Already know Python but want to learn more? Dive into a variety of topics used in real-world applications.

Covers regular expressions, Internet/network programming, GUIs, SQL/databases/ORMs, threading, and Web development.

Learn about contemporary development trends such as Google+, Twitter, MongoDB, OAuth, Python 3 migration, and Java/Pytho.

Presents brand new material on Django, Google App Engine, CSV/JSON/XML, and Microsoft Office.

Includes Python 2 and 3 code samples to get you started right away!

Provides code snippets, interactive examples, and practical exercises to help build your Python skills.

WESLEY J. CHUN

FREE SAMPLE CHAPTER

SHARE WITH OTHERS
“The simplified yet deep level of detail, comprehensive coverage of material, and informative historical references make this book perfect for the classroom... An easy read, with complex examples presented simply, and great historical references rarely found in such books. Awesome!”

—Gloria W.

**Praise for the Previous Edition**

“The long-awaited second edition of Wesley Chun’s *Core Python Programming* proves to be well worth the wait—its deep and broad coverage and useful exercises will help readers learn and practice good Python.”

—Alex Martelli, author of *Python in a Nutshell* and editor of *Python Cookbook*

“There has been lot of good buzz around Wesley Chun’s *Core Python Programming*. It turns out that all the buzz is well earned. I think this is the best book currently available for learning Python. I would recommend Chun’s book over *Learning Python* (O’Reilly), *Programming Python* (O’Reilly), or *The Quick Python Book* (Manning).”

—David Mertz, Ph.D., IBM DeveloperWorks

“I have been doing a lot of research [on] Python for the past year and have seen a number of positive reviews of your book. The sentiment expressed confirms the opinion that *Core Python Programming* is now considered the standard introductory text.”

—Richard Ozaki, Lockheed Martin

“Finally, a book good enough to be both a textbook and a reference on the Python language now exists.”

—Michael Baxter, *Linux Journal*

“Very well written. It is the clearest, friendliest book I have come across yet for explaining Python, and putting it in a wider context. It does not presume a large amount of other experience. It does go into some important Python topics carefully and in depth. Unlike too many beginner books, it never condescends or tortures the reader with childish hide-and-seek prose games. [It] sticks to gaining a solid grasp of Python syntax and structure.”

—http://python.org bookstore Web site
“[If] I could only own one Python book, it would be Core Python Programming by Wesley Chun. This book manages to cover more topics in more depth than Learning Python but includes it all in one book that also more than adequately covers the core language. [If] you are in the market for just one book about Python, I recommend this book. You will enjoy reading it, including its wry programmer’s wit. More importantly, you will learn Python. Even more importantly, you will find it invaluable in helping you in your day-to-day Python programming life. Well done, Mr. Chun!”

—Ron Stephens, Python Learning Foundation

“I think the best language for beginners is Python, without a doubt. My favorite book is Core Python Programming.”

—s003apr, MP3Car.com Forums

“Personally, I really like Python. It’s simple to learn, completely intuitive, amazingly flexible, and pretty darned fast. Python has only just started to claim mindshare in the Windows world, but look for it to start gaining lots of support as people discover it. To learn Python, I’d start with Core Python Programming by Wesley Chun.”

—Bill Boswell, MCSE, Microsoft Certified Professional Magazine Online

“If you learn well from books, I suggest Core Python Programming. It is by far the best I’ve found. I’m a Python newbie as well and in three months’ time I’ve been able to implement Python in projects at work (automating MSOffice, SQL DB stuff, etc.).”

—ptonman, Dev Shed Forums

“Python is simply a beautiful language. It’s easy to learn, it’s cross-platform, and it works. It has achieved many of the technical goals that Java strives for. A one-sentence description of Python would be: ‘All other languages appear to have evolved over time—but Python was designed.’ And it was designed well. Unfortunately, there aren’t a large number of books for Python. The best one I’ve run across so far is Core Python Programming.”

—Chris Timmons, C. R. Timmons Consulting

“If you like the Prentice Hall Core series, another good full-blown treatment to consider would be Core Python Programming. It addresses in elaborate concrete detail many practical topics that get little, if any, coverage in other books.”

—Mitchell L. Model, MLM Consulting
The Core Series is designed to provide you – the experienced programmer – with the essential information you need to quickly learn and apply the latest, most important technologies.

Authors in The Core Series are seasoned professionals who have pioneered the use of these technologies to achieve tangible results in real-world settings. These experts:
• Share their practical experiences
• Support their instruction with real-world examples
• Provide an accelerated, highly effective path to learning the subject at hand

The resulting book is a no-nonsense tutorial and thorough reference that allows you to quickly produce robust, production-quality code.

Visit informit.com/coreseries for a complete list of available publications.
Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsonedgroup.com
For sales outside the United States please contact:
International Sales
international@pearson.com
Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data
Chun, Wesley.
Core python applications programming / Wesley J. Chun. — 3rd ed.
p. cm.
Includes index.
1. Python (Computer program language) I. Chun, Wesley. Core Python programming. II. Title.
QA76.73.P98C48 2012
005.1'17—dc23 2011052903

Copyright © 2012 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.


Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
Second printing, June 2013
To my parents,
who taught me that everybody is different.

And to my wife,
who lives with someone who is different.
This page intentionally left blank
## CONTENTS

- Preface xv
- Acknowledgments xxvii
- About the Author xxxi

### Part I General Application Topics 1

#### Chapter 1 Regular Expressions 2
1.1 Introduction/Motivation 3
1.2 Special Symbols and Characters 6
1.3 Regexes and Python 16
1.4 Some Regex Examples 36
1.5 A Longer Regex Example 41
1.6 Exercises 48

#### Chapter 2 Network Programming 53
2.1 Introduction 54
2.2 What Is Client/Server Architecture? 54
2.3 Sockets: Communication Endpoints 58
2.4 Network Programming in Python 61
2.5 *The SocketServer Module 79
2.6 *Introduction to the Twisted Framework 84
2.7 Related Modules 88
2.8 Exercises 89
### Chapter 3  Internet Client Programming  94
3.1 What Are Internet Clients?  95
3.2 Transferring Files  96
3.3 Network News  104
3.4 E-Mail  114
3.5 In Practice
   3.5.1 E-Mail Composition  131
   3.5.2 E-Mail Parsing  134
   3.5.3 Web-Based Cloud E-Mail Services  135
   3.5.4 Best Practices: Security, Refactoring  136
   3.5.5 Yahoo! Mail  138
   3.5.6 Gmail  144
3.6 Related Modules  146
3.7 Exercises  148

### Chapter 4  Multithreaded Programming  156
4.1 Introduction/Motivation  157
4.2 Threads and Processes  158
4.3 Threads and Python  160
4.4 The thread Module  164
4.5 The threading Module  169
4.6 Comparing Single vs. Multithreaded Execution  180
4.7 Multithreading in Practice  182
4.8 Producer-Consumer Problem and the Queue/queue Module  202
4.9 Alternative Considerations to Threads  206
4.10 Related Modules  209
4.11 Exercises  210

### Chapter 5  GUI Programming  213
5.1 Introduction  214
5.2 Tkinter and Python Programming  216
5.3 Tkinter Examples  221
5.4 A Brief Tour of Other GUIs  236
5.5 Related Modules and Other GUIs  247
5.6 Exercises  250

### Chapter 6  Database Programming  253
6.1 Introduction  254
6.2 The Python DB-API  259
6.3 ORMs  289
6.4 Non-Relational Databases  309
6.5 Related References  316
6.6 Exercises  319
## Chapter 7  *Programming Microsoft Office*  324

7.1 Introduction  325  
7.2 COM Client Programming with Python  326  
7.3 Introductory Examples  328  
7.4 Intermediate Examples  338  
7.5 Related Modules/Packages  357  
7.6 Exercises  357  

## Chapter 8  Extending Python  364

8.1 Introduction/Motivation  365  
8.2 Extending Python by Writing Extensions  368  
8.3 Related Topics  384  
8.4 Exercises  388  

## Part II  Web Development  389

## Chapter 9  Web Clients and Servers  390

9.1 Introduction  391  
9.2 Python Web Client Tools  396  
9.3 Web Clients  410  
9.4 Web (HTTP) Servers  428  
9.5 Related Modules  433  
9.6 Exercises  436  

## Chapter 10  Web Programming: CGI and WSGI  441

10.1 Introduction  442  
10.2 Helping Web Servers Process Client Data  442  
10.3 Building CGI Applications  446  
10.4 Using Unicode with CGI  464  
10.5 Advanced CGI  466  
10.6 Introduction to WSGI  478  
10.7 Real-World Web Development  487  
10.8 Related Modules  488  
10.9 Exercises  490  

## Chapter 11  Web Frameworks: Django  493

11.1 Introduction  494  
11.2 Web Frameworks  494  
11.3 Introduction to Django  496  
11.4 Projects and Apps  501  
11.5 Your “Hello World” Application (A Blog)  507  
11.6 Creating a Model to Add Database Service  509  
11.7 The Python Application Shell  514  
11.8 The Django Administration App  518  
11.9 Creating the Blog’s User Interface  527
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.10</td>
<td>Improving the Output</td>
<td>537</td>
</tr>
<tr>
<td>11.11</td>
<td>Working with User Input</td>
<td>542</td>
</tr>
<tr>
<td>11.12</td>
<td>Forms and Model Forms</td>
<td>546</td>
</tr>
<tr>
<td>11.13</td>
<td>More About Views</td>
<td>551</td>
</tr>
<tr>
<td>11.14</td>
<td>*Look-and-Feel Improvements</td>
<td>553</td>
</tr>
<tr>
<td>11.15</td>
<td>*Unit Testing</td>
<td>554</td>
</tr>
<tr>
<td>11.16</td>
<td>*An Intermediate Django App: The TweetApprover</td>
<td>564</td>
</tr>
<tr>
<td>11.17</td>
<td>Resources</td>
<td>597</td>
</tr>
<tr>
<td>11.18</td>
<td>Conclusion</td>
<td>597</td>
</tr>
<tr>
<td>11.19</td>
<td>Exercises</td>
<td>598</td>
</tr>
</tbody>
</table>

**Chapter 12 Cloud Computing: Google App Engine**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1</td>
<td>Introduction</td>
<td>605</td>
</tr>
<tr>
<td>12.2</td>
<td>What Is Cloud Computing?</td>
<td>605</td>
</tr>
<tr>
<td>12.3</td>
<td>The Sandbox and the App Engine SDK</td>
<td>612</td>
</tr>
<tr>
<td>12.4</td>
<td>Choosing an App Engine Framework</td>
<td>617</td>
</tr>
<tr>
<td>12.5</td>
<td>Python 2.7 Support</td>
<td>626</td>
</tr>
<tr>
<td>12.6</td>
<td>Comparisons to Django</td>
<td>628</td>
</tr>
<tr>
<td>12.7</td>
<td>Starting “Hello World”</td>
<td>628</td>
</tr>
<tr>
<td>12.8</td>
<td>Creating “Hello World” Manually (Zip File Users)</td>
<td>629</td>
</tr>
<tr>
<td>12.9</td>
<td>Uploading your Application to Google</td>
<td>629</td>
</tr>
<tr>
<td>12.10</td>
<td>Morphing “Hello World” into a Simple Blog</td>
<td>631</td>
</tr>
<tr>
<td>12.11</td>
<td>Adding Memcache Service</td>
<td>647</td>
</tr>
<tr>
<td>12.12</td>
<td>Static Files</td>
<td>651</td>
</tr>
<tr>
<td>12.13</td>
<td>Adding Users Service</td>
<td>652</td>
</tr>
<tr>
<td>12.14</td>
<td>Remote API Shell</td>
<td>654</td>
</tr>
<tr>
<td>12.15</td>
<td>Lightning Round (with Python Code)</td>
<td>656</td>
</tr>
<tr>
<td>12.16</td>
<td>Sending Instant Messages by Using XMPP</td>
<td>660</td>
</tr>
<tr>
<td>12.17</td>
<td>Processing Images</td>
<td>662</td>
</tr>
<tr>
<td>12.18</td>
<td>Task Queues (Unscheduled Tasks)</td>
<td>663</td>
</tr>
<tr>
<td>12.19</td>
<td>Profiling with Appstats</td>
<td>670</td>
</tr>
<tr>
<td>12.20</td>
<td>The URLfetch Service</td>
<td>672</td>
</tr>
<tr>
<td>12.21</td>
<td>Lightning Round (without Python Code)</td>
<td>673</td>
</tr>
<tr>
<td>12.22</td>
<td>Vendor Lock-In</td>
<td>675</td>
</tr>
<tr>
<td>12.23</td>
<td>Resources</td>
<td>676</td>
</tr>
<tr>
<td>12.24</td>
<td>Conclusion</td>
<td>679</td>
</tr>
<tr>
<td>12.25</td>
<td>Exercises</td>
<td>680</td>
</tr>
</tbody>
</table>

**Chapter 13 Web Services**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1</td>
<td>Introduction</td>
<td>685</td>
</tr>
<tr>
<td>13.2</td>
<td>The Yahoo! Finance Stock Quote Server</td>
<td>685</td>
</tr>
<tr>
<td>13.3</td>
<td>Microblogging with Twitter</td>
<td>690</td>
</tr>
<tr>
<td>13.4</td>
<td>Exercises</td>
<td>707</td>
</tr>
</tbody>
</table>
Part III Supplemental/Experimental 713

Chapter 14 Text Processing 714
14.1 Comma-Separated Values 715
14.2 JavaScript Object Notation 719
14.3 Extensible Markup Language 724
14.4 References 738
14.5 Related Modules 740
14.6 Exercises 740

Chapter 15 Miscellaneous 743
15.1 Jython 744
15.2 Google+ 748
15.3 Exercises 759

Appendix A Answers to Selected Exercises 763

Appendix B Reference Tables 768

Appendix C Python 3: The Evolution of a Programming Language 798
C.1 Why Is Python Changing? 799
C.2 What Has Changed? 799
C.3 Migration Tools 805
C.4 Conclusion 806
C.5 References 806

Appendix D Python 3 Migration with 2.6+ 807
D.1 Python 3: The Next Generation 807
D.2 Integers 809
D.3 Built-In Functions 812
D.4 Object-Oriented Programming: Two Different Class Objects 814
D.5 Strings 815
D.6 Exceptions 816
D.7 Other Transition Tools and Tips 817
D.8 Writing Code That is Compatible in Both Versions 2.x and 3.x 818
D.9 Conclusion 822

Index 823
This page intentionally left blank
Welcome to the Third Edition of Core Python Applications Programming!

We are delighted that you have engaged us to help you learn Python as quickly and as deeply as possible. The goal of the Core Python series of books is not to just teach developers the Python language; we want you to develop enough of a personal knowledge base to be able to develop software in any application area.

In our other Core Python offerings, Core Python Programming and Core Python Language Fundamentals, we not only teach you the syntax of the Python language, but we also strive to give you in-depth knowledge of how Python works under the hood. We believe that armed with this knowledge, you will write more effective Python applications, whether you’re a beginner to the language or a journeyman (or journeywoman!).

Upon completion of either or any other introductory Python books, you might be satisfied that you have learned Python and learned it well. By completing many of the exercises, you’re probably even fairly confident in your newfound Python coding skills. Still, you might be left wondering, “Now what? What kinds of applications can I build with Python?” Perhaps you learned Python for a work project that’s constrained to a very narrow focus. “What else can I build with Python?”
About this Book

In *Core Python Applications Programming*, you will take all the Python knowledge gained elsewhere and develop new skills, building up a toolset with which you’ll be able to use Python for a variety of general applications. These advanced topics chapters are meant as intros or “quick dives” into a variety of distinct subjects. If you’re moving toward the specific areas of application development covered by any of these chapters, you’ll likely discover that they contain more than enough information to get you pointed in the right direction. Do not expect an in-depth treatment because that will detract from the breadth-oriented treatment that this book is designed to convey.

Like all other Core Python books, throughout this one, you will find many examples that you can try right in front of your computer. To hammer the concepts home, you will also find fun and challenging exercises at the end of every chapter. These easy and intermediate exercises are meant to test your learning and push your Python skills. There simply is no substitute for hands-on experience. We believe you should not only pick up Python programming skills but also be able to master them in as short a time period as possible.

Because the best way for you to extend your Python skills is through practice, you will find these exercises to be one of the greatest strengths of this book. They will test your knowledge of chapter topics and definitions as well as motivate you to code as much as possible. There is no substitute for improving your skills more effectively than by building applications. You will find easy, intermediate, and difficult problems to solve. It is also here that you might need to write one of those “large” applications that many readers wanted to see in the book, but rather than scripting them—which frankly doesn’t do you all that much good—you gain by jumping right in and doing it yourself. Appendix A, “Answers to Selected Exercises,” features answers to selected problems from each chapter. As with the second edition, you’ll find useful reference tables collated in Appendix B, “Reference Tables.”

I’d like to personally thank all readers for your feedback and encouragement. You’re the reason why I go through the effort of writing these books. I encourage you to keep sending your feedback and help us make a fourth edition possible, and even better than its predecessors!
Who Should Read This Book?

This book is meant for anyone who already knows some Python but wants to know more and expand their application development skillset. Python is used in many fields, including engineering, information technology, science, business, entertainment, and so on. This means that the list of Python users (and readers of this book) includes but is not limited to

- Software engineers
- Hardware design/CAD engineers
- QA/testing and automation framework developers
- IS/IT/system and network administrators
- Scientists and mathematicians
- Technical or project management staff
- Multimedia or audio/visual engineers
- SCM or release engineers
- Web masters and content management staff
- Customer/technical support engineers
- Database engineers and administrators
- Research and development engineers
- Software integration and professional services staff
- Collegiate and secondary educators
- Web service engineers
- Financial software engineers
- And many others!

Some of the most famous companies that use Python include Google, Yahoo!, NASA, Lucasfilm/Industrial Light and Magic, Red Hat, Zope, Disney, Pixar, and Dreamworks.
The Author and Python

I discovered Python over a decade ago at a company called Four11. At the time, the company had one major product, the Four11.com White Page directory service. Python was being used to design its next product: the Rocketmail Web-based e-mail service that would eventually evolve into what today is Yahoo! Mail.

It was fun learning Python and being on the original Yahoo! Mail engineering team. I helped re-design the address book and spell checker. At the time, Python also became part of a number of other Yahoo! sites, including People Search, Yellow Pages, and Maps and Driving Directions, just to name a few. In fact, I was the lead engineer for People Search.

Although Python was new to me then, it was fairly easy to pick up—much simpler than other languages I had learned in the past. The scarcity of textbooks at the time led me to use the Library Reference and Quick Reference Guide as my primary learning tools; it was also a driving motivation for the book you are reading right now.

Since my days at Yahoo!, I have been able to use Python in all sorts of interesting ways at the jobs that followed. In each case, I was able to harness the power of Python to solve the problems at hand, in a timely manner. I have also developed several Python courses and have used this book to teach those classes—truly eating my own dogfood.

Not only are the Core Python books great learning devices, but they’re also among the best tools with which to teach Python. As an engineer, I know what it takes to learn, understand, and apply a new technology. As a professional instructor, I also know what is needed to deliver the most effective sessions for clients. These books provide the experience necessary to be able to give you real-world analogies and tips that you cannot get from someone who is “just a trainer” or “just a book author.”

What to Expect of the Writing Style: Technical, Yet Easy Reading

Rather than being strictly a “beginners” book or a pure, hard-core computer science reference book, my instructional experience has taught me that an easy-to-read, yet technically oriented book serves the purpose the best, which is to get you up to speed on Python as quickly as possible so that you can apply it to your tasks posthaste. We will introduce concepts
coupled with appropriate examples to expedite the learning process. At the end of each chapter you will find numerous exercises to reinforce some of the concepts and ideas acquired in your reading.

We are thrilled and humbled to be compared with Bruce Eckel’s writing style (see the reviews to the first edition at the book’s Web site, http://corepython.com). This is not a dry college textbook. Our goal is to have a conversation with you, as if you were attending one of my well-received Python training courses. As a lifelong student, I constantly put myself in my student’s shoes and tell you what you need to hear in order to learn the concepts as quickly and as thoroughly as possible. You will find reading this book fast and easy, without losing sight of the technical details.

As an engineer, I know what I need to tell you in order to teach you a concept in Python. As a teacher, I can take technical details and boil them down into language that is easy to understand and grasp right away. You are getting the best of both worlds with my writing and teaching styles, but you will enjoy programming in Python even more.

Thus, you’ll notice that even though I’m the sole author, I use the “third-person plural” writing structure; that is to say, I use verbiage such as “we” and “us” and “our,” because in the grand scheme of this book, we’re all in this together, working toward the goal of expanding the Python programming universe.

**About This Third Edition**

At the time the first edition of this book was published, Python was entering its second era with the release of version 2.0. Since then, the language has undergone significant improvements that have contributed to the overall continued success, acceptance, and growth in the use of the language. Deficiencies have been removed and new features added that bring a new level of power and sophistication to Python developers worldwide. The second edition of the book came out in 2006, at the height of Python’s ascendance, during the time of its most popular release to date, 2.5.

The second edition was released to rave reviews and ended up outselling the first edition. Python itself had won numerous accolades since that time as well, including the following:

- Tiobe (www.tiobe.com)
  - Language of the Year (2007, 2010)
These awards and honors have helped propel Python even further. Now it’s on its next generation with Python 3. Likewise, Core Python Programming is moving towards its “third generation,” too, as I’m exceedingly pleased that Prentice Hall has asked me to develop this third edition. Because version 3.x is backward-incompatible with Python 1 and 2, it will take some time before it is universally adopted and integrated into industry. We are happy to guide you through this transition. The code in this edition will be presented in both Python 2 and 3 (as appropriate—not everything has been ported yet). We’ll also discuss various tools and practices when porting.

The changes brought about in version 3.x continue the trend of iterating and improving the language, taking a larger step toward removing some of its last major flaws, and representing a bigger jump in the continuing evolution of the language. Similarly, the structure of the book is also making a rather significant transition. Due to its size and scope, Core Python Programming as it has existed wouldn’t be able to handle all the new material introduced in this third edition.

Therefore, Prentice Hall and I have decided the best way of moving forward is to take that logical division represented by Parts I and II of the previous editions, representing the core language and advanced applications topics, respectively, and divide the book into two volumes at this juncture. You are holding in your hands (perhaps in eBook form) the second half of the third edition of Core Python Programming. The good news is that the first half is not required in order to make use of the rich amount of content in this volume. We only recommend that you have intermediate Python experience. If you’ve learned Python recently and are fairly comfortable with using it, or have existing Python skills and want to take it to the next level, then you’ve come to the right place!

As existing Core Python Programming readers already know, my primary focus is teaching you the core of the Python language in a comprehensive manner, much more than just its syntax (which you don’t really need a book to learn, right?). Knowing more about how Python works under the hood—including the relationship between data objects and memory management—will make you a much more effective Python programmer.
right out of the gate. This is what Part I, and now *Core Python Language Fundamentals*, is all about.

