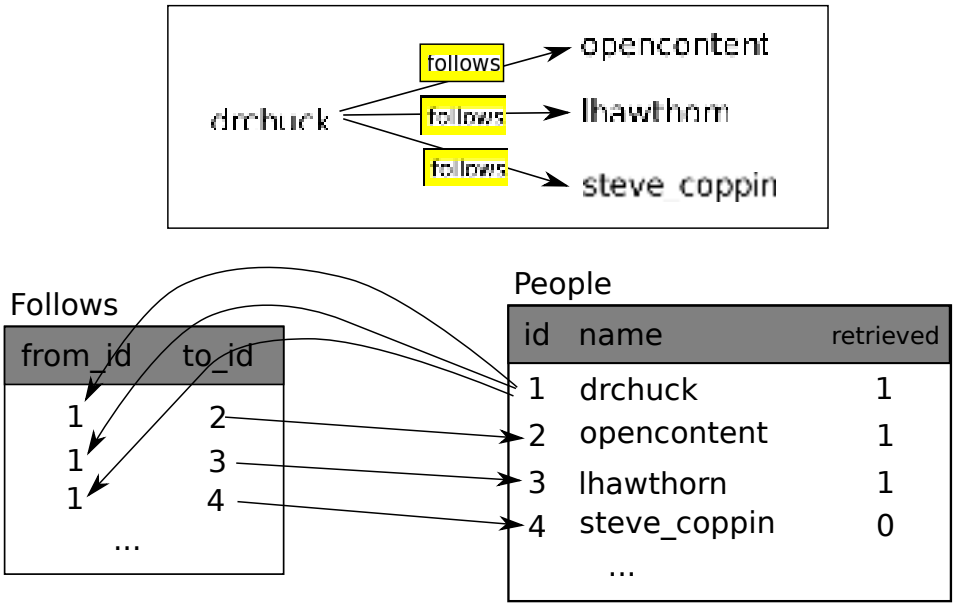

Figure 15.4: Relationships Between Tables

```
TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('friends.sqlite')
cur = conn.cursor()

cur.execute('''CREATE TABLE IF NOT EXISTS People
            (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
            (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')

while True:
    acct = input('Enter a Twitter account, or quit: ')
    if (acct == 'quit'): break
    if (len(acct) < 1):
        cur.execute('SELECT id, name FROM People WHERE retrieved = 0 LIMIT 1')
        try:
            (id, acct) = cur.fetchone()
        except:
            print('No unretrieved Twitter accounts found')
            continue
    else:
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
                    (acct, ))
        try:
            id = cur.fetchone()[0]
        except:
            cur.execute('''INSERT OR IGNORE INTO People
                        (name, retrieved) VALUES (?, 0)''', (acct, ))
```

```
            conn.commit()
            if cur.rowcount != 1:
                print('Error inserting account:', acct)
                continue
            id = cur.lastrowid

    url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '5'})
    print('Retrieving account', acct)
    connection = urllib.request.urlopen(url)
    data = connection.read().decode()
    headers = dict(connection.getheaders())

    print('Remaining', headers['x-rate-limit-remaining'])

    js = json.loads(data)
    # Debugging
    # print json.dumps(js, indent=4)

    cur.execute('UPDATE People SET retrieved=1 WHERE name = ?', (acct, ))

    countnew = 0
    countold = 0
    for u in js['users']:
        friend = u['screen_name']
        print(friend)
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
                    (friend, ))
        try:
            friend_id = cur.fetchone()[0]
            countold = countold + 1
        except:
            cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
                        VALUES (?, 0)''', (friend, ))
            conn.commit()
            if cur.rowcount != 1:
                print('Error inserting account:', friend)
                continue
            friend_id = cur.lastrowid
            countnew = countnew + 1
        cur.execute('''INSERT OR IGNORE INTO Follows (from_id, to_id)
                    VALUES (?, ?)''', (id, friend_id))
    print('New accounts=', countnew, ' revisited=', countold)
    conn.commit()
cur.close()

# Code: http://www.pythonlearn.com/code3/twfriends.py
```

This program is starting to get a bit complicated, but it illustrates the patterns that we need to use when we are using integer keys to link tables. The basic patterns are:

1. Create tables with primary keys and constraints.

2. When we have a logical key for a person (i.e., account name) and we need the `id` value for the person, depending on whether or not the person is already in the `People` table we either need to: (1) look up the person in the `People` table and retrieve the `id` value for the person or (2) add the person to the `People` table and get the `id` value for the newly added row.

3. Insert the row that captures the "follows" relationship.

We will cover each of these in turn.

15.8.1 Constraints in database tables

As we design our table structures, we can tell the database system that we would like it to enforce a few rules on us. These rules help us from making mistakes and introducing incorrect data into out tables. When we create our tables:

```
cur.execute('''CREATE TABLE IF NOT EXISTS People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')
```

We indicate that the `name` column in the `People` table must be `UNIQUE`. We also indicate that the combination of the two numbers in each row of the `Follows` table must be unique. These constraints keep us from making mistakes such as adding the same relationship more than once.

We can take advantage of these constraints in the following code:

```
cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
    VALUES ( ?, 0)''', ( friend, ) )
```

We add the `OR IGNORE` clause to our `INSERT` statement to indicate that if this particular `INSERT` would cause a violation of the "`name` must be unique" rule, the database system is allowed to ignore the `INSERT`. We are using the database constraint as a safety net to make sure we don't inadvertently do something incorrect.

Similarly, the following code ensures that we don't add the exact same `Follows` relationship twice.

```
cur.execute('''INSERT OR IGNORE INTO Follows
    (from_id, to_id) VALUES (?, ?)''', (id, friend_id) )
```

Again, we simply tell the database to ignore our attempted `INSERT` if it would violate the uniqueness constraint that we specified for the `Follows` rows.

