Python for Everybody

Exploring Data Using Python 3

Charles R. Severance

Credits

Editorial Support: Elliott Hauser, Sue Blumenberg
Cover Design: Aimee Andrion

Printing History

- 2016-Jul-05 First Complete Python 3.0 version
- 2015-Dec-20 Initial Python 3.0 rough conversion

Copyright Details

Copyright ~2009- Charles Severance.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. This license is available at

http://creativecommons.org/licenses/by-nc-sa/3.0/

You can see what the author considers commercial and non-commercial uses of this material as well as license exemptions in the Appendix titled “Copyright Detail”.

Preface

Remixing an Open Book

It is quite natural for academics who are continuously told to "publish or perish" to want to always create something from scratch that is their own fresh creation. This book is an experiment in not starting from scratch, but instead "remixing" the book titled Think Python: How to Think Like a Computer Scientist written by Allen B. Downey, Jeff Elkner, and others.

In December of 2009, I was preparing to teach SI502 - Networked Programming at the University of Michigan for the fifth semester in a row and decided it was time to write a Python textbook that focused on exploring data instead of understanding algorithms and abstractions. My goal in SI502 is to teach people lifelong data handling skills using Python. Few of my students were planning to be professional computer programmers. Instead, they planned to be librarians, managers, lawyers, biologists, economists, etc., who happened to want to skillfully use technology in their chosen field.

I never seemed to find the perfect data-oriented Python book for my course, so I set out to write just such a book. Luckily at a faculty meeting three weeks before I was about to start my new book from scratch over the holiday break, Dr. Atul Prakash showed me the Think Python book which he had used to teach his Python course that semester. It is a well-written Computer Science text with a focus on short, direct explanations and ease of learning.

The overall book structure has been changed to get to doing data analysis problems as quickly as possible and have a series of running examples and exercises about data analysis from the very beginning.

Chapters 2–10 are similar to the Think Python book, but there have been major changes. Number-oriented examples and exercises have been replaced with data-oriented exercises. Topics are presented in the order needed to build increasingly sophisticated data analysis solutions. Some topics like `try` and `except` are pulled forward and presented as part of the chapter on conditionals. Functions are given very light treatment until they are needed to handle program complexity rather than introduced as an early lesson in abstraction. Nearly all user-defined functions have been removed from the example code and exercises outside of Chapter 4. The word "recursion" [1] does not appear in the book at all.

In chapters 1 and 11–16, all of the material is brand new, focusing on real-world uses and simple examples of Python for data analysis including regular expressions for searching and parsing, automating tasks on your computer, retrieving data across the network, scraping web pages for data, object-oriented programming, using web services, parsing XML and JSON data, creating and using databases using Structured Query Language, and visualizing data.

The ultimate goal of all of these changes is a shift from a Computer Science to an Informatics focus is to only include topics into a first technology class that can be useful even if one chooses not to become a professional programmer.

Students who find this book interesting and want to further explore should look at Allen B. Downey's Think Python book. Because there is a lot of overlap between the two books,

[1] Except, of course, for this line.

students will quickly pick up skills in the additional areas of technical programming and algorithmic thinking that are covered in Think Python. And given that the books have a similar writing style, they should be able to move quickly through Think Python with a minimum of effort.

As the copyright holder of Think Python, Allen has given me permission to change the book's license on the material from his book that remains in this book from the GNU Free Documentation License to the more recent Creative Commons Attribution — Share Alike license. This follows a general shift in open documentation licenses moving from the GFDL to the CC-BY-SA (e.g., Wikipedia). Using the CC-BY-SA license maintains the book's strong copyleft tradition while making it even more straightforward for new authors to reuse this material as they see fit.

I feel that this book serves an example of why open materials are so important to the future of education, and want to thank Allen B. Downey and Cambridge University Press for their forward-looking decision to make the book available under an open copyright. I hope they are pleased with the results of my efforts and I hope that you the reader are pleased with our collective efforts.

I would like to thank Allen B. Downey and Lauren Cowles for their help, patience, and guidance in dealing with and resolving the copyright issues around this book.

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
September 9, 2013

Charles Severance is a Clinical Associate Professor at the University of Michigan School of Information.

Contents

1 为什么要学编程? 1

1.1 创新与动机 . . . 2
1.2 计算机硬件架构 . . . 2
1.3 理解编程 . . . 3
1.4 词汇与句子 . . . 4
1.5 与Python对话 . . . 5
1.6 术语：解释器与编译器 . . . 6
1.7 编写一个程序 . . . 8
1.8 什么是程序? . . . 8
1.9 构成程序的小积木 . . . 9
1.10 什么可能会出错? . . . 10
1.11 学习之旅 . . . 11
1.12 术语表 . . . 11
1.13 习题 . . . 12

2 变量、表达式、语句 15

2.1 值与类型 . . . 15
2.2 变量 . . . 16
2.3 变量名与关键字 . . . 16
2.4 语句 . . . 17
2.5 运算符和运算对象 . . . 18
2.6 表达式 . . . 18
2.7 运算顺序 . . . 19
2.8 模运算 . . . 19
2.9 字符串运算符 . . . 19
2.10 请求用户输入 . . . 20

2.11 注释 . . . 21
2.12 助记变量命名法 . . . 21
2.13 调试 . . . 23
2.14 术语表 . . . 23
2.15 习题 . . . 24

3 条件执行 25

3.1 布尔表达式 . . . 25
3.2 逻辑运算符 . . . 26
3.3 Conditional execution . . . 26
3.4 Alternative execution . . . 27
3.5 Chained conditionals . . . 27
3.6 Nested conditionals . . . 29
3.7 Catching exceptions using try and except . . . 30
3.8 Short-circuit evaluation of logical expressions . . . 31
3.9 Debugging . . . 32
3.10 Glossary . . . 33
3.11 Exercises . . . 33

4 Functions 35

4.1 Function calls . . . 35
4.2 Built-in functions . . . 35
4.3 Type conversion functions . . . 36
4.4 Random numbers . . . 37
4.5 Math functions . . . 38
4.6 Adding new functions . . . 39
4.7 Definitions and uses . . . 40
4.8 Flow of execution . . . 41
4.9 Parameters and arguments . . . 41
4.10 Fruitful functions and void functions . . . 42
4.11 Why functions? . . . 44
4.12 Debugging . . . 44
4.13 Glossary . . . 45
4.14 Exercises . . . 45

5 Iteration **47**

5.1 Updating variables . . . 47

5.2 The `while` statement . . . 47

5.3 Infinite loops . . . 48

5.4 "Infinite loops" and `break` . . . 48

5.5 Finishing iterations with `continue` . . . 49

5.6 Definite loops using `for` . . . 50

5.7 Loop patterns . . . 51

5.7.1 Counting and summing loops . . . 51

5.7.2 Maximum and minimum loops . . . 52

5.8 Debugging . . . 54

5.9 Glossary . . . 54

5.10 Exercises . . . 54

6 Strings **57**

6.1 A string is a sequence . . . 57

6.2 Getting the length of a string using `len` . . . 58

6.3 Traversal through a string with a loop . . . 58

6.4 String slices . . . 59

6.5 Strings are immutable . . . 60

6.6 Looping and counting . . . 60

6.7 The `in` operator . . . 61

6.8 String comparison . . . 61

6.9 `string` methods . . . 61

6.10 Parsing strings . . . 64

6.11 Format operator . . . 64

6.12 Debugging . . . 65

6.13 Glossary . . . 66

6.14 Exercises . . . 67

7 Files **69**

7.1 Persistence . . . 69

7.2 Opening files . . . 69

7.3 Text files and lines . . . 71

7.4 Reading files . . . 72

7.5 Searching through a file . . . 73
7.6 Letting the user choose the file name . . . 75
7.7 Using `try`, `except`, and `open` . . . 75
7.8 Writing files . . . 77
7.9 Debugging . . . 78
7.10 Glossary . . . 78
7.11 Exercises . . . 78

8 Lists **81**

8.1 A list is a sequence . . . 81
8.2 Lists are mutable . . . 82
8.3 Traversing a list . . . 82
8.4 List operations . . . 83
8.5 List slices . . . 84
8.6 List methods . . . 84
8.7 Deleting elements . . . 85
8.8 Lists and functions . . . 86
8.9 Lists and strings . . . 87
8.10 Parsing lines . . . 88
8.11 Objects and values . . . 88
8.12 Aliasing . . . 89
8.13 List arguments . . . 90
8.14 Debugging . . . 91
8.15 Glossary . . . 95
8.16 Exercises . . . 95

9 Dictionaries **97**

9.1 Dictionary as a set of counters . . . 99
9.2 Dictionaries and files . . . 100
9.3 Looping and dictionaries . . . 101
9.4 Advanced text parsing . . . 102
9.5 Debugging . . . 104
9.6 Glossary . . . 105
9.7 Exercises . . . 105

10 Tuples **107**

10.1 Tuples are immutable . . . 107
10.2 Comparing tuples . . . 108
10.3 Tuple assignment . . . 110
10.4 Dictionaries and tuples . . . 111
10.5 Multiple assignment with dictionaries . . . 112
10.6 The most common words . . . 112
10.7 Using tuples as keys in dictionaries . . . 114
10.8 Sequences: strings, lists, and tuples - Oh My! . . . 114
10.9 Debugging . . . 115
10.10 Glossary . . . 116
10.11 Exercises . . . 116

11 Regular expressions **119**

11.1 Character matching in regular expressions . . . 120
11.2 Extracting data using regular expressions . . . 121
11.3 Combining searching and extracting . . . 124
11.4 Escape character . . . 127
11.5 Summary . . . 128
11.6 Bonus section for Unix / Linux users . . . 129
11.7 Debugging . . . 129
11.8 Glossary . . . 130
11.9 Exercises . . . 130

12 Networked programs **133**

12.1 HyperText Transport Protocol - HTTP . . . 133
12.2 The World's Simplest Web Browser . . . 134
12.3 Retrieving an image over HTTP . . . 135
12.4 Retrieving web pages with `urllib` . . . 137
12.5 Parsing HTML and scraping the web . . . 138
12.6 Parsing HTML using regular expressions . . . 139
12.7 Parsing HTML using BeautifulSoup . . . 140
12.8 Reading binary files using urllib . . . 142
12.9 Glossary . . . 143
12.10 Exercises . . . 144

13 Using Web Services **145**

13.1 eXtensible Markup Language - XML . . . 145
13.2 Parsing XML . . . 145
13.3 Looping through nodes . . . 147
13.4 JavaScript Object Notation - JSON . . . 147
13.5 Parsing JSON . . . 148
13.6 Application Programming Interfaces . . . 149
13.7 Google geocoding web service . . . 150
13.8 Security and API usage . . . 153
13.9 Glossary . . . 157
13.10 Exercises . . . 158

14 Object-Oriented Programming **159**

14.1 Managing Larger Programs . . . 159
14.2 Getting Started . . . 159
14.3 Using Objects . . . 160
14.4 Starting with Programs . . . 161
14.5 Subdividing a Problem - Encapsulation . . . 163
14.6 Our First Python Object . . . 164
14.7 Classes as Types . . . 166
14.8 Object Lifecycle . . . 167
14.9 Many Instances . . . 168
14.10 Inheritance . . . 169
14.11 Summary . . . 170
14.12 Glossary . . . 171

15 Using databases and SQL **173**

15.1 What is a database? . . . 173
15.2 Database concepts . . . 173
15.3 Database Browser for SQLite . . . 174
15.4 Creating a database table . . . 174
15.5 Structured Query Language summary . . . 177
15.6 Spidering Twitter using a database . . . 179
15.7 Basic data modeling . . . 184
15.8 Programming with multiple tables . . . 185

15.8.1 Constraints in database tables 188
15.8.2 Retrieve and/or insert a record 189
15.8.3 Storing the friend relationship 190
15.9 Three kinds of keys 191
15.10 Using JOIN to retrieve data 191
15.11 Summary 194
15.12 Debugging 194
15.13 Glossary 195

16 Visualizing data **197**

16.1 Building a Google map from geocoded data 197
16.2 Visualizing networks and interconnections 199
16.3 Visualizing mail data 202

A Contributions **207**

A.1 Contributor List for Python for Everybody 207
A.2 Contributor List for Python for Informatics 207
A.3 Preface for "Think Python" 207
A.3.1 The strange history of "Think Python" 207
A.3.2 Acknowledgements for "Think Python" 208
A.4 Contributor List for "Think Python" 209

B Copyright Detail **211**

Chapter 1

为什么要学编程?

编程是一项极具创造性和有益的活动。编程的原因很多，大到为谋生去解决一个困难的数据分析问题，小到因为帮助别人解决一个问题而获得快乐。本书假定**每个人**都需要知道怎样编程，一旦学会编程，你就会想要用这个新技能做些什么了。

我们的日常生活中计算机无处不在，大到笔记本电脑，小到手机。这些计算机可视为帮助我们打理很多事情的“私人助理”。　在本质上，如今计算机硬件的构建，就是在不断地问我们一个问题，即“你（用户）想让我（计算机）下一步做什么”。

程序员在硬件之上添加了操作系统和应用程序，我们手中拿到的成品是一个很有用的个人数字助理（PDA， Personal Digital Assistant），它能够帮我们处理很多不同的事情。

计算机运行速度很快并拥有大量的内存，如果我们学会了与计算机沟通的语言，告诉计算机我们想要它“接下来做什么”，那么它就会对我们有非常大的帮助。我们能让计算机根据我们的意愿完成一些重复性工作。有趣的是，计算机能够胜任并且做得很好的工作通常是那些让我们人类感到无聊、令人头脑麻木的事情。

例如，阅读本章的前三段，找出出现频率最高的词是哪一个，以及这个词总共出现了多少次。尽管你能在短时间内阅读和理解这些文字，但要对它们进行统计就很痛苦了，因为这类问题不是人的大脑擅长解决的。计算机恰好相反，它很难像人一样阅读和理解一段文字，但是对文字进行统计并告诉你出现频率最高的词及其出现次数，对计算机而言是非常容易的:

```
python words.py
Enter file:words.txt
to 16
```

“个人信息分析助理”很快告诉我们，单词“to“在本章前三段中一共出现了16次。

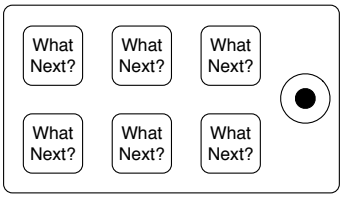


Figure 1.1: 个人数字助理

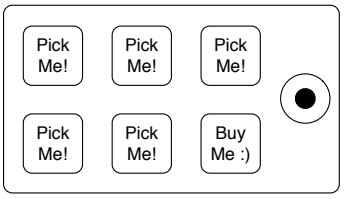


Figure 1.2: 程序员在对你说

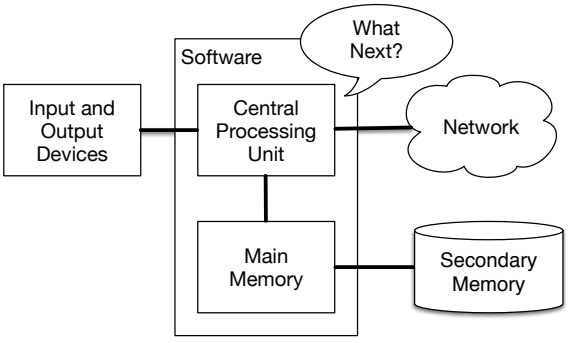


Figure 1.3: 硬件架构

事实上，计算机擅长做人类不擅长做的事，这就是为什么你需要熟练掌握一门“与计算机对话的语言”。一旦学会这门新语言，你就可以将枯燥的工作指派给你的搭档（计算机）了。留出更多的时间去做适合你自己的事。在这种合作关系中，你的贡献是才思、直觉力和创造力。

1.1 创新与动机

这本书不是为专业程序员准备的，专业编程是份非常有前途的工作，可算是物质与精神双丰收。为他人创造有用的、简洁的与智能的程序是一项创新性很强的活动。你的计算机或PDA通常安装了来自许多不同程序员开发的各种软件，每一款软件都想要吸引你的注意力和兴趣。它们尽其所能来满足你的需求，在使用过程中让你获得优质的用户体验。在某些情况下，当你选择了一个软件，这个软件的开发者就会因为你的选择而直接获得收益。

如果将程序看作是一群程序员的创造性产出，那么下图就是一个更形象的PDA模型：

本书的写作初衷不是为了赚钱或者取悦最终用户，而是让我们能更好地处理生活中的数据与信息。开始学编程，你既是程序员，也是你所写程序的最终用户。当你获得了程序员的技能，如果编程让你感到有创新活力的话，到时你的想法也许会发生改变，转向为他人开发程序也说不定。

1.2 计算机硬件架构

学习这种向计算机发指令来开发软件的语言之前，我们需要了解一下计算机的构成。如果拆开你的计算机或者手机，仔细观察就会发现以下这些组件：

这些组件的一般定义如下：

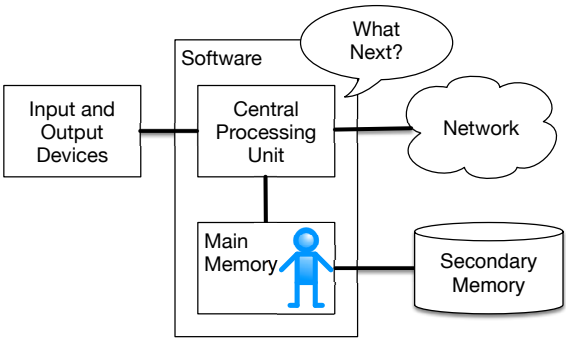


Figure 1.4: Where Are You?

- **中央处理器**（Central Processing Unit，CPU）是专门为解决“下一步做什么”而存在的组件。如果计算机处理速度达到3.0 GHz，这就意味着CPU每秒会提问30亿次“下一步做什么？”。你不得不学会如何跟CPU如此快速地交谈与保持同步。
- **主存储器**（Main Memory）用来存储CPU即刻需要的信息。主存储器的速度几乎与CPU不相上下。但是，关闭计算机之后主存储器里的信息也就消失了。
- **辅助存储器**（Secondary Memory）也是用来存储信息的，但是它比主存储器速度慢很多。辅助存储器的优点是，它可以在计算机不带电情况下存储信息。常见辅助存储器包括磁盘和闪存。闪存通常用在U盘和便携式音乐播放器上。
- **输入输出设备**（Input and Output Devices）包括屏幕、键盘、鼠标、麦克风、扬声器以及触摸板等。这些都是用来与计算机进行交互的设备。
- 如今大多数计算机之间还建立了**网络连接**，通过网络获取信息。我们可以将网络看成信息存储与检索速度很慢的一个空间，而且不总是那么稳定。从某种意义上讲，网络是速度很慢且并不是那么可靠的**辅助存储器**。

这些组件的工作原理细节最好还是交给计算机厂商吧。这里只是为了掌握一些术语，在编程时方便提及这些组件。

作为一名程序员，你的工作就是利用并协调这些资源来解决问题和分析数据。作为程序员，你主要与CPU打交道，告诉它下一步做什么。有时，你要告诉CPU调用主存储器、辅助存储器、网络或输入输出设备。

你需要成为回答CPU“下一步做什么”的人。但把你压缩到5毫米高，塞入计算机，让你每秒发出3亿次命令，想必这样会很不舒服。所以，你必须提前写好你的指令。我们把这些存储下来的指令称为**程序**，编写指令并进行调试的活动称之为**编程**。

1.3 理解编程

在本书其他章节中，我们尝试把你培养成长为一名理解编程艺术并具备一定编程能力的人。最后，你会成为一个**程序员**，也许不是专业的。但至少你掌握了如何看待数据（信息）分析问题，并开发出解决问题的程序。

从某种意义上来说，程序员的养成需要两种技能：

- 首先，需要掌握编程语言（Python）本身—— 熟悉词汇和语法。能够准确地拼写这门新语言中的单词，并且掌握如何使用这门新语言正确地“造句”。
- 其次，学会讲故事。在写故事的过程中，通过文字和句式的组合，向读者传达思想。编故事的艺术与能力通过写作与反馈得以提高。在编程中，程序即故事，待解决的问题即传达的想法。

当掌握一种编程语言（如Python）之后，你会发现学习其他编程语言，如JavaScript或者C++，就会容易许多。虽然新的编程语言拥有很多不同的词汇和语法，但你已经学会解决问题的技能，所有编程语言本质上都是相通的。

Python的词汇和句式上手很快，但要能写出一些连贯的程序来解决一个全新的问题，尚需时日磨练。讲授编程就像讲授写作一样。先对程序进行阅读和解释，然后编写简单的程序，接着逐步编写更复杂的程序。当达到一定水平，你就形成了自己的编程风格，自然而然地去应对问题，通过编写程序解决它。一旦修炼到这个程度，编程就变成一个愉悦且富有创造力的过程了。

我们从Python程序的词汇和结构讲起。第一次阅读时，一定要耐心学习那些简单的例子。

1.4 词汇与句子

与人类语言不同的是，Python的词汇数量实际上相当少。我们称这些“词汇”为“保留字”，它们是Python中具有特殊意义的词汇。对于Python来说，程序中出现的这些词汇，它们有且仅有一个含义。等下你在编程时，你自己定义的词汇称为**变量**。变量命名非常自由，但有一点，你不能使用Python的保留字作为变量名。

从某种意义上讲，我们训练一只狗时会使用一些特殊的词汇，比如“坐下”、“停下”和“拿来”。跟狗说话时不用这些保留字的话，它们就会傻傻地看着你，直到你对它说出保留字。举例来说，“我希望更多的人通过散步来促进健康。”，而大多数狗听到的可能是，“吧啦吧啦**散步**吧啦吧啦。”这是因为在狗的语言中“散步”是保留字。很多人可能觉得人类和猫之间的语言没有保留字[1].