As with all editions of this book, I will continue to update the book’s Web site and my blog with updates, downloads, and other related articles to keep this publication as contemporary as possible, regardless to which new release of Python you have migrated.

For existing readers, the new topics we have added to this edition include:

- Web-based e-mail examples (Chapter 3)
- Using Tile/Ttk (Chapter 5)
- Using MongoDB (Chapter 6)
- More significant Outlook and PowerPoint examples (Chapter 7)
- Web server gateway interface (WSGI) (Chapter 10)
- Using Twitter (Chapter 13)
- Using Google+ (Chapter 15)

In addition, we are proud to introduce three brand new chapters to the book: Chapter 11, “Web Frameworks: Django,” Chapter 12, “Cloud Computing: Google App Engine,” and Chapter 14, “Text Processing.” These represent new or ongoing areas of application development for which Python is used quite often. All existing chapters have been refreshed and updated to the latest versions of Python, possibly including new material. Take a look at the chapter guide that follows for more details on what to expect from every part of this volume.

**Chapter Guide**

This book is divided into three parts. The first part, which takes up about two-thirds of the text, gives you treatment of the “core” members of any application development toolset (with Python being the focus, of course). The second part concentrates on a variety of topics, all tied to Web programming. The book concludes with the supplemental section which provides experimental chapters that are under development and hopefully will grow into independent chapters in future editions.

All three parts provide a set of various advanced topics to show what you can build by using Python. We are certainly glad that we were at least able to provide you with a good introduction to many of the key areas of Python development including some of the topics mentioned previously.

Following is a more in-depth, chapter-by-chapter guide.
Part I: General Application Topics

Chapter 1—Regular Expressions
Regular expressions are a powerful tool that you can use for pattern matching, extracting, and search-and-replace functionality.

Chapter 2—Network Programming
So many applications today need to be network oriented. In this chapter, you learn to create clients and servers using TCP/IP and UDP/IP as well as get an introduction to SocketServer and Twisted.

Chapter 3—Internet Client Programming
Most Internet protocols in use today were developed using sockets. In Chapter 3, we explore some of those higher-level libraries that are used to build clients of these Internet protocols. In particular, we focus on file transfer (FTP), the Usenet news protocol (NNTP), and a variety of e-mail protocols (SMTP, POP3, IMAP4).

Chapter 4—Multithreaded Programming
Multithreaded programming is one way to improve the execution performance of many types of applications by introducing concurrency. This chapter ends the drought of written documentation on how to implement threads in Python by explaining the concepts and showing you how to correctly build a Python multithreaded application and what the best use cases are.

Chapter 5—GUI Programming
Based on the Tk graphical toolkit, Tkinter (renamed to tkinter in Python 3) is Python’s default GUI development library. We introduce Tkinter to you by showing you how to build simple GUI applications. One of the best ways to learn is to copy, and by building on top of some of these applications, you will be on your way in no time. We conclude the chapter by taking a brief look at other graphical libraries, such as Tix, Pmw, wxPython, PyGTK, and Ttk/Tile.
Chapter 6—Database Programming

Python helps simplify database programming, as well. We first review basic concepts and then introduce you to the Python database application programmer’s interface (DB-API). We then show you how you can connect to a relational database and perform queries and operations by using Python. If you prefer a hands-off approach that uses the Structured Query Language (SQL) and want to just work with objects without having to worry about the underlying database layer, we have object-relational managers (ORMs) just for that purpose. Finally, we introduce you to the world of non-relational databases, experimenting with MongoDB as our NoSQL example.

Chapter 7—Programming Microsoft Office

Like it or not, we live in a world where we will likely have to interact with Microsoft Windows-based PCs. It might be intermittent or something we have to deal with on a daily basis, but regardless of how much exposure we face, the power of Python can be used to make our lives easier. In this chapter, we explore COM Client programming by using Python to control and communicate with Office applications, such as Word, Excel, PowerPoint, and Outlook. Although experimental in the previous edition, we’re glad we were able to add enough material to turn this into a standalone chapter.

Chapter 8—Extending Python

We mentioned earlier how powerful it is to be able to reuse code and extend the language. In pure Python, these extensions are modules and packages, but you can also develop lower-level code in C/C++, C#, or Java. Those extensions then can interface with Python in a seamless fashion. Writing your extensions in a lower-level programming language gives you added performance and some security (because the source code does not have to be revealed). This chapter walks you step-by-step through the extension building process using C.
Part II: Web Development

Chapter 9—Web Clients and Servers
Extending our discussion of client-server architecture in Chapter 2, we apply this concept to the Web. In this chapter, we not only look at clients, but also explore a variety of Web client tools, parsing Web content, and finally, we introduce you to customizing your own Web servers in Python.

Chapter 10—Web Programming: CGI and WSGI
The main job of Web servers is to take client requests and return results. But how do servers get that data? Because they’re really only good at returning results, they generally do not have the capabilities or logic necessary to do so; the heavy lifting is done elsewhere. CGI gives servers the ability to spawn another program to do this processing and has historically been the solution, but it doesn’t scale and is thus not really used in practice; however, its concepts still apply, regardless of what framework(s) you use, so we’ll spend most of the chapter learning CGI. You will also learn how WSGI helps application developers by providing them a common programming interface. In addition, you’ll see how WSGI helps framework developers who have to connect to Web servers on one side and application code on the other so that application developers can write code without having to worry about the execution platform.

Chapter 11—Web Frameworks: Django
Python features a host of Web frameworks with Django being one of the most popular. In this chapter, you get an introduction to this framework and learn how to write simple Web applications. With this knowledge, you can then explore other Web frameworks as you wish.

Chapter 12—Cloud Computing: Google App Engine
Cloud computing is taking the industry by storm. While the world is most familiar with infrastructure services like Amazon’s AWS and online applications such as Gmail and Yahoo! Mail, platforms present a powerful alternative that take advantage of infrastructure without user involvement but give more flexibility than cloud software because you control the application and its code. In this chapter, you get a comprehensive introduction to the first platform service using Python, Google App Engine. With the knowledge gained here, you can then explore similar services in the same space.
Chapter 13—Web Services

In this chapter, we explore higher-level services on the Web (using HTTP). We look at an older service (Yahoo! Finance) and a newer one (Twitter). You learn how to interact with both of these services by using Python as well as knowledge you’ve gained from earlier chapters.

Part III: Supplemental/Experimental

Chapter 14—Text Processing

Our first supplemental chapter introduces you to text processing using Python. We first explore CSV, then JSON, and finally XML. In the last part of this chapter, we take our client/server knowledge from earlier in the book and combine it with XML to look at how you can create online remote procedure calls (RPC) services by using XML-RPC.

Chapter 15—Miscellaneous

This chapter consists of bonus material that we will likely develop into full, individual chapters in a future edition. Topics covered here include Java/Pytho and Google+.

Conventions

All program output and source code are in monospaced font. Python keywords appear in Bold-monospaced font. Lines of output with three leading greater than signs (>>>) represent the Python interpreter prompt. A leading asterisk (*) in front of a chapter, section, or exercise, indicates that this is advanced and/or optional material.

- Represents Core Notes
- Represents Core Module
- Represents Core Tips
- New features to Python are highlighted with this icon, with the number representing version(s) of Python in which the features first appeared.
Book Resources

We welcome any and all feedback—the good, the bad, and the ugly. If you have any comments, suggestions, kudos, complaints, bugs, questions, or anything at all, feel free to contact me at corepython@yahoo.com.

You will find errata, source code, updates, upcoming talks, Python training, downloads, and other information at the book’s Web site located at: http://corepython.com. You can also participate in the community discussion around the “Core Python” books at their Google+ page, which is located at: http://plus.ly/corepython.
Acknowledgments for the Third Edition

Reviewers and Contributors

Gloria Willadsen (lead reviewer)
Darlene Wong
Bryce Verdier
Eric Walstad
Paul Bissex (coauthor of Python Web Development with Django)
Johan “proppy” Euphrosine
Anthony Vallone

Inspiration

My wife Faye, who has continued to amaze me by being able to run the household, take care of the kids and their schedule, feed us all, handle the finances, and be able to do this while I’m off on the road driving cloud adoption or under foot at home, writing books.
Acknowledgments

Editorial
Mark Taub (Editor-in-Chief)
Debra Williams Cauley (Acquisitions Editor)
John Fuller (Managing Editor)
Elizabeth Ryan (Project Editor)
Bob Russell, Octal Publishing, Inc. (Copy Editor)
Dianne Russell, Octal Publishing, Inc. (Production and Management Services)

Acknowledgments for the Second Edition

Reviewers and Contributors
Shannon -jj Behrens (lead reviewer)
Michael Santos (lead reviewer)
Rick Kwan
Lindell Aldermann (coauthor of the Unicode section in Chapter 6)
Wai-Yip Tung (coauthor of the Unicode example in Chapter 20)
Eric Foster-Johnson (coauthor of Beginning Python)
Alex Martelli (editor of Python Cookbook and author of Python in a Nutshell)
Larry Rosenstein
Jim Orosz
Krishna Srinivasan
Chuck Kung

Inspiration
My wonderful children and pet hamster.
Acknowledgments for the First Edition

Reviewers and Contributors

Guido van Rossum (creator of the Python language)
Dowson Tong
James C. Ahlstrom (coauthor of Internet Programming with Python)
S. Candelaria de Ram
Cay S. Horstmann (coauthor of Core Java and Core JavaServer Faces)
Michael Santos
Greg Ward (creator of distutils package and its documentation)
Vincent C. Rubino
Martijn Faassen
Emile van Sebille
Raymond Tsai
Albert L. Anders (coauthor of MT Programming chapter)
Fredrik Lundh (author of Python Standard Library)
Cameron Laird
Fred L. Drake, Jr. (coauthor of Python & XML and editor of the official Python documentation)
Jeremy Hylton
Steve Yoshimoto
Aahz Maruch (author of Python for Dummies)
Jeffrey E. F. Friedl (author of Mastering Regular Expressions)
Pieter Claerhout
Catriona (Kate) Johnston
David Ascher (coauthor of Learning Python and editor of Python Cookbook)
Reg Charney
Christian Tismer (creator of Stackless Python)
Jason Stillwell
and my students at UC Santa Cruz Extension

Inspiration

I would like to extend my great appreciation to James P. Prior, my high school programming teacher.

To Louise Moser and P. Michael Melliar-Smith (my graduate thesis advisors at The University of California, Santa Barbara), you have my deepest gratitude.)
Thanks to Alan Parsons, Eric Woolfson, Andrew Powell, Ian Bairnson, Stuart Elliott, David Paton, all other Project participants, and fellow Projectologists and Roadkillers (for all the music, support, and good times).

I would like to thank my family, friends, and the Lord above, who have kept me safe and sane during this crazy period of late nights and abandonment, on the road and off. I want to also give big thanks to all those who believed in me for the past two decades (you know who you are!)—I couldn’t have done it without you.

Finally, I would like to thank you, my readers, and the Python community at large. I am excited at the prospect of teaching you Python and hope that you enjoy your travels with me on this, our third journey.

Wesley J. Chun  
Silicon Valley, CA  
(It’s not so much a place as it is a state of sanity.)  
October 2001; updated July 2006,  
March 2009, March 2012
ABOUT THE AUTHOR

Wesley Chun was initiated into the world of computing during high school, using BASIC and 6502 assembly on Commodore systems. This was followed by Pascal on the Apple IIe, and then ForTran on punch cards. It was the last of these that made him a careful/cautious developer, because sending the deck out to the school district’s mainframe and getting the results was a one-week round-trip process. Wesley also converted the journalism class from typewriters to Osborne 1 CP/M computers. He got his first paying job as a student-instructor teaching BASIC programming to fourth, fifth, and sixth graders and their parents.

After high school, Wesley went to University of California at Berkeley as a California Alumni Scholar. He graduated with an AB in applied math (computer science) and a minor in music (classical piano). While at Cal, he coded in Pascal, Logo, and C. He also took a tutoring course that featured videotape training and psychological counseling. One of his summer internships involved coding in a 4GL and writing a “Getting Started” user manual. He then continued his studies several years later at University of California, Santa Barbara, receiving an MS in computer science (distributed systems). While there, he also taught C programming. A paper based on his master’s thesis was nominated for Best Paper at the 29th HICSS conference, and a later version appeared in the University of Singapore’s Journal of High Performance Computing.
Wesley has been in the software industry since graduating and has continued to teach and write, publishing several books and delivering hundreds of conference talks and tutorials, plus Python courses, both to the public as well as private corporate training. Wesley’s Python experience began with version 1.4 at a startup where he designed the Yahoo! Mail spellchecker and address book. He then became the lead engineer for Yahoo! People Search. After leaving Yahoo!, he wrote the first edition of this book and then traveled around the world. Since returning, he has used Python in a variety of ways, from local product search, anti-spam and antivirus e-mail appliances, and Facebook games/applications to something completely different: software for doctors to perform spinal fracture analysis.

In his spare time, Wesley enjoys piano, bowling, basketball, bicycling, ultimate frisbee, poker, traveling, and spending time with his family. He volunteers for Python users groups, the Tutor mailing list, and PyCon. He also maintains the Alan Parsons Project Monster Discography. If you think you’re a fan but don’t have “Freudiana,” you had better find it! At the time of this writing, Wesley was a Developer Advocate at Google, representing its cloud products. He is based in Silicon Valley, and you can follow him at @wescpy or plus.ly/wescpy.
> With Python you can start a thread, but you can’t stop it.
> Sorry. You’ll have to wait until it reaches the end of execution.

So, just the same as [comp.lang.python], then?
—Cliff Wells, Steve Holden
(and Timothy Delaney), February 2002

In this chapter...

• Introduction/Motivation
• Threads and Processes
• Threads and Python
• The thread Module
• The threading Module
• Comparing Single vs. Multithreaded Execution
• Multithreading in Practice
• Producer-Consumer Problem and the Queue/queue Module
• Alternative Considerations to Threads
• Related Modules
In this section, we will explore the different ways by which you can achieve more parallelism in your code. We will begin by differentiating between processes and threads in the first few sections of this chapter. We will then introduce the notion of multithreaded programming and present some multithreaded programming features found in Python. (Those of you already familiar with multithreaded programming can skip directly to Section 4.3.5.) The final sections of this chapter present some examples of how to use the threading and Queue modules to accomplish multithreaded programming with Python.

4.1 Introduction/Motivation

Before the advent of multithreaded (MT) programming, the execution of computer programs consisted of a single sequence of steps that were executed in synchronous order by the host’s CPU. This style of execution was the norm whether the task itself required the sequential ordering of steps or if the entire program was actually an aggregation of multiple subtasks. What if these subtasks were independent, having no causal relationship (meaning that results of subtasks do not affect other subtask outcomes)? Is it not logical, then, to want to run these independent tasks all at the same time? Such parallel processing could significantly improve the performance of the overall task. This is what MT programming is all about.

MT programming is ideal for programming tasks that are asynchronous in nature, require multiple concurrent activities, and where the processing of each activity might be nondeterministic, that is, random and unpredictable. Such programming tasks can be organized or partitioned into multiple streams of execution wherein each has a specific task to accomplish. Depending on the application, these subtasks might calculate intermediate results that could be merged into a final piece of output.

While CPU-bound tasks might be fairly straightforward to divide into subtasks and executed sequentially or in a multithreaded manner, the task of managing a single-threaded process with multiple external sources of input is not as trivial. To achieve such a programming task without multithreading, a sequential program must use one or more timers and implement a multiplexing scheme.

A sequential program will need to sample each I/O terminal channel to check for user input; however, it is important that the program does not block when reading the I/O terminal channel, because the arrival of user input is nondeterministic, and blocking would prevent processing of other I/O channels. The sequential program must use non-blocked I/O or blocked I/O with a timer (so that blocking is only temporary).
Because the sequential program is a single thread of execution, it must juggle the multiple tasks that it needs to perform, making sure that it does not spend too much time on any one task, and it must ensure that user response time is appropriately distributed. The use of a sequential program for this type of task often results in a complicated flow of control that is difficult to understand and maintain.

Using an MT program with a shared data structure such as a Queue (a multithreaded queue data structure, discussed later in this chapter), this programming task can be organized with a few threads that have specific functions to perform:

- **UserRequestThread**: Responsible for reading client input, perhaps from an I/O channel. A number of threads would be created by the program, one for each current client, with requests being entered into the queue.

- **RequestProcessor**: A thread that is responsible for retrieving requests from the queue and processing them, providing output for yet a third thread.

- **ReplyThread**: Responsible for taking output destined for the user and either sending it back (if in a networked application) or writing data to the local file system or database.

Organizing this programming task with multiple threads reduces the complexity of the program and enables an implementation that is clean, efficient, and well organized. The logic in each thread is typically less complex because it has a specific job to do. For example, the UserRequestThread simply reads input from a user and places the data into a queue for further processing by another thread, etc. Each thread has its own job to do; you merely have to design each type of thread to do one thing and do it well. Use of threads for specific tasks is not unlike Henry Ford’s assembly line model for manufacturing automobiles.

### 4.2 Threads and Processes

#### 4.2.1 What Are Processes?

Computer *programs* are merely executables, binary (or otherwise), which reside on disk. They do not take on a life of their own until loaded into memory and invoked by the operating system. A *process* (sometimes called
a heavyweight process) is a program in execution. Each process has its own address space, memory, a data stack, and other auxiliary data to keep track of execution. The operating system manages the execution of all processes on the system, dividing the time fairly between all processes. Processes can also fork or spawn new processes to perform other tasks, but each new process has its own memory, data stack, etc., and cannot generally share information unless interprocess communication (IPC) is employed.

### 4.2.2 What Are Threads?

Threads (sometimes called lightweight processes) are similar to processes except that they all execute within the same process, and thus all share the same context. They can be thought of as “mini-processes” running in parallel within a main process or “main thread.”

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running. It can be preempted (interrupted) and temporarily put on hold (also known as sleeping) while other threads are running—this is called yielding.

Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with one another more easily than if they were separate processes. Threads are generally executed in a concurrent fashion, and it is this parallelism and data sharing that enable the coordination of multiple tasks. Naturally, it is impossible to run truly in a concurrent manner in a single CPU system, so threads are scheduled in such a way that they run for a little bit, then yield to other threads (going to the proverbial back of the line to await more CPU time again). Throughout the execution of the entire process, each thread performs its own, separate tasks, and communicates the results with other threads as necessary.

Of course, such sharing is not without its dangers. If two or more threads access the same piece of data, inconsistent results can arise because of the ordering of data access. This is commonly known as a race condition. Fortunately, most thread libraries come with some sort of synchronization primitives that allow the thread manager to control execution and access.

Another caveat is that threads cannot be given equal and fair execution time. This is because some functions block until they have completed. If not written specifically to take threads into account, this skews the amount of CPU time in favor of such greedy functions.
4.3 Threads and Python

In this section, we discuss how to use threads in Python. This includes the limitations of threads due to the global interpreter lock and a quick demo script.

4.3.1 Global Interpreter Lock

Execution of Python code is controlled by the Python Virtual Machine (a.k.a. the interpreter main loop). Python was designed in such a way that only one thread of control may be executing in this main loop, similar to how multiple processes in a system share a single CPU. Many programs can be in memory, but only one is live on the CPU at any given moment. Likewise, although multiple threads can run within the Python interpreter, only one thread is being executed by the interpreter at any given time.

Access to the Python Virtual Machine is controlled by the global interpreter lock (GIL). This lock is what ensures that exactly one thread is running. The Python Virtual Machine executes in the following manner in an MT environment:

1. Set the GIL
2. Switch in a thread to run
3. Execute either of the following:
   a. For a specified number of bytecode instructions, or
   b. If the thread voluntarily yields control (can be accomplished `time.sleep(0)`)        
4. Put the thread back to sleep (switch out thread)
5. Unlock the GIL
6. Do it all over again (lather, rinse, repeat)

When a call is made to external code—that is, any C/C++ extension built-in function—the GIL will be locked until it has completed (because there are no Python bytecodes to count as the interval). Extension programmers do have the ability to unlock the GIL, however, so as the Python developer, you shouldn’t have to worry about your Python code locking up in those situations.

As an example, for any Python I/O-oriented routines (which invoke built-in operating system C code), the GIL is released before the I/O call is made, allowing other threads to run while the I/O is being performed. Code that doesn’t have much I/O will tend to keep the processor (and GIL)
for the full interval a thread is allowed before it yields. In other words, I/O-bound Python programs stand a much better chance of being able to take advantage of a multithreaded environment than CPU-bound code.

Those of you who are interested in the source code, the interpreter main loop, and the GIL can take a look at the Python/ceval.c file.

4.3.2 Exiting Threads

When a thread completes execution of the function it was created for, it exits. Threads can also quit by calling an exit function such as thread.exit(), or any of the standard ways of exiting a Python process such as sys.exit() or raising the SystemExit exception. You cannot, however, go and “kill” a thread.

We will discuss in detail the two Python modules related to threads in the next section, but of the two, the thread module is the one we do not recommend. There are many reasons for this, but an obvious one is that when the main thread exits, all other threads die without cleanup. The other module, threading, ensures that the whole process stays alive until all “important” child threads have exited. (For a clarification of what important means, read the upcoming Core Tip, “Avoid using the thread module.”)

Main threads should always be good managers, though, and perform the task of knowing what needs to be executed by individual threads, what data or arguments each of the spawned threads requires, when they complete execution, and what results they provide. In so doing, those main threads can collate the individual results into a final, meaningful conclusion.

4.3.3 Accessing Threads from Python

Python supports multithreaded programming, depending on the operating system on which it’s running. It is supported on most Unix-based platforms, such as Linux, Solaris, Mac OS X, *BSD, as well as Windows-based PCs. Python uses POSIX-compliant threads, or pthreads, as they are commonly known.

By default, threads are enabled when building Python from source (since Python 2.0) or the Win32 installed binary. To determine whether threads are available for your interpreter, simply attempt to import the thread module from the interactive interpreter, as shown here (no errors occur when threads are available):

```python
>>> import thread
>>>```
If your Python interpreter was *not* compiled with threads enabled, the module import fails:

```python
>>> import thread
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ImportError: No module named thread
```

In such cases, you might need to recompile your Python interpreter to get access to threads. This usually involves invoking the `configure` script with the `--with-thread` option. Check the README file for your distribution to obtain specific instructions on how to compile Python with threads for your system.

### 4.3.4 Life Without Threads

For our first set of examples, we are going to use the `time.sleep()` function to show how threads work. `time.sleep()` takes a floating point argument and “sleeps” for the given number of seconds, meaning that execution is temporarily halted for the amount of time specified.

Let’s create two time loops: one that sleeps for 4 seconds (`loop0()`) and one that sleeps for 2 seconds (`loop1()`), respectively. (We use the names “loop0” and “loop1” as a hint that we will eventually have a sequence of loops.) If we were to execute `loop0()` and `loop1()` sequentially in a one-process or single-threaded program, as `onethr.py` does in Example 4-1, the total execution time would be at least 6 seconds. There might or might not be a 1-second gap between the starting of `loop0()` and `loop1()` as well as other execution overhead which can cause the overall time to be bumped to 7 seconds.

#### Example 4-1  Loops Executed by a Single Thread (`onethr.py`)

This script executes two loops consecutively in a single-threaded program. One loop must complete before the other can begin. The total elapsed time is the sum of times taken by each loop.