15.8.2 Retrieve and/or insert a record

When we prompt the user for a Twitter account, if the account exists, we must look up its `id` value. If the account does not yet exist in the `People` table, we must insert the record and get the `id` value from the inserted row.

This is a very common pattern and is done twice in the program above. This code shows how we look up the `id` for a friend's account when we have extracted a `screen_name` from a `user` node in the retrieved Twitter JSON.

Since over time it will be increasingly likely that the account will already be in the database, we first check to see if the `People` record exists using a `SELECT` statement.

If all goes well[2] inside the `try` section, we retrieve the record using `fetchone()` and then retrieve the first (and only) element of the returned tuple and store it in `friend_id`.

If the `SELECT` fails, the `fetchone()[0]` code will fail and control will transfer into the `except` section.

```
friend = u['screen_name']
cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
    (friend, ) )
try:
    friend_id = cur.fetchone()[0]
    countold = countold + 1
except:
    cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
        VALUES ( ?, 0)''', ( friend, ) )
    conn.commit()
    if cur.rowcount != 1 :
        print 'Error inserting account:',friend
        continue
    friend_id = cur.lastrowid
    countnew = countnew + 1
```

If we end up in the `except` code, it simply means that the row was not found, so we must insert the row. We use `INSERT OR IGNORE` just to avoid errors and then call `commit()` to force the database to really be updated. After the write is done, we can check the `cur.rowcount` to see how many rows were affected. Since we are attempting to insert a single row, if the number of affected rows is something other than 1, it is an error.

If the `INSERT` is successful, we can look at `cur.lastrowid` to find out what value the database assigned to the `id` column in our newly created row.

15.8.3 Storing the friend relationship

Once we know the key value for both the Twitter user and the friend in the JSON, it is a simple matter to insert the two numbers into the `Follows` table with the

[2] In general, when a sentence starts with "if all goes well" you will find that the code needs to use try/except.

following code:

```
cur.execute('INSERT OR IGNORE INTO Follows (from_id, to_id) VALUES (?, ?)',
    (id, friend_id) )
```

Notice that we let the database take care of keeping us from "double-inserting" a relationship by creating the table with a uniqueness constraint and then adding `OR IGNORE` to our `INSERT` statement.

Here is a sample execution of this program:

```
Enter a Twitter account, or quit:
No unretrieved Twitter accounts found
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

We started with the `drchuck` account and then let the program automatically pick the next two accounts to retrieve and add to our database.

The following is the first few rows in the `People` and `Follows` tables after this run is completed:

```
People:
(1, 'drchuck', 1)
(2, 'opencontent', 1)
(3, 'lhawthorn', 1)
(4, 'steve_coppin', 0)
(5, 'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
```

You can see the `id`, `name`, and `visited` fields in the `People` table and you see the numbers of both ends of the relationship in the `Follows` table. In the `People` table, we can see that the first three people have been visited and their data has been retrieved. The data in the `Follows` table indicates that `drchuck` (user 1) is a friend to all of the people shown in the first five rows. This makes sense because the first data we retrieved and stored was the Twitter friends of `drchuck`. If you were to print more rows from the `Follows` table, you would see the friends of users 2 and 3 as well.

15.9 Three kinds of keys

Now that we have started building a data model putting our data into multiple linked tables and linking the rows in those tables using *keys*, we need to look at some terminology around keys. There are generally three kinds of keys used in a database model.

- A *logical key* is a key that the "real world" might use to look up a row. In our example data model, the `name` field is a logical key. It is the screen name for the user and we indeed look up a user's row several times in the program using the `name` field. You will often find that it makes sense to add a `UNIQUE` constraint to a logical key. Since the logical key is how we look up a row from the outside world, it makes little sense to allow multiple rows with the same value in the table.
- A *primary key* is usually a number that is assigned automatically by the database. It generally has no meaning outside the program and is only used to link rows from different tables together. When we want to look up a row in a table, usually searching for the row using the primary key is the fastest way to find the row. Since primary keys are integer numbers, they take up very little storage and can be compared or sorted very quickly. In our data model, the `id` field is an example of a primary key.
- A *foreign key* is usually a number that points to the primary key of an associated row in a different table. An example of a foreign key in our data model is the `from_id`.

We are using a naming convention of always calling the primary key field name `id` and appending the suffix `_id` to any field name that is a foreign key.

15.10 Using JOIN to retrieve data

Now that we have followed the rules of database normalization and have data separated into two tables, linked together using primary and foreign keys, we need to be able to build a `SELECT` that reassembles the data across the tables.

SQL uses the `JOIN` clause to reconnect these tables. In the `JOIN` clause you specify the fields that are used to reconnect the rows between the tables.

The following is an example of a `SELECT` with a `JOIN` clause:

```
SELECT * FROM Follows JOIN People
    ON Follows.from_id = People.id WHERE People.id = 1
```

The `JOIN` clause indicates that the fields we are selecting cross both the `Follows` and `People` tables. The `ON` clause indicates how the two tables are to be joined: Take the rows from `Follows` and append the row from `People` where the field `from_id` in `Follows` is the same the `id` value in the `People` table.

The result of the JOIN is to create extra-long "metarows" which have both the fields from `People` and the matching fields from `Follows`. Where there is more

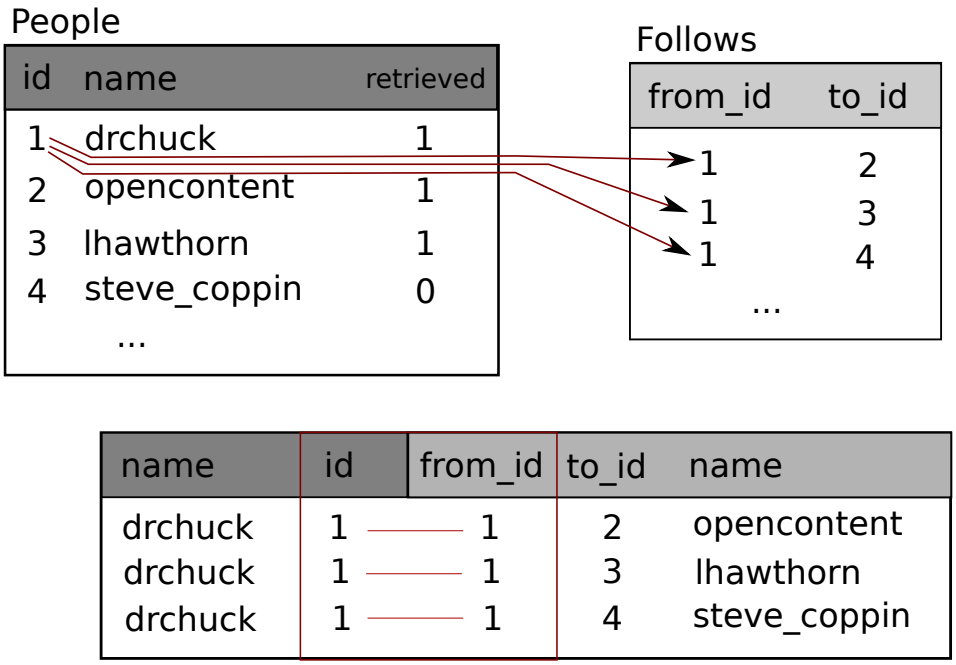

Figure 15.5: Conencting Tables Using JOIN

than one match between the `id` field from `People` and the `from_id` from `People`, then JOIN creates a metarow for *each* of the matching pairs of rows, duplicating data as needed.

The following code demonstrates the data that we will have in the database after the multi-table Twitter spider program (above) has been run several times.

```
import sqlite3

conn = sqlite3.connect('friends.sqlite')
cur = conn.cursor()

cur.execute('SELECT * FROM People')
count = 0
print('People:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')

cur.execute('SELECT * FROM Follows')
count = 0
print('Follows:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')
```

```
cur.execute('''SELECT * FROM Follows JOIN People
            ON Follows.to_id = People.id
            WHERE Follows.from_id = 2''')
count = 0
print('Connections for id=2:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')

cur.close()

# Code: http://www.pythonlearn.com/code3/twjoin.py
```

In this program, we first dump out the `People` and `Follows` and then dump out a subset of the data in the tables joined together.

Here is the output of the program:

```
python twjoin.py
People:
(1, 'drchuck', 1)
(2, 'opencontent', 1)
(3, 'lhawthorn', 1)
(4, 'steve_coppin', 0)
(5, 'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
Connections for id=2:
(2, 1, 1, 'drchuck', 1)
(2, 28, 28, 'cnxorg', 0)
(2, 30, 30, 'kthanos', 0)
(2, 102, 102, 'SomethingGirl', 0)
(2, 103, 103, 'ja_Pac', 0)
20 rows.
```

You see the columns from the `People` and `Follows` tables and the last set of rows is the result of the `SELECT` with the `JOIN` clause.

In the last select, we are looking for accounts that are friends of "opencontent" (i.e., `People.id=2`).

In each of the "metarows" in the last select, the first two columns are from the `Follows` table followed by columns three through five from the `People` table. You can also see that the second column (`Follows.to_id`) matches the third column (`People.id`) in each of the joined-up "metarows".

15.11 Summary

This chapter has covered a lot of ground to give you an overview of the basics of using a database in Python. It is more complicated to write the code to use a database to store data than Python dictionaries or flat files so there is little reason to use a database unless your application truly needs the capabilities of a database. The situations where a database can be quite useful are: (1) when your application needs to make small many random updates within a large data set, (2) when your data is so large it cannot fit in a dictionary and you need to look up information repeatedly, or (3) when you have a long-running process that you want to be able to stop and restart and retain the data from one run to the next.

You can build a simple database with a single table to suit many application needs, but most problems will require several tables and links/relationships between rows in different tables. When you start making links between tables, it is important to do some thoughtful design and follow the rules of database normalization to make the best use of the database's capabilities. Since the primary motivation for using a database is that you have a large amount of data to deal with, it is important to model your data efficiently so your programs run as fast as possible.

15.12 Debugging

One common pattern when you are developing a Python program to connect to an SQLite database will be to run a Python program and check the results using the Database Browser for SQLite. The browser allows you to quickly check to see if your program is working properly.

You must be careful because SQLite takes care to keep two programs from changing the same data at the same time. For example, if you open a database in the browser and make a change to the database and have not yet pressed the "save" button in the browser, the browser "locks" the database file and keeps any other program from accessing the file. In particular, your Python program will not be able to access the file if it is locked.

So a solution is to make sure to either close the database browser or use the *File* menu to close the database in the browser before you attempt to access the database from Python to avoid the problem of your Python code failing because the database is locked.

15.13 Glossary

attribute One of the values within a tuple. More commonly called a "column" or "field".

constraint When we tell the database to enforce a rule on a field or a row in a table. A common constraint is to insist that there can be no duplicate values in a particular field (i.e., all the values must be unique).

cursor A cursor allows you to execute SQL commands in a database and retrieve data from the database. A cursor is similar to a socket or file handle for network connections and files, respectively.

database browser A piece of software that allows you to directly connect to a database and manipulate the database directly without writing a program.

foreign key A numeric key that points to the primary key of a row in another table. Foreign keys establish relationships between rows stored in different tables.

index Additional data that the database software maintains as rows and inserts into a table to make lookups very fast.

logical key A key that the "outside world" uses to look up a particular row. For example in a table of user accounts, a person's email address might be a good candidate as the logical key for the user's data.

normalization Designing a data model so that no data is replicated. We store each item of data at one place in the database and reference it elsewhere using a foreign key.

primary key A numeric key assigned to each row that is used to refer to one row in a table from another table. Often the database is configured to automatically assign primary keys as rows are inserted.

relation An area within a database that contains tuples and attributes. More typically called a "table".

tuple A single entry in a database table that is a set of attributes. More typically called "row".

Chapter 16

Visualizing data

So far we have been learning the Python language and then learning how to use Python, the network, and databases to manipulate data.

In this chapter, we take a look at three complete applications that bring all of these things together to manage and visualize data. You might use these applications as sample code to help get you started in solving a real-world problem.

Each of the applications is a ZIP file that you can download and extract onto your computer and execute.

16.1 Building a Google map from geocoded data

In this project, we are using the Google geocoding API to clean up some user-entered geographic locations of university names and then placing the data on a Google map.

To get started, download the application from:

www.pythonlearn.com/code3/geodata.zip

The first problem to solve is that the free Google geocoding API is rate-limited to a certain number of requests per day. If you have a lot of data, you might need to stop and restart the lookup process several times. So we break the problem into two phases.

In the first phase we take our input "survey" data in the file *where.data* and read it one line at a time, and retrieve the geocoded information from Google and store it in a database *geodata.sqlite*. Before we use the geocoding API for each user-entered location, we simply check to see if we already have the data for that particular line of input. The database is functioning as a local "cache" of our geocoding data to make sure we never ask Google for the same data twice.

You can restart the process at any time by removing the file *geodata.sqlite*.

Run the *geoload.py* program. This program will read the input lines in *where.data* and for each line check to see if it is already in the database. If we don't have the

Figure 16.1: A Google Map

data for the location, it will call the geocoding API to retrieve the data and store it in the database.

Here is a sample run after there is already some data in the database:

```
Found in database  Northeastern University
Found in database  University of Hong Kong, ...
Found in database  Technion
Found in database  Viswakarma Institute, Pune, India
Found in database  UMD
Found in database  Tufts University

Resolving Monash University
Retrieving http://maps.googleapis.com/maps/api/
    geocode/json?sensor=false&address=Monash+University
Retrieved 2063 characters {    "results" : [
{'status': 'OK', 'results': ... }

Resolving Kokshetau Institute of Economics and Management
Retrieving http://maps.googleapis.com/maps/api/
    geocode/json?sensor=false&address=Kokshetau+Inst ...
Retrieved 1749 characters {    "results" : [
{'status': 'OK', 'results': ... }
...
```

The first five locations are already in the database and so they are skipped. The program scans to the point where it finds new locations and starts retrieving them.

The *geoload.py* program can be stopped at any time, and there is a counter that you can use to limit the number of calls to the geocoding API for each run. Given that the *where.data* only has a few hundred data items, you should not run into the daily rate limit, but if you had more data it might take several runs over several days to get your database to have all of the geocoded data for your input.

Once you have some data loaded into *geodata.sqlite*, you can visualize the data using the *geodump.py* program. This program reads the database and writes the file *where.js* with the location, latitude, and longitude in the form of executable JavaScript code.

A run of the *geodump.py* program is as follows:

```
Northeastern University, ... Boston, MA 02115, USA 42.3396998 -71.08975
Bradley University, 1501 ... Peoria, IL 61625, USA 40.6963857 -89.6160811
...
Technion, Viazman 87, Kesalsaba, 32000, Israel 32.7775 35.0216667
Monash University Clayton ... VIC 3800, Australia -37.9152113 145.134682
Kokshetau, Kazakhstan 53.2833333 69.3833333
...
12 records written to where.js
Open where.html to view the data in a browser
```

The file *where.html* consists of HTML and JavaScript to visualize a Google map. It reads the most recent data in *where.js* to get the data to be visualized. Here is the format of the *where.js* file:

```
myData = [
[42.3396998,-71.08975, 'Northeastern Uni ... Boston, MA 02115'],
[40.6963857,-89.6160811, 'Bradley University, ... Peoria, IL 61625, USA'],
[32.7775,35.0216667, 'Technion, Viazman 87, Kesalsaba, 32000, Israel'],
   ...
];
```

This is a JavaScript variable that contains a list of lists. The syntax for JavaScript list constants is very similar to Python, so the syntax should be familiar to you.

Simply open *where.html* in a browser to see the locations. You can hover over each map pin to find the location that the geocoding API returned for the user-entered input. If you cannot see any data when you open the *where.html* file, you might want to check the JavaScript or developer console for your browser.

16.2 Visualizing networks and interconnections

In this application, we will perform some of the functions of a search engine. We will first spider a small subset of the web and run a simplified version of the Google page rank algorithm to determine which pages are most highly connected, and then visualize the page rank and connectivity of our small corner of the web. We will use the D3 JavaScript visualization library http://d3js.org/ to produce the visualization output.

You can download and extract this application from:

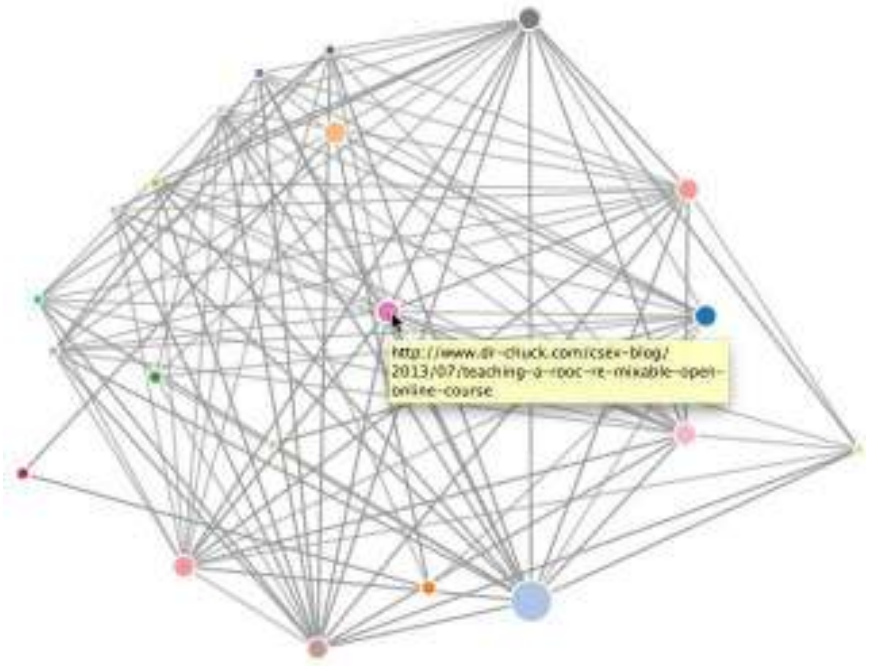

Figure 16.2: A Page Ranking

www.pythonlearn.com/code3/pagerank.zip

The first program (*spider.py*) program crawls a web site and pulls a series of pages into the database (*spider.sqlite*), recording the links between pages. You can restart the process at any time by removing the *spider.sqlite* file and rerunning *spider.py*.