Python的保留字如下:

```
and       del       global      not       with
as        elif      if          or        yield
assert    else      import      pass
break     except    in          raise
class     finally   is          return
continue  for       lambda      try
def       from      nonlocal    while
```

就这么多词汇。Python比狗训练有素多了。当你说“try”，Python会毫无差错地执行try。

后续章节会介绍这些保留字及它们的适用场合。现在，我们只关注怎么与Python**对话**（就像人跟狗说话）。教Python说话是件有意思的事情，把想要说的话用引号括起来就可以了。

```
print('Hello world!')
```

这就是我们写出的第一个语法正确的Python语句。我们的句子以函数 **print** 开头，后面跟一个文本字符串，用单引号括起来。

[1] http://xkcd.com/231/

1.5 与Python对话

我们已经掌握了Python的一个词汇与一个简单语句，接下来需要了解如何与Python对话，测试我们的新语言技能。

与Python对话之前，必须先在计算机上安装Python软件，学会如何启动Python。本章包含许多细节，建议查看 www.py4e.com，网站上有Python在Mac和Windows系统上配置和启动的详细说明和视频演示。当打开终端或者命令行窗口，输入python，Python解释器会以交互模式启动，如下所示：

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:54:25)
[MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

‘>>>’提示符表示Python解释器在询问，“你希望我下一步做什么？”。Python已经准备好与你对话。你需要掌握的是怎样说Python语言，发起一个对话。

举个例子，你对Python语言最简单的词汇或句子一无所知，想要使用宇航员的标准用语（喊话）。宇航员在一个遥远的星球登陆，试着和这个星球的居民用以下语句对话：

```
>>> I come in peace, please take me to your leader
  File "<stdin>", line 1
    I come in peace, please take me to your leader
         ^
SyntaxError: invalid syntax
>>>
```

事情进展好像并不顺利。除非你反应迅速，否则这个星球的居民可能会拿长矛刺向你，向你吐口水，然后把你放在火上烤，当成晚饭吃掉。

幸运的是，旅行时你带了这本书，及时翻到了这一页，再试一次：

```
>>> print('Hello world!')
Hello world!
```

这次看起来效果不错，试着与他们继续对话：

```
>>> print('You must be the legendary god that comes from the sky')
You must be the legendary god that comes from the sky
>>> print('We have been waiting for you for a long time')
We have been waiting for you for a long time
>>> print('Our legend says you will be very tasty with mustard')
Our legend says you will be very tasty with mustard
>>> print 'We will have a feast tonight unless you say
  File "<stdin>", line 1
    print 'We will have a feast tonight unless you say
                                                      ^
SyntaxError: Missing parentheses in call to 'print'
>>>
```

上面的对话一直进展顺利，只到你在使用Python语言的时候犯了一个非常小的错误，Python就又把长矛拿出来了。

此时，你应该意识到，Python虽然非常复杂与强大，但在语法上非常挑剔，并**不**那么智能。对话中必须使用正确的语法。

在某种意义上，当你使用别人写的程序时，Python就在你和其他程序员之间充当中间人。Python是程序编写者将对话进行下去的一种方式。在阅读完短短几章之后，你将成为Python程序员中的一员，与你的程序使用者进行对话。

结束与Python解释器的第一次谈话之前，你可能要知道如何正确地与这个星球的居民说“再见”：

```
>>> good-bye
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'good' is not defined
>>> if you don't mind, I need to leave
  File "<stdin>", line 1
    if you don't mind, I need to leave
              ^
SyntaxError: invalid syntax
>>> quit()
```

你会发现前两个错误提示是不同的。由于 **if** 是保留字，Python看到保留字会认为我们想说些什么，但句子的语法是错的。

跟Python说“再见”的正确方法是，在交互模式的提示符’>>>’后输入 **quit()**。猜出这个命令这可能会花费一些时间，所以手头备本书可能会派上用场。

1.6 术语：解释器与编译器

Python是一种**高级语言**，旨在较为方便地让人类进行读写，让计算进行读取与处理。其他高级语言包括：Java、C++、PHP、Ruby、Basic、Perl以及JavaScript等。CPU里的硬件并不能理解任何一种高级语言。

CPU能理解的语言称之为**机器语言**。机器语言非常简单，坦白讲，编写起来非常无聊。它全部由0和1组成：

```
00101000111010010010101010000001111
11100110000011101010010101101101
...
```

虽然机器语言表面看起来很简单，只有0和1，但它的语法比Python复杂得多。所以，很少有程序员用机器语言编程。相反，借助各种翻译器，程序员可以编写像Python或JavaScript这样的高级语言，这些翻译器会将程序转换成机器语言，再交由CPU执行。

因为机器语言依附于计算机硬件，所以不能在不同类型硬件之间移植。使用高级语言编写的程序可以在不同的计算机之间移植，通过在另一台计算机上使用不同的编译器，或者重新编译代码，生成一个适合这台计算机的机器语言版本。

编程语言的翻译器大体可分为两类：(1)解释器 与 (2) 编译器。

解释器读取程序员所写程序的源代码，解析源代码并实时解释指令。Python是一种解释器。当交互式执行Python时，输入一行Python语句，Python就会立即处理它，并做好准备让我们输入下一条语句。

Python语句中有一些地方会告诉Python，你想要Python记住等下会用到的一些数据。这时就需要为数据挑选一个名称来记住它，这样之后就可以通过这个名称来获取对应的数据了。我们使用**变量**（variable）来代表存储的数据。

```
>>> x = 6
>>> print(x)
6
>>> y = x * 7
>>> print(y)
42
>>>
```

在这个例子中，我们让Python记住数值6，并将6赋值给变量 **x**，以便后续使用。为了确认Python已经记住这个数值，使用 **print** 命令打印它。接下来，我们让Python获取变量 **x** 的值并乘以7，然后将结果赋给新变量 **y**。最后，打印出变量 **y** 的当前值。

尽管我们一次只输入了一行命令，但Python将其视为一个语句序列，后面的语句可以获取前面语句的数据。四句组成的段落以一种有逻辑的和有意义的顺序编写，这就是我们写出的第一个简单的多行程序。

如上所示，解释器的本质是进行交互式对话。而编译器需要将整个程序放在一个文件中，将高层次的源代码翻译成低层次的机器语言，然后编译器将生成的机器语言放到一个文件中以便后续执行。

如果你使用Windows系统，这些可执行的机器语言程序通常带有“.exe”或“.dll”后缀，分别代表这是“可执行的”和“动态可加载库”。在Linux和Mac中没有这样的后缀来明确表示文件是否是可执行的。

如果在文本编辑器中打开一个可执行文件，满眼望去完全看不懂：

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^F^@
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H^S
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
....
```

机器语言的读写并不容易，好在借助**解释器**和**编译器**，能够使用Python或C这样的高级语言编写程序。

通过对解释器与编译器的讨论，你应该对Python解释器本身有了一些了解。你应该想问那它又是用什么语言写的？是用编译语言写的吗?当我们输入“python”，究竟发生了什么?

Python的解释器是用C语言编写的。你可以访问http://www.python.org网站，查看Python解释器的源代码，如有你有意愿改造这些源代码也是可以的。Python本身就是一个程序，它被编译成机器代码。当你（或计硬件供应商）在计算机上安装了Python，实际是上将一份编译好的Python程序的机器代码拷贝到你的计算机系统。在Windows中，Python可执行的机器代码很可能位于以下文件夹中：

```
C:\Python35\python.exe
```

要成为一名Python程序员，你并不一定的需要知道上述这些。但在一开始，花一些时间解释这些细节问题，还是值得的。

1.7 编写一个程序

在Python解释器中输入命令是体验Python功能非常好的方式，但不建议采用这种方式来解决复杂的问题。

编程时，我们在文本编辑器里把Python指令写到一个文件里，这个文件称为**脚本**。一般而言，Python脚本以`.py`命名结尾。

要执行脚本，你必须告诉Python解释器脚本文件的名称。在Unix或Windows命令窗口中，你可以像下面这样执行`python hello.py`:

```
csev$ cat hello.py
print('Hello world!')
csev$ python hello.py
Hello world!
csev$
```

“csev$”是操作系统提示符，“cat hello.py”是查看“hello.py”文件的内容，其中包含了一行打印字符串的Python程序。

我们调用Python解释器，告诉它从“hello.py”文件中读取源代码，而不是用命令行交互式一行一行地执行Python代码。

你会发现，没有必要在Python程序文件末尾加上 **quit()**。Python在读取源代码文件时，到达文件末尾它会自己停止。

1.8 什么是程序?

程序的基本定义是，完成特定任务的一组Python语句序列。就算是前面简单的 **hello.py** 脚本也是一个程序，不过只是一行代码的程序罢了，作用并不大。不过从最严格的定义上来说，它是一个Python程序。

思考一个可以被程序解决的问题，然后看一看解决这个问题的程序，这可能是理解程序的最简单方式。

假设，你想对Facebook上的发帖进行社会计算方面的研究，并且你感兴趣的问题是一系列帖子中最常用的词汇是什么。你可以打印出这些发帖，然后通读文本，从中寻找最常见的词，但这需要很长时间而且很容易出错。你也可以通过编写Python程序来快速且准确地处理这个任务，这样会比较明智，周末你就可以做些其他有趣的事了。

举例来说，阅读以下内容，这是关于一个小丑和一辆车的文本，找出出现次数最多的单词，并统计它出现的次数。

```
the clown ran after the car and the car ran into the tent
and the tent fell down on the clown and the car
```

然后想象一下，你将要做的是对数百万行文本来完成这个统计任务会。坦率地说，学习Python，编写一个Python程序来统计的话，要比人工查看单词快得多。

一个更好的消息是，我已经写了一个简单的程序来在文本文件中找到最常见的单词。我编写并测试了它，现在，我把这个程序给你，这样就可以节省你的一些时间。

```
name = input('Enter file:')
handle = open(name, 'r')
counts = dict()

for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None
for word, count in list(counts.items()):
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)

# Code: http://www.py4e.com/code3/words.py
```

你甚至不需要知道Python就可以使用这个程序。你需要通读到本书第10章来完全理解所用到的Python编程技术。你现在是最终用户，只需使用这个程序，你就会惊叹于它的聪明，感叹如何让你摆脱繁重的人工查看。你只需输入代码，保存到 **words.py** 文件并执行它，或者你也可以从http://www.py4e.com/code3/下载源码并运行。

这个示例充分体现了Python以及这门语言在你（最终用户）与我（程序员）之间扮演的中间人角色。通过Python这门通用语言，我们可以交流有用的指令序列（即程序），这样任何人在电脑里面安装了Python的人都可以使用它。实际上，交流双方并不是**跟Python**交流，而是**通过Python**与对方交流。

1.9 构成程序的小积木

在接下来的几章中，我们将会学习更多Python的词汇、句子结构，段落结构，和故事结构。我们将学习Python强大的功能，以及如何将这些功能组合起来创建有用的程序。

程序的构造包含一些低层次的概念模式。这些构造模式不仅仅针对Python程序，而是每一种编程语言，从机器语言到高级语言，都通用的组成部分。

输入 从 “外部世界”获取数据，可以是从文件中读取数据，或者从某种传感器，比如麦克风或GPS获取数据。在我们最初的程序中，输入是用户通过键盘输入的数据。

输出 将程序的结果显示在屏幕上，或保存在一个文件，或写入一个设备，如扬声器来播放音乐或朗读文字。

顺序执行 按照脚本中的语句顺序，一句接一句执行。

条件执行 根据特定条件执行或者跳过特定语句序列。

重复执行 重复执行一些语句，这些语句在每次执行时通常会有些许变化。
重用 编写好一组指令，并为它们命名，之后在整个程序中根据需要重复使用这些指令。

这听起来太简单了以至于大家不会相信，然而事实上并不那么简单。这就好比，走路很简单吧，只要“把一只脚放在另一只脚前面”就好了。编程的“艺术”就是不断地编织、组合这些基本元素，来创造对用户有用的东西。

除了“重用”模式之外，上面的词频统计程序几乎用到了上面提及的所有模式。

1.10 什么可能会出错?

从前面与Python的对话中可以看到，我们编写Python代码时必须非常精确，很小的偏差和错误都会导致Python放弃执行程序。

初学者通常认为，Python不能容忍犯错，给人留下刻薄、可恨与粗暴的印象。虽然Python看上去与每个人一样，但它知道每个人的个性并且对他们持有一种怨恨的态度。由于这种怨恨，Python拒绝执行我们写得非常好的程序，把他们称之为“不合格”来折磨我们。

```
>>> primt 'Hello world!'
  File "<stdin>", line 1
    primt 'Hello world!'
                       ^
SyntaxError: invalid syntax
>>> primt ('Hello world')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'primt' is not defined

>>> I hate you Python!
  File "<stdin>", line 1
    I hate you Python!
         ^
SyntaxError: invalid syntax
>>> if you come out of there, I would teach you a lesson
  File "<stdin>", line 1
    if you come out of there, I would teach you a lesson
               ^
SyntaxError: invalid syntax
>>>
```

与Python争论并没什么好处。它只是一个工具，没有情感，并且它很高兴随时准备为你服务。它的错误信息看起来很苛刻，但这只是Python在寻求帮助。Python看到了你输入的内容，它只是不理解你输入的是什么意思。

Python更像一只狗，无条件爱你，只能理解很少的关键词，一直用它那甜美的表情(>>>)看着你，等待你输入一些它能理解的东西。当Python说道：“SyntaxError: invalid syntax”（语法错误：无效语法），它只是在摇着尾巴说：“你似乎说了什么，但我不太明白，不过希望你继续跟我说话(>>>)。”

当程序变得越来越复杂，你会遇到以下三种一般类型的错误：

语法错误 这是你遇到的第一种错误，很容易解决。语法错误意味着，你违反了Python的“语法”规则。Python会尽其所能地指出它不能理解的那一行和相应字符。语法错误唯一棘手的是，有时候程序中需要修改的错误可能位于Python**指出**的困惑的位置之前。所以，Python指出的语法错误位置可能只是你排查问题的起点。

逻辑错误 逻辑错误是在程序没有语法错误的前提下，语句顺序或语句关系存在错误。逻辑错误的一个形象例子是，“打开水瓶喝水，把它放到书包里，走到图书馆，然后再把水瓶盖上。”

语义错误 语义错误是程序的语法完美且逻辑正确，但就是有一个错误。也就是说，程序完全正确，但它不能做到你**想要**它做的事。一个简单的例子，如果你给人指路怎么去一家餐馆，“...当你走到有加油站的十字路口时，向左转，继续走一英里，你的左手边有一栋红色建筑，餐馆就在那儿。”过了很长时间，你的朋友打来电话，他们正在一个农场，在一个谷仓附件转悠，并没有看到餐馆的标志。”然后，你问：“你们在加油站左转还是右转了？”他们说：“完全是按你指示的方向走，我还写到纸上，在加油站左转，继续走一英里”。然后，你说：“非常抱歉，虽然我的方向指示在语法上没错，但其中悲催地包含了一个很小的、没有被发现的语义错误。”

重申一下，对于所有这三种错误，Python只是在尽最大努力按照你的要求准确地去执行。

1.11 学习之旅

当阅读本书时，刚开始如果遇到某些概念不能很好理解时，不要担心。小时候学说话时，头几年只能发出一些可爱的喃喃之音，这不是什么问题。花6个月的时间，从能说简单的词汇发展到能表达简单的句子；再用5到6年的时间，从句子上升到段落；再过几年能依据自身兴趣，独立写出一篇有趣而完整的小故事，这样的发展过程是正常的.

我们希望你能更快地学习Python，所以我们会在接下来的几章中把它们全部教给你。

不过学习一门新语言需要花时间去吸收和理解，然后才能做到运用自如。因此可能会给你带来一些疑惑，因为我们会定义很多小的片段，它们之后会拼接成一副全景。而为了尝试让你看到全景，我们会一再提及一些片段。由于本书的编写是线性的，并且如果你参加一门课，其推进方式是线性的话，你完全也可以以一种非线性的方式去学习。可以前后来回翻阅，蜻蜓点水式的阅读，可以浏览一些有难度的内容，但并不一定要理解所有的细节，这样有助于更好的理解编程之“道”。尽管你正在阅读的地方看起来可能有些费解，但通过回顾之前的内容，或者重做之前做过的练习，你将会感觉收获颇多。

通常在学习第一门编程语言时，会有一些值得欢呼雀跃的时刻。这就像你在用斧凿精心雕琢一块岩石，然后后退了几步，发现你真的在建造一尊美丽的雕塑。

如果有些事看起来特别困难，通宵熬夜耗着是没有意义的。休息一下，打个盹，吃点零食，向某人（或许是你的狗）倾诉下你当下遇到的问题，然后，以全新的眼光回过头来再看这个问题。我保证，一旦你从本书中学会了编程的概念，回头看时，你会发现编程真的是非常简单、优雅的，只是需要花一些时间去吸收罢了。

1.12 术语表

漏洞 程序中的错误。

中央处理器 所有计算机的心脏。我们编写的软件都由它来执行，也称为“CPU”或者“处理器”。

编译 把高级语言编写的程序一次性翻译成低级语言，为后续执行做好准备。

高级语言 易于人类阅读和编写的编程语言，如Python。
交互模式 Python解释器的一种使用方法，即在提示符后输入命令和表达式。
解释 采用一次翻译一行的方式来执行高级语言编写的程序。
低级语言 一种旨在便于计算机执行的编程语言，也称为“机器代码”或“汇编语言”。
机器代码 最低级的软件编写语言，可直接由中央处理器（CPU）执行。
主存储器 存储程序和数据。关闭电源后主存储器的信息会丢失。
解析 检查程序和分析语法结构。
可移植性 程序的一个属性，即程序可在不同类型的计算机上运行。
print函数 能让Python解释器在屏幕上显示数据的指令。
问题解决 描述问题、寻找解决方案、描述解决方案的过程。
程序 实现特定计算的一组指令集。
提示 程序显示一个消息，等待用户的输入。
辅助存储器 存储程序和数据，电源关闭后数据不会丢失。辅助存储器的速度通常比主存储器慢。辅助存储器的例子有磁盘驱动器、U盘中的闪存等。
语义 程序的含义（程序要做的事情，译者注）。
语义错误 程序的一种错误。即程序并未按照程序员的意愿做事。
源代码 程序的高级语言代码。

1.13 习题

习题 1: 计算机中的辅助存储器的功能是什么?

a) 执行程序的所有计算和逻辑
b) 在互联网上检索网页
c) 长期存储信息，就算重启之后信息也不会丢失
d) 接收用户的输入

习题 2: 什么是程序?

习题 3: 编译器和解释器有什么区别?

习题 4: 下面哪一个含有“机器代码”？

a) Python解释器
b) 键盘
c) Python源文件
d) 文本文档

习题 5: 请找出下面代码的错误:

```
>>> primt 'Hello world!'
  File "<stdin>", line 1
    primt 'Hello world!'
                       ^
SyntaxError: invalid syntax
>>>
```

习题 6: 执行以下Python语句后，变量“x”存于何处?

```
x = 123
```

a) 中央处理器
b) 主存储器
c) 辅助存储器
d) 输入设备
e) 输出设备

习题 7: 以下程序会输出什么?