```python
#!/usr/bin/env python

from time import sleep, ctime

def loop0():
    print 'start loop 0 at:', ctime()
    sleep(4)
```

4.3 Threads and Python

We can verify this by executing `onethr.py`, which renders the following output:

```
$ onethr.py
starting at: Sun Aug 13 05:03:34 2006
start loop 0 at: Sun Aug 13 05:03:34 2006
loop 0 done at: Sun Aug 13 05:03:38 2006
start loop 1 at: Sun Aug 13 05:03:38 2006
loop 1 done at: Sun Aug 13 05:03:40 2006
all DONE at: Sun Aug 13 05:03:40 2006
```

Now, assume that rather than sleeping, `loop0()` and `loop1()` were separate functions that performed individual and independent computations, all working to arrive at a common solution. Wouldn’t it be useful to have them run in parallel to cut down on the overall running time? That is the premise behind MT programming that we now introduce.

### 4.3.5 Python Threading Modules

Python provides several modules to support MT programming, including the `thread`, `threading`, and `Queue` modules. Programmers can use the `thread` and `threading` modules to create and manage threads. The `thread` module provides basic thread and locking support; `threading` provides higher-level, fully-featured thread management. With the `Queue` module, users can create a queue data structure that can be shared across multiple threads. We will take a look at these modules individually and present examples and intermediate-sized applications.
CORE TIP: Avoid using the thread module

We recommend using the high-level threading module instead of the thread module for many reasons. threading is more contemporary, has better thread support, and some attributes in the thread module can conflict with those in the threading module. Another reason is that the lower-level thread module has few synchronization primitives (actually only one) while threading has many.

However, in the interest of learning Python and threading in general, we do present some code that uses the thread module. We present these for learning purposes only; hopefully they give you a much better insight as to why you would want to avoid using thread. We will also show you how to use more appropriate tools such as those available in the threading and Queue modules.

Another reason to avoid using thread is because there is no control of when your process exits. When the main thread finishes, any other threads will also die, without warning or proper cleanup. As mentioned earlier, at least threading allows the important child threads to finish first before exiting.

Use of the thread module is recommended only for experts desiring lower-level thread access. To emphasize this, it is renamed to _thread in Python 3.

Any multithreaded application you create should utilize threading and perhaps other higher-level modules.

4.4 The thread Module

Let’s take a look at what the thread module has to offer. In addition to being able to spawn threads, the thread module also provides a basic synchronization data structure called a lock object (a.k.a. primitive lock, simple lock, mutual exclusion lock, mutex, and binary semaphore). As we mentioned earlier, such synchronization primitives go hand in hand with thread management.

Table 4-1 lists the more commonly used thread functions and LockType lock object methods.
The key function of the thread module is `start_new_thread()`. It takes a function (object) plus arguments and optionally, keyword arguments. A new thread is spawned specifically to invoke the function.

Let's take our `onethr.py` example and integrate threading into it. By slightly changing the call to the `loop*()` functions, we now present `mtsleepA.py` in Example 4-2:

```python
#!/usr/bin/env python
import thread
from time import sleep, ctime

def loop0():
    print 'start loop 0 at:', ctime()

# Example 4-2 Using the thread Module (mtsleepA.py)
The same loops from `onethr.py` are executed, but this time using the simple multithreaded mechanism provided by the thread module. The two loops are executed concurrently (with the shorter one finishing first, obviously), and the total elapsed time is only as long as the slowest thread rather than the total time for each separately.

(Continued)```
start_new_thread() requires the first two arguments, so that is the reason for passing in an empty tuple even if the executing function requires no arguments.

Upon execution of this program, our output changes drastically. Rather than taking a full 6 or 7 seconds, our script now runs in 4 seconds, the length of time of our longest loop, plus any overhead.

```
starting at: Sun Aug 13 05:04:50 2006
start loop 0 at: Sun Aug 13 05:04:50 2006
start loop 1 at: Sun Aug 13 05:04:50 2006
loop 1 done at: Sun Aug 13 05:04:52 2006
loop 0 done at: Sun Aug 13 05:04:54 2006
all DONE at: Sun Aug 13 05:04:56 2006
```

The pieces of code that sleep for 4 and 2 seconds now occur concurrently, contributing to the lower overall runtime. You can even see how loop 1 finishes before loop 0.

The only other major change to our application is the addition of the sleep(6) call. Why is this necessary? The reason is that if we did not stop the main thread from continuing, it would proceed to the next statement, displaying “all done” and exit, killing both threads running loop0() and loop1().

We did not have any code that directed the main thread to wait for the child threads to complete before continuing. This is what we mean by threads requiring some sort of synchronization. In our case, we used another sleep() call as our synchronization mechanism. We used a value

```
start_new_thread() requires the first two arguments, so that is the reason for passing in an empty tuple even if the executing function requires no arguments.

Upon execution of this program, our output changes drastically. Rather than taking a full 6 or 7 seconds, our script now runs in 4 seconds, the length of time of our longest loop, plus any overhead.

```

start_new_thread() requires the first two arguments, so that is the reason for passing in an empty tuple even if the executing function requires no arguments.

Upon execution of this program, our output changes drastically. Rather than taking a full 6 or 7 seconds, our script now runs in 4 seconds, the length of time of our longest loop, plus any overhead.

```
start_new_thread() requires the first two arguments, so that is the reason for passing in an empty tuple even if the executing function requires no arguments.

Upon execution of this program, our output changes drastically. Rather than taking a full 6 or 7 seconds, our script now runs in 4 seconds, the length of time of our longest loop, plus any overhead.

```
start_new_thread() requires the first two arguments, so that is the reason for passing in an empty tuple even if the executing function requires no arguments.

Upon execution of this program, our output changes drastically. Rather than taking a full 6 or 7 seconds, our script now runs in 4 seconds, the length of time of our longest loop, plus any overhead.

```
start_new_thread() requires the first two arguments, so that is the reason for passing in an empty tuple even if the executing function requires no arguments.

Upon execution of this program, our output changes drastically. Rather than taking a full 6 or 7 seconds, our script now runs in 4 seconds, the length of time of our longest loop, plus any overhead.

```
start_new_thread() requires the first two arguments, so that is the reason for passing in an empty tuple even if the executing function requires no arguments.

Upon execution of this program, our output changes drastically. Rather than taking a full 6 or 7 seconds, our script now runs in 4 seconds, the length of time of our longest loop, plus any overhead.
of 6 seconds because we know that both threads (which take 4 and 2 seconds) should have completed by the time the main thread has counted to 6.

You are probably thinking that there should be a better way of managing threads than creating that extra delay of 6 seconds in the main thread. Because of this delay, the overall runtime is no better than in our single-threaded version. Using \texttt{sleep()} for thread synchronization as we did is not reliable. What if our loops had independent and varying execution times? We could be exiting the main thread too early or too late. This is where locks come in.

Making yet another update to our code to include locks as well as getting rid of separate loop functions, we get \texttt{mtsleepB.py}, which is presented in Example 4-3. Running it, we see that the output is similar to \texttt{mtsleepA.py}. The only difference is that we did not have to wait the extra time for \texttt{mtsleepA.py} to conclude. By using locks, we were able to exit as soon as both threads had completed execution. This renders the following output:

```
$ mtsleepB.py
starting at: Sun Aug 13 16:34:41 2006
start loop 0 at: Sun Aug 13 16:34:41 2006
start loop 1 at: Sun Aug 13 16:34:41 2006
loop 1 done at: Sun Aug 13 16:34:43 2006
loop 0 done at: Sun Aug 13 16:34:45 2006
all DONE at: Sun Aug 13 16:34:45 2006
```

**Example 4-3** Using thread and Locks (\texttt{mtsleepB.py})

Rather than using a call to \texttt{sleep()} to hold up the main thread as in \texttt{mtsleepA.py}, the use of locks makes more sense.
So how did we accomplish our task with locks? Let’s take a look at the source code.

**Line-by-Line Explanation**

**Lines 1–6**
After the Unix startup line, we import the `thread` module and a few familiar attributes of the `time` module. Rather than hardcoding separate functions to count to 4 and 2 seconds, we use a single `loop()` function and place these constants in a list, `loops`.

**Lines 8–12**
The `loop()` function acts as a proxy for the deleted `loop*()` functions from our earlier examples. We had to make some cosmetic changes to `loop()` so that it can now perform its duties using locks. The obvious changes are that we need to be told which loop number we are as well as the sleep duration. The last piece of new information is the lock itself. Each thread will be allocated an acquired lock. When the `sleep()` time has concluded, we release the corresponding lock, indicating to the main thread that this thread has completed.
The bulk of the work is done here in `main()`, using three separate `for` loops. We first create a list of locks, which we obtain by using the `thread.allocate_lock()` function and acquire (each lock) with the `acquire()` method. Acquiring a lock has the effect of “locking the lock.” Once it is locked, we add the lock to the lock list, `locks`. The next loop actually spawns the threads, invoking the `loop()` function per thread, and for each thread, provides it with the loop number, the sleep duration, and the acquired lock for that thread. So why didn’t we start the threads in the lock acquisition loop? There are two reasons. First, we wanted to synchronize the threads, so that all the horses started out the gate around the same time, and second, locks take a little bit of time to be acquired. If your thread executes too fast, it is possible that it completes before the lock has a chance to be acquired.

It is up to each thread to unlock its lock object when it has completed execution. The final loop just sits and spins (pausing the main thread) until both locks have been released before continuing execution. Because we are checking each lock sequentially, we might be at the mercy of all the slower loops if they are more toward the beginning of the set of loops. In such cases, the majority of the wait time may be for the first loop(s). When that lock is released, remaining locks may have already been unlocked (meaning that corresponding threads have completed execution). The result is that the main thread will fly through those lock checks without pause. Finally, you should be well aware that the final pair of lines will execute `main()` only if we are invoking this script directly.

As hinted in the earlier Core Note, we presented the `thread` module only to introduce the reader to threaded programming. Your MT application should use higher-level modules such as the `threading` module, which we discuss in the next section.

### 4.5 The `threading` Module

We will now introduce the higher-level `threading` module, which gives you not only a `Thread` class but also a wide variety of synchronization mechanisms to use to your heart’s content. Table 4-2 presents a list of all the objects available in the `threading` module.
In this section, we will examine how to use the `Thread` class to implement threading. Because we have already covered the basics of locking, we will not cover the locking primitives here. The `Thread()` class also contains a form of synchronization, so explicit use of locking primitives is not necessary.

**Table 4-2 threading Module Objects**

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Object that represents a single thread of execution</td>
</tr>
<tr>
<td>Lock</td>
<td>Primitive lock object (same lock as in <code>thread</code> module)</td>
</tr>
<tr>
<td>RLock</td>
<td>Re-entrant lock object provides ability for a single thread to (re)acquire an already-held lock (recursive locking)</td>
</tr>
<tr>
<td>Condition</td>
<td>Condition variable object causes one thread to wait until a certain “condition” has been satisfied by another thread, such as changing of state or of some data value</td>
</tr>
<tr>
<td>Event</td>
<td>General version of condition variables, whereby any number of threads are waiting for some event to occur and all will awaken when the event happens</td>
</tr>
<tr>
<td>Semaphore</td>
<td>Provides a “counter” of finite resources shared between threads; block when none are available</td>
</tr>
<tr>
<td>BoundedSemaphore</td>
<td>Similar to a Semaphore but ensures that it never exceeds its initial value</td>
</tr>
<tr>
<td>Timer</td>
<td>Similar to <code>Thread</code>, except that it waits for an allotted period of time before running</td>
</tr>
<tr>
<td>Barrier(^a)</td>
<td>Creates a “barrier,” at which a specified number of threads must all arrive before they’re all allowed to continue</td>
</tr>
</tbody>
</table>

\(^a\) New in Python 3.2.
CORE TIP:Daemon threads

Another reason to avoid using the thread module is that it does not support the concept of daemon (or daemonic) threads. When the main thread exits, all child threads will be killed, regardless of whether they are doing work. The concept of daemon threads comes into play here if you do not desire this behavior.

Support for daemon threads is available in the threading module, and here is how they work: a daemon is typically a server that waits for client requests to service. If there is no client work to be done, the daemon sits idle. If you set the daemon flag for a thread, you are basically saying that it is non-critical, and it is okay for the process to exit without waiting for it to finish. As you have seen in Chapter 2, “Network Programming,” server threads run in an infinite loop and do not exit in normal situations.

If your main thread is ready to exit and you do not care to wait for the child threads to finish, then set their daemon flags. A value of true denotes a thread is not important or more likely, not doing anything but waiting for a client.

To set a thread as daemonic, make this assignment: thread.daemon = True before you start the thread. (The old-style way of calling thread.setDaemon(True) is deprecated.) The same is true for checking on a thread’s daemonic status; just check that value (versus calling thread.isDaemon()). A new child thread inherits its daemonic flag from its parent. The entire Python program (read as: the main thread) will stay alive until all non-daemonic threads have exited—in other words, when no active non-daemonic threads are left.

4.5.1 The Thread Class

The Thread class of the threading module is your primary executive object. It has a variety of functions not available to the thread module. Table 4-3 presents a list of attributes and methods.
### Table 4-3 Thread Object Attributes and Methods

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread object data attributes</strong></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>The name of a thread.</td>
</tr>
<tr>
<td>ident</td>
<td>The identifier of a thread.</td>
</tr>
<tr>
<td>daemon</td>
<td>Boolean flag indicating whether a thread is daemonic.</td>
</tr>
<tr>
<td><strong>Thread object methods</strong></td>
<td></td>
</tr>
<tr>
<td><strong>init</strong></td>
<td>Instantiate a Thread object, taking target callable and any args or kwargs. A name or group can also be passed but the latter is unimplemented. A verbose flag is also accepted. Any daemon value sets the thread.daemon attribute/flag.</td>
</tr>
<tr>
<td>start()</td>
<td>Begin thread execution.</td>
</tr>
<tr>
<td>run()</td>
<td>Method defining thread functionality (usually overridden by application writer in a subclass).</td>
</tr>
<tr>
<td>join(timeout=None)</td>
<td>Suspend until the started thread terminates; blocks unless timeout (in seconds) is given.</td>
</tr>
<tr>
<td>getName()</td>
<td>Return name of thread.</td>
</tr>
<tr>
<td>setName(name)</td>
<td>Set name of thread.</td>
</tr>
<tr>
<td>isVisible/is_alive()</td>
<td>Boolean flag indicating whether thread is still running.</td>
</tr>
<tr>
<td>isDaemon()</td>
<td>Return True if thread daemonic, False otherwise.</td>
</tr>
<tr>
<td>setDaemon(daemonic)</td>
<td>Set the daemon flag to the given Boolean daemonic value (must be called before thread start()).</td>
</tr>
</tbody>
</table>

a. Deprecated by setting (or getting) thread.name attribute or passed in during instantiation.
b. CamelCase names deprecated and replaced starting in Python 2.6.
c. is/isDaemon() deprecated by setting thread.daemon attribute; thread.daemon can also be set during instantiation via the optional daemon value—new in Python 3.3.
There are a variety of ways by which you can create threads using the Thread class. We cover three of them here, all quite similar. Pick the one you feel most comfortable with, not to mention the most appropriate for your application and future scalability (we like the final choice the best):

- Create Thread instance, passing in function
- Create Thread instance, passing in callable class instance
- Subclass Thread and create subclass instance

You’ll discover that you will pick either the first or third option. The latter is chosen when a more object-oriented interface is desired and the former, otherwise. The second, honestly, is a bit more awkward and slightly harder to read, as you’ll discover.

**Create Thread Instance, Passing in Function**

In our first example, we will just instantiate Thread, passing in our function (and its arguments) in a manner similar to our previous examples. This function is what will be executed when we direct the thread to begin execution. Taking our mtsleepB.py script from Example 4-3 and tweaking it by adding the use of Thread objects, we have mtsleepC.py, as shown in Example 4-4.

**Example 4-4 Using the threading Module (mtsleepC.py)**

The Thread class from the threading module has a join() method that lets the main thread wait for thread completion.

```python
#!/usr/bin/env python
import threading
from time import sleep, ctime

loops = [4, 2]

def loop(nloop, nsec):
    print 'start loop', nloop, 'at:', ctime()
    sleep(nsec)
    print 'loop', nloop, 'done at:', ctime()

def main():
    print 'starting at:', ctime()
    threads = []

(Continued)```
When we run the script in Example 4-4, we see output similar to that of its predecessors:

```
$ mtsleepC.py
starting at: Sun Aug 13 18:16:38 2006
start loop 0 at: Sun Aug 13 18:16:38 2006
start loop 1 at: Sun Aug 13 18:16:38 2006
loop 1 done at: Sun Aug 13 18:16:40 2006
loop 0 done at: Sun Aug 13 18:16:42 2006
all DONE at: Sun Aug 13 18:16:42 2006
```

So what *did* change? Gone are the locks that we had to implement when using the thread module. Instead, we create a set of Thread objects. When each Thread is instantiated, we dutifully pass in the function (target) and arguments (args) and receive a Thread instance in return. The biggest difference between instantiating Thread (calling Thread()) and invoking thread.start_new_thread() is that the new thread does not begin execution right away. This is a useful synchronization feature, especially when you don’t want the threads to start immediately.

Once all the threads have been allocated, we let them go off to the races by invoking each thread’s start() method, but not a moment before that. And rather than having to manage a set of locks (allocating, acquiring, releasing, checking lock state, etc.), we simply call the join() method for each thread. join() will wait until a thread terminates, or, if provided, a timeout occurs. Use of join() appears much cleaner than an infinite loop that waits for locks to be released (which is why these locks are sometimes known as *spin locks*).
One other important aspect of `join()` is that it does not need to be called at all. Once threads are started, they will execute until their given function completes, at which point, they will exit. If your main thread has things to do other than wait for threads to complete (such as other processing or waiting for new client requests), it should do so. `join()` is useful only when you want to wait for thread completion.

### Create Thread Instance, Passing in Callable Class Instance

A similar offshoot to passing in a function when creating a thread is having a callable class and passing in an instance for execution—this is the more object-oriented approach to MT programming. Such a callable class embodies an execution environment that is much more flexible than a function or choosing from a set of functions. You now have the power of a class object behind you, as opposed to a single function or a list/tuple of functions.

Adding our new class `ThreadFunc` to the code and making other slight modifications to `mtsleepC.py`, we get `mtsleepD.py`, shown in Example 4-5.

```python
#!/usr/bin/env python
import threading
from time import sleep, ctime

loops = [4,2]

class ThreadFunc(object):
    def __init__(self, func, args, name=' '):
        self.name = name
        self.func = func
        self.args = args
    def __call__(self):
        self.func(*self.args)

(Continued)
```
When we run `mtsleepD.py`, we get the expected output:

```
$ mtsleepD.py
start loop 0 at: Sun Aug 13 18:49:17 2006
start loop 1 at: Sun Aug 13 18:49:17 2006
loop 1 done at: Sun Aug 13 18:49:19 2006
loop 0 done at: Sun Aug 13 18:49:21 2006
all DONE at: Sun Aug 13 18:49:21 2006
```

So what are the changes this time? The addition of the `ThreadFunc` class and a minor change to instantiate the `Thread` object, which also instantiates `ThreadFunc`, our callable class. In effect, we have a double instantiation going on here. Let’s take a closer look at our `ThreadFunc` class.

We want to make this class general enough to use with functions other than our `loop()` function, so we added some new infrastructure, such as having this class hold the arguments for the function, the function itself, and also a function name string. The constructor `__init__()` just sets all the values.

When the `Thread` code calls our `ThreadFunc` object because a new thread is created, it will invoke the `__call__()` special method. Because we already have our set of arguments, we do not need to pass it to the `Thread()` constructor and can call the function directly.

---

**Example 4-5 Using Callable classes (mtsleepD.py) (Continued)**

```python
18 def loop(nloop, nsec):
19     print 'start loop', nloop, 'at:', ctime()
20     sleep(nsec)
21     print 'loop', nloop, 'done at:', ctime()
22
def main():
23     print 'starting at:', ctime()
24     threads = []
25     nloops = range(len(loops))
26
27     for i in nloops: # create all threads
28         t = threading.Thread(
29             target=ThreadFunc(loop, (i, loops[i]),
30                 loop.__name__),
31             threads.append(t)
32
33     for i in nloops: # start all threads
34         threads[i].start()
35
36     for i in nloops: # wait for completion
37         threads[i].join()
38
39     print 'all DONE at:', ctime()
40
41     if __name__ == '__main__':
42         main()
```
4.5 The threading Module

Subclass Thread and Create Subclass Instance

The final introductory example involves subclassing `Thread()`, which turns out to be extremely similar to creating a callable class as in the previous example. Subclassing is a bit easier to read when you are creating your threads (lines 29–30). We will present the code for `mtsleepE.py` in Example 4-6 as well as the output obtained from its execution, and leave it as an exercise for you to compare `mtsleepE.py` to `mtsleepD.py`.

Example 4-6 Subclassing Thread (`mtsleepE.py`)

Rather than instantiating the `Thread` class, we subclass it. This gives us more flexibility in customizing our threading objects and simplifies the thread creation call.

```python
#!/usr/bin/env python

import threading
from time import sleep, ctime

loops = (4, 2)

class MyThread(threading.Thread):
    def __init__(self, func, args, name=''):  
        threading.Thread.__init__(self)
        self.name = name
        self.func = func
        self.args = args

    def run(self):
        self.func(*self.args)

    def loop(nloop, nsec):
        print('start loop', nloop, 'at:', ctime())
        sleep(nsec)
        print('loop', nloop, 'done at:', ctime())

    def main():
        print('starting at:', ctime())
        threads = []
        nloops = range(len(loops))

        for i in nloops:
            t = MyThread(loop, (i, loops[i]), loop.__name__)
            threads.append(t)
```

(Continued)
Here is the output for mtsleepE.py. Again, it’s just as we expected:

```
$ mtsleepE.py
starting at: Sun Aug 13 19:14:26 2006
start loop 0 at: Sun Aug 13 19:14:26 2006
start loop 1 at: Sun Aug 13 19:14:26 2006
loop 1 done at: Sun Aug 13 19:14:28 2006
loop 0 done at: Sun Aug 13 19:14:30 2006
all DONE at: Sun Aug 13 19:14:30 2006
```

While you compare the source between the mtsleep4 and mtsleep5 modules, we want to point out the most significant changes: 1) our MyThread subclass constructor must first invoke the base class constructor (line 9), and 2) the former special method \_call\_() must be called run() in the subclass.

We now modify our MyThread class with some diagnostic output and store it in a separate module called myThread (look ahead to Example 4-7) and import this class for the upcoming examples. Rather than simply calling our functions, we also save the result to instance attribute self.res, and create a new method to retrieve that value, getResult().