```
Enter web url or enter: http://www.dr-chuck.com/
['http://www.dr-chuck.com']
How many pages:2
1 http://www.dr-chuck.com/ 12
2 http://www.dr-chuck.com/csev-blog/ 57
How many pages:
```

In this sample run, we told it to crawl a website and retrieve two pages. If you restart the program and tell it to crawl more pages, it will not re-crawl any pages already in the database. Upon restart it goes to a random non-crawled page and starts there. So each successive run of *spider.py* is additive.

```
Enter web url or enter: http://www.dr-chuck.com/
['http://www.dr-chuck.com']
How many pages:3
3 http://www.dr-chuck.com/csev-blog 57
4 http://www.dr-chuck.com/dr-chuck/resume/speaking.htm 1
5 http://www.dr-chuck.com/dr-chuck/resume/index.htm 13
How many pages:
```

You can have multiple starting points in the same database—within the program,

these are called "webs". The spider chooses randomly amongst all non-visited links across all the webs as the next page to spider.

If you want to dump the contents of the *spider.sqlite* file, you can run *spdump.py* as follows:

```
(5, None, 1.0, 3, 'http://www.dr-chuck.com/csev-blog')
(3, None, 1.0, 4, 'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, None, 1.0, 2, 'http://www.dr-chuck.com/csev-blog/')
(1, None, 1.0, 5, 'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

This shows the number of incoming links, the old page rank, the new page rank, the id of the page, and the url of the page. The *spdump.py* program only shows pages that have at least one incoming link to them.

Once you have a few pages in the database, you can run page rank on the pages using the *sprank.py* program. You simply tell it how many page rank iterations to run.

```
How many iterations:2
1 0.546848992536
2 0.226714939664
[(1, 0.559), (2, 0.659), (3, 0.985), (4, 2.135), (5, 0.659)]
```

You can dump the database again to see that page rank has been updated:

```
(5, 1.0, 0.985, 3, 'http://www.dr-chuck.com/csev-blog')
(3, 1.0, 2.135, 4, 'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, 1.0, 0.659, 2, 'http://www.dr-chuck.com/csev-blog/')
(1, 1.0, 0.659, 5, 'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

You can run *sprank.py* as many times as you like and it will simply refine the page rank each time you run it. You can even run *sprank.py* a few times and then go spider a few more pages sith *spider.py* and then run *sprank.py* to reconverge the page rank values. A search engine usually runs both the crawling and ranking programs all the time.

If you want to restart the page rank calculations without respidering the web pages, you can use *spreset.py* and then restart *sprank.py*.

```
How many iterations:50
1 0.546848992536
2 0.226714939664
3 0.0659516187242
4 0.0244199333
5 0.0102096489546
6 0.00610244329379
...
42 0.000109076928206
43 9.91987599002e-05
```

```
44 9.02151706798e-05
45 8.20451504471e-05
46 7.46150183837e-05
47 6.7857770908e-05
48 6.17124694224e-05
49 5.61236959327e-05
50 5.10410499467e-05
[(512, 0.0296), (1, 12.79), (2, 28.93), (3, 6.808), (4, 13.46)]
```

For each iteration of the page rank algorithm it prints the average change in page rank per page. The network initially is quite unbalanced and so the individual page rank values change wildly between iterations. But in a few short iterations, the page rank converges. You should run *prank.py* long enough that the page rank values converge.

If you want to visualize the current top pages in terms of page rank, run *spjson.py* to read the database and write the data for the most highly linked pages in JSON format to be viewed in a web browser.

```
Creating JSON output on spider.json...
How many nodes? 30
Open force.html in a browser to view the visualization
```

You can view this data by opening the file *force.html* in your web browser. This shows an automatic layout of the nodes and links. You can click and drag any node and you can also double-click on a node to find the URL that is represented by the node.

If you rerun the other utilities, rerun *spjson.py* and press refresh in the browser to get the new data from *spider.json*.

16.3 Visualizing mail data

Up to this point in the book, you have become quite familiar with our *mbox-short.txt* and *mbox.txt* data files. Now it is time to take our analysis of email data to the next level.

In the real world, sometimes you have to pull down mail data from servers. That might take quite some time and the data might be inconsistent, error-filled, and need a lot of cleanup or adjustment. In this section, we work with an application that is the most complex so far and pull down nearly a gigabyte of data and visualize it.

You can download this application from:

www.pythonlearn.com/code3/gmane.zip

We will be using data from a free email list archiving service called www.gmane.org. This service is very popular with open source projects because it provides a nice searchable archive of their email activity. They also have a very liberal policy regarding accessing their data through their API. They have no rate limits, but ask that you don't overload their service and take only the data you need. You can read gmane's terms and conditions at this page:

Figure 16.3: A Word Cloud from the Sakai Developer List

http://gmane.org/export.php

It is very important that you make use of the gmane.org data responsibly by adding delays to your access of their services and spreading long-running jobs over a longer period of time. Do not abuse this free service and ruin it for the rest of us.

When the Sakai email data was spidered using this software, it produced nearly a Gigabyte of data and took a number of runs on several days. The file *README.txt* in the above ZIP may have instructions as to how you can download a pre-spidered copy of the *content.sqlite* file for a majority of the Sakai email corpus so you don't have to spider for five days just to run the programs. If you download the pre-spidered content, you should still run the spidering process to catch up with more recent messages.

The first step is to spider the gmane repository. The base URL is hard-coded in the *gmane.py* and is hard-coded to the Sakai developer list. You can spider another repository by changing that base url. Make sure to delete the *content.sqlite* file if you switch the base url.

The *gmane.py* file operates as a responsible caching spider in that it runs slowly and retrieves one mail message per second so as to avoid getting throttled by gmane. It stores all of its data in a database and can be interrupted and restarted as often as needed. It may take many hours to pull all the data down. So you may need to restart several times.

Here is a run of *gmane.py* retrieving the last five messages of the Sakai developer list:

```
How many messages:10
```

```
http://download.gmane.org/gmane.comp.cms.sakai.devel/51410/51411 9460
    nealcaidin@sakaifoundation.org 2013-04-05 re: [building ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51411/51412 3379
    samuelgutierrezjimenez@gmail.com 2013-04-06 re: [building ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51412/51413 9903
    da1@vt.edu 2013-04-05 [building sakai] melete 2.9 oracle ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51413/51414 349265
    m.shedid@elraed-it.com 2013-04-07 [building sakai] ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51414/51415 3481
    samuelgutierrezjimenez@gmail.com 2013-04-07 re: ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51415/51416 0

Does not start with From
```

The program scans *content.sqlite* from one up to the first message number not already spidered and starts spidering at that message. It continues spidering until it has spidered the desired number of messages or it reaches a page that does not appear to be a properly formatted message.

Sometimes gmane.org is missing a message. Perhaps administrators can delete messages or perhaps they get lost. If your spider stops, and it seems it has hit a missing message, go into the SQLite Manager and add a row with the missing id leaving all the other fields blank and restart *gmane.py*. This will unstick the spidering process and allow it to continue. These empty messages will be ignored in the next phase of the process.

One nice thing is that once you have spidered all of the messages and have them in *content.sqlite*, you can run *gmane.py* again to get new messages as they are sent to the list.

The *content.sqlite* data is pretty raw, with an inefficient data model, and not compressed. This is intentional as it allows you to look at *content.sqlite* in the SQLite Manager to debug problems with the spidering process. It would be a bad idea to run any queries against this database, as they would be quite slow.

The second process is to run the program *gmodel.py*. This program reads the raw data from *content.sqlite* and produces a cleaned-up and well-modeled version of the data in the file *index.sqlite*. This file will be much smaller (often 10X smaller) than *content.sqlite* because it also compresses the header and body text.

Each time *gmodel.py* runs it deletes and rebuilds *index.sqlite*, allowing you to adjust its parameters and edit the mapping tables in *content.sqlite* to tweak the data cleaning process. This is a sample run of *gmodel.py*. It prints a line out each time 250 mail messages are processed so you can see some progress happening, as this program may run for a while processing nearly a Gigabyte of mail data.

```
Loaded allsenders 1588 and mapping 28 dns mapping 1
1 2005-12-08T23:34:30-06:00 ggolden22@mac.com
251 2005-12-22T10:03:20-08:00 tpamsler@ucdavis.edu
501 2006-01-12T11:17:34-05:00 lance@indiana.edu
751 2006-01-24T11:13:28-08:00 vrajgopalan@ucmerced.edu
...
```

The *gmodel.py* program handles a number of data cleaning tasks.

Domain names are truncated to two levels for .com, .org, .edu, and .net. Other domain names are truncated to three levels. So si.umich.edu becomes umich.edu and caret.cam.ac.uk becomes cam.ac.uk. Email addresses are also forced to lower case, and some of the @gmane.org address like the following

```
arwhyte-63aXycvo3TyHXe+LvDLADg@public.gmane.org
```

are converted to the real address whenever there is a matching real email address elsewhere in the message corpus.

In the *content.sqlite* database there are two tables that allow you to map both domain names and individual email addresses that change over the lifetime of the email list. For example, Steve Githens used the following email addresses as he changed jobs over the life of the Sakai developer list:

```
s-githens@northwestern.edu
sgithens@cam.ac.uk
swgithen@mtu.edu
```

We can add two entries to the Mapping table in *content.sqlite* so *gmodel.py* will map all three to one address:

```
s-githens@northwestern.edu ->  swgithen@mtu.edu
sgithens@cam.ac.uk -> swgithen@mtu.edu
```

You can also make similar entries in the DNSMapping table if there are multiple DNS names you want mapped to a single DNS. The following mapping was added to the Sakai data:

```
iupui.edu -> indiana.edu
```

so all the accounts from the various Indiana University campuses are tracked together.

You can rerun the *gmodel.py* over and over as you look at the data, and add mappings to make the data cleaner and cleaner. When you are done, you will have a nicely indexed version of the email in *index.sqlite*. This is the file to use to do data analysis. With this file, data analysis will be really quick.

The first, simplest data analysis is to determine "who sent the most mail?" and "which organization sent the most mail"? This is done using *gbasic.py*:

```
How many to dump? 5
Loaded messages= 51330 subjects= 25033 senders= 1584

Top 5 Email list participants
steve.swinsburg@gmail.com 2657
azeckoski@unicon.net 1742
ieb@tfd.co.uk 1591
csev@umich.edu 1304
david.horwitz@uct.ac.za 1184
```

```
Top 5 Email list organizations
gmail.com 7339
umich.edu 6243
uct.ac.za 2451
indiana.edu 2258
unicon.net 2055
```

Note how much more quickly *gbasic.py* runs compared to *gmane.py* or even *gmodel.py*. They are all working on the same data, but *gbasic.py* is using the compressed and normalized data in *index.sqlite*. If you have a lot of data to manage, a multistep process like the one in this application may take a little longer to develop, but will save you a lot of time when you really start to explore and visualize your data.

You can produce a simple visualization of the word frequency in the subject lines in the file *gword.py*:

```
Range of counts: 33229 129
Output written to gword.js
```

This produces the file *gword.js* which you can visualize using *gword.htm* to produce a word cloud similar to the one at the beginning of this section.

A second visualization is produced by *gline.py*. It computes email participation by organizations over time.