```
x = 43
x = x + 1
print(x)
```

a) 43
b) 44
c) x + 1
d) 报错，因为x = x + 1在数学上讲不通

习题 8: 以人作类比，解释以下事物： (1) 中央处理器, (2) 主存储器, (3) 辅助存储器, (4) 输入设备, 和 (5) 输出设备. 例如，“计算机的中央处理单元相当于人体哪个部位”?

习题 9: 如何解决一个“语法错误”?

Chapter 2

变量、表达式、语句

2.1 值与类型

值 是程序要处理的一个基本要素，如一个字母或一个数字。目前为止，我们接触到的值有1、2和“Hello, World!”。

这些值属于不同的**类型**：2是整数，“Hello, World!”是**字符串**（因包含一“串”字母而得名）。因为字符串都在引号当中，你（以及解释器）可以根据引号来识别它们。

`print`语句也可以打印整数。输入python命令启动解释器。

```
python
>>> print(4)
4
```

如果不确定一个值属于哪种类型，可以用解释器来确定。

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
```

显而易见，字符串属于`str`类型，整数属于`int`类型。需要注意的是，带小数点的数字使用**浮点**（floating-point）格式表示，称为`float`类型。

```
>>> type(3.2)
<class 'float'>
```

那么，像’17’和’3.2’这种属于哪种类型呢？看起来像数字，但它们和字符串一样被放在单引号里面。

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

它们是字符串。

输入较大的数字时，你可能会在每三个数字之间加一个逗号，例如，`1,000,000`。在Python中这不是一个合法的整数，但这句话是合法的：

```
>>> print(1,000,000)
1 0 0
```

不过，这根本不是我们想要的！Python把`1,000,000`解释成了一个逗号分隔的整数序列，它把三部分依次打印出来了，中间用空格分隔。

这是我们遇到的第一个语义错误例子：代码成功运行，没有任何错误信息，但是它并没有做“正确的事“。

2.2 变量

编程语言最强大的功能之一体现在对**变量**的操控能力。变量是指向一个值的名称。

赋值 语句用来创建新变量并对其赋值：

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

这个例子列举了三个赋值语句。第一条语句将字符串赋值给变量`message`；第二条语句将整数`17`赋值给变量`n`，第三条语句将 π 的(近似)值赋值给变量`pi`。

你可以使用打印语句来显示一个变量的值：

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

变量的类型就是它所指向的值的类型。

```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

2.3 变量名与关键字

程序员通常会选择有意义的变量名，并且在说明书中写明其用途。

变量名不限长度，可以同时包含字母和数字，但是不能以数字开头。使用大写字母也是合法的，但以小写字母开头会更好（之后你会明白原因）。

下划线（_）可以出现在变量名中。它经常用在含有多个词的变量名中，例如，my_name和airspeed_of_u
变量名可以采用下划线开头，但我们一般会避免这样命名，除非是在编写供他人使用的Python库
代码。

如果使用不合法的变量名，你就会遇到一个语法错误：

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

76trombones是不合法的变量名，因为它是以数字开头的。more@也是不合法的，因为它包含了一个不合法的字符@。不过变量名class错在哪呢？

原因在于，class是Python的**关键字**。Python解释器使用关键字来识别程序的结构，因此，关键字不能用作变量名。

Python保留了31个关键字：

```
and       del       from      None      True
as        elif      global    nonlocal  try
assert    else      if        not       while
break     except    import    or        with
class     False     in        pass      yield
continue  finally   is        raise
def       for       lambda    return
```

你可以在手边存留一份。如果解释器在一个变量名那里报错，而你又不知道为什么，那么检查一下它是否在这个列表里面。

2.4 语句

语句 是Python解释器能够执行的代码单元。我们已经见到过两种语句：print和assignment。

当你在交互模式中输入一条语句，解释器就会执行它并打印出结果（如果有结果的话）。

一个脚本通常包含一连串的语句。如果超过一句的话，结果会随着程序的执行，一句一句地产生。

如，以下脚本：

```
print(1)
x = 2
print(x)
```

产生下列结果：

```
1
2
```

其中，赋值语句没有输出结果。

2.5 运算符和运算对象

运算符 是表示运算的特殊符号，例如，加法与乘法。运算符操作的值称为运算对象(或运算数，译者注)。

+、-、*、/和**五个运算符分别代表加、减、乘、除和次方的运算，请看如下示例：

```
20+32    hour-1    hour*60+minute    minute/60    5**2    (5+9)*(15-7)
```

对于除法运算符而言，Python 2.x 版本和 Python 3.x 版本有一点区别。在 Python 3.x 中，除法的结果是一个浮点数：

```
>>> minute = 59
>>> minute/60
0.9833333333333333
```

在 Python 2.0 中，两个整数相除，会得到一个被截断的整数（0.59，会被截断成0，译者注）：

```
>>> minute = 59
>>> minute/60
0
```

要想在 Python 3.0 中得到和上面一样的结果，要使用地板除法（ // 整数 ）。

```
>>> minute = 59
>>> minute//60
0
```

在 Python 3.0 中，整数相除的结果会像你所期望的那样，和你使用计算器得到的结果一样。

2.6 表达式

表达式 是值、变量和运算符的组合。值本身可以是一个表达式，变量亦如此。所以下面都是合法的表达式（假设变量x已被赋值）：

```
17
x
x + 17
```

如果在交互模式中输入一个表达式，解释器就会**运算**它并把结果打印出来：

```
>>> 1 + 1
2
```

然而，在一个程序中，表达式本身并不能做任何事情！这是初学者容易混淆的一点。

习题 1: 在Python解释器中输入下面的语句并查看结果：

```
5
x = 5
x + 1
```

2.7 运算顺序

当一个表达式中出现多个运算符时，运算顺序由优先级规则来确定。对于数学运算符来说，Python遵照数学运算习惯，即“括号、次方、乘除、加减”。

- **括号** 拥有最高运算优先级，可以强制表达式按特定顺序运算。括号内的表达式最先进行运算，例如，`2 * (3-1)` 等于4，`(1+1)**(5-2)` 等于8。有时候，使用括号即便没有改变运算结果，但阅读起来会更加方便，例如，(minute * 100) / 60。
- **幂运算**（次方、乘方）的优先级仅次于括号，例如，`2**1+1` 等于3，而不是4，`3*1**3`等于3，而不是27。
- **乘法** 和 **除法** 具有相同的优先级，**加法** 和 **减法** 也具有相同的优先级，且乘除高于加减。所以，`2*3 -1`等于5，而不是 4，`6+4/2`等于8.0，而不是5。
- 相同优先级的运算符按从左到右的顺序依次运算。所以，`5-3-1`等于1，而不是3。先计算`5-3`得到2，然后再减1。

当不能确定运算顺序时，通常使用括号来确保我们想要的运算顺序。

2.8 模运算

模 的运算对象是整数，得到的是第一个整数除以第二个整数的余数。在Python中，模运算符用百分号（`%`）表示，语法与其他运算符一样：

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

如，7被3所除的商是2，余数是1。

模运算非常实用。举例来说，你可以检验一个数是否能被另一个数整除，如果`x%y`的结果是0，那么`x`能被`y`整除。

另外，模运算也可以提取一个数字最右边的数位。举例来说，`x%10`可以提取`x`最右边的一位数字（以10为基数）。同理，`x%100`可以提取最右边的两位数字。

2.9 字符串运算符

加号 `+` 也可以操作于字符串，但是这时它不是数学里面加法的含义，而是，它会把字符串首尾**串联**。例如：

```
>>> first = 10
>>> second = 15
>>> print(first+second)
```

```
25
>>> first = '100'
>>> second = '150'
>>> print(first + second)
100150
```

这个程序的输出结果是100150。

2.10 请求用户输入

有时候我们希望获取用户通过键盘输入的值。Python提供了一个内置函数叫input，用来获取键盘输入[1]。当调用这个函数时，程序会暂停运行，等待用户的输入。当用户按下 (Return或Enter)时，程序就会恢复运行，input函数会以字符串形式返回用户输入的值。

```
>>> input = input()
Some silly stuff
>>> print(input)
Some silly stuff
```

在请求用户输入之前，最好打印一条提示语句，告诉用户需要输入些什么。你可以通过在input中插入一个字符串来提示用户。

```
>>> name = input('What is your name?\n')
What is your name?
Chuck
>>> print(name)
Chuck
```

提示语结尾的 \n 表示**换行符**，它是一个用于截断当前行，并开始下一行的特殊字符。这样一来，用户输入的位置就在提示语句的下面。

如果希望用户输入一个整数，你可以尝试用int()函数将返回的值转换成**整数**型:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
>>> int(speed) + 5
22
```

但是，如果用户输入的不是由数字组成的字符串，那么就会报错:

[1] 在Python 2.0中，这个函数叫做raw_input

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
```

之后，我们会学习该如何处理这类错误。

2.11 注释

当程序变得越来越长并且越来越复杂时，阅读难度也随之增大。正式的程序代码很密集，经常会遇到看不懂这段代码是做什么的，或者为什么要这样写。

为解决这个问题，在程序代码中加入自然语言说明，来解释这段代码的作用，这会是一个不错的主意。这些说明称为**注释**，它们以#号开头：

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

上面这种情况，注释本身占一行。你也可以把它加到一行代码的末尾：

```
percentage = (minute * 100) / 60      # percentage of an hour
```

从\#号开始到这一行的最后，解释时都会被忽略掉，它们不会对程序产生任何影响。

对代码不显著的特征进行注释是非常有用的。我们可以合理假设读者能够理解代码在**做什么**，但是更有用的是，解释一下**为什么**。

下面这行注释就是多余的，没什么作用：

```
v = 5      # assign 5 to v
```

而下面的这行注释则包含了有用的信息，是单纯地看代码看不出来的：

```
v = 5      # velocity in meters/second.
```

清晰易懂的变量名能够减少注释的使用，但是变量名如果太长，就会使复杂的表达式变得更加难懂，所以需要权衡利弊。

2.12 助记变量命名法

只要遵循变量命名的简单规则，避免使用保留字，你给变量取名字的时候还是有很多种选择的。

编程入门阶段，你在阅读别人的程序和编写自己的程序时，对变量的命名可能会感到困惑。例如，下面三个程序所完成的任务在实质上是一样的，但是阅读和理解起来差别却很大。

```
a = 35.0
b = 12.50
c = a * b
print(c)
```

```
hours = 35.0
rate = 12.50
pay = hours * rate
print(pay)
```

```
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

Python解释器看到这三个程序时，会觉得是**完全一样**的。但是对于人而言，阅读和理解它们却是非常不一样的。读者能够快速看懂的是第二个程序的**目的**，这是因为该程序员选择了能够代表变量取值含义的变量名。

这种变量命名法称为“助记变量命名法”。助记^[对于“助记”的详细介绍，请参见 http://en.wikipedia.org/wiki/Mnemonic的意思就是帮助记忆。选择易于记忆的变量名，有助于我们记住当初创建这个变量是为了做什么。

这看起来不错，使用助记变量命名法是一个好主意，但可能也会妨碍初学者解析并理解代码。这是由于初学者可能还没有记全Python的33个保留关键字，如果变量名中包含太多描述性的词语，精心命名的变量看上去会像是Python语言的一部分，对初学者理解上造成干扰。

下面两行简单的Python代码实现了循环。循环将在第5章介绍，这里尝试猜猜这两行代码的含义：

```
for word in words:
    print(word)
```

这里发生了什么呢？上面哪些（for, word, in, 等）是保留字，哪些只是变量名呢？Python能理解单词的基本含义吗？初学者很难分辨出代码中哪些部分**必须**照抄示例中的，而哪些部分是可以由程序员自主选择的。

下面的代码和上面的在实质上是一样的：

```
for slice in pizza:
    print(slice)
```

初学者可以较容易的从这段代码中判断哪些是Python定义的保留字，哪些是程序员选择的变量名。Python显然不能理解 pizza（披萨） 和 slices（块） 的含义，更不用说披萨可以被切成很多块这个事实了。

但是，如果我们的程序实际上是要读取数据并在数据中查找单词的话，`pizza` 和 `slice` 就是非常不易记的变量名了。选择它们作为变量名就会偏离程序的本意。

用不了多久，你就会熟悉最常用的保留字，并会在程序中注意到它们：

```
word *in* words*:*\ *print* word
```

这段代码中，由Python定义的部分被加粗了（for，in，print，和 :），程序员选择的变量名（word 和 words）则没有被加粗。很多文本编辑器能感知到Python的语法，并且会用不同的颜色来标记保留字，以让你能够更好的区分变量名与保留字。熟悉一段时间后，你就会很快地区分哪些是变量名，哪些保留字了。

2.13 调试

目前，你最容易犯的语法错误应该是使用了一个不合法的变量名，比如 class 和 yield，它们是保留字，又或者 odd~job 和 US$，它们含有不合法的字符。

如果你在变量名中放一个空格，Python会认为它是没有运算符的两个运算对象：

```
>>> bad name = 5
SyntaxError: invalid syntax

>>> month = 09
  File "<stdin>", line 1
    month = 09
              ^
SyntaxError: invalid token
```

对语法错误而言，错误信息并不能提供多少帮助。最常见的信息是 SyntaxError: invalid syntax 和 SyntaxError: invalid token，而这两条都没什么信息量。

最常遇到的运行错误是“use before def;（定义前就使用）”，也就是，你在尝试使用一个还没有被赋值的变量。变量名拼写不正确就会导致这个错误：

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

变量名区分大小写，所以，LaTeX 和 latex 是不一样的。

目前，最容易犯的语义错误在运算顺序上。比如，要运算 $1/2\pi$，你可能会写成这样：

```
>>> 1.0 / 2.0 * pi
```

而这样的话，就是先进行除法运算，得到的是 $\pi/2$，和你的目的不一样！但是Python并不明白你想要表达的是什么，所以在这种情况下，它并不会报错，但是你得到的答案不是你想要的。

2.14 术语表

赋值 给变量赋予一个值的语句。
串联 将两个运算对象首位相连。
注释 程序里面包含的信息，旨在帮助其他程序员（或任何查看源码的人）理解程序，而不会对程序的执行产生任何影响。
求值 对表达式进行运算，得到单个值。

表达式 变量、运算符和值的组合，表示单个结果值。
浮点数 代表有小数部分的数值类型。
整数 代表整数类型
关键字 Python解释器用来解析程序的保留字。变量名不能使用保留字，如 `if`, `def`, 和 `while` 等。
助记法 一种辅助记忆的方法。我们通常使用易记的变量名来帮助我们记住变量存储的内容。
模运算 一种运算符，用百分号（`%`）表示，求两个整数相除的余数。
运算对象 运算符操作的值。
运算符 代表着简单运算的一类特殊符号，如加法、乘法和字符串串联。
运算优先级 一组运算规则，用来规定在有多个运算符和运算对象的表达式中的求值顺序。
语句 表示一个命令的一段代码。目前为止，我们见到的语句有赋值语句和打印语句。
字符串 由字符序列组成的一种数据类型。
类型 表示一类值。目前，我们已经见到的类型有整数（`int`），浮点数（`float`），和字符串（`str`）。
值 数据的一种基本单元。如一个数字或一个字符串，可以被程序操作。
变量 一个值的引用名称。

2.15 习题

习题 2: 使用 `input` 编写一个程序，提示用户输入姓名，然后打印欢迎语。

```
Enter your name: Chuck
Hello Chuck
```

习题 3: 编写一个程序，提示用户输入工时和时薪，然后计算出总工资。

```
Enter Hours: 35
Enter Rate: 2.75
Pay: 96.25
```

我们暂时不用担心我们计算的结果是否能正好精确到小数点后两位。如果你非常想的话，可以试一试Python内置的 `round` 函数，它可以把结果约到两位数。

Exercise 4: 假设我们执行了下面的赋值语句:

```
width = 17
height = 12.0
```

对于下面每一个表达式，写出它的结果，及其结果类型。

1. `width//2`
2. `width/2.0`
3. `height/3`
4. `1 + 2 \* 5`

使用Python解释器检查你的结果。

Exercise 5: 写一个程序，提示用户输入摄氏温度，然后将其转化成华氏温度，并且把结果打印出来。

Chapter 3

条件执行

3.1 布尔表达式

布尔表达式 是具有真或假状态的一种表达式。下面的例子用运算符==来比较两个运算对象，若两者相等则返回True，否则返回False：

```
>>> 5 == 5
True
>>> 5 == 6
False
{}
```

True和False（ 和 ）是一种特殊的值，属于布尔类型；它们并不是字符串：

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

==是一个**比较运算符**，其他的比较运算符如下：

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
x is y               # x is the same as y
x is not y           # x is not the same as y
```

虽然你可能很熟悉这些运算符，但要注意这些Python符号并不等同于数学符号。一个常见的错误是用了单等号（=），而没有用双等号（==）。请记住，=是赋值运算符，==是比较运算符。不存在=<或=>这样的运算符。

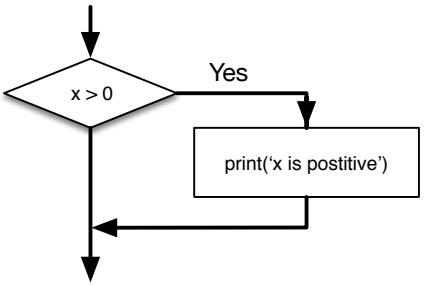


Figure 3.1: If Logic

3.2 逻辑运算符

逻辑运算符 包括and（与）、or（或）与not（非）三个。这些运算符的语义与它们的英文含义相似。例如，

```
x > 0 and x < 10
```

只在x大于0**并且**小于10时为真。

若n%2 == 0 or n%3 == 0其中有一个条件为真，也就是说，这个数字能被2**或者**3整除，那么整句话就为真。

最后一个，not运算符会对一个布尔表达式的结果取反。所以，若x > y为假，也就是x小于或等于y，则not (x > y)为真；

严格讲，逻辑运算符的运算对象应该是布尔表达式，但在Python中并不是很严格。任何非零数字都可看作是“真”。

```
>>> 17 and True
True
```

这种灵活性的存在是有用的，但也会产生一些微妙的困惑。除非你清楚自己在做什么，否则不要乱用。

3.3 Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the `if` statement:

```
if x > 0 :
    print('x is positive')
```

The boolean expression after the `if` statement is called the condition. We end the `if` statement with a colon character (:) and the line(s) after the if statement are indented.

If the logical condition is true, then the indented statement gets executed. If the logical condition is false, the indented statement is skipped.

`if` statements have the same structure as function definitions or `for` loops[1]. The state-

[1] We will learn about functions in Chapter 4 and loops in Chapter 5.

ment consists of a header line that ends with the colon character (:) followed by an indented block. Statements like this are called compound statements because they stretch across more than one line.

There is no limit on the number of statements that can appear in the body, but there must be at least one. Occasionally, it is useful to have a body with no statements (usually as a place holder for code you haven't written yet). In that case, you can use the `pass` statement, which does nothing.

```
if x < 0 :
    pass          # need to handle negative values!
```

If you enter an `if` statement in the Python interpreter, the prompt will change from three chevrons to three dots to indicate you are in the middle of a block of statements, as shown below:

```
>>> x = 3
>>> if x < 10:
...     print('Small')
...
Small
>>>
```

3.4 Alternative execution

A second form of the `if` statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x%2 == 0 :
    print('x is even')
else :
    print('x is odd')
```

If the remainder when `x` is divided by 2 is 0, then we know that `x` is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed.

Since the condition must either be true or false, exactly one of the alternatives will be executed. The alternatives are called branches, because they are branches in the flow of execution.

3.5 Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional:

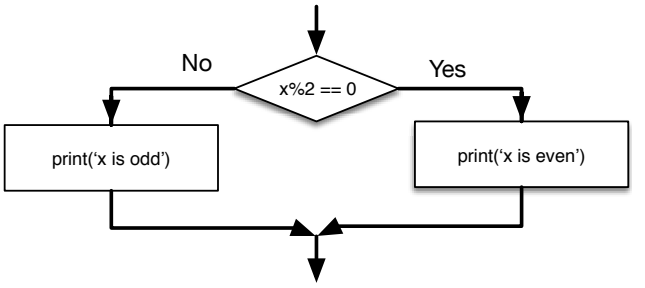


Figure 3.2: If-Then-Else Logic

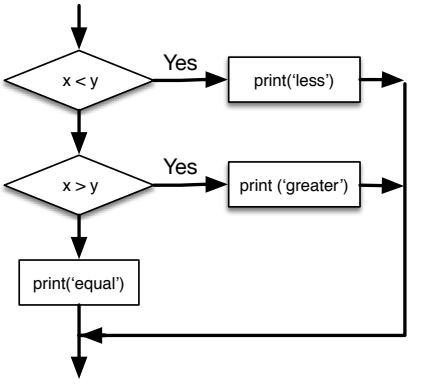


Figure 3.3: If-Then-ElseIf Logic

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

`elif` is an abbreviation of "else if." Again, exactly one branch will be executed.

There is no limit on the number of `elif` statements. If there is an `else` clause, it has to be at the end, but there doesn't have to be one.

```
if choice == 'a':
    print('Bad guess')
elif choice == 'b':
    print('Good guess')
elif choice == 'c':
    print('Close, but not correct')
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

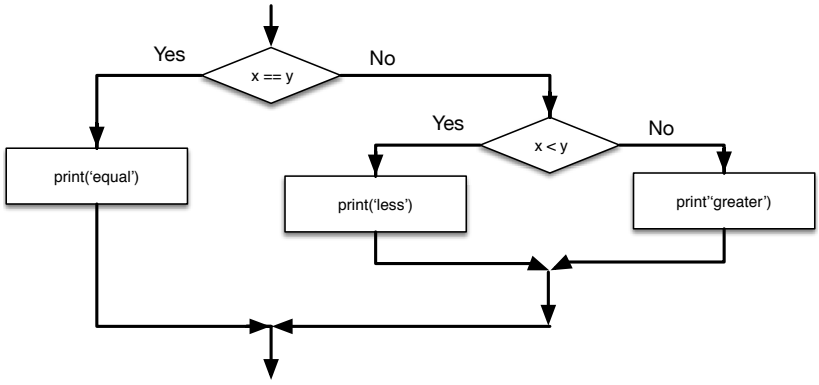


Figure 3.4: Nested If Statements

3.6 Nested conditionals

One conditional can also be nested within another. We could have written the three-branch example like this:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another `if` statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

The `print` statement is executed only if we make it past both conditionals, so we can get the same effect with the `and` operator:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

3.7 Catching exceptions using try and except

Earlier we saw a code segment where we used the `input` and `int` functions to read and parse an integer number entered by the user. We also saw how treacherous doing this could be:

```
>>> prompt = "What...is the airspeed velocity of an unladen swallow?\n"
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
>>>
```

When we are executing these statements in the Python interpreter, we get a new prompt from the interpreter, think "oops", and move on to our next statement.

However if you place this code in a Python script and this error occurs, your script immediately stops in its tracks with a traceback. It does not execute the following statement.

Here is a sample program to convert a Fahrenheit temperature to a Celsius temperature:

```
inp = input('Enter Fahrenheit Temperature: ')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)

# Code: http://www.py4e.com/code3/fahren.py
```

If we execute this code and give it invalid input, it simply fails with an unfriendly error message:

```
python fahren.py
Enter Fahrenheit Temperature:72
22.22222222222222

python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: could not convert string to float: 'fred'
```

There is a conditional execution structure built into Python to handle these types of expected and unexpected errors called "try / except". The idea of `try` and `except` is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the except block) are ignored if there is no error.

You can think of the `try` and `except` feature in Python as an "insurance policy" on a sequence of statements.

We can rewrite our temperature converter as follows:

```
inp = input('Enter Fahrenheit Temperature:')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print('Please enter a number')

# Code: http://www.py4e.com/code3/fahren2.py
```

Python starts by executing the sequence of statements in the `try` block. If all goes well, it skips the `except` block and proceeds. If an exception occurs in the `try` block, Python jumps out of the `try` block and executes the sequence of statements in the `except` block.

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.22222222222222


python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

Handling an exception with a `try` statement is called catching an exception. In this example, the `except` clause prints an error message. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

3.8 Short-circuit evaluation of logical expressions

When Python is processing a logical expression such as `x >= 2 and (x/y) > 2`, it evaluates the expression from left to right. Because of the definition of `and`, if `x` is less than 2, the expression `x >= 2` is `False` and so the whole expression is `False` regardless of whether `(x/y) > 2` evaluates to `True` or `False`.

When Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression. When the evaluation of a logical expression stops because the overall value is already known, it is called short-circuiting the evaluation.

While this may seem like a fine point, the short-circuit behavior leads to a clever technique called the guardian pattern. Consider the following code sequence in the Python interpreter:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
```

```
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

The third calculation failed because Python was evaluating `(x/y)` and `y` was zero, which causes a runtime error. But the second example did not fail because the first part of the expression `x >= 2` evaluated to `False` so the `(x/y)` was not ever executed due to the short-circuit rule and there was no error.

We can construct the logical expression to strategically place a guard evaluation just before the evaluation that might cause an error as follows:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

In the first logical expression, `x >= 2` is `False` so the evaluation stops at the `and`. In the second logical expression, `x >= 2` is `True` but `y != 0` is `False` so we never reach `(x/y)`.

In the third logical expression, the `y != 0` is after the `(x/y)` calculation so the expression fails with an error.

In the second expression, we say that `y != 0` acts as a guard to insure that we only execute `(x/y)` if `y` is non-zero.

3.9 Debugging

The traceback Python displays when an error occurs contains a lot of information, but it can be overwhelming. The most useful parts are usually:

- What kind of error it was, and
- Where it occurred.

Syntax errors are usually easy to find, but there are a few gotchas. Whitespace errors can be tricky because spaces and tabs are invisible and we are used to ignoring them.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

In this example, the problem is that the second line is indented by one space. But the error message points to `y`, which is misleading. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.

In general, error messages tell you where the problem was discovered, but that is often not where it was caused.

3.10 Glossary

body The sequence of statements within a compound statement.
boolean expression An expression whose value is either `True` or `False`.
branch One of the alternative sequences of statements in a conditional statement.
chained conditional A conditional statement with a series of alternative branches.
comparison operator One of the operators that compares its operands: `==`, `!=`, `>`, `<`, `>=`, and `<=`.
conditional statement A statement that controls the flow of execution depending on some condition.
condition The boolean expression in a conditional statement that determines which branch is executed.
compound statement A statement that consists of a header and a body. The header ends with a colon (:). The body is indented relative to the header.
guardian pattern Where we construct a logical expression with additional comparisons to take advantage of the short-circuit behavior.
logical operator One of the operators that combines boolean expressions: `and`, `or`, and `not`.
nested conditional A conditional statement that appears in one of the branches of another conditional statement.
traceback A list of the functions that are executing, printed when an exception occurs.
short circuit When Python is part-way through evaluating a logical expression and stops the evaluation because Python knows the final value for the expression without needing to evaluate the rest of the expression.

3.11 Exercises

Exercise 1: Rewrite your pay computation to give the employee 1.5 times the hourly rate for hours worked above 40 hours.

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

Exercise 2: Rewrite your pay program using `try` and `except` so that your program handles non-numeric input gracefully by printing a message and exiting the program. The following shows two executions of the program:

```
Enter Hours: 20
Enter Rate: nine
Error, please enter numeric input

Enter Hours: forty
Error, please enter numeric input
```

Exercise 3: Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error message. If the score is between 0.0 and 1.0, print a grade using the following table:

```
Score   Grade
>= 0.9     A
>= 0.8     B
>= 0.7     C
>= 0.6     D
< 0.6    F
~~~
```

Enter score: 0.95 A ~~~~

```
Enter score: perfect
Bad score

Enter score: 10.0
Bad score

Enter score: 0.75
C

Enter score: 0.5
F
```

Run the program repeatedly as shown above to test the various different values for input.