```
Example 4-7   MyThread Subclass of Thread (myThread.py)

To generalize our subclass of Thread from mtsleepE.py, we move the subclass to a separate module and add a getResult() method for callables that produce return values.

1   #!/usr/bin/env python
2
3   import threading
4   from time import ctime
```
4.5 The threading Module

4.5.2 Other Threading Module Functions

In addition to the various synchronization and threading objects, the Threading module also has some supporting functions, as detailed in Table 4-4.

Table 4-4 threading Module Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>activeCount/active_count()&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Number of currently active Thread objects</td>
</tr>
<tr>
<td>currentThread()/current_thread&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Returns the current Thread object</td>
</tr>
<tr>
<td>enumerate()</td>
<td>Returns list of all currently active Threads</td>
</tr>
<tr>
<td>settrace(func)&lt;sup&gt;b&lt;/sup&gt;</td>
<td>Sets a trace function for all threads</td>
</tr>
<tr>
<td>setprofile(func)&lt;sup&gt;b&lt;/sup&gt;</td>
<td>Sets a profile function for all threads</td>
</tr>
<tr>
<td>stack_size(size=0)&lt;sup&gt;c&lt;/sup&gt;</td>
<td>Returns stack size of newly created threads; optional size can be set for subsequently created threads</td>
</tr>
</tbody>
</table>

---

<sup>a</sup> CamelCase names deprecated and replaced starting in Python 2.6.

<sup>b</sup> New in Python 2.3.

<sup>c</sup> An alias to thread.stack_size(); (both) new in Python 2.5.

```python
6    class MyThread(threading.Thread):
7        def __init__(self, func, args, name=''):  
8            threading.Thread.__init__(self)       
9                self.name = name               
10               self.func = func              
11               self.args = args             
12        def getResult(self):
13            return self.res                
14        def run(self):
15            print 'starting', self.name, 'at:', 
16                ctime()          
17            self.res = self.func(*self.args) 
18            print self.name, 'finished at:', 
19                ctime()
20```

### 4.6 Comparing Single vs. Multithreaded Execution

The mtfacfib.py script, presented in Example 4-8 compares execution of the recursive Fibonacci, factorial, and summation functions. This script runs all three functions in a single-threaded manner. It then performs the same task by using threads to illustrate one of the advantages of having a threading environment.

**Example 4-8 Fibonacci, Factorial, Summation (mtfacfib.py)**

In this MT application, we execute three separate recursive functions—first in a single-threaded fashion, followed by the alternative with multiple threads.

```python
#!/usr/bin/env python

from myThread import MyThread
from time import ctime, sleep

def fib(x):
    sleep(0.005)
    if x < 2:
        return 1
    return (fib(x-2) + fib(x-1))

def fac(x):
    sleep(0.1)
    if x < 2:
        return 1
    return (x * fac(x-1))

def sum(x):
    sleep(0.1)
    if x < 2:
        return 1
    return (x + sum(x-1))

funcs = [fib, fac, sum]
n = 12

def main():
    nfuncs = range(len(funcs))

    print '*** SINGLE THREAD'
    for i in nfuncs:
        print 'starting', funcs[i].__name__, 'at:', ctime()
        print funcs[i](n)
        print funcs[i].__name__, 'finished at:', ctime()

    print '
*** MULTIPLE THREADS'
    threads = []
```
4.6 Comparing Single vs. Multithreaded Execution

Running in single-threaded mode simply involves calling the functions one at a time and displaying the corresponding results right after the function call.

When running in multithreaded mode, we do not display the result right away. Because we want to keep our MyThread class as general as possible (being able to execute callables that do and do not produce output), we wait until the end to call the getResult() method to finally show you the return values of each function call.

Because these functions execute so quickly (well, maybe except for the Fibonacci function), you will notice that we had to add calls to sleep() to each function to slow things down so that we can see how threading can improve performance, if indeed the actual work had varying execution times—you certainly wouldn’t pad your work with calls to sleep(). Anyway, here is the output:

```
$ mtfacfib.py
*** SINGLE THREAD
starting fib at: Wed Nov 16 18:52:20 2011
233
fib finished at: Wed Nov 16 18:52:24 2011
starting fac at: Wed Nov 16 18:52:24 2011
479001600
fac finished at: Wed Nov 16 18:52:26 2011
starting sum at: Wed Nov 16 18:52:26 2011
78
sum finished at: Wed Nov 16 18:52:27 2011

*** MULTIPLE THREADS
starting fib at: Wed Nov 16 18:52:27 2011
starting fac at: Wed Nov 16 18:52:27 2011
starting sum at: Wed Nov 16 18:52:27 2011
```
4.7 Multithreading in Practice

So far, none of the simplistic sample snippets we’ve seen so far represent code that you’d write in practice. They don’t really do anything useful beyond demonstrating threads and the different ways that you can create them—the way we’ve started them up and wait for them to finish are all identical, and they all just sleep, too.

We also mentioned earlier in Section 4.3.1 that due to the fact that the Python Virtual Machine is single-threaded (the GIL), greater concurrency in Python is only possible when threading is applied to an I/O-bound application (versus CPU-bound applications, which only do round-robin), so let’s look at an example of this, and for a further exercise, try to port it to Python 3 to give you a sense of what that process entails.

4.7.1 Book Rankings Example

The bookrank.py script shown in Example 4-9 is very straightforward. It goes to the one of my favorite online retailers, Amazon, and asks for the current rankings of books written by yours truly. In our sample code, you’ll see a function, getRanking(), that uses a regular expression to pull out and return the current ranking plus showRanking(), which displays the result to the user.

Note that, according to their Conditions of Use guidelines, “Amazon grants you a limited license to access and make personal use of this site and not to download (other than page caching) or modify it, or any portion of it, except with express written consent of Amazon.” For our application, all we’re doing is looking at the current book rankings for a specific book and then throwing everything away; we’re not even caching the page.

Example 4-9 is our first (but nearly-final) attempt at bookrank.py, which is a non-threaded version.
Example 4-9  Book Rankings “Screenscraper” (bookrank.py)

This script makes calls to download book ranking information via separate threads.

```python
#!/usr/bin/env python
from atexit import register
from re import compile
from threading import Thread
from time import ctime
from urllib2 import urlopen as uopen

REGEX = compile('#([\d,]+) in Books ')
AMZN = 'http://amazon.com/dp/
ISBNs = {'0132269937': 'Core Python Programming',
       '0132356139': 'Python Web Development with Django',
       '0137143419': 'Python Fundamentals',
       }

def getRanking(isbn):
    page = uopen('%s%s' % (AMZN, isbn))
    data = page.read()
    page.close()
    return REGEX.findall(data)[0]

def _showRanking(isbn):
    print '- %r ranked %s' % (ISBNs[isbn], getRanking(isbn))

def _main():
    print 'At', ctime(), 'on Amazon...
    for isbn in ISBNs:
        _showRanking(isbn)

@register
def _atexit():
    print 'all DONE at:', ctime()

if __name__ == '__main__':
    main()
```

Line-by-Line Explanation

Lines 1–7
These are the startup and import lines. We'll use the `atexit.register()` function to tell us when the script is over (you'll see why later). We'll also use the regular expression `re.compile()` function for the pattern that matches a book's ranking on Amazon's product pages. Then, we save the
threading.Thread import for future improvement (coming up a bit later), time.ctime() for the current timestamp string, and urllib2.urlopen() for accessing each link.

**Lines 9–15**

We use three constants in this script: REGEX, the regular expression object (compiled from the regex pattern that matches a book’s ranking); AMZN, the base Amazon product link—all we need to complete each link is a book’s International Standard Book Number (ISBN), which serves as a book’s ID, differentiating one written work from all others. There are two standards: the ISBN-10 ten-character value and its successor, the ISBN-13 thirteen-character ISBN. Currently, Amazon’s systems understand both ISBN types, so we’ll just use ISBN-10 because they’re shorter. These are stored in the ISBNs dictionary along with the corresponding book titles.

**Lines 17–21**

The purpose of getRanking() is to take an ISBN, create the final URL with which to communicate to Amazon’s servers, and then call urllib2.urlopen() on it. We used the string format operator to put together the URL (on line 18) but if you’re using version 2.6 and newer, you can also try the str.format() method, for example, '{0}{1}'.format(AMZN,isbn).

Once you have the full URL, call urllib2.urlopen()—we shortened it to uopen()—and expect the file-like object back once the Web server has been contacted. Then the read() call is issued to download the entire Web page, and “file” is closed. If the regex is as precise as we have planned, there should only be exactly one match, so we grab it from the generated list (any additional would be dropped) and return it back to the caller.

**Lines 23–25**

The _showRanking() function is just a short snippet of code that takes an ISBN, looks up the title of the book it represents, calls getRanking() to get its current ranking on Amazon’s Web site, and then outputs both of these values to the user. The leading single-underscore notation indicates that this is a special function only to be used by code within this module and should not be imported by any other application using this as a library or utility module.
4.7 Multithreading in Practice

Lines 27–30

_main() is also a special function, only executed if this module is run directly from the command-line (and not imported for use by another module). It shows the start and end times (to let users know how long it took to run the entire script) and calls _showRanking() for each ISBN to lookup and display each book’s current ranking on Amazon.

Lines 32–37

These lines present something completely different. What is `atexit.register()`? It’s a function (used in a decorator role here) that registers an exit function with the Python interpreter, meaning it’s requesting a special function be called just before the script quits. (Instead of the decorator, you could have also done `register(_atexit())`.

Why are we using it here? Well, right now, it’s definitely not needed. The print statement could very well go at the end of _main() in lines 27–31, but that’s not a really great place for it. Plus this is functionality that you might really want to use in a real production application at some point. We assume that you know what lines 36–37 are about, so onto the output:

```
$ python bookrank.py
At Wed Mar 30 22:11:19 2011 PDT on Amazon...
- 'Core Python Programming' ranked 87,118
- 'Python Fundamentals' ranked 851,816
- 'Python Web Development with Django' ranked 184,735
```

If you’re wondering, we’ve separated the process of retrieving (getRanking()) and displaying (_showRanking() and _main()) the data in case you wish to do something other than dumping the results out to the user via the terminal. In practice, you might need to send this data back via a Web template, store it in a database, text it to a mobile phone, etc. If you put all of this code into a single function, it makes it harder to reuse and/or repurpose.

Also, if Amazon changes the layout of their product pages, you might need to modify the regular expression “screenscraper” to continue to be able to extract the data from the product page. By the way, using a regex (or even plain old string processing) for this simple example is fine, but you might need a more powerful markup parser, such as HTMLParser from the standard library or third-party tools like BeautifulSoup, html5lib, or lxml. (We demonstrate a few of these in Chapter 9, “Web Clients and Servers.”)
Add threading

Okay, you don’t have to tell me that this is still a silly single-threaded program. We’re going to change our application to use threads instead. It is an I/O-bound application, so this is a good candidate to do so. To simplify things, we won’t use any of the classes and object-oriented programming; instead, we’ll use threading.Thread directly, so you can think of this more as a derivative of mtsleepF.py than any of the succeeding examples. We’ll just spawn the threads and start them up immediately.

Take your application and modify the _showRanking(isbn) call to the following:

    Thread(target=_showRanking, args=(isbn,)).start().

That’s it! Now you have your final version of bookrank.py and can see that the application (typically) runs faster because of the added concurrency. But, your still only as fast as the slowest response.

    $ python bookrank.py
    At Thu Mar 31 10:11:32 2011 on Amazon...
    - 'Python Fundamentals' ranked 869,010
    - 'Core Python Programming' ranked 36,481
    - 'Python Web Development with Django' ranked 219,228
    all DONE at: Thu Mar 31 10:11:35 2011

As you can see from the output, instead of taking six seconds as our single-threaded version, our threaded version only takes three. Also note that the output is in “by completion” order, which is variable, versus the single-threaded display. With the non-threaded version, the order is always by key, but now the queries all happen in parallel with the output coming as each thread completes its work.

In the earlier mtsleepX.py examples, we used Thread.join() on all the threads to block execution until each thread exits. This effectively prevents the main thread from continuing until all threads are done, so the print statement of “all DONE at” is called at the correct time.

In those examples, it’s not necessary to join() all the threads because none of them are daemon threads. The main thread is not going to exit the script until all the spawned threads have completed anyway. Because of this reasoning, we’ve dropped all the join()s in mtsleepF.py. However, realize that if we displayed “all done” from the same spot, it would be incorrect.

The main thread would have displayed “all done” before the threads have completed, so we can’t have that print call above in _main(). There are only 2 places we can put this print: after line 37 when _main() returns (the very final line executed of our script), or use atexit.register() to
register an exit function. Because the latter is something we haven’t discussed before and might be something useful to you later on, we thought this would be a good place to introduce it to you. This is also one interface that remains constant between Python 2 and 3, our upcoming challenge.

**Porting to Python 3**

The next thing we want is a working Python 3 version of this script. As projects and applications continue down the migration path, this is something with which you need to become familiar, anyway. Fortunately, there are few tools to help you, one of them being the 2to3 tool. There are generally two ways of using it:

```
$ 2to3 foo.py    # only output diff
$ 2to3 -w foo.py # overwrites w/3.x code
```

In the first command, the 2to3 tool just displays the differences between the version 2.x original script and its generated 3.x equivalent. The -w flag instructs 2to3 to overwrite the original script with the newly minted 3.x version while renaming the 2.x version to foo.py.bak.

Let’s run 2to3 on bookrank.py, writing over the existing file. It not only spits out the differences, it also saves the new version, as we just described:

```
$ 2to3 -w bookrank.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
--- bookrank.py (original)
+++ bookrank.py (refactored)
@@ -4,7 +4,7 @@
    from re import compile
    from threading import Thread
    from time import ctime
-    from urllib2 import urlopen as uopen
+    from urllib.request import urlopen as uopen
    REGEX = compile('#([\d.,]+) in Books ')  
    AMZN = 'http://amazon.com/dp/'
@@ -21,17 +21,17 @@
        return REGEX.findall(data)[0]
    def _showRanking(isbn):
        print '- %r ranked %s' % (
-            ISBNs[isbn], getRanking(isbn))
+        print('- %r ranked %s' % (
+            ISBNs[isbn], getRanking(isbn)))
```
def _main():
    print 'At', ctime(), 'on Amazon...
    print('At', ctime(), 'on Amazon...')
    for isbn in ISBNs:
        Thread(target=_showRanking,
                args=(isbn,)).start()#_showRanking(isbn)

@register
def _atexit():
    print 'all DONE at:', ctime()
    print('all DONE at:', ctime())

if __name__ == '__main__':
    _main()

The following step is optional for readers, but we renamed our files to
bookrank.py and bookrank3.py by using these POSIX commands (Windows-
based PC users should use the ren command):

$ mv bookrank.py bookrank3.py
$ mv bookrank.py.bak bookrank.py

If you try to run our new next-generation script, it’s probably wishful
thinking that it’s a perfect translation and that you’re done with your
work. Something bad happened, and you’ll get the following exception in
each thread (this output is for just one thread as they’re all the same):

$ python3 bookrank3.py
Exception in thread Thread-1:
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/threading.py", line 736, in _bootstrap_inner
    self.run()
  File "/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/threading.py", line 689, in run
    self._target(*self._args, **self._kwargs)
  File "bookrank3.py", line 25, in _showRanking
    ISBNs[isbn], getRanking(isbn))
  File "bookrank3.py", line 21, in getRanking
    return REGEX.findall(data)[0]
TypeError: can't use a string pattern on a bytes-like object:

Darn it! Apparently the problem is that the regular expression is a (Uni-
code) string, whereas the data that comes back from urlopen() file-like
object’s read() method is an ASCII/bytes string. The fix here is to compile
a bytes object instead of a text string. Therefore, change line 9 so that
re.compile() is compiling a bytes string (by adding the bytes string. To
do this, add the bytes string designation b just before the opening quote, as shown here:

```python
REGEX = compile(b'#([\d,]+) in Books ')```

Now let's try it again:

```bash
$ python3 bookrank3.py
At Sun Apr  3 00:45:46 2011 on Amazon...
- 'Core Python Programming' ranked b'108,796'
- 'Python Web Development with Django' ranked b'268,660'
- 'Python Fundamentals' ranked b'969,149'
all DONE at: Sun Apr  3 00:45:49 2011
```

Aargh! What's wrong now? Well, it's a little bit better (no errors), but the output looks weird. The ranking values grabbed by the regular expressions, when passed to str() show the b and quotes. Your first instinct might be to try ugly string slicing:

```python
>>> x = b'xxx'
>>> repr(x)
'b'xxx'"
>>> str(x)
'b'xxx'"
>>> str(x)[2:-1]
'xxx'
```

However, it's just more appropriate to convert it to a real (Unicode string, perhaps using UTF-8):

```python
>>> str(x, 'utf-8')
'xxx'
```

To do that in our script, make a similar change to line 53 so that it now reads as:

```python
return str(REGEX.findall(data)[0], 'utf-8')```

Now, the output of our Python 3 script matches that of our Python 2 script:

```bash
$ python3 bookrank3.py
At Sun Apr  3 00:47:31 2011 on Amazon...
- 'Python Fundamentals' ranked 969,149
- 'Python Web Development with Django' ranked 268,660
- 'Core Python Programming' ranked 108,796
all DONE at: Sun Apr  3 00:47:34 2011
```

In general, you'll find that porting from version 2.x to version 3.x follows a similar pattern: you ensure that all your unit and integration tests pass, knock down all the basics using 2to3 (and other tools), and then clean up the aftermath by getting the code to run and pass the same tests. We'll try this exercise again with our next example which demonstrates the use of synchronization with threads.
4.7.2 Synchronization Primitives

In the main part of this chapter, we looked at basic threading concepts and how to utilize threading in Python applications. However, we neglected to mention one very important aspect of threaded programming: synchronization. Often times in threaded code, you will have certain functions or blocks in which you don’t (or shouldn’t) want more than one thread executing. Usually these involve modifying a database, updating a file, or anything similar that might cause a race condition, which, if you recall from earlier in the chapter, is when different code paths or behaviors are exhibited or inconsistent data was rendered if one thread ran before another one and vice versa. (You can read more about race conditions on the Wikipedia page at http://en.wikipedia.org/wiki/Race_condition.)

Such cases require synchronization. Synchronization is used when any number of threads can come up to one of these critical sections of code (http://en.wikipedia.org/wiki/Critical_section), but only one is allowed through at any given time. The programmer makes these determinations and chooses the appropriate synchronization primitives, or thread control mechanisms to perform the synchronization. There are different types of process synchronization (see http://en.wikipedia.org/wiki/Synchronization_(computer_science)) and Python supports several types, giving you enough choices to select the best one to get the job done.

We introduced them all to you earlier at the beginning of this section, so here we’d like to demonstrate a couple of sample scripts that use two types of synchronization primitives: locks/mutexes, and semaphores. A lock is the simplest and lowest-level of all these mechanisms; while semaphores are for situations in which multiple threads are contending for a finite resource. Locks are easier to explain, so we’ll start there, and then discuss semaphores.

4.7.3 Locking Example

Locks have two states: locked and unlocked (surprise, surprise). They support only two functions: acquire and release. These actions mean exactly what you think.

As multiple threads vie for a lock, the first thread to acquire one is permitted to go in and execute code in the critical section. All other threads coming along are blocked until the first thread wraps up, exits the critical section, and releases the lock. At this moment, any of the other waiting threads can acquire the lock and enter the critical section. Note that there
is no ordering (first come, first served) for the blocked threads; the selection of the “winning” thread is not deterministic and can vary between different implementations of Python.

Let’s see why locks are necessary. mtsleepF.py is an application that spawns a random number of threads, each of which outputs when it has completed. Take a look at the core chunk of (Python 2) source here:

```python
from atexit import register
from random import randrange
from threading import Thread, currentThread
from time import sleep, ctime

class CleanOutputSet(set):
    def __str__(self):
        return ', '.join(x for x in self)

loops = (randrange(2,5) for x in xrange(randrange(3,7)))
remaining = CleanOutputSet()

def loop(nsec):
    myname = currentThread().name
    remaining.add(myname)
    print '[%s] Started %s' % (ctime(), myname)
    sleep(nsec)
    remaining.remove(myname)
    print '[%s] Completed %s (%d secs)' % (ctime(), myname, nsec)
    print '    (remaining: %s)' % (remaining or 'NONE')

def _main():
    for pause in loops:
        Thread(target=loop, args=(pause,)).start()
    
@register
def _atexit():
    print 'all DONE at:', ctime()
```

We’ll have a longer line-by-line explanation once we’ve finalized our code with locking, but basically what mtsleepF.py does is expand on our earlier examples. Like bookrank.py, we simplify the code a bit by skipping object-oriented programming, drop the list of thread objects and thread join()s, and (re)use atexit.register() (for all the same reasons as bookrank.py).

Also as a minor change to the earlier mtsleepX.py examples, instead of hardcoding a pair of loops/threads sleeping for 4 and 2 seconds, respectively, we wanted to mix it up a little by randomly creating between 3 and 6 threads, each of which can sleep anywhere between 2 and 4 seconds.
One of the new features that stands out is the use of a set to hold the names of the remaining threads still running. The reason why we're subclassing the set object instead of using it directly is because we just want to demonstrate another use case, altering the default printable string representation of a set.

When you display a set, you get output such as `set([X, Y, Z, ...])`. The issue is that the users of our application don't (and shouldn't) need to know anything about sets or that we're using them. We just want to display something like `X, Y, Z, ...`, instead; thus the reason why we derived from `set` and implemented its `__str__()` method.