```
Loaded messages= 51330 subjects= 25033 senders= 1584
Top 10 Oranizations
['gmail.com', 'umich.edu', 'uct.ac.za', 'indiana.edu',
'unicon.net', 'tfd.co.uk', 'berkeley.edu', 'longsight.com',
'stanford.edu', 'ox.ac.uk']
Output written to gline.js
```

Its output is written to *gline.js* which is visualized using *gline.htm*.

This is a relatively complex and sophisticated application and has features to do some real data retrieval, cleaning, and visualization.

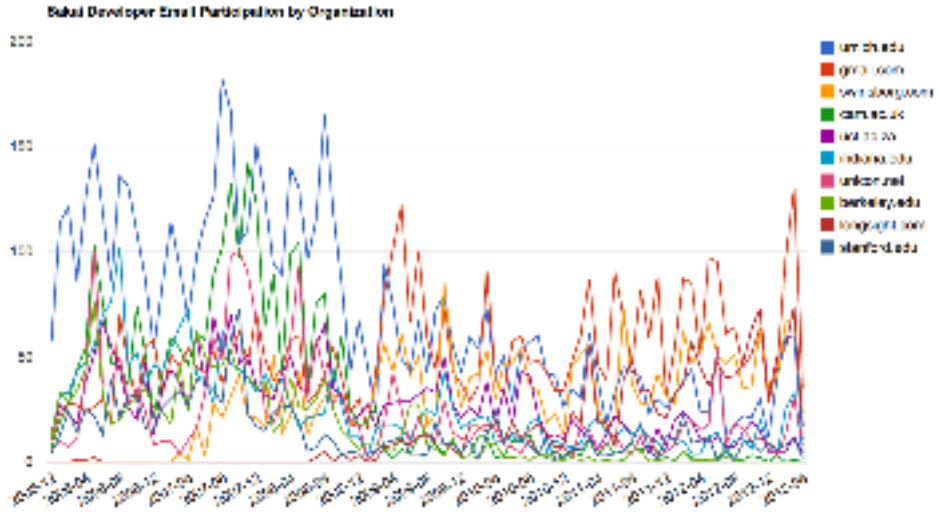

Figure 16.4: Sakai Mail Activity by Organization

Appendix A

Contributions

A.1 Contributor List for Python for Everybody

Elliott Hauser, Stephen Catto, Sue Blumenberg, Tamara Brunnock, Mihaela Mack, Chris Kolosiwsky, Dustin Farley, Jens Leerssen, Naveen KT, Mirza Ibrahimovic, Naveen (@togarnk), Zhou Fangyi, Alistair Walsh, Erica Brody, Jih-Sheng Huang, Louis Luangkesorn, and Michael Fudge

You can see contribution details at:

https://github.com/csev/pythonlearn/graphs/contributors

A.2 Contributor List for Python for Informatics

Bruce Shields for copy editing early drafts, Sarah Hegge, Steven Cherry, Sarah Kathleen Barbarow, Andrea Parker, Radaphat Chongthammakun, Megan Hixon, Kirby Urner, Sarah Kathleen Barbrow, Katie Kujala, Noah Botimer, Emily Alinder, Mark Thompson-Kular, James Perry, Eric Hofer, Eytan Adar, Peter Robinson, Deborah J. Nelson, Jonathan C. Anthony, Eden Rassette, Jeannette Schroeder, Justin Feezell, Chuanqi Li, Gerald Gordinier, Gavin Thomas Strassel, Ryan Clement, Alissa Talley, Caitlin Holman, Yong-Mi Kim, Karen Stover, Cherie Edmonds, Maria Seiferle, Romer Kristi D. Aranas (RK), Grant Boyer, Hedemarrie Dussan,

A.3 Preface for "Think Python"

A.3.1 The strange history of "Think Python"

(Allen B. Downey)

In January 1999 I was preparing to teach an introductory programming class in Java. I had taught it three times and I was getting frustrated. The failure rate in

the class was too high and, even for students who succeeded, the overall level of achievement was too low.

One of the problems I saw was the books. They were too big, with too much unnecessary detail about Java, and not enough high-level guidance about how to program. And they all suffered from the trap door effect: they would start out easy, proceed gradually, and then somewhere around Chapter 5 the bottom would fall out. The students would get too much new material, too fast, and I would spend the rest of the semester picking up the pieces.

Two weeks before the first day of classes, I decided to write my own book. My goals were:

- Keep it short. It is better for students to read 10 pages than not read 50 pages.
- Be careful with vocabulary. I tried to minimize the jargon and define each term at first use.
- Build gradually. To avoid trap doors, I took the most difficult topics and split them into a series of small steps.
- Focus on programming, not the programming language. I included the minimum useful subset of Java and left out the rest.

I needed a title, so on a whim I chose *How to Think Like a Computer Scientist*.

My first version was rough, but it worked. Students did the reading, and they understood enough that I could spend class time on the hard topics, the interesting topics and (most important) letting the students practice.

I released the book under the GNU Free Documentation License, which allows users to copy, modify, and distribute the book.

What happened next is the cool part. Jeff Elkner, a high school teacher in Virginia, adopted my book and translated it into Python. He sent me a copy of his translation, and I had the unusual experience of learning Python by reading my own book.

Jeff and I revised the book, incorporated a case study by Chris Meyers, and in 2001 we released *How to Think Like a Computer Scientist: Learning with Python*, also under the GNU Free Documentation License. As Green Tea Press, I published the book and started selling hard copies through Amazon.com and college book stores. Other books from Green Tea Press are available at greenteapress.com.

In 2003 I started teaching at Olin College and I got to teach Python for the first time. The contrast with Java was striking. Students struggled less, learned more, worked on more interesting projects, and generally had a lot more fun.

Over the last five years I have continued to develop the book, correcting errors, improving some of the examples and adding material, especially exercises. In 2008 I started work on a major revision—at the same time, I was contacted by an editor at Cambridge University Press who was interested in publishing the next edition. Good timing!

I hope you enjoy working with this book, and that it helps you learn to program and think, at least a little bit, like a computer scientist.

A.3.2 Acknowledgements for "Think Python"

(Allen B. Downey)

First and most importantly, I thank Jeff Elkner, who translated my Java book into Python, which got this project started and introduced me to what has turned out to be my favorite language.