Chapter 4

Functions

4.1 Function calls

In the context of programming, a function is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name. We have already seen one example of a function call:

```
>>> type(32)
<class 'int'>
```

The name of the function is `type`. The expression in parentheses is called the argument of the function. The argument is a value or variable that we are passing into the function as input to the function. The result, for the `type` function, is the type of the argument.

It is common to say that a function "takes" an argument and "returns" a result. The result is called the return value.

4.2 Built-in functions

Python provides a number of important built-in functions that we can use without needing to provide the function definition. The creators of Python wrote a set of functions to solve common problems and included them in Python for us to use.

The `max` and `min` functions give us the largest and smallest values in a list, respectively:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

The `max` function tells us the "largest character" in the string (which turns out to be the letter "w") and the `min` function shows us the smallest character (which turns out to be a space).

Another very common built-in function is the `len` function which tells us how many items are in its argument. If the argument to `len` is a string, it returns the number of characters in the string.

```
>>> len('Hello world')
11
>>>
```

These functions are not limited to looking at strings. They can operate on any set of values, as we will see in later chapters.

You should treat the names of built-in functions as reserved words (i.e., avoid using "max" as a variable name).

4.3 Type conversion functions

Python also provides built-in functions that convert values from one type to another. The `int` function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

`int` can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finally, `str` converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

4.4 Random numbers

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be deterministic. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to use algorithms that generate pseudorandom numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

The `random` module provides functions that generate pseudorandom numbers (which I will simply call "random" from here on).

The function `random` returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call `random`, you get the next number in a long series. To see a sample, run this loop:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

This program produces the following list of 10 random numbers between 0.0 and up to but not including 1.0.

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

Exercise 1: Run the program on your system and see what numbers you get. Run the program more than once and see what numbers you get.

The `random` function is only one of many functions that handle random numbers. The function `randint` takes the parameters `low` and `high`, and returns an integer between `low` and `high` (including both).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

To choose an element from a sequence at random, you can use `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

The `random` module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

4.5 Math functions

Python has a `math` module that provides most of the familiar mathematical functions. Before we can use the module, we have to import it:

```
>>> import math
```

This statement creates a module object named math. If you print the module object, you get some information about it:

```
>>> print(math)
<module 'math' (built-in)>
```

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called dot notation.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

The first example computes the logarithm base 10 of the signal-to-noise ratio. The math module also provides a function called `log` that computes logarithms base `e`.

The second example finds the sine of `radians`. The name of the variable is a hint that `sin` and the other trigonometric functions (`cos`, `tan`, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by 2π:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476
```

The expression `math.pi` gets the variable `pi` from the math module. The value of this variable is an approximation of π, accurate to about 15 digits.

If you know your trigonometry, you can check the previous result by comparing it to the square root of two divided by two:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

4.6 Adding new functions

So far, we have only been using the functions that come with Python, but it is also possible to add new functions. A function definition specifies the name of a new function and the sequence of statements that execute when the function is called. Once we define a function, we can reuse the function over and over throughout our program.

Here is an example:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
```

`def` is a keyword that indicates that this is a function definition. The name of the function is `print_lyrics`. The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn't take any arguments. Later we will build functions that take arguments as their inputs.

The first line of the function definition is called the header; the rest is called the body. The header has to end with a colon and the body has to be indented. By convention, the indentation is always four spaces. The body can contain any number of statements.

The strings in the print statements are enclosed in quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

If you type a function definition in interactive mode, the interpreter prints ellipses (...) to let you know that the definition isn't complete:

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...     print('I sleep all night and I work all day.')
...
```

To end the function, you have to enter an empty line (this is not necessary in a script).

Defining a function creates a variable with the same name.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<class 'function'>
```

The value of `print_lyrics` is a function object, which has type "function".

The syntax for calling the new function is the same as for built-in functions:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called `repeat_lyrics`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

And then call `repeat_lyrics`:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

But that's not really how the song goes.

4.7 Definitions and uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')


def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()

# Code: http://www.py4e.com/code3/lyrics.py
```

This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not get executed until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

Exercise 2: Move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.

Exercise 3: Move the function call back to the bottom and move the definition of `print_lyrics` after the definition of `repeat_lyrics`. What happens when you run this program?

4.8 Flow of execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the flow of execution.

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

What's the moral of this sordid tale? When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

4.9 Parameters and arguments

Some of the built-in functions we have seen require arguments. For example, when you call `math.sin` you pass a number as an argument. Some functions take more than one argument: `math.pow` takes two, the base and the exponent.

Inside the function, the arguments are assigned to variables called parameters. Here is an example of a user-defined function that takes an argument:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

This function assigns the argument to a parameter named `bruce`. When the function is called, it prints the value of the parameter (whatever it is) twice.

This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for `print_twice`:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in the examples the expressions ”Spam’*4**and**math.cos(math.pi)‘ are only evaluated once.

You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

The name of the variable we pass as an argument (`michael`) has nothing to do with the name of the parameter (`bruce`). It doesn’t matter what the value was called back home (in the caller); here in `print_twice`, we call everybody `bruce`.

4.10 Fruitful functions and void functions

Some of the functions we are using, such as the math functions, yield results; for lack of a better name, I call them fruitful functions. Other functions, like `print_twice`, perform an action but don’t return a value. They are called void functions.

When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

When you call a function in interactive mode, Python displays the result:

```
>>> math.sqrt(5)
2.23606797749979
```

But in a script, if you call a fruitful function and do not store the result of the function in a variable, the return value vanishes into the mist!

```
math.sqrt(5)
```

This script computes the square root of 5, but since it doesn't store the result in a variable or display the result, it is not very useful.

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called `None`.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

The value `None` is not the same as the string "None". It is a special value that has its own type:

```
>>> print(type(None))
<class 'NoneType'>
```

To return a result from a function, we use the `return` statement in our function. For example, we could make a very simple function called `addtwo` that adds two numbers together and returns a result.

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)

# Code: http://www.py4e.com/code3/addtwo.py
```

When this script executes, the `print` statement will print out "8" because the `addtwo` function was called with 3 and 5 as arguments. Within the function, the parameters `a` and `b` were 3 and 5 respectively. The function computed the sum of the two numbers and placed it in the local function variable named `added`. Then it used the `return` statement to send the computed value back to the calling code as the function result, which was assigned to the variable `x` and printed out.

4.11 Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Throughout the rest of the book, often we will use a function definition to explain a concept. Part of the skill of creating and using functions is to have a function properly capture an idea such as "find the smallest value in a list of values". Later we will show you code that finds the smallest in a list of values and we will present it to you as a function named `min` which takes a list of values as its argument and returns the smallest value in the list.

4.12 Debugging

If you are using a text editor to write your scripts, you might run into problems with spaces and tabs. The best way to avoid these problems is to use spaces exclusively (no tabs). Most text editors that know about Python do this by default, but some don't.

Tabs and spaces are usually invisible, which makes them hard to debug, so try to find an editor that manages indentation for you.

Also, don't forget to save your program before you run it. Some development environments do this automatically, but some don't. In that case, the program you are looking at in the text editor is not the same as the program you are running.

Debugging can take a long time if you keep running the same incorrect program over and over!

Make sure that the code you are looking at is the code you are running. If you're not sure, put something like `print("hello")` at the beginning of the program and run it again. If you don't see `hello`, you're not running the right program!

4.13 Glossary

algorithm A general process for solving a category of problems.
argument A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.
body The sequence of statements inside a function definition.
composition Using an expression as part of a larger expression, or a statement as part of a larger statement.
deterministic Pertaining to a program that does the same thing each time it runs, given the same inputs.
dot notation The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.
flow of execution The order in which statements are executed during a program run.
fruitful function A function that returns a value.
function A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.
function call A statement that executes a function. It consists of the function name followed by an argument list.
function definition A statement that creates a new function, specifying its name, parameters, and the statements it executes.
function object A value created by a function definition. The name of the function is a variable that refers to a function object.
header The first line of a function definition.
import statement A statement that reads a module file and creates a module object.
module object A value created by an `import` statement that provides access to the data and code defined in a module.
parameter A name used inside a function to refer to the value passed as an argument.
pseudorandom Pertaining to a sequence of numbers that appear to be random, but are generated by a deterministic program.
return value The result of a function. If a function call is used as an expression, the return value is the value of the expression.
void function A function that does not return a value.

4.14 Exercises

Exercise 4: What is the purpose of the "def" keyword in Python?

a) It is slang that means "the following code is really cool"
b) It indicates the start of a function
c) It indicates that the following indented section of code is to be stored for later
d) b and c are both true
e) None of the above

Exercise 5: What will the following Python program print out?

```
def fred():
   print("Zap")

def jane():
   print("ABC")
```

```
jane()
fred()
jane()
```

a) Zap ABC jane fred jane
b) Zap ABC Zap
c) ABC Zap jane
d) ABC Zap ABC
e) Zap Zap Zap

Exercise 6: Rewrite your pay computation with time-and-a-half for overtime and create a function called `computepay` which takes two parameters (`hours` and `rate`).

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

Exercise 7: Rewrite the grade program from the previous chapter using a function called `computegrade` that takes a score as its parameter and returns a grade as a string.

```
Score   Grade
> 0.9     A
> 0.8     B
> 0.7     C
> 0.6     D
<= 0.6    F

Program Execution:

Enter score: 0.95
A

Enter score: perfect
Bad score

Enter score: 10.0
Bad score

Enter score: 0.75
C

Enter score: 0.5
F
```

Run the program repeatedly to test the various different values for input.

Chapter 5

Iteration

5.1 Updating variables

A common pattern in assignment statements is an assignment statement that updates a variable, where the new value of the variable depends on the old.

```
x = x + 1
```

This means "get the current value of `x`, add 1, and then update `x` with the new value."

If you try to update a variable that doesn't exist, you get an error, because Python evaluates the right side before it assigns a value to `x`:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Before you can update a variable, you have to initialize it, usually with a simple assignment:

```
>>> x = 0
>>> x = x + 1
```

Updating a variable by adding 1 is called an increment; subtracting 1 is called a decrement.

5.2 The `while` statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. Because iteration is so common, Python provides several language features to make it easier.

One form of iteration in Python is the `while` statement. Here is a simple program that counts down from five and then says "Blastoff!".

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')
```

You can almost read the `while` statement as if it were English. It means, "While `n` is greater than 0, display the value of `n` and then reduce the value of `n` by 1. When you get to 0, exit the `while` statement and display the word `Blastoff!`"

More formally, here is the flow of execution for a `while` statement:

1. Evaluate the condition, yielding `True` or `False`.
2. If the condition is false, exit the `while` statement and continue execution at the next statement.
3. If the condition is true, execute the body and then go back to step

    1.

This type of flow is called a loop because the third step loops back around to the top. We call each time we execute the body of the loop an iteration. For the above loop, we would say, "It had five iterations", which means that the body of the loop was executed five times.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. We call the variable that changes each time the loop executes and controls when the loop finishes the iteration variable. If there is no iteration variable, the loop will repeat forever, resulting in an infinite loop.

5.3 Infinite loops

An endless source of amusement for programmers is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop because there is no iteration variable telling you how many times to execute the loop.

In the case of `countdown`, we can prove that the loop terminates because we know that the value of `n` is finite, and we can see that the value of `n` gets smaller each time through the loop, so eventually we have to get to 0. Other times a loop is obviously infinite because it has no iteration variable at all.

5.4 "Infinite loops" and `break`

Sometimes you don't know it's time to end a loop until you get half way through the body. In that case you can write an infinite loop on purpose and then use the `break` statement to jump out of the loop.

This loop is obviously an infinite loop because the logical expression on the `while` statement is simply the logical constant `True`:

```
n = 10
while True:
    print(n, end=' ')
    n = n - 1
print('Done!')
```

If you make the mistake and run this code, you will learn quickly how to stop a runaway Python process on your system or find where the power-off button is on your computer. This program will run forever or until your battery runs out because the logical expression at the top of the loop is always true by virtue of the fact that the expression is the constant value `True`.

While this is a dysfunctional infinite loop, we can still use this pattern to build useful loops as long as we carefully add code to the body of the loop to explicitly exit the loop using `break` when we have reached the exit condition.

For example, suppose you want to take input from the user until they type `done`. You could write:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')

# Code: http://www.py4e.com/code3/copytildone1.py
```

The loop condition is `True`, which is always true, so the loop runs repeatedly until it hits the break statement.

Each time through, it prompts the user with an angle bracket. If the user types `done`, the `break` statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here's a sample run:

```
> hello there
hello there
> finished
finished
> done
Done!
```

This way of writing `while` loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens.").

5.5 Finishing iterations with `continue`

Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration. In that case you can use the `continue` statement to skip to the next iteration without finishing the body of the loop for the current iteration.

Here is an example of a loop that copies its input until the user types "done", but treats lines that start with the hash character as lines not to be printed (kind of like Python comments).

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')

# Code: http://www.py4e.com/code3/copytildone2.py
```

Here is a sample run of this new program with `continue` added.

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

All the lines are printed except the one that starts with the hash sign because when the `continue` is executed, it ends the current iteration and jumps back to the `while` statement to start the next iteration, thus skipping the `print` statement.

5.6 Definite loops using `for`

Sometimes we want to loop through a set of things such as a list of words, the lines in a file, or a list of numbers. When we have a list of things to loop through, we can construct a definite loop using a `for` statement. We call the `while` statement an indefinite loop because it simply loops until some condition becomes `False`, whereas the `for` loop is looping through a known set of items so it runs through as many iterations as there are items in the set.

The syntax of a `for` loop is similar to the `while` loop in that there is a `for` statement and a loop body:

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print('Happy New Year:', friend)
print('Done!')
```

In Python terms, the variable `friends` is a list[1] of three strings and the `for` loop goes through the list and executes the body once for each of the three strings in the list resulting in this output:

[1] We will examine lists in more detail in a later chapter.

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

Translating this `for` loop to English is not as direct as the `while`, but if you think of friends as a set, it goes like this: "Run the statements in the body of the for loop once for each friend in the set named friends."

Looking at the `for` loop, for and in are reserved Python keywords, and `friend` and `friends` are variables.

```
for friend in friends:
    print('Happy New Year:', friend)
```

In particular, `friend` is the iteration variable for the for loop. The variable `friend` changes for each iteration of the loop and controls when the `for` loop completes. The iteration variable steps successively through the three strings stored in the `friends` variable.

5.7 Loop patterns

Often we use a `for` or `while` loop to go through a list of items or the contents of a file and we are looking for something such as the largest or smallest value of the data we scan through.

These loops are generally constructed by:

- Initializing one or more variables before the loop starts
- Performing some computation on each item in the loop body, possibly changing the variables in the body of the loop
- Looking at the resulting variables when the loop completes

We will use a list of numbers to demonstrate the concepts and construction of these loop patterns.

5.7.1 Counting and summing loops

For example, to count the number of items in a list, we would write the following `for` loop:

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print('Count: ', count)
```

We set the variable `count` to zero before the loop starts, then we write a `for` loop to run through the list of numbers. Our iteration variable is named `itervar` and while we do not use `itervar` in the loop, it does control the loop and cause the loop body to be executed once for each of the values in the list.

In the body of the loop, we add 1 to the current value of `count` for each of the values in the list. While the loop is executing, the value of `count` is the number of values we have seen "so far".

Once the loop completes, the value of `count` is the total number of items. The total number "falls in our lap" at the end of the loop. We construct the loop so that we have what we want when the loop finishes.

Another similar loop that computes the total of a set of numbers is as follows:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print('Total: ', total)
```

In this loop we do use the iteration variable. Instead of simply adding one to the `count` as in the previous loop, we add the actual number (3, 41, 12, etc.) to the running total during each loop iteration. If you think about the variable `total`, it contains the "running total of the values so far". So before the loop starts `total` is zero because we have not yet seen any values, during the loop `total` is the running total, and at the end of the loop `total` is the overall total of all the values in the list.

As the loop executes, `total` accumulates the sum of the elements; a variable used this way is sometimes called an accumulator.

Neither the counting loop nor the summing loop are particularly useful in practice because there are built-in functions `len()` and `sum()` that compute the number of items in a list and the total of the items in the list respectively.

5.7.2 Maximum and minimum loops

To find the largest value in a list or sequence, we construct the following loop:

```
largest = None
print('Before:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest :
        largest = itervar
    print('Loop:', itervar, largest)
print('Largest:', largest)
```

When the program executes, the output is as follows:

```
Before: None
Loop: 3 3
Loop: 41 41
```

```
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

The variable `largest` is best thought of as the "largest value we have seen so far". Before the loop, we set `largest` to the constant `None`. `None` is a special constant value which we can store in a variable to mark the variable as "empty".

Before the loop starts, the largest value we have seen so far is `None` since we have not yet seen any values. While the loop is executing, if `largest` is `None` then we take the first value we see as the largest so far. You can see in the first iteration when the value of `itervar` is 3, since `largest` is `None`, we immediately set `largest` to be 3.

After the first iteration, `largest` is no longer `None`, so the second part of the compound logical expression that checks `itervar > largest` triggers only when we see a value that is larger than the "largest so far". When we see a new "even larger" value we take that new value for `largest`. You can see in the program output that `largest` progresses from 3 to 41 to 74.

At the end of the loop, we have scanned all of the values and the variable `largest` now does contain the largest value in the list.

To compute the smallest number, the code is very similar with one small change:

```
smallest = None
print('Before:', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
        smallest = itervar
    print('Loop:', itervar, smallest)
print('Smallest:', smallest)
```

Again, `smallest` is the "smallest so far" before, during, and after the loop executes. When the loop has completed, `smallest` contains the minimum value in the list.

Again as in counting and summing, the built-in functions `max()` and `min()` make writing these exact loops unnecessary.

The following is a simple version of the Python built-in `min()` function:

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

In the function version of the smallest code, we removed all of the `print` statements so as to be equivalent to the `min` function which is already built in to Python.

5.8 Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more places for bugs to hide.

One way to cut your debugging time is "debugging by bisection." For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a `print` statement (or something else that has a verifiable effect) and run the program.

If the mid-point check is incorrect, the problem must be in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is much less than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the "middle of the program" is and not always possible to check it. It doesn't make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

5.9 Glossary

accumulator A variable used in a loop to add up or accumulate a result.

counter A variable used in a loop to count the number of times something happened. We initialize a counter to zero and then increment the counter each time we want to "count" something.

decrement An update that decreases the value of a variable.

initialize An assignment that gives an initial value to a variable that will be updated.

increment An update that increases the value of a variable (often by one).

infinite loop A loop in which the terminating condition is never satisfied or for which there is no terminating condition.

iteration Repeated execution of a set of statements using either a function that calls itself or a loop.

5.10 Exercises

Exercise 1: Write a program which repeatedly reads numbers until the user enters "done". Once "done" is entered, print out the total, count, and average of the numbers. If the user enters anything other than a number, detect their mistake using `try` and `except` and print an error message and skip to the next number.