With this change, and if you're lucky, the output will be all nice and lined up properly:

```python
$ python mtsleepF.py
[Sat Apr 2 11:37:26 2011] Started Thread-1
[Sat Apr 2 11:37:26 2011] Started Thread-3
[Sat Apr 2 11:37:29 2011] Completed Thread-2 (3 secs)
  (remaining: Thread-3, Thread-1)
[Sat Apr 2 11:37:30 2011] Completed Thread-1 (4 secs)
  (remaining: Thread-3)
[Sat Apr 2 11:37:30 2011] Completed Thread-3 (4 secs)
  (remaining: NONE)
all DONE at: Sat Apr 2 11:37:30 2011
```

However, if you're unlucky, you might get strange output such as this pair of example executions:

```python
$ python mtsleepF.py
[Sat Apr 2 11:37:09 2011] Started Thread-1
[Sat Apr 2 11:37:09 2011] Started Thread-3
[Sat Apr 2 11:37:12 2011] Completed Thread-1 (3 secs)
  (remaining: Thread-3)
  (remaining: Thread-3)
[Sat Apr 2 11:37:12 2011] Completed Thread-2 (3 secs)
  (remaining: Thread-3)
  (remaining: Thread-3)
[Sat Apr 2 11:37:12 2011] Completed Thread-3 (3 secs)
  (remaining: NONE)
all DONE at: Sat Apr 2 11:37:12 2011
```

```python
$ python mtsleepF.py
[Sat Apr 2 11:37:56 2011] Started Thread-1
[Sat Apr 2 11:37:56 2011] Started Thread-3
  (remaining: Thread-3, Thread-1)
[Sat Apr 2 11:37:58 2011] Completed Thread-4 (2 secs)
  (remaining: Thread-3, Thread-1)
```
What’s wrong? Well, for one thing, the output might appear partially garbled (because multiple threads might be executing I/O in parallel). You can see some examples of preceding code in which the output is interleaved, too. Another problem identified is when you have two threads modifying the same variable (the set containing the names of the remaining threads).

Both the I/O and access to the same data structure are part of critical sections; therefore, we need locks to prevent more than one thread from entering them at the same time. To add locking, you need to add a line of code to import the Lock (or RLock) object and create a lock object, so add/modify your code to contain these lines in the right places:

```python
from threading import Thread, Lock, currentThread
lock = Lock()
```

Now you must use your lock. The following code highlights the acquire() and release() calls that we should insert into our loop() function:

```python
def loop(nsec):
    myname = currentThread().name
    lock.acquire()
    remaining.add(myname)
    print '[%s] Started %s' % (ctime(), myname)
    lock.release()
    sleep(nsec)
    lock.acquire()
    remaining.remove(myname)
    print '[%s] Completed %s (%d secs)' % (ctime(), myname, nsec)
    print '    (remaining: %s)' % (remaining or 'NONE')
    lock.release()
```

Once the changes are made, you should no longer get strange output:

```
$ python mtsleepF.py
[Sun Apr  3 23:16:59 2011] Started Thread-1
[Sun Apr  3 23:16:59 2011] Started Thread-4
[Sun Apr  3 23:17:01 2011] Completed Thread-3 (2 secs) (remaining: Thread-4, Thread-2, Thread-1)
[Sun Apr  3 23:17:01 2011] Completed Thread-4 (2 secs) (remaining: Thread-2, Thread-1)
```
The modified (and final) version of \texttt{mtsleepF.py} is shown in Example 4-10.

**Example 4-10  Locks and More Randomness (**\texttt{mtsleepF.py}\texttt{**})**

In this example, we demonstrate the use of locks and other threading tools.

```python
#!/usr/bin/env python
from atexit import register
from random import randrange
from threading import Thread, Lock, currentThread
from time import sleep, ctime

class CleanOutputSet(set):
    def __str__(self):
        return ', '.join(x for x in self)

lock = Lock()
loops = (randrange(2, 5) for x in xrange(randrange(3, 7)))
remaining = CleanOutputSet()

def loop(nsec):
    myname = currentThread().name
    lock.acquire()
    remaining.add(myname)
    print ' [%s] Started %s' % (ctime(), myname)
    lock.release()
    sleep(nsec)
    lock.acquire()
    remaining.remove(myname)
    print ' [%s] Completed %s (%d secs)' % (ctime(), myname, nsec)
    print '    (remaining: %s)' % (remaining or 'NONE')
    lock.release()

def _main():
    for pause in loops:
        Thread(target=loop, args=(pause,)).start()

@register
def _atexit():
    print 'all DONE at:', ctime()

if __name__ == '__main__':
    main()
```
Line-by-Line Explanation

Lines 1–6
These are the usual startup and import lines. Be aware that threading.currentThread() is renamed to threading.current_thread() starting in version 2.6 but with the older name remaining intact for backward compatibility.

Lines 8–10
This is the set subclass we described earlier. It contains an implementation of __str__() to change the output from the default to a comma-delimited string of its elements.

Lines 12–14
Our global variables consist of the lock, an instance of our modified set from above, and a random number of threads (between three and six), each of which will pause or sleep for between two and four seconds.

Lines 16–28
The loop() function saves the name of the current thread executing it, then acquires a lock so that the addition of that name to the remaining set and an output indicating the thread has started is atomic (where no other thread can enter this critical section). After releasing the lock, this thread sleeps for the predetermined random number of seconds, then re-acquires the lock in order to do its final output before releasing it.

Lines 30–39
The _main() function is only executed if this script was not imported for use elsewhere. Its job is to spawn and execute each of the threads. As mentioned before, we use atexit.register() to register the _atexit() function that the interpreter can execute before exiting.

As an alternative to maintaining your own set of currently running threads, you might consider using threading.enumerate(), which returns a list of all threads that are still running (including daemon threads, but not those which haven’t started yet). We didn’t use it for our example here because it gives us two extra threads that we need to remove to keep our output short: the current thread (because it hasn’t completed yet) as well as the main thread (not necessary to show this either).
Also don’t forget that you can also use the str.format() method instead of the string format operator if you’re using Python 2.6 or newer (including version 3.x). In other words, this print statement

```
print '[%s] Started %s' % (ctime(), myname)
```

2.6-2.7 can be replaced by this one in 2.6+

```
print '[{0}] Started {1}'.format(ctime(), myname)
```

3.x or this call to the print() function in version 3.x:

```
print('[{0}] Started {1}'.format(ctime(), myname))
```

If you just want a count of currently running threads, you can use threading.activeCount() (renamed to active_count() starting in version 2.6), instead.

### Using Context Management

Another option for those of you using Python 2.5 and newer is to have neither the lock acquire() nor release() calls at all, simplifying your code. When using the with statement, the context manager for each object is responsible for calling acquire() before entering the suite and release() when the block has completed execution.

The threading module objects Lock, RLock, Condition, Semaphore, and BoundedSemaphore, all have context managers, meaning they can be used with the with statement. By using with, you can further simplify loop() to:

```
from __future__ import with_statement # 2.5 only
def loop(nsec):
    myname = currentThread().name
    with lock:
        remaining.add(myname)
        print '[%s] Started %s' % (ctime(), myname)
    sleep(nsec)
    with lock:
        remaining.remove(myname)
        print '[%s] Completed %s (%d secs)' % (ctime(), myname, nsec)
        print '    (remaining: %s)' % (remaining or 'NONE',)
```

### Porting to Python 3

Now let’s do a seemingly easy port to Python 3.x by running the 2to3 tool on the preceding script (this output is truncated because we saw a full diff dump earlier):
$ 2to3 -w mtsleepF.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma

RefactoringTool: Files that were modified:
RefactoringTool: mtsleepF.py
After renaming mtsleepF.py to mtsleepF3.py and mtsleep.py.bak to
mtsleepF.py, we discover, much to our pleasant surprise, that this is one
script that ported perfectly, with no issues:

$ python3 mtsleepF3.py
[Sun Apr  3 23:29:41 2011] Completed Thread-3 (2 secs)
(remaining: Thread-2, Thread-1)
(remaining: Thread-1)
(remaining: NONE)
all DONE at: Sun Apr  3 23:29:43 2011

Now let’s take our knowledge of locks, introduce semaphores, and look
at an example that uses both.

4.7.4 Semaphore Example

As stated earlier, locks are pretty simple to understand and implement. It’s
also fairly easy to decide when you should need them. However, if the sit-
uation is more complex, you might need a more powerful synchronization
primitive, instead. For applications with finite resources, using sema-
phores might be a better bet.

Semaphores are some of the oldest synchronization primitives out
there. They’re basically counters that decrement when a resource is being
consumed (and increment again when the resource is released). You can
think of semaphores representing their resources as either available or
unavailable. The action of consuming a resource and decrementing the
counter is traditionally called P() (from the Dutch word probeer/proberen)
but is also known as wait, try, acquire, pend, or procure. Conversely, when a
thread is done with a resource, it needs to return it back to the pool. To do
this, the action used is named “V()” (from the Dutch word verhogen/
verhoog) but also known as signal, increment, release, post, vacate. Python
simplifies all the naming and uses the same function/method names as
locks: acquire and release. Semaphores are more flexible than locks because you can have multiple threads, each using one of the instances of the finite resource.

For our example, we’re going to simulate an oversimplified candy vending machine as an example. This particular machine has only five slots available to hold inventory (candy bars). If all slots are taken, no more candy can be added to the machine, and similarly, if there are no more of one particular type of candy bar, consumers wishing to purchase that product are out-of-luck. We can track these finite resources (candy slots) by using a semaphore.

Example 4-11 shows the source code (candy.py).

Example 4-11  Candy Vending Machine and Semaphores (candy.py)

This script uses locks and semaphores to simulate a candy vending machine.

```python
#!/usr/bin/env python
from atexit import register
from random import randrange
from threading import BoundedSemaphore, Lock, Thread
from time import sleep, ctime

lock = Lock()
MAX = 5
candytray = BoundedSemaphore(MAX)

def refill():
    lock.acquire()
    print 'Refilling candy...',
    try:
        candytray.release()
    except ValueError:
        print 'full, skipping'
    else:
        print 'OK'
    lock.release()

def buy():
    lock.acquire()
    print 'Buying candy...',
    if candytray.acquire(False):
        print 'OK'
    else:
        print 'empty, skipping'
    lock.release()
```
4.7 Multithreading in Practice 199

Line-by-Line Explanation

Lines 1–6
The startup and import lines are quite similar to examples earlier in this chapter. The only thing new is the semaphore. The threading module comes with two semaphore classes, Semaphore and BoundedSemaphore. As you know, semaphores are really just counters; they start off with some fixed number of a finite resource.

This counter decrements when one unit of this is allocated, and when that unit is returned to the pool, the counter increments. The additional feature you get with a BoundedSemaphore is that the counter can never increment beyond its initial value; in other words, it prevents the aberrant use case where a semaphore is released more times than it’s acquired.

Lines 8–10
The global variables in this script are the lock, a constant representing the maximum number of items that can be inventoried, and the tray of candy.
Lines 12–21
The refill() function is performed when the owner of the fictitious vending machines comes to add one more item to inventory. The entire routine represents a critical section; this is why acquiring the lock is the only way to execute all lines. The code outputs its action to the user as well as warns when someone has exceeded the maximum inventory (lines 17–18).

Lines 23–30
buy() is the converse of refill(); it allows a consumer to acquire one unit of inventory. The conditional (line 26) detects when all finite resources have been consumed already. The counter can never go below zero, so this call would normally block until the counter is incremented again. By passing the nonblocking flag as False, this instructs the call to not block but to return a False if it would’ve blocked, indicating no more resources.

Lines 32–40
The producer() and consumer() functions merely loop and make corresponding calls to refill() and buy(), pausing momentarily between calls.

Lines 42–55
The remainder of the code contains the call to _main() if the script was executed from the command-line, the registration of the exit function, and finally, _main(), which seeds the newly created pair of threads representing the producer and consumer of the candy inventory.

The additional math in the creation of the consumer/buyer is to randomly suggest positive bias where a customer might actually consume more candy bars than the vendor/producer puts in the machine (otherwise, the code would never enter the situation in which the consumer attempts to buy a candy bar from an empty machine).

Running the script results in output similar to the following:

$ python candy.py
starting at: Mon Apr  4 00:56:02 2011
THE CANDY MACHINE (full with 5 bars)!
Buying candy... OK
Refilling candy... OK
Refilling candy... full, skipping
Buying candy... OK
Buying candy... OK
Refilling candy... OK
Buying candy... OK
Buying candy... OK
Buying candy... OK
all DONE at: Mon Apr  4 00:56:08 2011
CORE TIP: Debugging might involve intervention

At some point, you might need to debug a script that uses semaphores, but to do this, you might need to know exactly what value is in the semaphore’s counter at any given time. In one of the exercises at the end of the chapter, you will implement such a solution to candy.py, perhaps calling it candydebug.py, and give it the ability to display the counter’s value. To do this, you’ll need to look at the source code for threading.py (and probably in both the Python 2 and Python 3 versions).

You’ll discover that the threading module’s synchronization primitives are not class names even though they use CamelCase capitalization to look like a class. In fact, they’re really just one-line functions that instantiate the objects you’re expecting. There are two problems to consider: the first one is that you can’t subclass them (because they’re functions); the second problem is that the variable name changed between version 2.x and 3.x.

The entire issue could be avoided if the object gives you clean/easy access to a counter, which it doesn’t. You can directly access the counter’s value because it’s just an attribute of the class, as we just mentioned, the variable name changed from self.__value, meaning self._Semaphore__value, in Python 2 to self._value in Python 3.

For developers, the cleanest application programming interface (API) (at least in our opinion) is to derive from threading._BoundedSemaphore class and implement an __len__() method but use the correct counter value we just discussed if you plan to support this on both version 2.x and version 3.x.

Porting to Python 3

Similar to mtsleepF.py, candy.py is another example of how the 2to3 tool is sufficient to generate a working Python 3 version, which we have renamed to candy3.py. We’ll leave this as an exercise for the reader to confirm.

Summary

We’ve demonstrated only a couple of the synchronization primitives that come with the threading module. There are plenty more for you to explore. However, keep in mind that that’s still only what they are: “primitives.” There’s nothing wrong with using them to build your own classes and data structures that are thread-safe. The Python Standard Library comes with one, the Queue object.
4.8 Producer-Consumer Problem and the Queue/queue Module

The final example illustrates the producer-consumer scenario in which a producer of goods or services creates goods and places it in a data structure such as a queue. The amount of time between producing goods is non-deterministic, as is the consumer consuming the goods produced by the producer.

We use the Queue module (Python 2.x; renamed to queue in version 3.x) to provide an interthread communication mechanism that allows threads to share data with each other. In particular, we create a queue into which the producer (thread) places new goods and the consumer (thread) consumes them. Table 4-5 itemizes the various attributes that can be found in this module.

Table 4-5 Common Queue/queue Module Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Queue/queue Module Classes</strong></td>
<td></td>
</tr>
<tr>
<td><code>Queue(maxsize=0)</code></td>
<td>Creates a FIFO queue of given <code>maxsize</code> where inserts block until there is more room, or (if omitted), unbounded</td>
</tr>
<tr>
<td><code>LifoQueue(maxsize=0)</code></td>
<td>Creates a LIFO queue of given <code>maxsize</code> where inserts block until there is more room, or (if omitted), unbounded</td>
</tr>
<tr>
<td><code>PriorityQueue(maxsize=0)</code></td>
<td>Creates a priority queue of given <code>maxsize</code> where inserts block until there is more room, or (if omitted), unbounded</td>
</tr>
</tbody>
</table>

| **Queue/queue Exceptions**                                                                 |
|-------------------------------|----------------------------------------------------------------------------------|
| `Empty`                       | Raised when a `get*()` method called for an empty queue                           |
| `Full`                        | Raised when a `put*()` method called for a full queue                              |
4.8 Producer-Consumer Problem and the Queue/queue Module

We'll use Example 4-12 (prodcons.py), to demonstrate producer-consumer Queue/queue. The following is the output from one execution of this script:

```
$ prodcons.py
starting writer at: Sun Jun 18 20:27:07 2006
producing object for Q... size now 1
starting reader at: Sun Jun 18 20:27:07 2006
consumed object from Q... size now 0
producing object for Q... size now 1
consumed object from Q... size now 1
producing object for Q... size now 1
consumed object from Q... size now 0
producing object for Q... size now 2
producing object for Q... size now 3
consumed object from Q... size now 2
consumed object from Q... size now 1
writer finished at: Sun Jun 18 20:27:17 2006
consumed object from Q... size now 0
all DONE
```
Example 4-12  Producer-Consumer Problem (prodcons.py)

This implementation of the Producer–Consumer problem uses Queue objects and a random number of goods produced (and consumed). The producer and consumer are individually—and concurrently—executing threads.

```python
#!/usr/bin/env python

from random import randint
from time import sleep
from queue import Queue
from myThread import MyThread

def writeQ(queue):
    print('producing object for Q...',
    queue.put('xxx', 1)
    print "size now", queue.qsize()

def readQ(queue):
    val = queue.get(1)
    print 'consumed object from Q... size now', \
        queue.qsize()

def writer(queue, loops):
    for i in range(loops):
        writeQ(queue)
        sleep(randint(1, 3))

def reader(queue, loops):
    for i in range(loops):
        readQ(queue)
        sleep(randint(2, 5))

funcs = [writer, reader]

if __name__ == '__main__':
    main()```
As you can see, the producer and consumer do not necessarily alternate in execution. (Thank goodness for random numbers!) Seriously, though, real life is generally random and non-deterministic.

**Line-by-Line Explanation**

*Lines 1–6*
In this module, we use the `Queue.Queue` object as well as our thread class `myThread.MyThread`, seen earlier. We use `random.randint()` to make production and consumption somewhat varied. (Note that `random.randint()` works just like `random.randrange()` but is *inclusive* of the upper/end value).

*Lines 8–16*
The `writeQ()` and `readQ()` functions each have a specific purpose: to place an object in the queue—we are using the string 'xxx', for example—and to consume a queued object, respectively. Notice that we are producing one object and reading one object each time.

*Lines 18–26*
The `writer()` is going to run as a single thread whose sole purpose is to produce an item for the queue, wait for a bit, and then do it again, up to the specified number of times, chosen randomly per script execution. The `reader()` will do likewise, with the exception of consuming an item, of course.

You will notice that the random number of seconds that the writer sleeps is in general shorter than the amount of time the reader sleeps. This is to discourage the reader from trying to take items from an empty queue. By giving the writer a shorter time period of waiting, it is more likely that there will already be an object for the reader to consume by the time their turn rolls around again.

*Lines 28–29*
These are just setup lines to set the total number of threads that are to be spawned and executed.
Finally, we have our `main()` function, which should look quite similar to the `main()` in all of the other scripts in this chapter. We create the appropriate threads and send them on their way, finishing up when both threads have concluded execution.

We infer from this example that a program that has multiple tasks to perform can be organized to use separate threads for each of the tasks. This can result in a much cleaner program design than a single-threaded program that attempts to do all of the tasks.

In this chapter, we illustrated how a single-threaded process can limit an application’s performance. In particular, programs with independent, non-deterministic, and non-causal tasks that execute sequentially can be improved by division into separate tasks executed by individual threads. Not all applications will benefit from multithreading due to overhead and the fact that the Python interpreter is a single-threaded application, but now you are more cognizant of Python’s threading capabilities and can use this tool to your advantage when appropriate.

### 4.9 Alternative Considerations to Threads

Before you rush off and do some threading, let’s do a quick recap: threading in general is a good thing. However, because of the restrictions of the GIL in Python, threading is more appropriate for I/O-bound applications (I/O releases the GIL, allowing for more concurrency) than for CPU-bound applications. In the latter case, to achieve greater parallelism, you’ll need processes that can be executed by other cores or CPUs.

Without going into too much detail here (some of these topics have already been covered in the “Execution Environment” chapter of *Core Python Programming* or *Core Python Language Fundamentals*), when looking at multiple threads or processes, the primary alternatives to the `threading` module include:

#### 4.9.1 The subprocess Module

This is the primary alternative when desiring to spawn processes, whether to purely execute stuff or to communicate with another process via the standard files (`stdin`, `stdout`, `stderr`). It was introduced to Python in version 2.4.
4.9.2 The multiprocessing Module

This module, added in Python 2.6, lets you spawn processes for multiple cores or CPUs but with an interface very similar to that of the threading module; it also contains various mechanisms to pass data between processes that are cooperating on shared work.

4.9.3 The concurrent.futures Module

This is a new high-level library that operates only at a “job” level, which means that you no longer have to fuss with synchronization, or managing threads or processes. You just specify a thread or process pool with a certain number of “workers,” submit jobs, and collate the results. It’s new in Python 3.2, but a port for Python 2.6+ is available at http://code.google.com/p/pythonfutures.

What would bookrank3.py look like with this change? Assuming everything else stays the same, here’s the new import and modified _main() function:

```python
from concurrent.futures import ThreadPoolExecutor

def _main():
    print('At', ctime(), 'on Amazon...')
    with ThreadPoolExecutor(3) as executor:
        for isbn in ISBNs:
            executor.submit(_showRanking, isbn)
    print('all DONE at:', ctime())
```

The argument given to concurrent.futures.ThreadPoolExecutor is the thread pool size, and our application is looking for the rankings of three books. Of course, this is an I/O-bound application for which threads are more useful. For a CPU-bound application, we would use concurrent.futures.ProcessPoolExecutor, instead.

Once we have an executor (whether threads or processes), which is responsible for dispatching the jobs and collating the results, we can call its submit() method to execute what we would have had to spawn a thread to run previously.

If we do a “full” port to Python 3 by replacing the string format operator with the str.format() method, making liberal use of the with statement, and using the executor’s map() method, we can actually delete _showRanking() and roll its functionality into _main(). In Example 4-13, you’ll find our final bookrank3CF.py script.
Line-by-Line Explanation

**Lines 1–14**
Outside of the new `import` statement, everything in the first half of this script is identical to the `bookrank3.py` file we looked at earlier in this chapter.

**Lines 16–18**
The new `getRanking()` uses the `with` statement and `str.format()`. You can make the same change to `bookrank.py` because both features are available in version 2.6+ (they are not unique to version 3.x).

**Lines 20–26**
In the previous code example, we used `executor.submit()` to spawn the jobs. Here, we tweak this slightly by using `executor.map()` because it
allows us to absorb the functionality from \_showRanking(), letting us remove it entirely from our code.

The output is nearly identical to what we’ve seen earlier:

```
$ python3 bookrank3CF.py
At Wed Apr 6 00:21:50 2011 on Amazon...
- 'Core Python Programming' ranked 43,992
- 'Python Fundamentals' ranked 1,018,454
- 'Python Web Development with Django' ranked 502,566
all DONE at: Wed Apr 6 00:21:55 2011
```

You can read more about the concurrent.futures module origins at the link below.

- http://docs.python.org/dev/py3k/library/concurrent.futures.html
- http://code.google.com/p/pythonfutures/
- http://www.python.org/dev/peps/pep-3148/

A summary of these options and other threading-related modules and packages can be found in the next section.

## 4.10 Related Modules

Table 4-6 lists some of the modules that you can use when programming multithreaded applications.

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread(^a)</td>
<td>Basic, lower-level thread module</td>
</tr>
<tr>
<td>threading</td>
<td>Higher-level threading and synchronization objects</td>
</tr>
<tr>
<td>multiprocessing(^b)</td>
<td>Spawn/use subprocesses with a “threading” interface</td>
</tr>
<tr>
<td>subprocess(^c)</td>
<td>Skip threads altogether and execute processes instead</td>
</tr>
<tr>
<td>Queue</td>
<td>Synchronized FIFO queue for multiple threads</td>
</tr>
<tr>
<td>mutex(^d)</td>
<td>Mutual exclusion objects</td>
</tr>
</tbody>
</table>

(Continued)
4.11 Exercises

4-1. *Processes versus Threads.* What are the differences between processes and threads?

4-2. *Python Threads.* Which type of multithreaded application will tend to fare better in Python, I/O-bound or CPU-bound?

4-3. *Threads.* Do you think anything significant happens if you have multiple threads on a multiple CPU system? How do you think multiple threads run on these systems?

4-4. *Threads and Files.*

   a) Create a function that obtains a byte value and a filename (as parameters or user input) and displays the number of times that byte appears in the file.

   b) Suppose now that the input file is extremely large. Multiple readers in a file is acceptable, so modify your solution to create multiple threads that count in different parts of the file such that each thread is responsible for a certain part of the file. Collate the data from each thread and provide the correct total. Use the `timeit` module to time both the single-threaded new multithreaded solutions and say something about the difference in performance, if any.

4-5. *Threads, Files, and Regular Expressions.* You have a very large mailbox file—if you don’t have one, put all of your e-mail messages together into a single text file. Your job is to take

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>concurrent.futurese</td>
<td>High-level library for asynchronous execution</td>
</tr>
<tr>
<td>SocketServer</td>
<td>Create/manage threaded TCP or UDP servers</td>
</tr>
</tbody>
</table>

---

*a.* Renamed to `_thread` in Python 3.0.

*b.* New in Python 2.6.

*c.* New in Python 2.4.

*d.* Deprecated in Python 2.6 and removed in version 3.0.

*e.* New in Python 3.2 (but available outside the standard library for version 2.6+).
the regular expressions you designed earlier in this book that recognize e-mail addresses and Web site URLs and use them to convert all e-mail addresses and URLs in this large file into live links so that when the new file is saved as an .html (or .htm) file, it will show up in a Web browser as live and clickable. Use threads to segregate the conversion process across the large text file and collate the results into a single new .html file. Test the results on your Web browser to ensure the links are indeed working.

4-6. *Threads and Networking.* Your solution to the chat service application in the previous chapter required you to use heavyweight threads or processes as part of your solution. Convert your solution to be multithreaded.

4-7. *Threads and Web Programming.* The Crawler application in Chapter 10, “Web Programming: CGI and WSGI,” is a single-threaded application that downloads Web pages. It would benefit from MT programming. Update crawl.py (you could call it mtcrawl.py) such that independent threads are used to download pages. Be sure to use some kind of locking mechanism to prevent conflicting access to the links queue.

4-8. *Thread Pools.* Instead of a producer thread and a consumer thread, change the code for prodcons.py, in Example 4-12 so that you have any number of consumer threads (a thread pool) which can process or consume more than one item from the Queue at any given moment.

4-9. *Files.* Create a set of threads to count how many lines there are in a set of (presumably large) text files. You can choose the number of threads to use. Compare the performance against a single-threaded version of this code. Hint: Review the exercises at the end of the Chapter 9, in Core Python Programming or Core Python Language Fundamentals.

4-10. *Concurrent Processing.* Take your solution to Exercise 4-9 and adopt it to a task of your selection, for example, processing a set of e-mail messages, downloading Web pages, processing RSS or Atom feeds, enhancing message processing as part of a chat server, solving a puzzle, etc.

4-11. *Synchronization Primitives.* Investigate each of the synchronization primitives in the threading module. Describe what they do, what they might be useful for, and create working code examples for each.
The next couple of exercises deal with the candy.py script featured in Example 4-11.

4-12. Porting to Python 3. Take the candy.py script and run the 2to3 tool on it to create a Python 3 version called candy3.py.

4-13. The threading module. Add debugging to the script. Specifically, for applications that use semaphores (whose initial value is going to be greater than 1), you might need to know exactly the counter’s value at any given time. Create a variation of candy.py, perhaps calling it candydebug.py, and give it the ability to display the counter’s value. You will need to look at the threading.py source code, as alluded to earlier in the CORE TIP sidebar. Once you’re done with the modifications, you can alter its output to look something like the following:

```bash
$ python candydebug.py
starting at: Mon Apr  4 00:24:28 2011
THE CANDY MACHINE (full with 5 bars)!
Buying candy... inventory: 4
Refilling candy... inventory: 5
Refilling candy... full, skipping
Buying candy... inventory: 4
Buying candy... inventory: 3
Refilling candy... inventory: 4
Buying candy... inventory: 3
Buying candy... inventory: 2
Buying candy... inventory: 1
Buying candy... inventory: 0
Buying candy... empty, skipping
all DONE at: Mon Apr  4 00:24:36 2011
```
Symbols
^ (carat) symbol
  for matching from start of string, 6, 10
  for negation, 12
? (question mark), in regex, 6, 12–13, 24, 47
. (dot) symbol, in regex, 6, 9, 23
(?:...) notation, 32
(?!...) notation, 33
(?=...) notation, 33
{ } (brace operators), 12
{ % } (percent signs and braces), for Django block tags, 529
{% block ... %} tag, 553
{% extends ... %} tag, 554
* (asterisk), in regex, 6, 12–13
** (exponentiation), 771
/ (division operator), 771, 810
// (floor division), 772, 803, 804, 810, 811
\ (backslash) to escape characters to include in search, 23
\s (special character, for whitespace characters, 14
& (ampersand), for key-value pairs, 403
# (hash symbol)
  for comment, 32
  for Django comments, 518
% (percent sign)
  for hexadecimal ordinal equivalents, 403
  for modulo, 772
  in string format operator conversion symbols, 776
+ (plus sign)
  for encoding, 403
  in regex, 6, 12–13
| (pipe symbol)
  for Django variable tag filters, 528
  in regex, 9
~ (bit inversion), 771
$ (dollar sign), for matching end of string, 6, 10
Numerics
2to3 tool, 187, 407, 805, 817
-3 switch, for Python 3 transition, 817
3to2 tool, 805, 818
500 HTTP error, 445
A
\A special character, for matching start of string, 10
abs() function, 770
__abs__() method, 793
AbstractFormatter object, 415
accept() method, 62, 65
access key, for Google+ API, 749
access token secret, for Twitter, 694
access token, for Twitter, 694
acquire() method (lock object), 165, 169, 190, 193
Active FTP mode, 98, 103
Active Record pattern, 295
active sheet in Excel, 329
activeCount() function (threading module), 179
active_count() function (threading module), 179
ActiveMapper, 295
ActiveX, 326
See also COM (Component Object Model)
programming adapter for database. See database adapters
add() function (set types), 785
add() method, 792, 794
addition sign (+). See + (plus sign)
address families, 58
Admin Console page, adding Appstats UI
as custom, 671
admin.py file, 559
to register data models, 580
administration app in Django, 518–527
setup, 518–519
ADMIN_MEDIA_PREFIX variable, 570
adodbapi, 317
AdvCGI class, 476
advcgi.py CGI application, 468–478
advertising on cloud services, 135
AF_INET sockets, 58
AF_INET6 sockets, 58
AF_LOCAL sockets, 58
AF_NETLINK sockets, 59
AF_TIPC sockets, 59
AF_UNIX sockets, 58
alpha() method, 298
alpha_categorize() function, 165
alphabet, for regular expressions, 5
alphanumeric character class, \w special class
for, 14
alphanumeric character, matching in regex, 7
alternation (|) operation, in regex, 9
Amazon, 608
“Conditions of Use” guidelines, 182
Amazon Web Services (AWS), 607
ampersand (&), for key-value pairs, 403
anchors, parsing, 418
and operator, 770
__and__() method, 793
animalGtk.pyw application, 242–244
animalTk.ppy application, 245
animalTk3.ppy application, 246
anonymous FTP login, 96, 102
Apache web server, 428, 446, 479
Django and, 497
apiclient.discovery.build() function, 755
API_KEY variable for Google+, 755
apislevel attribute (DB-API), 260
APIS (application programming interfaces), 685
Google App Engine and, 614–616
Twitter libraries, 691
App Engine Blobstore, 613
App Engine. See Google App Engine
App Identity API, 614
app.yaml file, 628
for handling inbound e-mail, 658
for tasks queues, 664
handler for Appstats, 671
inbound_services: section, 661
for remote API shell, 654
append() function, 772
append() method (list), 781
appengine_config.py file, 671
“application/x-www-form-urlencoded”, 466
applications
– event-driven, 80
– Google hosting of, 605
– recording events from, 671
– uploading to Google, 629
– visibility on desktop, 330
apps in Django, 501
– creating, 566
AppScale back-end system, 676
Appstats, 614, 670
– handler for, 671
APSW, 317
archive() view function, 543
arguments, default for widgets, 221
AssertionError, 788
arraysize attribute (DB-API Cursor object), 265
article() method (NNTP object), 107
as keyword, 802, 816
ASCII strings
– regular expression with, 188
vs. Unicode, 800–801, 815
ASCII symbols, vs. regular expression special
characters, 34
assertEqual() method, 555
authentication, 487
in Django, 574, 595
federated, 653
in Google, 755
with Google Accounts, 652
in Google App Engine, 574
SMTP, 118
for Twitter account, 694
urllib2 HTTP example, 405–407
vs. authorization, 569
authentication header, base64-encoded, in
HTTP request, 406
authorization
– vs. authentication, 569
– with Twitter, 694
auto_now_add feature, in Django, 578
B
\b special character, for word boundary
matches, 10
\b special character, for word boundary
matches, 10
backbone, 395
backend server, 394
Backends service/API, 614
background color of button, argument for, 227
backslash (\) to escape characters to include in search, 23
backward compatibility, 799
Barrier object (threading module), 170
base (Web) server, 429
base representation, 794
base64 module, 147
base64-encoded authentication header, in HTTP request, 406
BaseException, 788
BaseHTTPRequestHandler class, 429, 430, 447
BaseHTTPServer class, 430, 432, 489
BaseHTTPServer module, 429, 435
BaseRequestHandler class (SocketServer module), 79
BaseServer class (SocketServer module), 79
BeautifulSoup package, 185, 418, 421, 422, 424, 435, 489
BeautifulSoup.BeautifulSoup class importing, 427
Beazley, David, 384
beginning of string, matching from, 10
Berkeley sockets, 58
Bigtable, 610, 635
bind() function, 770
binary literals, 804, 810
binary operators, 792, 793
BINARY type object (DB-API), 267
Binary type object (DB-API), 267
binascii module, 147
bind() method, 62, 67, 74
binding, 233
binhex module, 147
Bissex, Paul, Python Web Development with Django, 496
bit inversion (~), 771
bitwise operators, 772
blacklist section, in dos.yaml file, 675
blank lines, in newsgroup article, 113
Blobstore, 614
resources, 676
block tags in Django, 529
blocking-oriented socket methods, 63
blog application
admin.py file, 559
code review, 557–563
from Google App Engine, 631–647
adding datastore service, 635–638
adding form, 633–635
iterative improvements, 640
plain text conversion to HTML, 632
manage.py to create, 507
models.py file, 558
reverse-chronological order for, 537
summary, 563
template file, 562
URL pattern creation, 529–533
urls.py file, 557
user interface, 527–537
view function creation, 533–537
views.py file, 560
blog.views.archive() function, 561
blog.views.create_blogpost() function, 561
blog/admin.py file, updating with BlogPostAdmin class, 525
BlogEntry.post() method, 637
blogging, 690
BlogPostAdmin class, 525
BlogPostForm object, 559
Boa Constructor module, 248
body() method (NNTP object), 107
boilerplate code, 370–377
include Python header file, 371
initModule() modules, initializer function, 376
PyMethodDef ModuleMethods[] array, 376
PyObject* Module_func() wrappers, 371–376
SWIG and, 384
boilerplate, base server as, 429
bookrank.py script, 182–189
adding threading, 186–187
non-threaded version, 182–185
porting to Python 3, 187–189
bookrank3CF.py script, 208–209
bool type, 809
Boolean operators, 770
borrowed reference, 383
bot, 410
bottle framework, App Engine and, 617, 676
BoundedSemaphore class, 199
BoundedSemaphore object (threading module), 170
code review, 557–563
context manager, 196
BoxSizer widget, 241
button widget, 515
brace operators ({}), 12
BSD Unix, 58
*BSD, Zip files for App Engine SDK, 620
BSON format, 311
buffer size, for timestamp server, 67
build() function, 754
build_absolute_url() method, 591
built-in functions in Python 3, 813
__builtins__ module, 285
burstiness rates, for task queues, 663
Button widget, 220, 222
Label and Scale widgets with, 224–225
Label widget with, 223
buy() function, 200
bytecode, 19
bytes literals, 815
bytes objects, and string format operator, 409
bytes type, 800, 815
C
C language
converting data between Python and, 372
C language (continued)
creating application for extension, 368–370
extensions in, 365
memory leak, 375
Python-wrapped version of library, 380–382
caching, 647
key for, 649
Memcache in App Engine for, 647–651
on proxy server, 394
__call__() method, 791
callable classes, for threads, 175–176
callables
as deferred tasks, 669
in Django templates, 528
WSGI applications defined as, 481
callbacks, 217
binding event to, 233
callproc() method (DB-API Cursor object), 265
camel capitalization in Twython, 703
candy.py script, 198–200
porting to Python 3, 201
Canvas widget, 220
Capabilities service/API, 614
capitalize() function, 773
capitalize() method (string), 778
capitalizing name in form, 460
carat (^) symbol
for matching from start of string, 6, 10
for negation, 12
Cascading Style Sheets (CSS), 553
C-compiled modules/packages, whitelist, 613
center() method (string), 773, 778
cformat() function, 285, 288
CGI (common gateway interface)
alternatives, 479–487
external processes, 480
server integration, 479
See also WSGI (Web Server Gateway Interface)
basics, 442–444
elements, exercise answer, 766
form encoding specifications, 466
scalability limitations, 494
CGI applications, 444
cookies, 466–478
form and results page generation, 452–456
form page creation, 448–450
fully interactive Web sites, 457–463
multivalued fields, 467
results page, 450–452
Unicode with, 464–465
Web server setup, 446–448
cgi module/package, 433, 445, 488
Channel service/API, 614
resources, 677
character classes, creating, 24
character sets
negation of matches, 14
special characters for, 14
characters
escaping to include in search, 23
hexadecimal ordinal equivalents of
disallowed, 403
matching any single, 6, 23
non-ASCII, 464
See also special characters
chat invitation, 660
chat score, for Google+ posts, 757
Checkbutton widget, 220
checkUserCookie() method, 476
Cheeseshop, 311, 418
CherryPy, 494
child threads, main thread need to wait for, 166
child widget, 217
chr() function, 770, 773
CIL (Common Intermediate Language), 387
class type in Python 3, 801
class wrapper, 486
class-based generic views, 553
classes, special methods for, 791–795
classic classes, 814
clear() function (set types), 785
clear() method (dictionary), 782
client/server architecture
exercise answer, 765
hardware, 55
network programming, 56–57
software, 55–56
Web surfing and, 391–392
window system, 216
XML-RPC and, 733–738
clientConnectionFailed() method, 87
clientConnectionLost() method, 87
clients, 54
awareness of server, 57
for NNTP, 108–114
for UDP
creating, 74–76
executing, 76
FTP
example program, 100–102
list of typical, 103
Internet, 95
location on Internet, 394–395
socket methods, 62–63
spawning threads to handle requests, 65
TCP
creating, 68–71
executing, 71–73
executing Twisted, 87
SocketServer execution, 83
SocketServer for creating, 82–83
Twisted for creating, 85–87
client-side COM programming, 326–327
with Excel, 328–330, 338–340
with Outlook, 334–337, 340–347
with PowerPoint, 332–334, 347–356
with Word, 331
close() method, 63, 66
for server, 72
for UDP server, 73
close() method (DB-API Connection object), 264
close() method (DB-API Cursor object), 265
close() method (file object), 786
close() method (IMAP4 object), 129
close() method (urlopen object), 401
closed() method (file object), 787
closing spreadsheet without saving, 330
close() method (DB-API Connection object), 264
close() method (DB-API Cursor object), 265
close() method (file object), 786
close() method (IMAP4 object), 129
close() method (urlopen object), 401
closed() method (file object), 787
closing spreadsheet without saving, 330
cloud computing, 605–611
levels of service, 607–609
Web-based SaaS, 135
Cloud SQL service/API, 615
Cloud Storage service/API, 615
clrDir() method, 235
CMDs dictionary, 702
cmp() function, 769, 773
__cmp__() method, 792
do coercion() function, 771
do __coerce__() method, 794
co-location, 395
columns in database tables, 255
column-stores, 310
COM (Component Object Model) programming, client-side, 326–327
basics, 325
with Excel, 328–330, 338–340
with Outlook, 334–337, 340–347
with PowerPoint, 332–334, 347–356
with Word, 331
ComboBox widget, 236, 238, 241
ComboBoxEntry widget, position of labels, 244
command shell, executing http.server module from, 447
command-line
FTP clients, 103
to start App Engine application, 629
comma-separated values (CSV), 715–719
Yahoo! Stock Quotes example, 717–719
Comment class, 579
Comment objects, for TweetApprover, 578
comments
hash symbol (#) for, 32
in regex, 8, 16
commit() method (DB-API Connection object), 264, 271
close() method (DB-API Connection object), 264, 271
common gateway interface. See CGI (common
gateway interface)
Common Intermediate Language (CIL), 387
communication endpoint, 58
See also sockets
comparisons, 769
compatibility library, for Tweepy and
twython, 693–706
compilation of regex, decision process, 19
bcmath() function, 17
compiled languages, vs. interpreted, 367
compiling extensions, 377–379
complex() function, 771
__complex__() method, 793
Concurrence networking framework, 89
concurrency, 626
 concurrent.futures module, 207, 210
 concurrent.futures.ProcessPoolExecutor, 207
 concurrent.futures.ThreadPoolExecutor, 207
Condition object (threading module), 170
context manager, 196
conditional expressions, 288
conditional regular expression matching, 34
connect() attribute (DB-API), 260
connect() function, 286
 for database access, 261–262
connect() method, 62
connect() method, 62
connection attribute (DB-API Connection object), 265
Connection objects (DB-API), 263–264
database adapters with, 271
connectionless socket, 60
connectionMade() method, 86
connection-oriented sockets, 60
constants
in Outlook, 336
in PowerPoint, 334
constructors (DB-API), 266–268
consumer key, for OAuth, 694
consumer secret, for OAuth, 694
consumer() function, 200
container environments, and Django install, 500
containers, widgets as, 217
__contains__() method, 794
context, for Django template variables, 528
continue statement, 113
Control widget, 236, 238
Conversion package/API, 615
converting data between Python and C/C++, 372
cookie jar, 476
Cookie module/package, 433, 476, 488
cookielib module/package, 433, 476, 488
cookies, 392, 487
CGI for, 466–478
expiration date, 467
copy() function (set types), 784
copy() method (dictionary), 782
costs, cloud computing services and, 606
CouchDB, 318
couchdb-python, 318
count() function, 773
count() method (list), 781
count() method (string), 778
counters
  semaphores as, 197, 199
  value display for debugging, 201
counting, by App Engine, 643
crawl.py script, 411–418
  sample invocation, 417–418
crawler, 410
Crawler class, 416
CREATE DATABASE statement (SQL), 256
CREATE TABLE statement (MySQL), 271
CREATE TABLE statement (SQL), 256
creator() function, for database table, 287
create_blogpost() view function, 562
create_connection() function, 77
cron job, 101
cron service, 615, 673
cron.yaml file, 673
cross-site request forgery, 544
cStringIO module/package, 413
cStringIO.StringIO class, 731
CSV (comma-separated values), 715–719
downloading files for importing into Excel or Quicken, 685
csv module, 740
  exercise answer, 766
  importing, 716
csv.DictReader class, 717
csv.DictWriter class, 717
csv.reader() function, 717
csv.reader() script, 718
csv.writer() function, 717
csvex.py script, 715–717
current_thread function (threading module), 179
currentThread() function (threading module), 179
cursor for databases, 255
cursor objects (DB-API), 263–266
cursor() method (DB-API Connection object), 264
custom views, 551
  customization of classes, special methods for, 791
cwd Tk string variable, 235
cwd() method (FTP objects), 99
cx_Oraclse, 318
Python, 385
D
  \d special character, for decimal digit, 14
daemon attribute (Thread object), 172
daemon threads, 171
data
  converting between Python and C/C++, 372
  in Python 3, 800
  manipulation, 3
data attributes (DB-API), 260–261
  “Data Mapper” pattern, 295
data models
  admin.py file to register, 580
  BlogPostForm object for, 559
  file for TweetApprover poster app, 578
  for blog application, 558
  for TweetApprover, 576–582
  in Django, experimenting with, 516–517
  repetition vs. DRY, 546
data set, script to generate, 41–43
data strings. See strings
data types, 267
database adapters, 258
  basics, 270
  example application, 275–288
  porting to Python 3, 279–288
  examples, 270–275
  MySQL, 271–272
  PostgreSQL, 272–274
  SQLite, 274–275
database application programmer’s interface (DB-API), 259–288
  changes between versions, 268
  Connection objects, 263–264
  cursor objects, 265–266
  exceptions, 263
  exercise answer, 766
  module attributes, 260–263
  data attributes, 260–261
  function attributes, 261–262
  relational databases, available interfaces, 269–270
  type objects and constructors, 266–268
  web resources, 268
database servers, 55
Database Source Names (DSNs), 294
DatabaseError exception (DB-API), 263
databases
  auto-generating records for testing, 538
  basics, 254–257
  creator() function for tables, 287
  creating engine to, 296
  Django model for, 509–514
  table creation, 512–514
  using MySQL, 510–511
  using SQLite, 511–512
  for Django, 498
  list of supported, 270
  non-relational, 309–315
  MongoDB, 310
  PyMongo, 311–315
  NoSQL, 498
  Python and, 257–258
  row insertion, update, and deletion, 297
  SQL, 256–257
testing, 556
user interface, 255
Web resources on modules/packages, 316
See also object relational managers (ORMs)
DATABASES variable, for TweetApprover, 570
DataError exception (DB-API), 263
datagram type of socket, 60
DatagramRequest-Handler class (SocketServer module), 79
dataReceived() method, 86
datastore admin, for App Engine, 655
Datastore service/API, 614, 615
date
  converting American style to world format, 29
  converting integer to, 43
Date type object (DB-API), 267
DateTime type object (DB-API), 267
days of the week, extracting from timestamp, 44
DB-API. See database application programmer’s interface (DB-API)
dbDump() function, 288
dbDump() method, 298, 307, 315
DB_EXC, 285
DCOracle2, 318
debugging, counter value display and, 201
decode() function, 773
decode() method (string), 778
default arguments, widgets with, 221
default radio button, 454
defered package, in Google App Engine, 668–670
defered defer() function, 668
Dejavu, 289
del() method (POP3 object), 125
delegation, for database operations, 298
DELETE FROM statement (MySQL), 272
DELETE FROM statement (SQL), 257
delete() function, for database adapter, 288
delete() method, 297, 298
delete() method (FTP objects), 99
deleteitem() method, 794, 795
deleteslice() method, 794
__demo_search() function, 706
Denial-of-Service protection, 675
Denial-of-Service service/API, 615
DeprecationWarning, 790
description attribute (DB-API Cursor object), 265
desktop, application visibility on, 330
detach() method, 63
developer servers, 446–448
development server in Django, 505–507
dict.fromkeys() function, 702
dict() factory function, 314
dict2json.py script, 722–724
dict2xml.py script, 725–729
dictionary type built-in methods, 782–783
Diesel, 496
difference_update() function (set types), 785
difference() function (set types), 784
digits
  \d special character for, 14
  matching single in regex, 7
dir() method (FTP objects), 99
directory tree traversal tool, 230–236
direct_to_template() generic view, 561
DirList class, defining constructor for, 232
discard() function (set types), 785
dispatch, static vs. dynamic, 329
Dispatch() function, 329
displayFirst20() function, 113
displaying sets, 192
Distribute, 290
distutils package, 377
distutils.log.warn() function, 279, 285, 693, 716, 722, 819, 821
__div__() method, 792
division from __future__ module, 811
division operator (/), 771, 810
Python 3 changes, 803–804
divmod() function, 771
__divmod__() method, 793
Django, 428, 494
administration app, 518–527
data manipulation, 522–527
setup, 518–519
trying out, 519–527
App Engine and, 617, 676
authentication in, 574, 595
auto_now_add feature, 578
basics, 496
caching, 650
data model experimenting, 516–517
development server in, 505–507
fixtures, 513
forms, 546–550
defining, 590
model forms, 547
ModelForm data processing, 549
ModelForm to generate HTML form, 548
“Hello World” application, 507
installation, 499–501
prerequisites, 497–499
labour-saving features, 563
look-and-feel improvements, 553
model for database service, 509–514
table creation, 512–514
using MySQL, 510–511
using SQLite, 511–512
non-relational databases and, 618
output improvement, 537–541
Django (continued)
model default ordering, 540
query change, 537–540
projects and apps, 501
basic files, 504
project creation, 502–505
Python application shell, 514–517
resources, 597
sending e-mail, 567
templates
directory for, 529
specifying location for Web pages, 570
tutorial, 597
unit testing, 554–557, ??–563
user input, 542–546
cross-site request forgery, 544
template for, 542
URLconf entry, 543
view, 543
user interface for, 527–537
template creation, 528–529
URL pattern creation, 529–533
view function creation, 533–537
views, 551–553
generic views, 552–553
semi-generic views, 551
vs. App Engine, 628–630
See also TweetApprover
Django’s Database API, 289
django-admin.py startproject command, 566
django-admin.py utility, 502, 505
Django-nonrel, 498
App Engine and, 617
resources, 597
.dmg file, for App Engine SDK, 620
document object model (DOM) tree-structure, 725
document stores, 310
documentation strings (docstrings), 518
testing, 554
DocXMLRPCServer module/package, 434, 733, 740
do_GET() method, 430, 432
do_HEAD() method, 432
dollar sign ($), for matching end of string, 6, 10
doS() method, 235
do_POST() method, 432
doResults() method, 477, 478
DOS Command window
Django project creation in, 503
for installing Django, 499
dos.yaml file, blacklist section, 675
dot (.) symbol, in regex, 6, 9, 23
double-slash division operator (/), 804, 811
Download service/API, 615
download() method, 415
downloading
CSV files for importing into Excel or
Quicken, 685
e-mail, Yahoo! Mail Plus account for, 139
file from Web site, 101
Google App Engine SDK, 620
HTML, urlretrieve() for, 402
stock quotes into Excel, 338–340
downloadStatusHook function, 402
DP-API See database application programmer’s interface (DB-API)
DROP DATABASE statement (SQL), 256
DROP TABLE statement (SQL), 256
DRY principle, 530, 532, 551, 560
resources on, 591
vs. repetition, 546
DSNs (Database Source Names), 294
Durus, 289
dynamic dispatch, 329, 346
E
East Asian fonts, 464
EasyGUI module, 248
easy_install (Setuptools), for Django, 499
ECMA-262 standard, 719
e-mail, 114–146
attachment, 131
best practices in security and refactoring, 136–138
composition, 131–134
definition of message, 114
Google App Engine for receiving, 658–660
Google App Engine for sending, 656
Google Gmail service, 144–146
handler for inbound, 659
IMAP, 121–122
Python and, 128
instructing Django to send, 567
multipart alternative messages, 133
parsing, 134
POP, 121–122
interactive example, 123–124
methods, 124–125
Python and, 122
Python modules, 146–147
receiving, 121
sending, 116–117
sending, as task, 666–668
system components and protocols, 115–116
Web-based SaaS cloud computing, 135
Yahoo! Mail, 138–144
Yahoo! Mail Plus account for downloading, 139
See also Outlook
e-mail addresses, regex for, 24–26
Email API, 614
e-mail module/package, 131, 147
e-mail.message_from_string() function, 134
e-mail.mime.multipart.MIMEMultipart class, 133
e-mail.mime.text.MIMEText class, 133
e-mail-examples.py script, 132–134
email module/package, 131, 147
end of string, matching from, 6, 10
Empty exception (Queue/queue module), 202
empty() method (queue object), 203
encode() function, 773
encode() method, 464
encode() method (string), 778
encoding() method (file object), 787
endswith() function, 773
endswith() method (string), 778
Environment setting, for Django database, 510
Entry widget, 220
enumerate() function (threading module), 179
environment variables, 481
Django project shell command setup of, 515
Django project shell command setup of, 515
wsGI.*, 483
EnvironmentError, 788
EOFError, 72, 788
__eq__() method, 792
eric module, 249
Error exception (DB-API), 263
error exception (socket module), 77
error page, for Advcgi script, 477
error submodule, 400
errorhandler() method (DB-API Connection object), 264
escaping characters, in regex, 9
ESMTP, 116
estock.pyw script, 338–340
/etc/services file, 59
Event object (threading module), 170
event-based processors for XML, 725
event-driven applications, 80
event-driven processing, 218
events, 217
Excel
COM programming with, 328–330, 338–340
downloading CSV files for importing into, 685
excel.pyw script, 328–330
Exception, 788
exceptions, 788–790
DB-API, 263
in Python 3, 816–817
Python 3 changes, 801–802
for socket module, 77
syntax for handling in database adapters, 280
exc_info, and start_response(), 482
execute() method (DB-API Cursor object), 265
execute*() method (DB-API Cursor object), 266
executemany() function, 287
executemany() method (DB-API Cursor object), 265
execution rates, for task queues, 663
executor.map(), 208
executor.submit(), 208
exit() function (thread module), 165
extending threads, 161
expandtabs() method (string), 773, 778
Expat streaming parser, 725
expiration date of cookies, 467
exponentiation (**), 771
extend() method (list), 773, 781
extension notations, for regex, 16, 31–34
extensions
basics, 365
creating boilerplate wrapper for, 370–377
C application code, 368–370
compilation, 377–379
creating on different platforms, 365–366
disadvantages, 367–368
Global Interpreter Lock and, 384
importing, 379
reasons for, 366–367
reference counting and, 382–383
testing, 379–382
threading and, 384
vs. embedding, 387
external processes, as CGI alternative, 480
Exttest2.c C library, 380–382
ExtJS, 495
F
fac() function, 368–370
Facebook, 690
scalability issues, 310
factorial function, thread for, 180–182
fake views, 533
family attribute, for socket object, 64
FastCGI, 480
fasterBS() function, 421, 422
federated authentication, 653
fetch() method (IMAP4 object), 129, 130
fetch*() method (DB-API Cursor object), 266
fetchall() method (DB-API Cursor object), 266, 288
fetching database rows, 255
fetchone() method (DB-API Cursor object), 266
Fibonacci function, 180–182
fields, multivalued in CGI, 467
FieldStorage class (cgi module), 445
instance, 451
file input type, 466
file objects, methods and data attributes, 786–787
file servers, 55
File Transfer Protocol (FTP), 96–98
client example, 100–102
interactive example, 100
miscellaneous notes, 103–104
file() method (file object), 786
file() method (socket object), 64
file() method (urlopen object), 401
file-oriented socket methods, 63
files, 254
uploading, 478
Files service/API, 615
fill parameter, for packer, 224
filter() function, 804
filter() method, 297
filter_by() method, 297
filters, in Django variable tags, 528
find() method (string), 778
findall() function, 33–34
findall() function/method, 17, 27
findAll() method, 421
finditer() function, 17, 28, 33–34
find_top_posts() function, 758
find_user() function, 758
finish() method, 299, 307
Firebird (InterBase), 317
firewalls, 394
first() method, 298
fixtures, 513
flags
for specialized regex compilation, 19
in regex, 8, 18
Flask framework, App Engine and, 617
Flask, App Engine and, 676
float type, division and, 810
float() function, 771
__float__() method, 793
FloatingPointError, 788
floor division (/), 772, 803, 804, 810, 811
__floordiv__() method, 792
flush() method (file object), 786
Foo, Michael, Python Cookbook, 407
Forier, Jeff, Python Web Development with Django, 496
foreground color of button, argument for, 227
forex() function, 736
ForgetSQL, 289
ForkingMixIn class (SocketServer module), 79
ForkingTCPServer class (SocketServer module), 79
ForkingUDPServer class (SocketServer module), 79
form variable, 451
format parameter style, for database parameters, 261
format() function, 773
format() method (string), 778
formatter module/package, 413
formatter object, 415
FormHandler class, 666
forms
CGI specifications on encodings, 466
classes to define, 559
"hidden" variable in, 454
hidden variable in, 467
in Django, 546–550
defining, 590
forward proxies, 394
Frame class, 241
Frame object, 224
Frame widget, 220, 233
Friedl, Jeffrey E.F., Mastering Regular Expressions, 48
friendsA.py script, 450
friendsB.py script, 453–456
friendsC.py script, 457–462
friendsC3.py script, 462–463
from module import *, 702
fromfd() function, 778
from-import module, 42
fromkeys() method (dictionary), 782
frozenset() function (set types), 784
FTP (File Transfer Protocol)
creating client, 98
support for, 399
ftplib module, 98, 148, 400
ftplib.FTP class
instantiating, 98
methods, 99–100
Full exception (Queue/queue module), 202
full() method (queue object), 203
full-stack systems, 494
function attributes (DB-API), 261–262
functions
PFAs for, 226
standard, 769
vs. methods, 19
functools module, reduced() moved to in Python 3, 813
functools.partial() method, 229
future_builtins module, 814
FutureWarning, 790
FXPy module, 249
G
Gadfly, 275, 286, 316
database, 258
GAE Framework, App Engine and, 617
Gage, John, 608
gaierror exception, for socket module, 77
__ge__() method, 792
Google App Engine development servers, 428
Google App Engine Oil (GAEO), 617
Google App Engine SDK, 613
downloading and installing, 620
Google Cloud SQL, 498
Google Gmail service, 135, 144–146
Google News server, connection to, 732
Google Web crawlers, 418
Google+ platform, 690, 748–759
basics, 748
chatter score for posts, 757
Python and, 749
social media analysis tool, 750–759
Google+ Ripples, 758
goognewsrss.py script, 730–733, 821
Gopher, support for, 399
gopherlib module, 400
GQL, 638
greediness, 13, 46
Grid (geometry manager), 219
Groovy, 610
group() method, 18, 20, 25–26, 106
group() method (NNTP objects), 107
groupdict() method, 18
groups in regex, parentheses for, 14–15, 45
groups() method, 18, 20, 25–26
__gt__() method, 792
GTK, importing, 243
GTKapp class, 243
guest downloads with FTP, 96
GUI programming, 216
basics, 217–219
event-driven processing, 218
gui managers, 218
default arguments, 221
FTP client, 103
related modules, 247–250
Swing example, 745–748
toolkit alternatives, 236–246
GTK+ and PyGTK, 242–244
PMW (Python MegaWidgets), 239
Tile/Tk, 244–246
Tix (Tk Interface eXtensions), 238
wxWidgets and wxPython, 240–242
GUI scripts
Button widget, 222
Label and Button widgets, 223
Label widget, 221–222
Label, Button and Scale widgets, 224–225

H
hacking, 394
Hammond, Mark, 326
handle() method, 81
handler class, 406
handlers, 430
for inbound e-mail, 659
for Google App Engine configuration, 623
handles, for urlopen() function, 400
handle_starttag() method, 423
hardware client/server architecture, 55
Harr, Lee, Python Cookbook, 407
hash symbol (#)
for Django comments, 518
for regex comment, 32
__hash__() method, 794
has_key() method (dictionary), 782
head() method (NNTP object), 107
headers, extracting from newsgroup articles, 112
heavyweight process, 159
"Hello World" application in Google App Engine, 620–626
morphed to blog, 631–647
in Django, 507
in Java, 746