I also thank Chris Meyers, who contributed several sections to *How to Think Like a Computer Scientist*.

And I thank the Free Software Foundation for developing the GNU Free Documentation License, which helped make my collaboration with Jeff and Chris possible.

I also thank the editors at Lulu who worked on *How to Think Like a Computer Scientist*.

I thank all the students who worked with earlier versions of this book and all the contributors (listed in an Appendix) who sent in corrections and suggestions.

And I thank my wife, Lisa, for her work on this book, and Green Tea Press, and everything else, too.

Allen B. Downey
Needham MA
Allen Downey is an Associate Professor of Computer Science at the Franklin W. Olin College of Engineering.

A.4 Contributor List for "Think Python"

(Allen B. Downey)

More than 100 sharp-eyed and thoughtful readers have sent in suggestions and corrections over the past few years. Their contributions, and enthusiasm for this project, have been a huge help.

For the detail on the nature of each of the contributions from these individuals, see the "Think Python" text.

Lloyd Hugh Allen, Yvon Boulianne, Fred Bremmer, Jonah Cohen, Michael Conlon, Benoit Girard, Courtney Gleason and Katherine Smith, Lee Harr, James Kaylin, David Kershaw, Eddie Lam, Man-Yong Lee, David Mayo, Chris McAloon, Matthew J. Moelter, Simon Dicon Montford, John Ouzts, Kevin Parks, David Pool, Michael Schmitt, Robin Shaw, Paul Sleigh, Craig T. Snydal, Ian Thomas, Keith Verheyden, Peter Winstanley, Chris Wrobel, Moshe Zadka, Christoph Zwerschke, James Mayer, Hayden McAfee, Angel Arnal, Tauhidul Hoque and Lex Berezhny, Dr. Michele Alzetta, Andy Mitchell, Kalin Harvey, Christopher P. Smith, David Hutchins, Gregor Lingl, Julie Peters, Florin Oprina, D. J. Webre, Ken, Ivo Wever, Curtis Yanko, Ben Logan, Jason Armstrong, Louis Cordier, Brian Cain, Rob Black, Jean-Philippe Rey at Ecole Centrale Paris, Jason Mader at George Washington University made a number Jan Gundtofte-Bruun, Abel David and Alexis Dinno, Charles Thayer, Roger Sperberg, Sam Bull, Andrew Cheung, C. Corey Capel, Alessandra, Wim Champagne, Douglas Wright, Jared Spindor,

Lin Peiheng, Ray Hagtvedt, Torsten Hübsch, Inga Petuhhov, Arne Babenhauserheide, Mark E. Casida, Scott Tyler, Gordon Shephard, Andrew Turner, Adam Hobart, Daryl Hammond and Sarah Zimmerman, George Sass, Brian Bingham, Leah Engelbert-Fenton, Joe Funke, Chao-chao Chen, Jeff Paine, Lubos Pintes, Gregg Lind and Abigail Heithoff, Max Hailperin, Chotipat Pornavalai, Stanislaw Antol, Eric Pashman, Miguel Azevedo, Jianhua Liu, Nick King, Martin Zuther, Adam Zimmerman, Ratnakar Tiwari, Anurag Goel, Kelli Kratzer, Mark Griffiths, Roydan Ongie, Patryk Wolowiec, Mark Chonofsky, Russell Coleman, Wei Huang, Karen Barber, Nam Nguyen, Stéphane Morin, Fernando Tardio, and Paul Stoop.

Appendix B

Copyright Detail

I would have preferred to license the book under the less restrictive CC-BY-SA license. But unfortunately there are a few unscrupulous organizations who search for and find freely licensed books, and then publish and sell virtually unchanged copies of the books on a print on demand service such as LuLu or CreateSpace. CreateSpace has (thankfully) added a policy that gives the wishes of the actual copyright holder preference over a non-copyright holder attempting to publish a freely licensed work. Unfortunately there are many print-on-demand services and very few have as well-considered a policy as CreateSpace.

Regretfully, I added the NC element to the license this book to give me recourse in case someone tries to clone this book and sell it commercially. Unfortunately, adding NC limits uses of this material that I would like to permit. So I have added this section of the document to describe specific situations where I am giving my permission in advance to use the material in this book in situations that some might consider commercial.

- If you are printing a limited number of copies of all or part of this book for use in a course (e.g., like a coursepack), then you are granted CC-BY license to these materials for that purpose.
- If you are a teacher at a university and you translate this book into a language other than English and teach using the translated book, then you can contact me and I will granted you a CC-BY-SA license to these materials with respect to the publication of your translation. In particular, you will be permitted to sell the resulting translated book commercially.

If you are intending to translate the book, you may want to contact me so we can make sure that you have all of the related course materials so you can translate them as well.

Of course, you are welcome to contact me and ask for permission if these clauses are not sufficient. In all cases, permission to reuse and remix this material will be

granted as long as there is clear added value or benefit to students or teachers that will accrue as a result of the new work.

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
September 9, 2013

Index