```
Enter a number: 4
Enter a number: 5
Enter a number: bad data
Invalid input
```

```
Enter a number: 7
Enter a number: done
16 3 5.333333333333333
```

Exercise 2: Write another program that prompts for a list of numbers as above and at the end prints out both the maximum and minimum of the numbers instead of the average.

Chapter 6

Strings

6.1 A string is a sequence

A string is a sequence of characters. You can access the characters one at a time with the bracket operator:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

The second statement extracts the character at index position 1 from the `fruit` variable and assigns it to the `letter` variable.

The expression in brackets is called an index. The index indicates which character in the sequence you want (hence the name).

But you might not get what you expect:

```
>>> print(letter)
a
```

For most people, the first letter of "banana" is `b`, not `a`. But in Python, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]
>>> print(letter)
b
```

So `b` is the 0th letter ("zero-eth") of "banana", `a` is the 1th letter ("one-eth"), and `n` is the 2th ("two-eth") letter.

You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

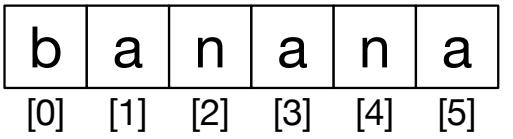


Figure 6.1: String Indexes

6.2 Getting the length of a string using `len`

`len` is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

The reason for the `IndexError` is that there is no letter in `'banana'` with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from `length`:

```
>>> last = fruit[length-1]
>>> print(last)
a
```

Alternatively, you can use negative indices, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

6.3 Traversal through a string with a loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a traversal. One way to write a traversal is with a `while` loop:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index \< len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

Exercise 1: Write a `while` loop that starts at the last character in the string and works its way backwards to the first character in the string, printing each letter on a separate line, except backwards.

Another way to write a traversal is with a `for` loop:

```
for char in fruit:
    print(char)
```

Each time through the loop, the next character in the string is assigned to the variable `char`. The loop continues until no characters are left.

6.4 String slices

A segment of a string is called a slice. Selecting a slice is similar to selecting a character:

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```

The operator returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

If the first index is greater than or equal to the second the result is an empty string, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Exercise 2: Given that `fruit` is a string, what does `fruit[:]` mean?

6.5 Strings are immutable

It is tempting to use the operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

The "object" in this case is the string and the "item" is the character you tried to assign. For now, an object is the same thing as a value, but we will refine that definition later. An item is one of the values in a sequence.

The reason for the error is that strings are immutable, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print(new_greeting)
Jello, world!
```

This example concatenates a new first letter onto a slice of `greeting`. It has no effect on the original string.

6.6 Looping and counting

The following program counts the number of times the letter `a` appears in a string:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

This program demonstrates another pattern of computation called a counter. The variable `count` is initialized to 0 and then incremented each time an `a` is found. When the loop exits, `count` contains the result: the total number of `a`'s.

Exercise 3:

Encapsulate this code in a function named `count`, and generalize it so that it accepts the string and the letter as arguments.

6.7 The `in` operator

The word `in` is a boolean operator that takes two strings and returns `True` if the first appears as a substring in the second:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

6.8 String comparison

The comparison operators work on strings. To see if two strings are equal:

```
if word == 'banana':
    print('All right, bananas.')
```

Other comparison operations are useful for putting words in alphabetical order:

```
if word < 'banana':
    print('Your word,' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word,' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters, so:

```
Your word, Pineapple, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

6.9 `string` methods

Strings are an example of Python objects. An object contains both data (the actual string itself) and methods, which are effectively functions that are built into the object and are available to any instance of the object.

Python has a function called `dir` which lists the methods available for an object. The `type` function shows the type of an object and the `dir` function shows the available methods.

```
>>> stuff = 'Hello world'
>>> type(stuff)
<class 'str'>
>>> dir(stuff)
['capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> str

    Return a capitalized version of S, i.e. make the first character
    have upper case and the rest lower case.
>>>
```

While the `dir` function lists the methods, and you can use `help` to get some simple documentation on a method, a better source of documentation for string methods would be https://docs.python.org/3.5/library/stdtypes.html#string-methods.

Calling a method is similar to calling a function (it takes arguments and returns a value) but the syntax is different. We call a method by appending the method name to the variable name using the period as a delimiter.

For example, the method `upper` takes a string and returns a new string with all uppercase letters:

Instead of the function syntax `upper(word)`, it uses the method syntax `word.upper()`.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print(new_word)
BANANA
```

This form of dot notation specifies the name of the method, `upper`, and the name of the string to apply the method to, `word`. The empty parentheses indicate that this method takes no argument.

A method call is called an invocation; in this case, we would say that we are invoking `upper` on the `word`.

For example, there is a string method named `find` that searches for the position of one string within another:

```
>>> word = 'banana'
>>> index = word.find('a')
```

```
>>> print(index)
1
```

In this example, we invoke **find** on **word** and pass the letter we are looking for as a parameter.

The **find** method can find substrings as well as characters:

```
>>> word.find('na')
2
```

It can take as a second argument the index where it should start:

```
>>> word.find('na', 3)
4
```

One common task is to remove white space (spaces, tabs, or newlines) from the beginning and end of a string using the **strip** method:

```
>>> line = '  Here we go  '
>>> line.strip()
'Here we go'
```

Some methods such as startswith return boolean values.

```
>>> line = 'Have a nice day'
>>> line.startswith('Have')
True
>>> line.startswith('h')
False
```

You will note that **startswith** requires case to match, so sometimes we take a line and map it all to lowercase before we do any checking using the **lower** method.

```
>>> line = 'Have a nice day'
>>> line.startswith('h')
False
>>> line.lower()
'have a nice day'
>>> line.lower().startswith('h')
True
```

In the last example, the method **lower** is called and then we use **startswith** to see if the resulting lowercase string starts with the letter "h". As long as we are careful with the order, we can make multiple method calls in a single expression.

Exercise 4:

There is a string method called **count** that is similar to the function in the previous exercise. Read the documentation of this method at https://docs.python.org/3.5/library/stdtypes.html#string-methods and write an invocation that counts the number of times the letter a occurs in "banana".

6.10 Parsing strings

Often, we want to look into a string and find a substring. For example if we were presented a series of lines formatted as follows:

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

and we wanted to pull out only the second half of the address (i.e., `uct.ac.za`) from each line, we can do this by using the `find` method and string slicing.

First, we will find the position of the at-sign in the string. Then we will find the position of the first space after the at-sign. And then we will use string slicing to extract the portion of the string which we are looking for.

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ',atpos)
>>> print(sppos)
31
>>> host = data[atpos+1:sppos]
>>> print(host)
uct.ac.za
>>>
```

We use a version of the `find` method which allows us to specify a position in the string where we want `find` to start looking. When we slice, we extract the characters from "one beyond the at-sign through up to but not including the space character".

The documentation for the `find` method is available at

https://docs.python.org/3.5/library/stdtypes.html#string-methods.

6.11 Format operator

The format operator, `%` allows us to construct strings, replacing parts of the strings with the data stored in variables. When applied to integers, `%` is the modulus operator. But when the first operand is a string, `%` is the format operator.

The first operand is the format string, which contains one or more format sequences that specify how the second operand is formatted. The result is a string.

For example, the format sequence "%d" means that the second operand should be formatted as an integer (`d` stands for "decimal"):

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string "42", which is not to be confused with the integer value 42.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple[1]. Each format sequence is matched with an element of the tuple, in order.

The following example uses "%d" to format an integer, "%g" to format a floating-point number (don't ask why), and "%s" to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple must match the number of format sequences in the string. The types of the elements also must match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

The format operator is powerful, but it can be difficult to use. You can read more about it at

https://docs.python.org/3.5/library/stdtypes.html#printf-style-string-formatting.

6.12 Debugging

A skill that you should cultivate as you program is always asking yourself, "What could go wrong here?" or alternatively, "What crazy thing might our user do to crash our (seemingly) perfect program?"

For example, look at the program which we used to demonstrate the `while` loop in the chapter on iteration:

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')

# Code: http://www.py4e.com/code3/copytildone2.py
```

[1] A tuple is a sequence of comma-separated values inside a pair of parenthesis. We will cover tuples in Chapter 10

Look what happens when the user enters an empty line of input:

```
> hello there
hello there
> # don't print this
> print this!
print this!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#':
IndexError: string index out of range
```

The code works fine until it is presented an empty line. Then there is no zero-th character, so we get a traceback. There are two solutions to this to make line three "safe" even if the line is empty.

One possibility is to simply use the `startswith` method which returns `False` if the string is empty.

```
    if line.startswith('#'):
```

Another way is to safely write the `if` statement using the guardian pattern and make sure the second logical expression is evaluated only where there is at least one character in the string.:

```
    if len(line) > 0 and line[0] == '#':
```

6.13 Glossary

counter A variable used to count something, usually initialized to zero and then incremented.

empty string A string with no characters and length 0, represented by two quotation marks.

format operator An operator, `%`, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.

format sequence A sequence of characters in a format string, like `%d`, that specifies how a value should be formatted.

format string A string, used with the format operator, that contains format sequences.

flag A boolean variable used to indicate whether a condition is true or false.

invocation A statement that calls a method.

immutable The property of a sequence whose items cannot be assigned.

index An integer value used to select an item in a sequence, such as a character in a string.

item One of the values in a sequence.

method A function that is associated with an object and called using dot notation.

object Something a variable can refer to. For now, you can use "object" and "value" interchangeably.

search A pattern of traversal that stops when it finds what it is looking for.

sequence An ordered set; that is, a set of values where each value is identified by an integer index.

slice A part of a string specified by a range of indices.

traverse To iterate through the items in a sequence, performing a similar operation on each.

6.14 Exercises

Exercise 5: Take the following Python code that stores a string: ‘

```
str = 'X-DSPAM-Confidence:0.8475'
```

Use `find` and string slicing to extract the portion of the string after the colon character and then use the `float` function to convert the extracted string into a floating point number.

Exercise 6:

Read the documentation of the string methods at

https://docs.python.org/3.5/library/stdtypes.html#string-methods

You might want to experiment with some of them to make sure you understand how they work. `strip` and `replace` are particularly useful.

The documentation uses a syntax that might be confusing. For example, in `find(sub[, start[, end]])`, the brackets indicate optional arguments. So `sub` is required, but `start` is optional, and if you include `start`, then `end` is optional.

Chapter 7

Files

7.1 Persistence

So far, we have learned how to write programs and communicate our intentions to the Central Processing Unit using conditional execution, functions, and iterations. We have learned how to create and use data structures in the Main Memory. The CPU and memory are where our software works and runs. It is where all of the "thinking" happens.

But if you recall from our hardware architecture discussions, once the power is turned off, anything stored in either the CPU or main memory is erased. So up to now, our programs have just been transient fun exercises to learn Python.

In this chapter, we start to work with Secondary Memory (or files). Secondary memory is not erased when the power is turned off. Or in the case of a USB flash drive, the data we write from our programs can be removed from the system and transported to another system.

We will primarily focus on reading and writing text files such as those we create in a text editor. Later we will see how to work with database files which are binary files, specifically designed to be read and written through database software.

7.2 Opening files

When we want to read or write a file (say on your hard drive), we first must open the file. Opening the file communicates with your operating system, which knows where the data for each file is stored. When you open a file, you are asking the operating system to find the file by name and make sure the file exists. In this example, we open the file `mbox.txt`, which should be stored in the same folder that you are in when you start Python. You can download this file from www.py4e.com/code3/mbox.txt

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
```

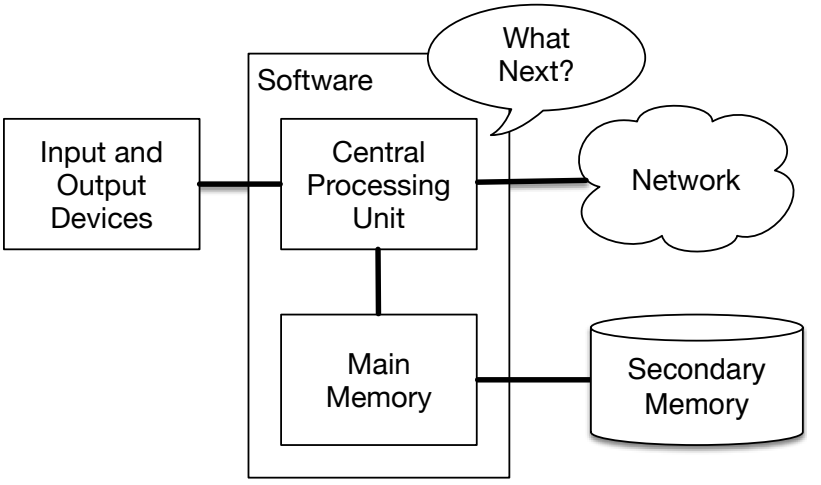


Figure 7.1: Secondary Memory

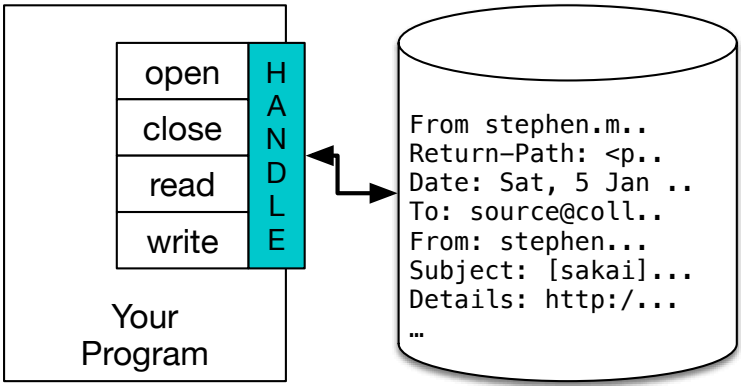


Figure 7.2: A File Handle

If the `open` is successful, the operating system returns us a file handle. The file handle is not the actual data contained in the file, but instead it is a "handle" that we can use to read the data. You are given a handle if the requested file exists and you have the proper permissions to read the file.

If the file does not exist, `open` will fail with a traceback and you will not get a handle to access the contents of the file:

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'
```

Later we will use `try` and `except` to deal more gracefully with the situation where we attempt to open a file that does not exist.

7.3 Text files and lines

A text file can be thought of as a sequence of lines, much like a Python string can be thought of as a sequence of characters. For example, this is a sample of a text file which records mail activity from various individuals in an open source project development team:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
...
```

The entire file of mail interactions is available from

www.py4e.com/code3/mbox.txt

and a shortened version of the file is available from

www.py4e.com/code3/mbox-short.txt

These files are in a standard format for a file containing multiple mail messages. The lines which start with "From" separate the messages and the lines which start with "From:" are part of the messages. For more information about the mbox format, see en.wikipedia.org/wiki/Mbox.

To break the file into lines, there is a special character that represents the "end of the line" called the newline character.

In Python, we represent the newline character as a backslash-n in string constants. Even though this looks like two characters, it is actually a single character. When we look at the variable by entering "stuff" in the interpreter, it shows us the `\n` in the string, but when we use `print` to show the string, we see the string broken into two lines by the newline character.

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print(stuff)
Hello
World!
>>> stuff = 'X\nY'
>>> print(stuff)
X
Y
>>> len(stuff)
3
```

You can also see that the length of the string `X\nY` is three characters because the newline character is a single character.

So when we look at the lines in a file, we need to imagine that there is a special invisible character called the newline at the end of each line that marks the end of the line.

So the newline character separates the characters in the file into lines.

7.4 Reading files

While the file handle does not contain the data for the file, it is quite easy to construct a `for` loop to read through and count each of the lines in a file:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    count = count + 1
print('Line Count:', count)

# Code: http://www.py4e.com/code3/open.py
```

We can use the file handle as the sequence in our `for` loop. Our `for` loop simply counts the number of lines in the file and prints them out. The rough translation of the `for` loop into English is, "for each line in the file represented by the file handle, add one to the `count` variable."

The reason that the `open` function does not read the entire file is that the file might be quite large with many gigabytes of data. The `open` statement takes the same amount of time regardless of the size of the file. The `for` loop actually causes the data to be read from the file.

When the file is read using a `for` loop in this manner, Python takes care of splitting the data in the file into separate lines using the newline character. Python reads each line through the newline and includes the newline as the last character in the `line` variable for each iteration of the `for` loop.

Because the `for` loop reads the data one line at a time, it can efficiently read and count the lines in very large files without running out of main memory to store the data. The above program can count the lines in any size file using very little memory since each line is read, counted, and then discarded.

If you know the file is relatively small compared to the size of your main memory, you can read the whole file into one string using the `read` method on the file handle.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp))
94626
>>> print(inp[:20])
From stephen.marquar
```

In this example, the entire contents (all 94,626 characters) of the file `mbox-short.txt` are read directly into the variable `inp`. We use string slicing to print out the first 20 characters of the string data stored in `inp`.

When the file is read in this manner, all the characters including all of the lines and newline characters are one big string in the variable inp. Remember that this form of the `open`

function should only be used if the file data will fit comfortably in the main memory of your computer.

If the file is too large to fit in main memory, you should write your program to read the file in chunks using a `for` or `while` loop.

7.5 Searching through a file

When you are searching through data in a file, it is a very common pattern to read through a file, ignoring most of the lines and only processing lines which meet a particular condition. We can combine the pattern for reading a file with string methods to build simple search mechanisms.

For example, if we wanted to read a file and only print out lines which started with the prefix "From:", we could use the string method startswith to select only those lines with the desired prefix:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    if line.startswith('From:'):
        print(line)

# Code: http://www.py4e.com/code3/search1.py
```

When this program runs, we get the following output:

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```

The output looks great since the only lines we are seeing are those which start with "From:", but why are we seeing the extra blank lines? This is due to that invisible newline character. Each of the lines ends with a newline, so the `print` statement prints the string in the variable line which includes a newline and then `print` adds another newline, resulting in the double spacing effect we see.

We could use line slicing to print all but the last character, but a simpler approach is to use the rstrip method which strips whitespace from the right side of a string as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:'):
        print(line)

# Code: http://www.py4e.com/code3/search2.py
```

When this program runs, we get the following output:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

As your file processing programs get more complicated, you may want to structure your search loops using `continue`. The basic idea of the search loop is that you are looking for "interesting" lines and effectively skipping "uninteresting" lines. And then when we find an interesting line, we do something with that line.

We can structure the loop to follow the pattern of skipping uninteresting lines as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Skip 'uninteresting lines'
    if not line.startswith('From:'):
        continue
    # Process our 'interesting' line
    print(line)

# Code: http://www.py4e.com/code3/search3.py
```

The output of the program is the same. In English, the uninteresting lines are those which do not start with "From:", which we skip using `continue`. For the "interesting" lines (i.e., those that start with "From:") we perform the processing on those lines.

We can use the `find` string method to simulate a text editor search that finds lines where the search string is anywhere in the line. Since `find` looks for an occurrence of a string within another string and either returns the position of the string or -1 if the string was not found, we can write the following loop to show lines which contain the string "@uct.ac.za" (i.e., they come from the University of Cape Town in South Africa):

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1: continue
    print(line)

# Code: http://www.py4e.com/code3/search4.py
```

Which produces the following output:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
```

```
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

7.6 Letting the user choose the file name

We really do not want to have to edit our Python code every time we want to process a different file. It would be more usable to ask the user to enter the file name string each time the program runs so they can use our program on different files without changing the Python code.

This is quite simple to do by reading the file name from the user using `input` as follows:

```
fname = input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)

# Code: http://www.py4e.com/code3/search6.py
```

We read the file name from the user and place it in a variable named `fname` and open that file. Now we can run the program repeatedly on different files.

```
python search6.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

Before peeking at the next section, take a look at the above program and ask yourself, "What could go possibly wrong here?" or "What might our friendly user do that would cause our nice little program to ungracefully exit with a traceback, making us look not-so-cool in the eyes of our users?"

7.7 Using `try`, `except`, and `open`

I told you not to peek. This is your last chance.

What if our user types something that is not a file name?

```
python search6.py
Enter the file name: missing.txt
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'

python search6.py
Enter the file name: na na boo boo
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'na na boo boo'
```

Do not laugh. Users will eventually do every possible thing they can do to break your programs, either on purpose or with malicious intent. As a matter of fact, an important part of any software development team is a person or group called Quality Assurance (or QA for short) whose very job it is to do the craziest things possible in an attempt to break the software that the programmer has created.

The QA team is responsible for finding the flaws in programs before we have delivered the program to the end users who may be purchasing the software or paying our salary to write the software. So the QA team is the programmer's best friend.

So now that we see the flaw in the program, we can elegantly fix it using the `try/except` structure. We need to assume that the `open` call might fail and add recovery code when the `open` fails as follows:

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)

# Code: http://www.py4e.com/code3/search7.py
```

The `exit` function terminates the program. It is a function that we call that never returns. Now when our user (or QA team) types in silliness or bad file names, we "catch" them and recover gracefully:

```
python search7.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search7.py
Enter the file name: na na boo boo
File cannot be opened: na na boo boo
```

Protecting the `open` call is a good example of the proper use of `try` and `except` in a Python program. We use the term "Pythonic" when we are doing something the "Python way". We might say that the above example is the Pythonic way to open a file.

Once you become more skilled in Python, you can engage in repartee with other Python programmers to decide which of two equivalent solutions to a problem is "more Pythonic". The goal to be "more Pythonic" captures the notion that programming is part engineering and part art. We are not always interested in just making something work, we also want our solution to be elegant and to be appreciated as elegant by our peers.

7.8 Writing files

To write a file, you have to open it with mode "w" as a second parameter:

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

The `write` method of the file handle object puts data into the file, returning the number of characters written. The default write mode is text for writing (and reading) strings.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

Again, the file object keeps track of where it is, so if you call `write` again, it adds the new data to the end.

We must make sure to manage the ends of lines as we write to the file by explicitly inserting the newline character when we want to end a line. The `print` statement automatically appends a newline, but the `write` method does not add the newline automatically.

```
>>> line2 = 'the emblem of our land.\n'
>>> fout.write(line2)
24
```

When you are done writing, you have to close the file to make sure that the last bit of data is physically written to the disk so it will not be lost if the power goes off.

```
>>> fout.close()
```

We could close the files which we open for read as well, but we can be a little sloppy if we are only opening a few files since Python makes sure that all open files are closed when the program ends. When we are writing files, we want to explicitly close the files so as to leave nothing to chance.

7.9 Debugging

When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because spaces, tabs, and newlines are normally invisible:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

The built-in function `repr` can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented `\n`. Others use a return character, represented `\r`. Some use both. If you move files between different systems, these inconsistencies might cause problems.

For most systems, there are applications to convert from one format to another. You can find them (and read more about this issue) at wikipedia.org/wiki/Newline. Or, of course, you could write one yourself.

7.10 Glossary

catch To prevent an exception from terminating a program using the `try` and `except` statements.

newline A special character used in files and strings to indicate the end of a line.

Pythonic A technique that works elegantly in Python. "Using try and except is the Pythonic way to recover from missing files".

Quality Assurance A person or team focused on insuring the overall quality of a software product. QA is often involved in testing a product and identifying problems before the product is released.

text file A sequence of characters stored in permanent storage like a hard drive.

7.11 Exercises

Exercise 1: Write a program to read through a file and print the contents of the file (line by line) all in upper case. Executing the program will look as follows:

```
python shout.py
Enter a file name: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN  5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
     BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
     SAT, 05 JAN 2008 09:14:16 -0500
```

You can download the file from

www.py4e.com/code3/mbox-short.txt

Exercise 2: Write a program to prompt for a file name, and then read through the file and look for lines of the form:

`X-DSPAM-Confidence:0.8475`

When you encounter a line that starts with "X-DSPAM-Confidence:" pull apart the line to extract the floating-point number on the line. Count these lines and then compute the total of the spam confidence values from these lines. When you reach the end of the file, print out the average spam confidence.

```
Enter the file name: mbox.txt
Average spam confidence: 0.894128046745

Enter the file name: mbox-short.txt
Average spam confidence: 0.750718518519
```

Test your file on the `mbox.txt` and `mbox-short.txt` files.

Exercise 3: Sometimes when programmers get bored or want to have a bit of fun, they add a harmless Easter Egg to their program Modify the program that prompts the user for the file name so that it prints a funny message when the user types in the exact file name "na na boo boo". The program should behave normally for all other files which exist and don't exist. Here is a sample execution of the program:

```
python egg.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python egg.py
Enter the file name: missing.tyxt
File cannot be opened: missing.tyxt

python egg.py
Enter the file name: na na boo boo
NA NA BOO BOO TO YOU - You have been punk'd!
```

We are not encouraging you to put Easter Eggs in your programs; this is just an exercise.

Chapter 8

Lists

8.1 A list is a sequence

Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called elements or sometimes items.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
~~~~ {.python}
```

```
The first example is a list of four integers. The second is a list of
three strings. The elements of a list don't have to be the same type.
The following list contains a string, a float, an integer, and (lo!)
another list:

~~~~ {.python}
['spam', 2.0, 5, [10, 20]]
```

A list within another list is nested.

A list that contains no elements is called an empty list; you can create one with empty brackets, [].

As you might expect, you can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

8.2 Lists are mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a string: the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>> print(cheeses[0])
Cheddar
```

Unlike strings, lists are mutable because you can change the order of items in a list or reassign an item in a list. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print(numbers)
[17, 5]
```

The one-eth element of `numbers`, which used to be 123, is now 5.

You can think of a list as a relationship between indices and elements. This relationship is called a mapping; each index "maps to" one of the elements.

List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an `IndexError`.
- If an index has a negative value, it counts backward from the end of the list.

The `in` operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

8.3 Traversing a list

The most common way to traverse the elements of a list is with a `for` loop. The syntax is the same as for strings:

```
for cheese in cheeses:
    print(cheese)
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions `range` and `len`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element. `len` returns the number of elements in the list. `range` returns a list of indices from 0 to $n - 1$, where n is the length of the list. Each time through the loop, `i` gets the index of the next element. The assignment statement in the body uses `i` to read the old value of the element and to assign the new value.

A `for` loop over an empty list never executes the body:

```
for x in empty:
    print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

8.4 List operations

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

Similarly, the operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats four times. The second example repeats the list three times.

8.5 List slices

The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle, or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

8.6 List methods

Python provides methods that operate on lists. For example, `append` adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

`extend` takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

This example leaves `t2` unmodified.

`sort` arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

Most list methods are void; they modify the list and return `None`. If you accidentally write `t = t.sort()`, you will be disappointed with the result.

8.7 Deleting elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use `pop`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

`pop` modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

If you don't need the removed value, you can use the `del` operator:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

If you know the element you want to remove (but not the index), you can use `remove`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

The return value from `remove` is `None`.

To remove more than one element, you can use `del` with a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

As usual, the slice selects all the elements up to, but not including, the second index.

8.8 Lists and functions

There are a number of built-in functions that can be used on lists that allow you to quickly look through a list without writing your own loops:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25
```

The `sum()` function only works when the list elements are numbers. The other functions (`max()`, `len()`, etc.) work with lists of strings and other types that can be comparable.

We could rewrite an earlier program that computed the average of a list of numbers entered by the user using a list.

First, the program to compute an average without a list:

```
total = 0
count = 0
while (True):
    inp = input('Enter a number: ')
    if inp == 'done': break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print('Average:', average)

# Code: http://www.py4e.com/code3/avenum.py
```

In this program, we have `count` and `total` variables to keep the number and running total of the user's numbers as we repeatedly prompt the user for a number.

We could simply remember each number as the user entered it and use built-in functions to compute the sum and count at the end.

```
numlist = list()
while (True):
    inp = input('Enter a number: ')
    if inp == 'done': break
    value = float(inp)
    numlist.append(value)
```

```
average = sum(numlist) / len(numlist)
print('Average:', average)

# Code: http://www.py4e.com/code3/avelist.py
```

We make an empty list before the loop starts, and then each time we have a number, we append it to the list. At the end of the program, we simply compute the sum of the numbers in the list and divide it by the count of the numbers in the list to come up with the average.

8.9 Lists and strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

Because `list` is the name of a built-in function, you should avoid using it as a variable name. I also avoid the letter `l` because it looks too much like the number `1`. So that's why I use `t`.

The `list` function breaks a string into individual letters. If you want to break a string into words, you can use the `split` method:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']
>>> print(t[2])
the
```

Once you have used `split` to break the string into a list of words, you can use the index operator (square bracket) to look at a particular word in the list.

You can call `split` with an optional argument called a delimiter that specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

`join` is the inverse of `split`. It takes a list of strings and concatenates the elements. `join` is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

In this case the delimiter is a space character, so `join` puts a space between words. To concatenate strings without spaces, you can use the empty string, " ", as a delimiter.

8.10 Parsing lines

Usually when we are reading a file we want to do something to the lines other than just printing the whole line. Often we want to find the "interesting lines" and then parse the line to find some interesting part of the line. What if we wanted to print out the day of the week from those lines that start with "From"?

```
From stephen.marquard@uct.ac.zaSatJan  5 09:14:16 2008
```

The `split` method is very effective when faced with this kind of problem. We can write a small program that looks for lines where the line starts with "From", `split` those lines, and then print out the third word in the line:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From '): continue
    words = line.split()
    print(words[2])

# Code: http://www.py4e.com/code3/search5.py
```

Here we also use the contracted form of the `if` statement where we put the `continue` on the same line as the `if`. This contracted form of the `if` functions the same as if the `continue` were on the next line and indented.

The program produces the following output:

```
Sat
Fri
Fri
Fri
    ...
```

Later, we will learn increasingly sophisticated techniques for picking the lines to work on and how we pull those lines apart to find the exact bit of information we are looking for.

8.11 Objects and values

If we execute these assignment statements:



Figure 8.1: Variables and Objects

```
a = 'banana'
b = 'banana'
```

we know that `a` and `b` both refer to a string, but we don't know whether they refer to the same string. There are two possible states:

In one case, `a` and `b` refer to two different objects that have the same value. In the second case, they refer to the same object.

To check whether two variables refer to the same object, you can use the `is` operator.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

In this example, Python only created one string object, and both `a` and `b` refer to it.

But when you create two lists, you get two objects:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

In this case we would say that the two lists are equivalent, because they have the same elements, but not identical, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

Until now, we have been using "object" and "value" interchangeably, but it is more precise to say that an object has a value. If you execute `a = [1,2,3]`, `a` refers to a list object whose value is a particular sequence of elements. If another list has the same elements, we would say it has the same value.

8.12 Aliasing

If `a` refers to an object and you assign `b = a`, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

The association of a variable with an object is called a reference. In this example, there are two references to the same object.

An object with more than one reference has more than one name, so we say that the object is aliased.

If the aliased object is mutable, changes made with one alias affect the other:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

For immutable objects like strings, aliasing is not as much of a problem. In this example:

```
a = 'banana'
b = 'banana'
```

it almost never makes a difference whether `a` and `b` refer to the same string or not.

8.13 List arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, `delete_head` removes the first element from a list:

```
def delete_head(t):
    del t[0]
```

Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
```

The parameter `t` and the variable `letters` are aliases for the same object.

It is important to distinguish between operations that modify lists and operations that create new lists. For example, the `append` method modifies a list, but the `+` operator creates a new list:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
>>> print(t2)
```

```
None

>>> t3 = t1 + [3]
>>> print(t3)
[1, 2, 3]
>>> t2 is t3
False
```

This difference is important when you write functions that are supposed to modify lists. For example, this function does not delete the head of a list:

```
def bad_delete_head(t):
    t = t[1:]               # WRONG!
```

The slice operator creates a new list and the assignment makes `t` refer to it, but none of that has any effect on the list that was passed as an argument.

An alternative is to write a function that creates and returns a new list. For example, `tail` returns all but the first element of a list:

```
def tail(t):
    return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print(rest)
['b', 'c']
```

Exercise 1:

Write a function called `chop` that takes a list and modifies it, removing the first and last elements, and returns `None`.

Then write a function called `middle` that takes a list and returns a new list that contains all but the first and last elements.

8.14 Debugging

Careless use of lists (and other mutable objects) can lead to long hours of debugging. Here are some common pitfalls and ways to avoid them:

1. Don't forget that most list methods modify the argument and return `None`. This is the opposite of the string methods, which return a new string and leave the original alone.

   If you are used to writing string code like this:

   ```
   word = word.strip()
   ```

It is tempting to write list code like this: ~~~~ {.python} t = t.sort() # WRONG! ~~~~

Because `sort` returns `None`, the next operation you perform with `t` is likely to fail.

Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode. The methods and operators that lists share with other sequences (like strings) are documented at https://docs.python.org/2/library/stdtypes.html#string-methods. The methods and operators that only apply to mutable sequences are documented at https://docs.python.org/2/library/stdtypes.html#mutable-sequence-types.

2. Pick an idiom and stick with it.

   Part of the problem with lists is that there are too many ways to do things. For example, to remove an element from a list, you can use `pop`, `remove`, `del`, or even a slice assignment.

   To add an element, you can use the `append` method or the `+` operator. But don't forget that these are right:

   ```
   t.append(x)
   t = t + [x]
   ```

   And these are wrong:

   ```
   t.append([x])          # WRONG!
   t = t.append(x)        # WRONG!
   t + [x]                # WRONG!
   t = t + x              # WRONG!
   ```

   Try out each of these examples in interactive mode to make sure you understand what they do. Notice that only the last one causes a runtime error; the other three are legal, but they do the wrong thing.

3. Make copies to avoid aliasing.

   If you want to use a method like `sort` that modifies the argument, but you need to keep the original list as well, you can make a copy.

   ```
   orig = t[:]
   t.sort()
   ```

   In this example you could also use the built-in function `sorted`, which returns a new, sorted list and leaves the original alone. But in that case you should avoid using `sorted` as a variable name!

4. Lists, `split`, and files

   When we read and parse files, there are many opportunities to encounter input that can crash our program so it is a good idea to revisit the guardian pattern when it comes writing programs that read through a file and look for a "needle in the haystack".

   Let's revisit our program that is looking for the day of the week on the from lines of our file:

   ```
   From stephen.marquard@uct.ac.zaSatJan  5 09:14:16 2008
   ```

Since we are breaking this line into words, we could dispense with the use of `startswith` and simply look at the first word of the line to determine if we are interested in the line at all. We can use `continue` to skip lines that don't have "From" as the first word as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
    words = line.split()
    if words[0] != 'From' : continue
    print(words[2])
```

This looks much simpler and we don't even need to do the `rstrip` to remove the newline at the end of the file. But is it better?

```
python search8.py
Sat
Traceback (most recent call last):
  File "search8.py", line 5, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

It kind of works and we see the day from the first line (Sat), but then the program fails with a traceback error. What went wrong? What messed-up data caused our elegant, clever, and very Pythonic program to fail?

You could stare at it for a long time and puzzle through it or ask someone for help, but the quicker and smarter approach is to add a `print` statement. The best place to add the print statement is right before the line where the program failed and print out the data that seems to be causing the failure.

Now this approach may generate a lot of lines of output, but at least you will immediately have some clue as to the problem at hand. So we add a print of the variable `words` right before line five. We even add a prefix "Debug:" to the line so we can keep our regular output separate from our debug output.

```
for line in fhand:
    words = line.split()
    print('Debug:', words)
    if words[0] != 'From' : continue
    print(words[2])
```

When we run the program, a lot of output scrolls off the screen but at the end, we see our debug output and the traceback so we know what happened just before the traceback.

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

Each debug line is printing the list of words which we get when we `split` the line into words. When the program fails, the list of words is empty `[]`. If we open the file in a text editor and look at the file, at that point it looks as follows:

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan  5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

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

The error occurs when our program encounters a blank line! Of course there are "zero words" on a blank line. Why didn't we think of that when we were writing the code? When the code looks for the first word (`word[0]`) to check to see if it matches "From", we get an "index out of range" error.

This of course is the perfect place to add some guardian code to avoid checking the first word if the first word is not there. There are many ways to protect this code; we will choose to check the number of words we have before we look at the first word:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    words = line.split()
    # print 'Debug:', words
    if len(words) == 0 : continue
    if words[0] != 'From' : continue
    print(words[2])
```

First we commented out the debug print statement instead of removing it, in case our modification fails and we need to debug again. Then we added a guardian statement that checks to see if we have zero words, and if so, we use `continue` to skip to the next line in the file.

We can think of the two `continue` statements as helping us refine the set of lines which are "interesting" to us and which we want to process some more. A line which has no words is "uninteresting" to us so we skip to the next line. A line which does not have "From" as its first word is uninteresting to us so we skip it.

The program as modified runs successfully, so perhaps it is correct. Our guardian statement does make sure that the `words[0]` will never fail, but perhaps it is not enough. When we are programming, we must always be thinking, "What might go wrong?"

Exercise 2: Figure out which line of the above program is still not properly guarded. See if you can construct a text file which causes the program to fail and then modify the program so that the line is properly guarded and test it to make sure it handles your new text file.

Exercise 3: Rewrite the guardian code in the above example without two `if` statements. Instead, use a compound logical expression using the `and` logical operator with a single `if` statement.

8.15 Glossary

aliasing A circumstance where two or more variables refer to the same object.
delimiter A character or string used to indicate where a string should be split.
element One of the values in a list (or other sequence); also called items.
equivalent Having the same value.
index An integer value that indicates an element in a list.
identical Being the same object (which implies equivalence).
list A sequence of values.
list traversal The sequential accessing of each element in a list.
nested list A list that is an element of another list.
object Something a variable can refer to. An object has a type and a value.
reference The association between a variable and its value.

8.16 Exercises

Exercise 4: Download a copy of the file from www.py4e.com/code3/romeo.txt

Write a program to open the file `romeo.txt` and read it line by line. For each line, split the line into a list of words using the `split` function.

For each word, check to see if the word is already in a list. If the word is not in the list, add it to the list.

When the program completes, sort and print the resulting words in alphabetical order.

```
Enter file: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']
```

Exercise 5: Write a program to read through the mail box data and when you find line that starts with "From", you will split the line into words using the `split` function. We are interested in who sent the message, which is the second word on the From line.

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

You will parse the From line and print out the second word for each From line, then you will also count the number of From (not From:) lines and print out a count at the end.

This is a good sample output with a few lines removed:

```
python fromcount.py
Enter a file name: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...some output removed...]
```

```
ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
There were 27 lines in the file with From as the first word
```

Exercise 6: Rewrite the program that prompts the user for a list of numbers and prints out the maximum and minimum of the numbers at the end when the user enters "done". Write the program to store the numbers the user enters in a list and use the `max()` and `min()` functions to compute the maximum and minimum numbers after the loop completes.

```
Enter a number: 6
Enter a number: 2
Enter a number: 9
Enter a number: 3
Enter a number: 5
Enter a number: done
Maximum: 9.0
Minimum: 2.0
```

Chapter 9

Dictionaries

A dictionary is like a list, but more general. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type.

You can think of a dictionary as a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key-value pair or sometimes an item.

As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

The function `dict` creates a new dictionary with no items. Because `dict` is the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()
>>> print(eng2sp)
{}
```

The curly brackets, `{}`, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

```
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key `'one'` to the value "uno". If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> print(eng2sp)
{'one': 'uno'}
```

This output format is also an input format. For example, you can create a new dictionary with three items. But if you print `eng2sp`, you might be surprised:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> print(eng2sp)
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> print(eng2sp['two'])
'dos'
```

The key `'two'` always maps to the value "dos" so the order of the items doesn't matter.

If the key isn't in the dictionary, you get an exception:

```
>>> print(eng2sp['four'])
KeyError: 'four'
```

The `len` function works on dictionaries; it returns the number of key-value pairs:

```
>>> len(eng2sp)
3
```

The `in` operator works on dictionaries; it tells you whether something appears as a key in the dictionary (appearing as a value is not good enough).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

To see whether something appears as a value in a dictionary, you can use the method `values`, which returns the values as a list, and then use the `in` operator:

```
>>> vals = list(eng2sp.values())
>>> 'uno' in vals
True
```

The `in` operator uses different algorithms for lists and dictionaries. For lists, it uses a linear search algorithm. As the list gets longer, the search time gets longer in direct proportion to the length of the list. For dictionaries, Python uses an algorithm called a hash table that has a remarkable property: the `in` operator takes about the same amount of time no matter how many items there are in a dictionary. I won't explain why hash functions are so magical, but you can read more about it at wikipedia.org/wiki/Hash_table.

Exercise 1: [wordlist2]

Write a program that reads the words in `words.txt` and stores them as keys in a dictionary. It doesn't matter what the values are. Then you can use the `in` operator as a fast way to check whether a string is in the dictionary.

9.1 Dictionary as a set of counters

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.
2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function `ord`), use the number as an index into the list, and increment the appropriate counter.
3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An implementation is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

```
word = 'brontosaurus'
d = dict()
for c in word:
    if c not in d:
        d[c] = 1
    else:
        d[c] = d[c] + 1
print(d)
```

We are effectively computing a histogram, which is a statistical term for a set of counters (or frequencies).

The `for` loop traverses the string. Each time through the loop, if the character `c` is not in the dictionary, we create a new item with key `c` and the initial value 1 (since we have seen this letter once). If `c` is already in the dictionary we increment `d[c]`.

Here's the output of the program:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

The histogram indicates that the letters `'a'` and "b" appear once; "o" appears twice, and so on.

Dictionaries have a method called `get` that takes a key and a default value. If the key appears in the dictionary, `get` returns the corresponding value; otherwise it returns the default value. For example:

```
>>> counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print(counts.get('jan', 0))
100
>>> print(counts.get('tim', 0))
0
```

We can use `get` to write our histogram loop more concisely. Because the `get` method automatically handles the case where a key is not in a dictionary, we can reduce four lines down to one and eliminate the `if` statement.

```
word = 'brontosaurus'
d = dict()
for c in word:
    d[c] = d.get(c,0) + 1
print(d)
```

The use of the `get` method to simplify this counting loop ends up being a very commonly used "idiom" in Python and we will use it many times in the rest of the book. So you should take a moment and compare the loop using the `if` statement and `in` operator with the loop using the `get` method. They do exactly the same thing, but one is more succinct.

9.2 Dictionaries and files

One of the common uses of a dictionary is to count the occurrence of words in a file with some written text. Let's start with a very simple file of words taken from the text of Romeo and Juliet.

For the first set of examples, we will use a shortened and simplified version of the text with no punctuation. Later we will work with the text of the scene with punctuation included.

```
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
```

We will write a Python program to read through the lines of the file, break each line into a list of words, and then loop through each of the words in the line and count each word using a dictionary.

You will see that we have two `for` loops. The outer loop is reading the lines of the file and the inner loop is iterating through each of the words on that particular line. This is an example of a pattern called nested loops because one of the loops is the outer loop and the other loop is the inner loop.

Because the inner loop executes all of its iterations each time the outer loop makes a single iteration, we think of the inner loop as iterating "more quickly" and the outer loop as iterating more slowly.

The combination of the two nested loops ensures that we will count every word on every line of the input file.

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Code: http://www.py4e.com/code3/count1.py
```

When we run the program, we see a raw dump of all of the counts in unsorted hash order. (the `romeo.txt` file is available at www.py4e.com/code3/romeo.txt)

```
python count1.py
Enter the file name: romeo.txt
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,
'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,
'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,
'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

It is a bit inconvenient to look through the dictionary to find the most common words and their counts, so we need to add some more Python code to get us the output that will be more helpful.

9.3 Looping and dictionaries

If you use a dictionary as the sequence in a `for` statement, it traverses the keys of the dictionary. This loop prints each key and the corresponding value:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    print(key, counts[key])
```

Here's what the output looks like:

```
jan 100
chuck 1
annie 42
```

Again, the keys are in no particular order.

We can use this pattern to implement the various loop idioms that we have described earlier. For example if we wanted to find all the entries in a dictionary with a value above ten, we could write the following code:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    if counts[key] > 10 :
        print(key, counts[key])
```

The `for` loop iterates through the keys of the dictionary, so we must use the index operator to retrieve the corresponding value for each key. Here's what the output looks like:

```
jan 100
annie 42
```

We see only the entries with a value above 10.

If you want to print the keys in alphabetical order, you first make a list of the keys in the dictionary using the `keys` method available in dictionary objects, and then sort that list and loop through the sorted list, looking up each key and printing out key-value pairs in sorted order as follows:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = list(counts.keys())
print(lst)
lst.sort()
for key in lst:
    print(key, counts[key])
```

Here's what the output looks like:

```
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

First you see the list of keys in unsorted order that we get from the `keys` method. Then we see the key-value pairs in order from the `for` loop.

9.4 Advanced text parsing

In the above example using the file `romeo.txt`, we made the file as simple as possible by removing all punctuation by hand. The actual text has lots of punctuation, as shown below.

```
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,
```

Since the Python `split` function looks for spaces and treats words as tokens separated by spaces, we would treat the words "soft!" and "soft" as different words and create a separate dictionary entry for each word.

Also since the file has capitalization, we would treat "who" and "Who" as different words with different counts.

We can solve both these problems by using the string methods `lower`, `punctuation`, and `translate`. The `translate` is the most subtle of the methods. Here is the documentation for `translate`:

```
line.translate(str.maketrans(fromstr, tostr, deletestr))
```

Replace the characters in *`fromstr`* with the character in the same position in *`tostr`* and delete all characters that are in *`deletestr`*. The *`fromstr`* and *`tostr`* can be empty strings and the *`deletestr`* parameter can be omitted.

We will not specify the `table` but we will use the `deletechars` parameter to delete all of the punctuation. We will even let Python tell us the list of characters that it considers "punctuation":

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

The parameters used by `translate` were different in Python 2.0.

We make the following modifications to our program:

```
import string

fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()
for line in fhand:
    line = line.rstrip()
    line = line.translate(line.maketrans('', '', string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
```

```
        counts[word] += 1

print(counts)

# Code: http://www.py4e.com/code3/count2.py
```

Part of learning the "Art of Python" or "Thinking Pythonically" is realizing that Python often has built-in capabilities for many common data analysis problems. Over time, you will see enough example code and read enough of the documentation to know where to look to see if someone has already written something that makes your job much easier.

The following is an abbreviated version of the output:

```
Enter the file name: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2,
'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love': 24, 'cloak': 1,
a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40,
'maiden': 1, 'whiteupturned': 1, 'juliet': 32, 'gentleman': 1,
'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

Looking through this output is still unwieldy and we can use Python to give us exactly what we are looking for, but to do so, we need to learn about Python tuples. We will pick up this example once we learn about tuples.

9.5 Debugging

As you work with bigger datasets it can become unwieldy to debug by printing and checking data by hand. Here are some suggestions for debugging large datasets:

Scale down the input If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest example you can find. You can either edit the files themselves, or (better) modify the program so it reads only the first `n` lines.

If there is an error, you can reduce `n` to the smallest value that manifests the error, and then increase it gradually as you find and correct errors.

Check summaries and types Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers.

A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.

Write self-checks Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a "sanity check" because it detects results that are "completely illogical".

Another kind of check compares the results of two different computations to see if they are consistent. This is called a "consistency check".

Pretty print the output Formatting debugging output can make it easier to spot an error.

Again, time you spend building scaffolding can reduce the time you spend debugging.

9.6 Glossary

dictionary A mapping from a set of keys to their corresponding values.
hashtable The algorithm used to implement Python dictionaries.
hash function A function used by a hashtable to compute the location for a key.
histogram A set of counters.
implementation A way of performing a computation.
item Another name for a key-value pair.
key An object that appears in a dictionary as the first part of a key-value pair.
key-value pair The representation of the mapping from a key to a value.
lookup A dictionary operation that takes a key and finds the corresponding value.
nested loops When there are one or more loops "inside" of another loop. The inner loop runs to completion each time the outer loop runs once.
value An object that appears in a dictionary as the second part of a key-value pair. This is more specific than our previous use of the word "value".

9.7 Exercises

Exercise 2: Write a program that categorizes each mail message by which day of the week the commit was done. To do this look for lines that start with "From", then look for the third word and keep a running count of each of the days of the week. At the end of the program print out the contents of your dictionary (order does not matter).

Sample Line:

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

Sample Execution:

```
python dow.py
Enter a file name: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

Exercise 3: Write a program to read through a mail log, build a histogram using a dictionary to count how many messages have come from each email address, and print the dictionary.

```
Enter file name: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

Exercise 4: Add code to the above program to figure out who has the most messages in the file.

After all the data has been read and the dictionary has been created, look through the dictionary using a maximum loop (see Section [maximumloop]) to find who has the most messages and print how many messages the person has.

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

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

Exercise 5: This program records the domain name (instead of the address) where the message was sent from instead of who the mail came from (i.e., the whole email address). At the end of the program, print out the contents of your dictionary.

```
python schoolcount.py
Enter a file name: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```

Chapter 10

Tuples

10.1 Tuples are immutable

A tuple[1] is a sequence of values much like a list. The values stored in a tuple can be any type, and they are indexed by integers. The important difference is that tuples are immutable. Tuples are also comparable and hashable so we can sort lists of them and use tuples as key values in Python dictionaries.

Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses to help us quickly identify tuples when we look at Python code:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include the final comma:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Without the comma Python treats `('a')` as an expression with a string in parentheses that evaluates to a string:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

Another way to construct a tuple is the built-in function `tuple`. With no argument, it creates an empty tuple:

[1] Fun fact: The word "tuple" comes from the names given to sequences of numbers of varying lengths: single, double, triple, quadruple, quituple, sextuple, septuple, etc.

```
>>> t = tuple()
>>> print(t)
()
```

If the argument is a sequence (string, list, or tuple), the result of the call to `tuple` is a tuple with the elements of the sequence:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

Because `tuple` is the name of a constructor, you should avoid using it as a variable name.

Most list operators also work on tuples. The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

And the slice operator selects a range of elements.

```
>>> print(t[1:3])
('b', 'c')
```

But if you try to modify one of the elements of the tuple, you get an error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

You can't modify the elements of a tuple, but you can replace one tuple with another:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

10.2 Comparing tuples

The comparison operators work with tuples and other sequences. Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

The `sort` function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on.

This feature lends itself to a pattern called DSU for

Decorate a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,
Sort the list of tuples using the Python built-in `sort`, and
Undecorate by extracting the sorted elements of the sequence.

[DSU]

For example, suppose you have a list of words and you want to sort them from longest to shortest:

```
txt = 'but soft what light in yonder window breaks'
words = txt.split()
t = list()
for word in words:
    t.append((len(word), word))

t.sort(reverse=True)

res = list()
for length, word in t:
    res.append(word)

print(res)

# Code: http://www.py4e.com/code3/soft.py
```

The first loop builds a list of tuples, where each tuple is a word preceded by its length.

`sort` compares the first element, length, first, and only considers the second element to break ties. The keyword argument `reverse=True` tells `sort` to go in decreasing order.

The second loop traverses the list of tuples and builds a list of words in descending order of length. The four-character words are sorted in reverse alphabetical order, so "what" appears before "soft" in the following list.

The output of the program is as follows:

```
['yonder', 'window', 'breaks', 'light', 'what',
'soft', 'but', 'in']
```

Of course the line loses much of its poetic impact when turned into a Python list and sorted in descending word length order.

10.3 Tuple assignment

One of the unique syntactic features of the Python language is the ability to have a tuple on the left side of an assignment statement. This allows you to assign more than one variable at a time when the left side is a sequence.

In this example we have a two-element list (which is a sequence) and assign the first and second elements of the sequence to the variables `x` and `y` in a single statement.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

It is not magic, Python roughly translates the tuple assignment syntax to be the following:[2]

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

Stylistically when we use a tuple on the left side of the assignment statement, we omit the parentheses, but the following is an equally valid syntax:

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
>>>
```

A particularly clever application of tuple assignment allows us to swap the values of two variables in a single statement:

```
>>> a, b = b, a
```

Both sides of this statement are tuples, but the left side is a tuple of variables; the right side is a tuple of expressions. Each value on the right side is assigned to its respective

[2] Python does not translate the syntax literally. For example, if you try this with a dictionary, it will not work as might expect.

variable on the left side. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right must be the same:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

More generally, the right side can be any kind of sequence (string, list, or tuple). For example, to split an email address into a user name and a domain, you could write:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

The return value from `split` is a list with two elements; the first element is assigned to `uname`, the second to `domain`.

```
>>> print(uname)
monty
>>> print(domain)
python.org
```

10.4 Dictionaries and tuples

Dictionaries have a method called `items` that returns a list of tuples, where each tuple is a key-value pair:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
[('b', 1), ('a', 10), ('c', 22)]
```

As you should expect from a dictionary, the items are in no particular order.

However, since the list of tuples is a list, and tuples are comparable, we can now sort the list of tuples. Converting a dictionary to a list of tuples is a way for us to output the contents of a dictionary sorted by key:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> t
[('b', 1), ('a', 10), ('c', 22)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

The new list is sorted in ascending alphabetical order by the key value.

10.5 Multiple assignment with dictionaries

Combining `items`, tuple assignment, and `for`, you can see a nice code pattern for traversing the keys and values of a dictionary in a single loop:

```
for key, val in list(d.items()):
    print(val, key)
```

This loop has two iteration variables because `items` returns a list of tuples and `key, val` is a tuple assignment that successively iterates through each of the key-value pairs in the dictionary.

For each iteration through the loop, both `key` and `value` are advanced to the next key-value pair in the dictionary (still in hash order).

The output of this loop is:

```
10 a
22 c
1 b
```

Again, it is in hash key order (i.e., no particular order).

If we combine these two techniques, we can print out the contents of a dictionary sorted by the value stored in each key-value pair.

To do this, we first make a list of tuples where each tuple is `(value, key)`. The `items` method would give us a list of `(key, value)` tuples, but this time we want to sort by value, not key. Once we have constructed the list with the value-key tuples, it is a simple matter to sort the list in reverse order and print out the new, sorted list.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for key, val in d.items() :
...     l.append( (val, key) )
...
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

By carefully constructing the list of tuples to have the value as the first element of each tuple, we can sort the list of tuples and get our dictionary contents sorted by value.

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(str.maketrans('', '', 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.py4e.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.py4e.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.py4e.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.py4e.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.py4e.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.py4e.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.py4e.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.py4e.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.py4e.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.py4e.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.py4e.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.py4e.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 "wild-card" 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:

^ 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.

[^A-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.pr4e.org` 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\r\n\r\n'.encode()
mysock.send(cmd)

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

# Code: http://www.py4e.com/code3/socket1.py
```

First the program makes a connection to port 80 on the server www.py4e.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
Content-Length: 167
Connection: close
Content-Type: text/plain
```

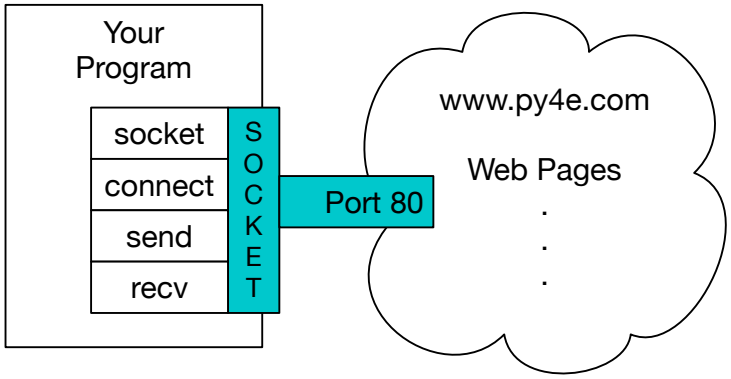


Figure 12.1: A Socket Connection

```
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/cover3.jpg HTTP/1.0\r\n\r\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.py4e.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

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

# Code: http://www.py4e.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

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')

counts = dict()
for line in fhand:
    words = line.decode().split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1
print(counts)

# Code: http://www.py4e.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.py4e.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.py4e.com/book.htm
http://www.greenteapress.com/thinkpython/thinkpython.html
http://allendowney.com/
http://www.py4e.com/code
http://www.lib.umich.edu/espresso-book-machine
http://www.py4e.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] 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.

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

# Or download the file
# http://www.py4e.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
import ssl
```

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

```
# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).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.py4e.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.py4e.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.py4e.com/code
http://www.py4e.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.py4e.com/code3/bs4.zip
# and unzip it in the same directory as this file


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

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
```

```
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urlopen(url, context=ctx).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.py4e.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/cover3.jpg').read()
fhand = open('cover3.jpg', 'wb')
fhand.write(img)
fhand.close()

# Code: http://www.py4e.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/cover3.jpg')
fhand = open('cover3.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.py4e.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.py4e.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.py4e.com/code3 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.

13.2 Parsing XML

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

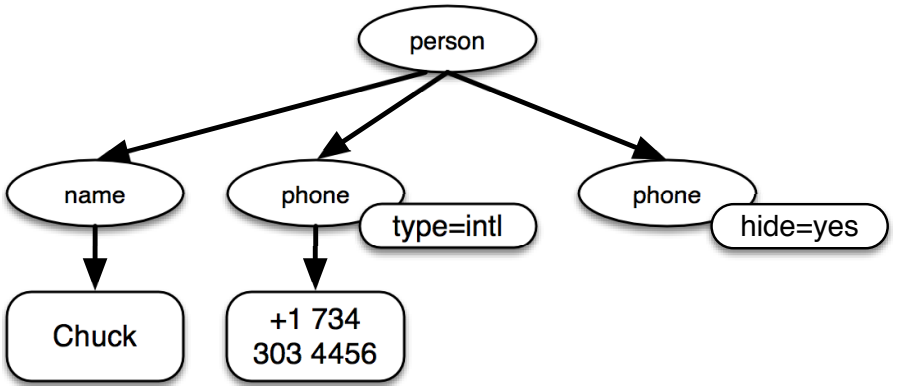


Figure 13.1: A Tree Representation of XML

```
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.py4e.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.py4e.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

data = '''
[
  { "id" : "001",
    "x" : "2",
    "name" : "Chuck"
  } ,
  { "id" : "009",
    "x" : "7",
    "name" : "Chuck"
  }
```

```
]'''

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

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

# Code: http://www.py4e.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

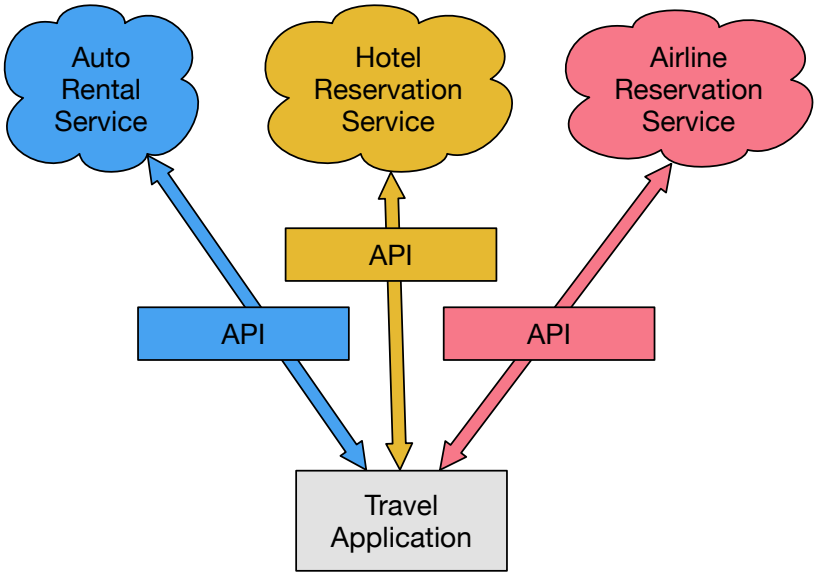


Figure 13.2: Service Oriented Architecture

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.

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

# Note that Google is increasingly requiring keys
# for this API
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(
        {'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.py4e.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.py4e.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.py4e.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 and get the four strings

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

# Code: http://www.py4e.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
import ssl

# https://apps.twitter.com/
# Create App and get the four strings, put them in hidden.py

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

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

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

# Code: http://www.py4e.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
import ssl

# https://apps.twitter.com/
# Create App and get the four strings, put them in hidden.py

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

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

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, context=ctx)
    data = connection.read().decode()

    js = json.loads(data)
    print(json.dumps(js, indent=2))

    headers = dict(connection.getheaders())
    print('Remaining', headers['x-rate-limit-remaining'])

    for u in js['users']:
        print(u['screen_name'])
        if 'status' not in u:
            print('   * No status found')
            continue
        s = u['status']['text']
        print('  ', s[:50])
```

 # Code: http://www.py4e.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.py4e.com/code3/geojson.py or www.py4e.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.py4e.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 convenient 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 capabilities of an object by looking at the output of the `dir()` function:

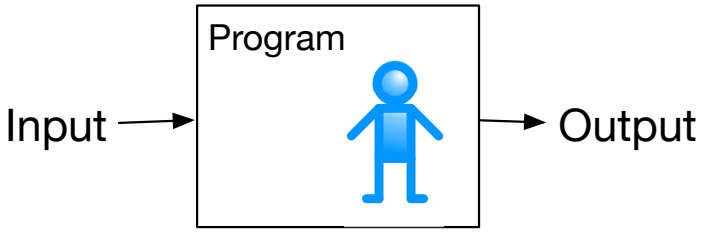


Figure 14.1: A Program

```
>>> 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.py4e.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".

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.py4e.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
import ssl

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).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.py4e.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.

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

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

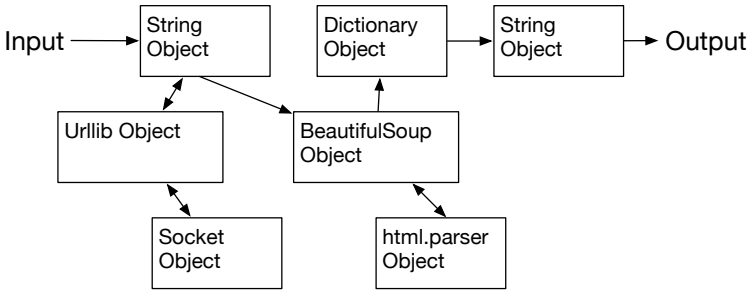


Figure 14.2: A Program as Network of Objects

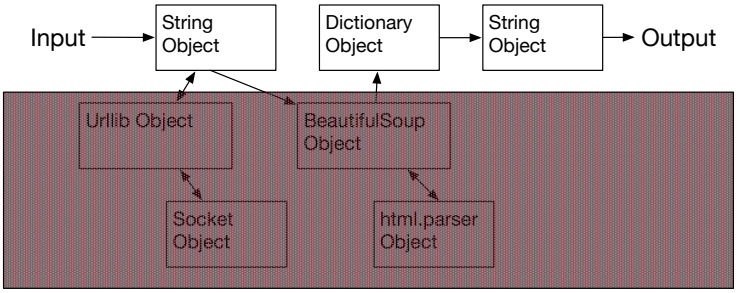


Figure 14.3: Ignoring Detail When Using an Object

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.

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

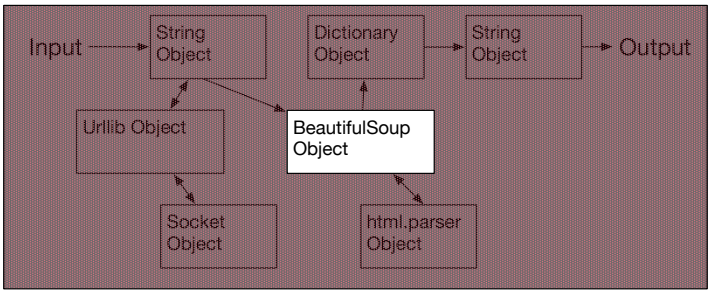


Figure 14.4: Ignoring Detail When Building an Object

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.py4e.com/code3/party2.py
```

Each method looks like a function, starting with the `def` keyword and consisting of an indented 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.

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



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:

```
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.py4e.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 variables 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.py4e.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 initial 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.py4e.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.py4e.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.py4e.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.py4e.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 attributes 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.

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 attributes 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' interchangeably with 'object'.

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

[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.

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.

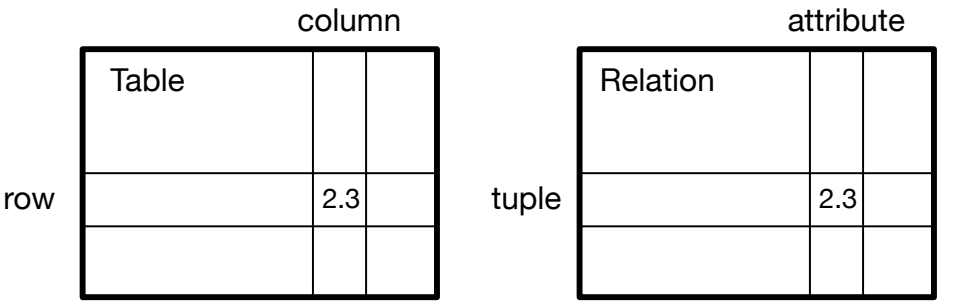


Figure 15.1: Relational Databases

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.

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

[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.

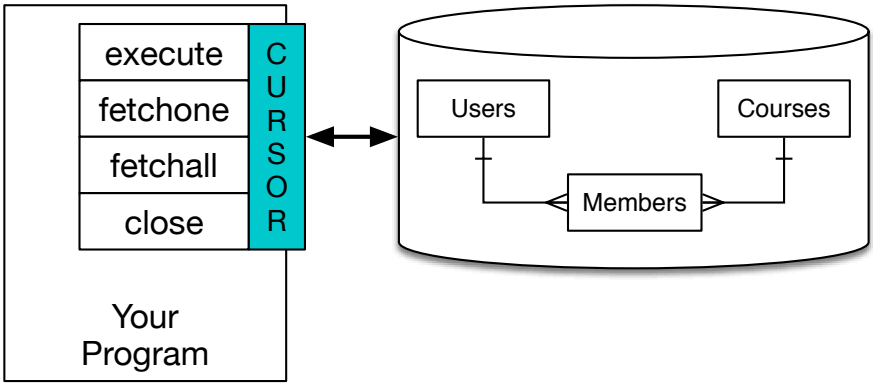


Figure 15.2: A Database Cursor

```
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.py4e.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.

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.py4e.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.

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`

Tracks

title	plays
Thunderstruck	20
My Way	15

Figure 15.3: Rows in a Table

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
import ssl

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)''')

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE
```

```
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, context=ctx)
    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.py4e.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.py4e.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:

```
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
import ssl

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

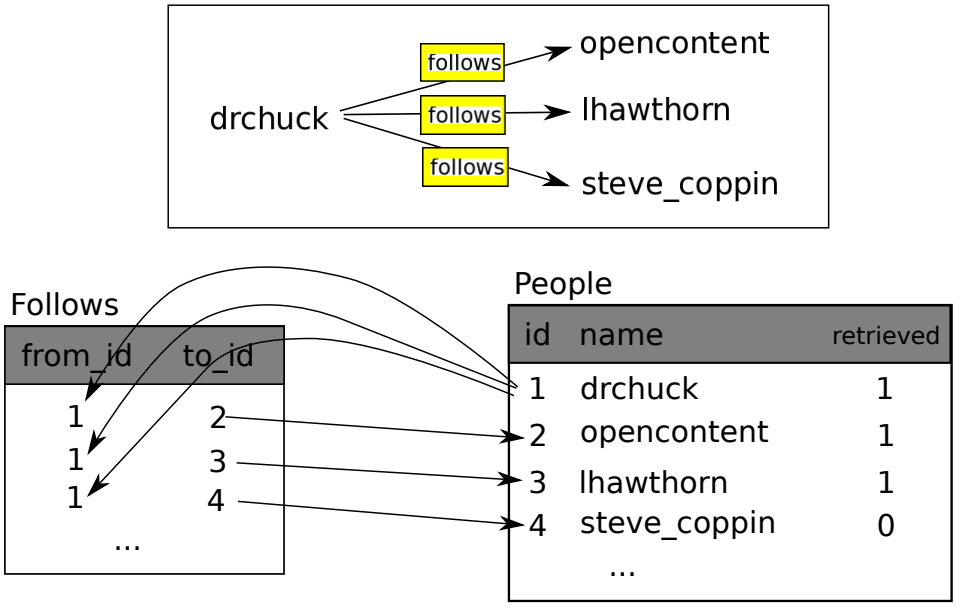


Figure 15.4: Relationships Between Tables

```
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))''')

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

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': '100'})
    print('Retrieving account', acct)
    try:
        connection = urllib.request.urlopen(url, context=ctx)
    except Exception as err:
        print('Failed to Retrieve', err)
        break

    data = connection.read().decode()
    headers = dict(connection.getheaders())

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

    try:
        js = json.loads(data)
    except:
        print('Unable to parse json')
        print(data)
        break

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

    if 'users' not in js:
        print('Incorrect JSON received')
        print(json.dumps(js, indent=4))
        continue

    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)
    print('Remaining', headers['x-rate-limit-remaining'])
    conn.commit()
cur.close()

# Code: http://www.py4e.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
```

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

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 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:

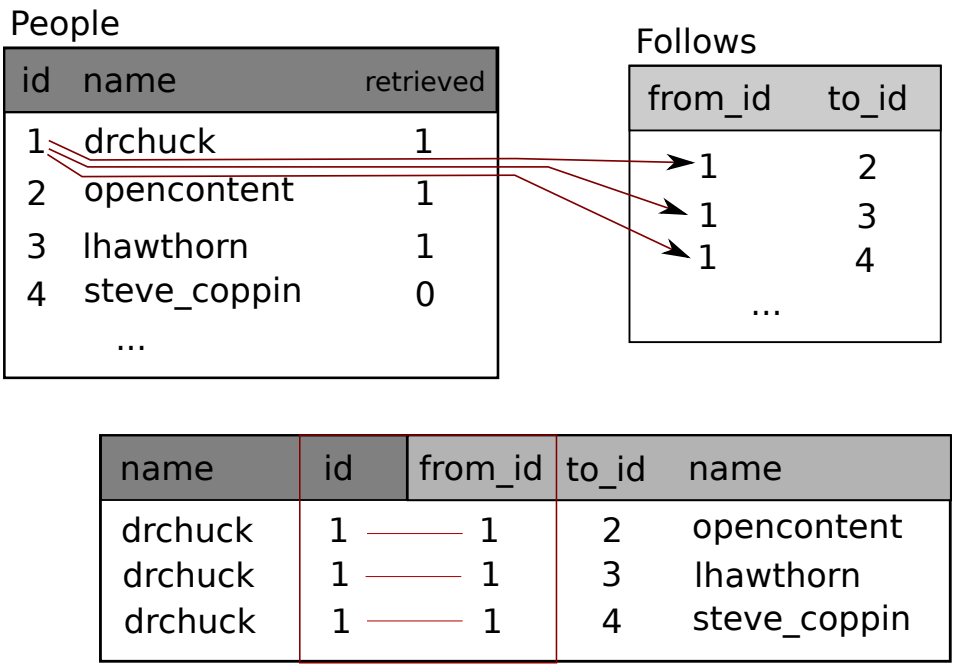


Figure 15.5: Connecting Tables Using JOIN

```
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 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.py4e.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.py4e.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 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:



Figure 16.1: A Google Map

```
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:

www.py4e.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.

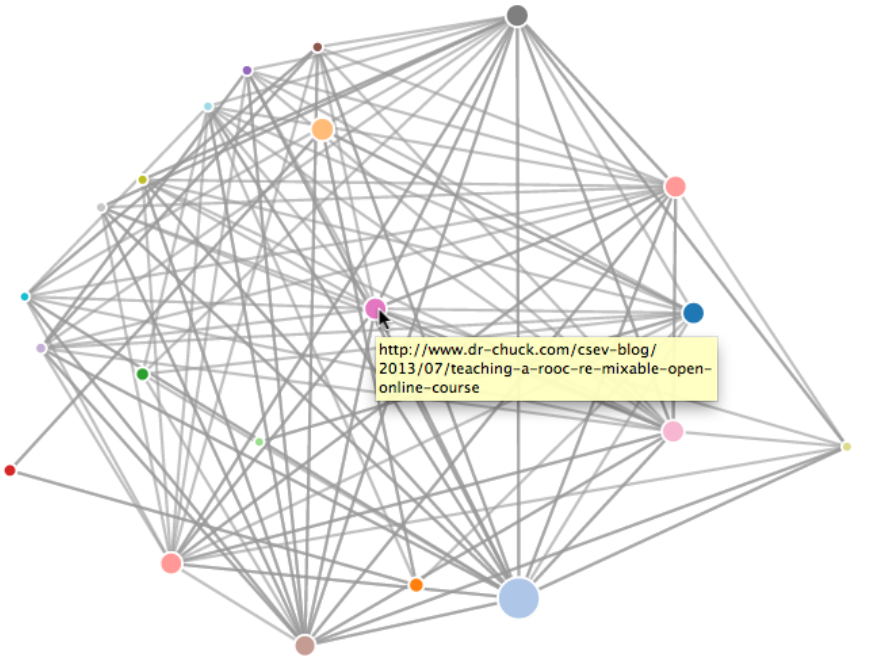


Figure 16.2: A Page Ranking

```
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 sprank.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.py4e.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:

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.



Figure 16.3: A Word Cloud from the Sakai Developer List

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:

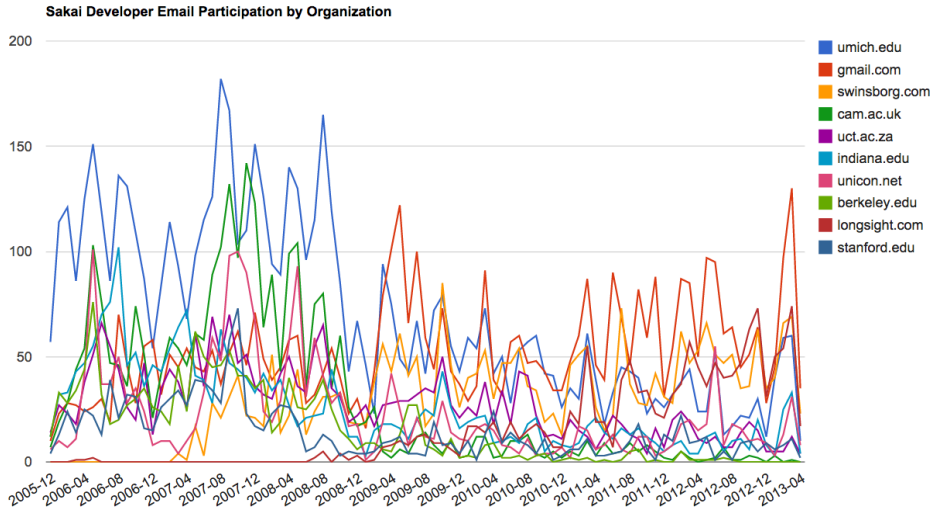


Figure 16.4: Sakai Mail Activity by Organization

```
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.

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/py4e/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, Chaochao 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

This work is licensed under a Creative Common Attribution-NonCommercial-ShareAlike 3.0 Unported License. This license is available at

creativecommons.org/licenses/by-nc-sa/3.0/.

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

access, 82
accumulator, 54
    sum, 52
algorithm, 45
aliasing, 89, 95
    copying to avoid, 92
alternative execution, 27
API, 157
    key, 153
append method, 84, 90
argument, 35, 39, 41, 42, 45, 90
    keyword, 109
    list, 90
    optional, 63, 87
arithmetic operator, 18
assignment, 81
    item, 60, 82, 108
    tuple, 110, 116
assignment statement, 16
attribute, 171, 195

BeautifulSoup, 140, 144, 162
binary file, 142
bisection, debugging by, 54
body, 33, 39, 45, 48
boolean expression, 33
boolean operator, 61
bracket
    squiggly, 97
bracket operator, 57, 82, 108
branch, 27, 33
break statement, 48
bug, 11
BY-SA, iv

cache, 197
catch, 78
CC-BY-SA, iv
celsius, 30
central processing unit, 11
chained conditional, 27, 33
character, 57
child class, 171
choice function, 38
class, 165, 171
    float, 15
    int, 15
    str, 15
class keyword, 164
close method, 77
colon, 39
comment, 21
comparable, 107, 116
comparison
    string, 61
    tuple, 108
compile, 11
composition, 42, 45
compound statement, 26, 33
concatenation, 60, 87
    list, 83, 90
condition, 26, 33, 48
conditional
    chained, 27, 33
    nested, 29, 33
conditional executions, 26
conditional statement, 26, 33
connect function, 175
consistency check, 104
constraint, 195
construct, 165
constructor, 167, 171
continue statement, 49
contributors, 209
conversion
    type, 36
copy
    slice, 59, 84
    to avoid aliasing, 92
count method, 63
counter, 54, 60, 66, 72, 99
counting and looping, 60
CPU, 11
Creative Commons License, iv

curl, 143
cursor, 195
cursor function, 175

data structure, 115, 116
database, 173
    indexes, 173
database browser, 195
database normalization, 195
debugging, 23, 32, 44, 65, 78, 91, 104, 115
    by bisection, 54
decorate-sort-undecorate pattern, 109
decrement, 47, 54
def keyword, 39
definition
    function, 39
del operator, 85
deletion, element of list, 85
delimiter, 87, 95
destructor, 167, 171
deterministic, 37, 45
development plan
    random walk programming, 115
dict function, 97
dictionary, 97, 105, 111
    looping with, 101
    traversal, 112
dir, 166
divisibility, 19
division
    floating-point, 18
dot notation, 38, 45, 62
DSU pattern, 109, 116

element, 81, 95
element deletion, 85
ElementTree, 145, 157
    find, 145
    findall, 147
    fromstring, 145
    get, 147
elif keyword, 28
ellipses, 39
else keyword, 27
email address, 111
empty list, 81
empty string, 66, 88
encapsulation, 60, 163
end of line character, 78
equivalence, 89
equivalent, 95

error
    semantic, 16
    shape, 115
error message, 16
evaluate, 18
exception
    IndexError, 58, 82
    IOError, 76
    KeyError, 98
    TypeError, 57, 60, 65, 108
    ValueError, 21, 111
experimental debugging, 115
expression, 18
    boolean, 33
extend method, 84
eXtensible Markup Language, 157

fahrenheit, 30
False 特殊値, 25
file, 69
    open, 69
    reading, 72
    writing, 77
file handle, 70
filter pattern, 73
findall, 122
flag, 66
float function, 36
float type, 15
floating-point division, 18
flow control, 137
flow of execution, 41, 45, 48
for loop, 58, 82
for statement, 50
foreign key, 195
format operator, 64, 66
format sequence, 64, 66
format string, 64, 66
Free Documentation License, GNU, 208, 209
frequency, 99
    letter, 117
fruitful function, 42, 45
function, 39, 45
    choice, 38
    connect, 175
    cursor, 175
    dict, 97
    float, 36
    int, 36
    len, 58, 98

list, 87
log, 38
open, 69, 76
print, 12
randint, 37
random, 37
repr, 78
reversed, 114
sorted, 114
sqrt, 39
str, 36
tuple, 107
function argument, 41
function call, 35, 45
function definition, 39, 40, 45
function object, 40
function parameter, 41
function, fruitful, 42
function, math, 38
function, reasons for, 44
function, trigonometric, 38
function, void, 42

gather, 116
geocoding, 150
get method, 99
GNU Free Documentation License, 208, 209
Google, 150
map, 197
page rank, 199
greedy, 121, 130, 139
greedy matching, 130
grep, 129, 130
guardian pattern, 31, 33, 66

hardware, 2
architecture, 2
hash function, 105
hash table, 98
hashable, 107, 114, 116
hashtable, 105
header, 39, 45
high-level language, 12
histogram, 99, 105
HTML, 140, 162

identical, 95
identity, 89
idiom, 92, 100, 102
if statement, 26
image
jpg, 135
immutability, 60, 66, 90, 107, 114
implementation, 99, 105
import statement, 45
in operator, 61, 82, 98
increment, 47, 54
indentation, 39
index, 57, 66, 82, 95, 97, 195
looping with, 83
negative, 58
slice, 59, 84
starting at zero, 57, 82
IndexError, 58, 82
infinite loop, 48, 54
inheritance, 171
initialization (before update), 47
instance, 165
int function, 36
int type, 15
interactive mode, 5, 12, 17, 43
interpret, 12
invocation, 62, 66
IOError, 76
is operator, 89
item, 66, 81
dictionary, 105
item assignment, 60, 82, 108
item update, 83
items method, 111
iteration, 47, 54

JavaScript Object Notation, 147, 157
join method, 87
jpg, 135
JSON, 147, 157

key, 97, 105
key-value pair, 97, 105, 111
keyboard input, 20
KeyError, 98
keys method, 102
keyword, 16, 17
def, 39
elif, 28
else, 27
keyword argument, 109

language
programming, 4
len function, 58, 98
letter frequency, 117

list, 81, 87, 95, 114
  as argument, 90
  concatenation, 83, 90
  copy, 84
  element, 82
  empty, 81
  function, 87
  index, 82
  membership, 82
  method, 84
  nested, 81, 83
  operation, 83
  repetition, 83
  slice, 84
  traversal, 82, 95
list object, 160
log function, 38
logical key, 195
lookup, 105
loop, 48
  for, 58, 82
  infinite, 48
  maximum, 52
  minimum, 52
  nested, 100, 105
  traversal, 58
  while, 47
looping
  with dictionaries, 101
  with indices, 83
  with strings, 60
looping and counting, 60
low-level language, 12

machine code, 12
main memory, 12
math function, 38
membership
  dictionary, 98
  list, 82
  set, 98
message, 171
method, 62, 66, 171
  append, 84, 90
  close, 77
  count, 63
  extend, 84
  get, 99
  items, 111
  join, 87
  keys, 102
  pop, 85
  remove, 85
  sort, 84, 92, 108
  split, 87, 111
  string, 67
  values, 98
  void, 85
method, list, 84
mnemonic, 21
module, 38, 45
  random, 37
  sqlite3, 174
module object, 38
modulus operator, 19
mutability, 60, 82, 84, 90, 107, 114

negative index, 58
nested conditional, 29, 33
nested list, 81, 83, 95
nested loops, 100, 105
newline, 20, 71, 77, 78
non-greedy, 139
None special value, 43, 52, 85
normalization, 195
number, random, 37

OAuth, 153
object, 60, 66, 88, 89, 95, 165
  function, 40
object lifecycle, 167
object-oriented, 159
open function, 69, 76
operand, 18
operator
  boolean, 61
  bracket, 57, 82, 108
  del, 85
  format, 64, 66
  in, 61, 82, 98
  is, 89
  modulus, 19
  slice, 59, 84, 91, 108
  string, 19
operator, arithmetic, 18
optional argument, 63, 87
order of operations, 19

parameter, 41, 45, 90
parent class, 171
parentheses
  argument in, 35

empty, 39, 62
overriding precedence, 19
parameters in, 41
regular expression, 125, 139
tuples in, 107
parse, 12
parsing
HTML, 140, 162
parsing HTML, 138
pass statement, 27
pattern
decorate-sort-undecorate, 109
DSU, 109
filter, 73
guardian, 31, 33, 66
search, 67
swap, 110
PEMDAS, 19
persistence, 69
pi, 39
pop method, 85
port, 144
portability, 12
primary key, 195
print function, 12
problem solving, 3, 12
program, 9, 12
programming language, 4
prompt, 12, 20
pseudorandom, 37, 45
Python 2.0, 18, 20
Python 3.0, 18
Pythonic, 77, 78

QA, 76, 78
Quality Assurance, 76, 78
quotation mark, 15, 59

radian, 38
randint function, 37
random function, 37
random module, 37
random number, 37
random walk programming, 115
rate limiting, 151
re module, 119
reference, 90, 95
aliasing, 89
regex, 119
character sets(brackets), 123
findall, 122
parentheses, 125, 139
search, 119
wild card, 120
regular expressions, 119
relation, 195
remove method, 85
repetition
list, 83
repr function, 78
return value, 35, 45
reversed function, 114
Romeo and Juliet, 95, 100, 102, 109, 112
rules of precedence, 19

sanity check, 104
scaffolding, 105
scatter, 116
script, 8
script mode, 17, 43
search pattern, 67
secondary memory, 12, 69
semantic error, 12, 16
semantics, 12
sequence, 57, 67, 81, 87, 107, 114
Service Oriented Architecture, 157
set membership, 98
shape, 116
shape error, 115
short circuit, 31, 33
sine function, 38
singleton, 107, 116
slice, 67
copy, 59, 84
list, 84
string, 59
tuple, 108
update, 84
slice operator, 59, 84, 91, 108
SOA, 157
socket, 144
sort method, 84, 92, 108
sorted function, 114
source code, 12
special value
None, 43, 52, 85
spider, 144
split method, 87, 111
sqlite3 module, 174
sqrt function, 39
squiggly bracket, 97
statement, 17

assignment, 16
break, 48
compound, 26
conditional, 26, 33
continue, 49
for, 50, 58, 82
if, 26
import, 45
pass, 27
try, 76
while, 47
str function, 36
string, 15, 87, 114
comparison, 61
empty, 88
find, 120
immutable, 60
method, 62
operation, 19
slice, 59
split, 125
startswith, 120
string method, 67
string representation, 78
string type, 15
swap pattern, 110

temperature conversion, 30
text file, 78
time, 137
time.sleep, 137
traceback, 30, 32, 33
traversal, 58, 67, 99, 101, 109
list, 82
traverse
dictionary, 112
trigonometric function, 38
True 特殊值, 25
try statement, 76
tuple, 107, 114, 116, 195
as key in dictionary, 114
assignment, 110
comparison, 108
in brackets, 114
singleton, 107
slice, 108
tuple assignment, 116
tuple function, 107
type, 15, 166
dict, 97
file, 69
list, 81
tuple, 107
type conversion, 36
TypeError, 57, 60, 65, 108
typographical error, 115

underscore character, 17
Unicode, 177
update, 47
item, 83
slice, 84
urllib
image, 135
use before def, 41

value, 15, 88, 89, 105
ValueError, 21, 111
values method, 98
variable, 16
updating, 47
Visualization
map, 197
networks, 199
page rank, 199
void function, 42, 45
void method, 85

web
scraping, 138
web service, 150
while loop, 47
whitespace, 33, 44, 78
wild card, 120, 130

XML, 157

zero, index starting at, 57, 82

与运算符, 26
串联, 19, 23
优先级, 24
例外, 23

值, 24
关键字, 24
助记法, 24
区分大小写, 变量名, 23
变量, 24
字符串, 24
定义前使用, 23
布尔类型, 25
布尔表达式, 25

或运算符, 26
整数, 24
模运算, 24
比较运算符, 25
注释, 23
浮点数, 24

特殊值
    False, 25
    True, 25
类型, 24
    布尔, 25

表达式, 24
    布尔, 25
语义错误, 23
语句, 24
语法错误, 23
赋值, 23
运算优先级, 24
运算对象, 24
运算符, 24
    与, 26
    或, 26
    模, 24
    比较, 25
    逻辑, 25, 26
    非, 26
运算顺序, 23
运行错误, 23

逻辑运算符, 25, 26
错误
    语义, 23
    语法, 23
    运行, 23
错误信息, 23
非运算符, 26