print statement vs. print() function, 820
in Python, 747
he1o() method (SMTP object), 119
error exception, for socket module, 77
hex() function, 771, 773
__hex__() method, 794
hexadecimal format, 810
hexadecimal ordinal equivalents, of
disallowed characters, 403
hidden variable in form, 454, 467
hops, 115
HOST setting, for Django database, 510
HOST variable, 67
for timestamp client, 70
host-port pairs for socket addresses, 59
howmany variable (Python), 451
HR variable for Google+ program, 754
HSC tool, 462
.htaccess file, 405
HTML (HyperText Markup Language), 401, 442
3rd-party tools for generating, 462
parsec tree format, 423
separating HTTP headers from, 451
separating HTTP MIME header from, 454
urlretrieve() to download, 402
HTML forms in Django for user input, 542
ModelForm to generate, 548
processing ModellForm data, 549
html5lib package, 185, 418, 423, 489
httplibdef module/package, 433, 488
htmlgen package, 435, 462
httplib module/package, 413, 433, 488
HTMLParser class, 415, 418
HTMLparser module/package, 185, 433, 488
htmlparser() function, 422
httplib() function, 78
htons() function, 78
htpasswd command, 405
HTTP (HyperText Transfer Protocol), 96, 392
separating headers from HTML, 451
separating MIME header from HTML body, 454
support for, 399
XML-RPC and, 733
http.cookies module, 476
http.server class, 430, 489
http.server module, 435, 447
HTTP_COOKIE environment variable, 468
httplib module, 148, 400, 404, 414, 433, 489
httplib2 library, 571
HTTPServer server class, 429
hybrid cloud, 606
hypertext, 442
Hyves social network, 89

I
IaaS (Infrastructure-as-a-Service), 607
ident attribute (Thread object), 172
if statement, 819
IIS (Internet Information Server), 428
Images API, 615, 662
IMAP (Internet Message Access Protocol), 121–122
interactive example, 128
Python and, 128
Yahoo! Mail example, 142–144
IMAP4 class, 128
IMAP4_SSL class, 128
IMAP4_stream class, 128
importlib module, 128, 148
importlib.IMAP4 class, methods, 129–131
import statement, 532
ImportError exception, 16, 789
importing
csv module, 716
extensions, 379
ordering guidelines for, 421, 561, 735
PyGTK, GTK, and Pango, 243
Tkinter module, 215
to create compatible code for Python 2.x and 3.x, 820–821
inbound e-mail, handler for, 659
InboundMailHandler class, 659
include Python header file, in boilerplate code, 371
include() directive, in Django project, 508
include() function, 530
IndentationError, 789
index() function, 773
index() method (list), 781
index() method (string), 778
IndexError, 789
inet_hton() function, 78
inet_ntoa() function, 78
inet_ntop() function, 78
inet_pton() function, 78
info() method (urlopen object), 401
Infrastructure-as-a-Service (IaaS), 607
ingmod, 318
Ingres, 318
Ingres DBI, 318
__init__ method (Thread object), 172
__init__.py file in Django project, 504, 508
__init__() method, 176, 414, 791
initModule() module initializer function, 376
input() function, 280
INSERT INTO statement (MySQL), 271
INSERT INTO statement (SQL), 257
insert() function, 287, 773
insert() method (list), 781
insert() method, for MongoDB collection, 314
inserting database rows, 255
INSTALLED_APPS variable, 571
installing
Django, 499–501
prerequisites, 497–499
Google App Engine SDK, 620
Tkinter, 215
Twython library, 571–572
instance attributes, local variable for, 703
instant messages
Google App Engine for sending, 660
receiving, 661
int type, 802, 809
int() function, 771
__int__() method, 793
integers
cverting to date, 43
Python 3 changes, 802–804
Python 3 migration, 809–812
IntegrityError exception (DB-API), 263
InterfaceError exception (DB-API), 263
“Internal Server Error” messages, 446
InternalError exception (DB-API), 263
International Standard Book Number (ISBN), 184
Internet, 392–395
protocols, related modules, 148
See also cloud computing
Internet addresses, 59
formatting, 121
Internet clients, 95
and servers location, 394–395
See also e-mail
Internet Protocol (IP), 60
Internet Server Application Programming Interface (ISAPI), 479
interpreted languages, vs. compiled, 367
intersection() function (set types), 784
intersection_update() function (set types), 785
__invert__() method, 793
io.BytesIO class, 731
int() method, 63
IOError, 789
IP (Internet Protocol), 60
IP address, binding, 62
IPv6 TCP client, creating, 71
IPython, 515
- starting and using commands, 516
IronPython, 325
is not operator, 770
is operator, 770
isAlive method (Thread object), 172
is_alive() method (Thread object), 172
isalnum() method (string), 773, 779
isalpha() method (string), 773, 779
isspace() method (string), 773, 779
issubset() function (set types), 784
issuperset() function (set types), 784
istitle() method (string), 773, 779
isupper() method (string), 774, 779
items() function, 804
items() method (dictionary), 783
__iter__() method, 794
__iter__() method (DB-API Cursor object), 266
iter() method (dictionary), 783
iterables, Python 3 changes, 804
itertools.izip() function, 731, 820
Java, 610
“Hello World” application, 746
Java and, 744
- vs. Python, 747
JavaScript, 610
JavaScript Object Notation (JSON), 719–724
join() function, 774
join() method, 298
join() method (queue object), 203
join() method (string), 779
join() method (thread object), 172, 174, 186
JOINs, Web resources on, 298
JPEG (Joint Photographic Experts Group), 401
jQuery, 495
K
Kantor, Brian, 105
Kay framework, App Engine and, 617
key for cache, 649
KeyboardInterrupt, 72, 788, 789
KeyError, 789
keys() function, 804
keys() method (dictionary), 783
keys-only counting, 643
key-value pairs
- in CGI, 445
url encode() encoding of, 403
key-value stores, 310
keyword module, 819
keywords, 768
KInterbasDB, 317
Klassa, John, 341
Kleene Closure, 12
Kuchling, Andrew, 799
Label widget, 220, 238, 241
- Button and Scale widgets with, 224–225
Button widget with, 223
LabelFrame widget, 220, 247
LAN (Local Area Network), 394
language runtimes of App Engine, 610
Lapsley, Phil, 105
last() method (NNTP object), 107
lastrowid attribute (DB-API Cursor object), 265
Launcher, 628
__le__() method, 792
len() function, 774
len() function (set types), 783
one() method (set types), 791, 794
libevent, 89
LibreOffice, 357
LibreOffice Calc, 685
LifoQueue class, 202
lightHTTPD, 428, 446, 494
lightweight processes, 159
limit() method, 297
line termination characters
- for Word documents, 331
links, parsing, 418
Linux
- package manager for Django install, 501
- Zip file for App Engine SDK, 620
list type built-in methods, 781–782
list() function, 774
list() method (POP3 object), 125
Listbox bind() method, 233
Listbox widget, 220
listdir() method (thread object), 720
load() method, 62, 67
list_tweet() method, 589
list_tweets() method, 587
literals
- binary and octal, 804
- bytes, 815
K
Kantor, Brian, 105
Kay framework, App Engine and, 617
key for cache, 649
KeyboardInterrupt, 72, 788, 789
KeyError, 789
keys() function, 804
keys() method (dictionary), 783
keys-only counting, 643
key-value pairs
- in CGI, 445
url encode() encoding of, 403
key-value stores, 310
keyword module, 819
keywords, 768
KInterbasDB, 317
Klassa, John, 341
Kleene Closure, 12
Kuchling, Andrew, 799
Label widget, 220, 238, 241
- Button and Scale widgets with, 224–225
Button widget with, 223
LabelFrame widget, 220, 247
LAN (Local Area Network), 394
language runtimes of App Engine, 610
Lapsley, Phil, 105
last() method (NNTP object), 107
lastrowid attribute (DB-API Cursor object), 265
Launcher, 628
__le__() method, 792
len() function, 774
len() function (set types), 783
one() method (set types), 791, 794
libevent, 89
LibreOffice, 357
LibreOffice Calc, 685
LifoQueue class, 202
lightHTTPD, 428, 446, 494
lightweight processes, 159
limit() method, 297
line termination characters
- for Word documents, 331
links, parsing, 418
Linux
- package manager for Django install, 501
- Zip file for App Engine SDK, 620
list type built-in methods, 781–782
list() function, 774
list() method (POP3 object), 125
Listbox bind() method, 233
Listbox widget, 220
listdir() method (thread object), 720
load() method, 62, 67
list_tweet() method, 589
list_tweets() method, 587
literals
- binary and octal, 804
- bytes, 815
migration to Python 3, 807–822
  built-in functions, 813
migration to Python (continued)
  exceptions, 816–817
  integers and, 809–812
  object-oriented programming, 814
print statement vs. print() function, 812
reduced() moved to functools module, 813
strings, 815
migration tools for Python 3, 805
MIME (Mail Interchange Message Extension), 131
MIME (Multipurpose Internet Mail Extension), 131
headers, 401
mimetypes module, 147
MimeWriter module, 147
MiniFieldStorage, 445
MIMEWriter module, 147
min() function, 774
__mul__() method, 792, 794
__neg__() method, 793
mode() method (file object), 787
model forms, in Django, 547
ModelForm
data processing, 549
HTML form generation with, 548
models
classes to define, 559
in Django, setting default ordering, 540
modelS.py file, 538
for Django app, 508
model-template view (MTV) pattern, 514
model-view-controller (MVC) pattern, 514
module initializer function, 376
modules, order for importing, 421
Modules/Setup file, Tkinter and, 215
mod_wsgi Apache module, Django and, 497
MongoDB, 310, 318, 498
mouse move event, 218
msg.get_payload() method, 134
msg.walk() method, 134
.msi file, for App Engine SDK, 620
mtfacfib.py script, 180–182
mtsleepA.py script, 165
mtsleepB.py script, 167–169, 173
mtsleepC.py script, 173
mtsleepD.py script, 175
mtsleepE.py script, 177–178
mtsleepF.py script, 191, 194–196
mbd3 module, 164–169
threads and processes, 158–159
multivalued fields in CGI, 467
mutex module, 209
MVC engine, 617
myhttpd.py script, 430
MySQL Connector/Python, 280, 316
MySQL for Django database, 510–511
MySQLdb package, 280, 286, 316
myThread.py script, 178
N
name attribute (Thread object), 172
name identifier, for saving matches, 32
NAME setting, for Django database, 510
name() method (file object), 787
named matches, 20
named parameter style, for database parameters, 261
NameError, 789
names
  for Django projects, 502
  for Google App Engine application, 631
  strategy for Python 2 to Python 3, 408
namespaces for App Engine, resources, 677
Namespaces service/API, 616
NDB (new database) service/API, 616
__ne__() method, 792
__neg__() method, 793
negation
  in regex, 12
  of character set matches, 14
negative lookahead assertion, 8, 33
.NET, 325
Netscape Server Application Programming Interface (NSAPI), 479
Netscape, cookies specification, 468
Network News Transfer Protocol (NNTP)
  additional resources, 114
  basics, 105
  client program example, 108–114
  interactive example, 108
  Python and, 105
  network programming
    for client/server architecture, 56–57
    related modules, 88–89
  socket module for, 61–62
  sockets, 58–61
  TCP server creation, 64–68
  Twisted framework, 84–87
networks, location components, 397
__new__() method, 791
NEWWLINE characters, to separate HTTP header from HTML, 451
newlines() method (file object), 787
newsgroups, 104–114
new-style classes, 814
next() method (DB-API Cursor object), 266
next() method (file object), 786
next() method (NNTP object), 107
nextset() method (DB-API Cursor object), 266
nlst() method (FTP objects), 99
NNTP. See Network News Transfer Protocol (NNTP)
nntplib class, 105
nntplib module, 148
nntplib.NNTP class, 105
methods, 107
non-ASCII characters, \u escape for, 464
non-blocking sockets, 65
nondeterministic activity, 157
non-relational databases, 309–315, 498
Django and, 618
MongoDB, 310
PyMongo, 311–315
Web resources, 319
non-validating, Expat parser as, 725
__nonzero__() method, 791
noop() method (IMAP4 object), 130
NoSQL, 310
not operator, 770
NotImplementedError, 789
NotSupportedError exception (DB-API), 263
now_int() function, 736
now_str() function, 736
ntohl() function, 78
ntohs() function, 78
NULL objects, 267
check for, 383
NUMBER type object (DB-API), 267
numeric conversion, 793
numeric parameter style, for database parameters, 261
numeric type operators, 770–772

O
OAuth, 494, 569
credentials for Twitter’s public API, 567 resources, 597, 678
Twitter and, 694
oauth2 library, 571
object comparisons, 770
object-level caching, 651
object-oriented programming, 814
object-relational managers (ORMs), 289–309
employee role database example, 291–309
SQLAlchemy for, 291–304
SQLObject for, 304–309
explicit “classical” access, 301–304
setup and installation, 290–291
Object-Relational Mapper (ORM)
App Engine and, 618
objects
comparison, 792
creating and caching, 329
oct() function, 771, 774
__oct__() method, 794
octal literals, 804
cotothorpe. See hash symbol (#)
offset() method, 298
oklook.pyw script, 335–337
one() method, 298
onethr.py script, 162–163
OpenDocument text (ODT) format, 357
OpenID service/API, 616, 653
OpenOffice, 356
OperationalError exception (DB-API), 263
operators, 769
numeric type, 770–772
sequence type, 772–776
summary, 795–797
OR
logical, 9
logical, brackets for, 11
or operator, 770
__*or__() method, 793
Oracle, 317, 498
Oracle Open Office, 357
ord() function, 771, 774
order_by() method, 297, 538
os module, 414
importing, 232
os.makedirs() function, 414
os.popen() command, 37
os.spawnv() function, 346
OSError, 789
Outlook
address book protection in, 336
COM programming with, 334–337
outlook_edit.pyw script, 341–347
output() function, 421
OverflowError exception, 788, 802
OverflowWarning, 790
owned reference, 382
P
PaaS (Platform-as-a-Service), 607
package manager, for Django install, 500
packer, 224
fill parameter, 224
Packer (geometry manager), 218
page views, persistent state across multiple, 467
PanedWindow widget, 220, 247
Panel widget, 241
Pango, importing, 243
parallel processing, 157
paramstyle attribute (DB-API), 260, 261
parent widget, 217
parentheses, for regex groups, 14–15
parse() function, 423
parse_links.py script, 419–424
parse_links() method, 415
parsing
data string, csv module for, 686
e-mail, 134
parsing (continued)
tree format for HTML documents, 423
Web content, 418–424
part.get_content_type() method, 134
Partial Function Application (PFA), 226–229
partition() function, 774
passwords
for anonymous FTP, 97
See also login
PATH environment variable
django-admin.py in, 502
easy_install and, 500
pattern-matching, 4
patterns() function, 531
PC COM client programming, 325
P_DETACH flag, 346
PDO, 289
PendingDeprecationWarning, 790
PEP 333, 496
PEP 3333, 487
PEP 444, 487
percent sign (%)
for hexadecimal ordinal equivalents, 403
for modulo, 772
in string format operator (%)
conversion symbols, 776
performance, interpreted vs. compiled languages, 367
period (.) symbol, in regex, 6, 9, 23
permission flags, in Django, 579
@permission_required decorator, 589
permissions, to access method, 589
persistence, in state across multiple page views, 467
persistent storage, 254, 488
databases and, 255
scalability issues, 310
pfaGUI2.py script, 227–229
PHP, 610
Pinax platform, 501
resources, 597
pip, for Django install, 499
pipe symbol (|)
for Django variable tag filters, 528
in regex, 9
Pipeline, 616
resources, 678
Placer (geometry manager), 218
plaintext
avoiding for login, 136, 142
See also comma-separated values (CSV)
planning for transition to Python 3, 817
platform.python_version() function, 142
Platform-as-a-Service (PaaS), 607
plus sign (+)
for encoding, 403
in regex, 6, 12–13
PlusService class, 755
plus.top_posts.py script, 752–759
sample execution, 750
PMW (Python MegaWidgets), 239, 248
PNG (Portable Network Graphics), 401
P_NOWAIT flag, 346
pop() function, 774
pop() function (set types), 785
pop() method (dictionary), 783
pop() method (list), 782
pop11b class, 122
pop11b module, 148
pop11b.POP3 class, 122
methods, 124–125
pop11b.POP3SSL class, 123
PoPy, 272
PORT setting, for Django database, 510
PORT variable, for timestamp client, 70
port, for Web server, 447
porting Python version 2 to version 3, 408
ports, 387
for Django development server, 506
for SMTP, 118
reserved numbers, 59
well-known numbers, 59
__pos__() method, 793
Positive Closure, 12
positive lookahead assertion, 8, 33
POSIX systems
http.server module on, 447
POSIX-compliant threads, 161
POST handler, for blog posts, 634
Post Office Protocol (POP), 121–122
example, 126–128
interactive example, 123–124
pop11b.POP3 class methods, 124–125
Python and, 122
Yahoo! Mail example, 142–144
POST request method, for HTTP requests, 400
post() method (FormHandler), 667
post() method (NNTP object), 107
Postel, Jonathan, 96, 116
PostgreSQL, 272–274, 317, 498
postings on newsgroups, 104
post-processing, 483
post_tweet.html template, 586
post_tweet() method, 584
pound sign (#) Seach character (#)
.P_OVERLAY flag, 346
pow() function, 736, 771
__pow__() method, 793
PowerPoint, COM programming with,
332–334, 347–356
ppoint.pyw script, 333
print.print() function, 732
precompiled code objects, performance, 19
preprocessing, 485
prettyprinting, 732
print servers, 55
print statement, 196
proxy for, 716
Python 2 vs. 3 versions, 279
vs. print() function, 799–800, 812, 819
print() function, 38
PriorityQueue class, 202
private cloud, 606
process() function, 424, 455
processes
synchronization, 190
threads and, 158–159
prodcons.py script, 204–206
producer() function, 200
production servers, 446
Apache as, 498
profiling with Appstats, 670
ProgrammingError exception (DB-API), 263
programs, vs. processes, 158
projects
file structure for TweetApprover, 565–571
in Django, 501
basic files, 504
creating, 502–505
proprietary source code, extensions to protect, 367
Prospective Search service/API, 616
proto attribute, for socket object, 64
proxy servers, 394
Psycho, 386
psycopg, 272, 317
Connection object setup code, 273
output, 273
pthread, 161
public cloud, 606
publish_tweet() method, 592
pull queues, 663, 666
purge() function/method, 18, 19
push queues, 663, 666
put() method (queue object), 203
put_nowait() method (queue object), 203
P_WAIT flag, 346
pwd() method (FTP objects), 99
_py_BuiltinValue() function, 372
PyArg_Parse*() functions, 372
PyArg_ParseTuple() function, 374
PyCon conference Web site, 425
Py_DECREF() function, 383
PyDO/PyDO2, 289, 318
pyFLTK module, 249
pyformat parameter style, for database parameters, 261
PyGSDL, 272, 317
Connection object setup code, 273
output, 273
PyGTK, 242–244
PyGTK module, 248
importing, 243
PyGUI module, 249
Py_INCREF() function, 383
Py_InitModule() function, 376
PyKDE module, 249
Pylons, 494, 495
resources, 597
PyMethodDef ModuleMethods[] array, 376
PyMongo, 311–315, 318
PyMongo3, 318
pymssql, 317
PyObject* Module_func() wrappers, 371–376
PyOpenGL module, 249
PyPgSQL, 272, 317
Connection object setup code, 273
output, 273
PyPy, 386
PyQt module, 249
PyQtGPL module, 249
Pyramid, 495
resources, 597
Pyramid framework, App Engine and, 617
Pyrex, 385
pysqlite, 274, 317
Python, 610
and App Engine, 609
converting data between C/C++ and, 372
“Hello World” application with Swing, 747
obtaining release number as string, 142
vs. Java, 747
Web servers with, 446
writing code compatible with versions 2.x and 3.x, 818–822
importing for, 820–821
Python 2.6+, 805
Python 3 changes, 798–806, 807–809
class type, 801
division, 803–804
exceptions, 801–802
integers, 802–804
migration tools, 805
print statement vs. print() function, 799–800
reasons for, 799
See also migration to Python 3
Python application shell in Django, 514–517, 407
Python dict
collection to JSON, 722–724
collection to XML, 725–729
Python Extensions for Windows, 327
Python interpreter, 655
compilation, enabled threads and, 162
Python MegaWidgets (PMW), 239
Python objects, wrapping in object to delegate lookup, 705
Python types, vs. JSON types, 721
Python Virtual Machine (PVM), 160–163
    extensions and, 384
Python/ceval.c file, 161
PythonCard module, 248
    .pyw extension, 237, 327
QLime, 289
qmark parameter style, for database parameters, 261
    -qnew switch, 811
qlsize() method (Queue object), 203
Quercus, 610
queries, 255
    change to reverse output order, 537–540
    in Google App Engine, documentation, 640
    speed of, caching and, 647
Query methods, Web resources on, 298
QuerySet, 537
question mark (?), in regex, 6, 12–13, 24, 47
Queue data structure, 158
Queue module, 163
queue.yaml file, 665
Queue/queue module, 202–206, 209
queues for tasks, 663
Quickzen, downloading CSV files for importing into, 685
quit Button, 238, 241
quit() method (FTP objects), 99
quit() method (NNTP object), 107
quit() method (SMTP object), 118, 119
quopri module, 147
quote() function, 404
quote*() functions, 402
quote_plus() function, 404
race conditions, 159, 190
radio buttons
    default, 454
    string to build list, 454
Radiobutton widget, 220
raising exceptions
    in Python 3, 817
    Python 3 changes, 802
randName() function, 287
random data, script to generate, 41
random.choice() function, 43
random.randint() method, 205
random.randrange() function, 43
range() function, 804
ranges (-) in regex, 12
raw strings, 27, 34, 36, 512
    note on use, 35
raw_input() function, 280, 774
rcp command (Unix), 96
RDBMS (relational database management system), 255
re module, 3, 16–35
    character classes creation, 24
    core functions and methods, 17–18
    match objects, 20
    match() function/method, 20–21
    matching any single character, 23
    matching multiple strings, 22
    search() function, 21–22
re.compile() function, 183, 189
re.IGNORECASE, 31
re.L/LOCALE flag, 34
re.MULTILINE, 31
re.DOTALL, 31
re.split() function, 39
re.UNICODE flag, 34
re.VERBOSE flag, 32
read() method (file object), 786
read() method (urlopen object), 401
reader() function, 205
readinto() method (file object), 786
readline() method, 81
readline() method (file object), 786
readline() method (urlopen object), 401
readlines() method (file object), 786
readlines() method (urlopen object), 401
readQ() function, 205
realms, 405
receiving e-mail, 121
    Google App Engine for, 658–660
recording events from application activity, 671
    records in database, autogenerating for testing, 538
recv() method, 63
recvfrom() method, 63
recvfrom_into() method, 63
redirect_to() method, 63
redirect_to() generic view, 552
reduced() function, Python 3 move to
    functools module, 813
refactoring, 136
reference counting, extensions and, 382–383
reference server, WSGI, 483
ReferenceError, 789
repl() function, 200
regex module, 16
regex. See regular expressions
registering password for login, 405
regrub module, 16
regular expressions, 3, 4
    alternation (|) operation, 9
    characters, escaping to include, 9
    comments, 8, 16
    compilation decision, 19
    conditional compilation, 34
    creating first, 5
Index 843

escaping characters to include, 23
examples, 36–41
in-depth, 41–48
exercise answers, 763
extension notations, 16, 31–34
for e-mail addresses, 24–26
grouping parts without saving, 32
groups, 14–15
matching from start or end of strings or
word boundaries, 10, 26–27
for obtaining current book ranking, 182
ranges (-) and negation (^), 12
repetition, 12–13, 24–26
special characters for character sets, 14
special symbols and characters, 6–16
splitting string based on, 30–31
Unicode string vs. ASCII/bytes string, 188
See also re module
relational databases, available interfaces,
269–270
release() method (lock object), 165, 190, 193
remote API shell, 654
remote procedure calls (RPCs), XML and,
733–738
remove() function, 774
remove() function (set types), 785
remove() method (list), 782
ren command, 188
rename() method (FTP objects), 99
render_to_response() method, 534, 536, 561
repetition, in regex, 12–13
replace() function, 774
replace() method (string), 780
replacing, searching and, 29
replenishment rates, for task queues, 663
ReplyThread, 158
repr() function, 769, 774
__repr__() method, 791
request context instance, 544
Request for Comments (RFCs), for cookies, 468
request in CGI, 444
RequestProcessor, 158
reserved port numbers, 59
reserved words, 768
reßtment variable, 451
resize() function, 225
response in CGI, 444
response submodule, 400
ResultsWrapper class, 705
for Twitter, 704
testing, 706
retasklist.py script, 40
retr() method (POP3 object), 125, 127
retrbinary() method (FTP objects), 99, 102
Retriever class, 414
retrlines() method (FTP objects), 99
retry parameters, for task queues, 663
reverse() function, 368–370, 375, 774
reverse() method (list), 782
reverse-chronological order
for blog, 537
query change for, 537–540
review_tweet() method, 587, 590
rewho.py script, 38
Reynolds, Joyce, 96
rfind() function, 774
 rfindex() method (string), 780
Rhino, 610
rich shells for Django, 515
rindex() function, 774
rindex() method (string), 780
rjust() function, 774
rjust() method (string), 780
Rlock object (threading module), 170
context manager, 196
rd() method (FTP objects), 99
road signs, PFA GUI application, 227–229
robotparser module, 400, 433, 489
rollback() method (DB-API Connection ob-
ject), 264
root window, 217
round() function, 771
rowcount attribute (DB-API Cursor object), 265
ROWID type object (DB-API), 267
rownumber attribute (DB-API Cursor object),
266
rows in database table, 255
inserting, 257
insertion, update, and deletion, 297
rpartition() function, 774
RPython, 387
*sinrshift__() method, 793
rsplit() function, 775
rstrip() function, 777
rstrip() method, 113, 780
rsync command (Unix), 96
Ruby, 610
run() method (Thread object), 172
run_bare_wsgi_app() function, 484
RuntimeError, 789
RuntimeWarning, 790
run_wsgi_app() function, 624
run_wsgi_app() method, 483
S
SaaS (Software-as-a-Service), 135, 607
Salesforce, 608
sandbox, 611
restrictions, 612–616
sapdb, 317
saving
matches from regex, 32
subgroup from regex, 7
SAX (Simple API for XML), 725
Scala, 610
scalability issues for storage, 310
Scale widget, 220
Label and Button widget with, 224–225
scanf() function, 280, 285
scp command (Unix), 96
scripts, standalone, 102
Scrollbar widget, 220, 233
Scrollbar.config() method, 233
sdb.dbapi, 317
search command (Twitter API), 695
search on Twitter, Tweepy library for, 692
Search service/API, 615
search() function/method, 4, 17, 21–22, 26–27
search() method (IMAP4 object), 130
searching
and replacing, 29
subgroups from, 27
vs. matching, 4, 21–22, 46–48
secret.pyc file, 136
Secure Socket Layer (SSL), 393, 404
security
e-mail and, 136
for Outlook address book, 337
seek() method (file object), 787
SELECT * FROM statement (MySQL), 272
select module, 88
select() function, 88
select() method (IMAP4 object), 130
self.api, 703
self.error variable, 477
self.service.people() function, 756
Semaphore class, 199
Semaphore object (threading module), 170
displaying, 192
context manager, 196
semi-generic views in Django, 551
send() method, 63
send11() method, 63
send_authorization_email() method, 591
sendData() method, 86
send_group_email() function, 667
sending e-mail, 116–117
Google App Engine for, 656
sendmail() method (SMTP object), 118, 119
sendmsg() method, 133
SendNewsletter class, 667
send_rejection_email() method, 591
send_review_email() method, 585
sendto() method, 63
sequence type operators, 772–776
sequential program, 157
server integration, as CGI alternative, 479
server.py module, 429
server.register_function() function, 736
servers, 54, 56
for UDP, 76
implementing exit scheme, 66, 72
as Internet providers, 95
location on Internet, 394–395
socket methods, 62
TCP
creating, 64–68
creating Twisted Reactor, 84–85
executing, 71–73
executing Twisted, 87
SocketServer execution, 83
timestamp from, 73
WSGI, 482
session management, 488
set types, operators and functions, 783–785
set() function, 783
_set__() method, 792
_setattr__() method, 792
setblocking() method, 63
“Set-Cookie” header, 468
setCPPCookies() method, 476, 477, 478
setDaemon() method (Thread object), 172
set_default() method (dictionary), 785
setDirAndGo() method, 235
setinputsizes() method (DB-API Cursor object), 266
_setitem__() method, 794, 795
setname() method (Thread object), 172
setoutputsize() method (DB-API Cursor object), 266
setprofile() function (threading module), 179
sets
displaying, 192
for names of running threads, 192
__setslice__() method, 794
socket() method, 63
settimeout() method, 63
settings file, for TweetApprover, 566–571
settings.py file
in Django project, 504
settings.py file in Django project
INSTALLED_APPS tuple in, 509
settrace() function (threading module), 179
setup.py script, creating, 377–378
SGML (Standard Generalized Markup Language), 724
sgmllib module/package, 418, 433, 489
sharded counter, 643
sharding, 498
Short Message Service (SMS), 691
shutdown() method, 63
Simple API for XML (SAX), 725
Simple Mail Transfer Protocol (SMTP), 116
authentication, 118

844 Index
example, 126–128
interactive example, 119–120
Python and, 118
web resources, 120
Yahoo! Mail example, 142–144
Simple Storage System (S3), 607
simple85() function, 421, 422
SimpleHTTPRequestHandler class, 429, 447
SimpleHTTPServer class, 430, 489
SimpleHTTPServer module, 432, 435
  handlers in, 430
simplejson library, 571, 720
simpletree format, for HTML documents, 423
simple_wsgi_app() app, 483
simple_wsgi_app() wrapping, 485
SimpleXMLRPCServer package, 434, 733, 740
single-threaded process, 157
six package, 822
Slashdot, and traffic, 674
sleep, 159
  vs. thread locks, 167
sleep() function, 166, 181, 354
SMS (Short Message Service), 691
smtpd module, 147
smtplib class, 118
smtplib module, 148
smtplib.SMTP class, 118
  methods, 118–119
SMTP_SSL class, 118, 139
SOAP, 733
social media analysis tool, 750–759
social networking, 690
  See also Twitter
SOCK_DGRAM socket, 61
Socket, 616
  socket module, 61–62, 88, 404
  attributes, 76–78
  socket.error, 143
  socket.socket() function, 61–62, 65, 74, 77
  socketpair() function, 77
  sockets, 58–61
    addresses with host-port pairs, 59
    built-in methods, 62–64
    connection-oriented vs. connectionless, 60–61
  data attributes, 64
  for FTP, 97
  related modules, 88–89
SocketServer class
  TCP client creation, 82–83
  TCP server and socket execution, 83
  TCP server creation, 80–82
SocketServer module, 65, 79–83, 88, 210
  classes, 79
SOCK_STREAM socket, 60
softspace() method (file object), 787
software client/server architecture, 55–56
Software-as-a-Service (SaaS), 135, 607
sort() function, 775
sort() method (list), 782
sorted() function, 758
SoupStrainer class, 419, 422
spaces, plus sign (+) for encoding, 403
spam e-mail, 127
special characters
  for character sets, 14
  regular expressions with, 5, 7
  vs. ASCII symbols, 34
spider, 410
spin locks, 174
spinbox widget, 220, 247
spinbutton widget, 236
  position of labels, 244
spinctrl widget, 241, 242
split() function, 775
split() function/method, 17
split() method, 30–31
split() method (string), 780
splitlines() function, 775
splitlines() method (string), 780
spreadsheets
  closing without saving, 330
  processing data from, 328
  See also Excel
SQL, 256–257
  viewing ORM-generated, 296
SQL Server, 317
SQLAlchemy, 289, 291–304, 318, 495
  setup and install, 290
SQLite, 274–275, 317, 498, 510
  for Django database, 511–?
  loading database adapter and, 286
SQLite for Django database, ??–512
sqlite3 package, 290
sqlite3a, 317
SQLObject, 289, 304–309, 318
  setup and install, 290
SQLObject2, 318
ss1() function, 77
standalone script, 102
standalone widgets, 217
Standard Generalized Markup Language (SGML), 724
StandardError, 798
StarOffice, 357
start of string, matching, 6, 10
  _start() function, 356
start() method (Thread object), 172, 174
  start_new_thread() function, 165, 166
startproject command, 502, 504
start_response() callable, 481
startswith() function, 775
startswith() method (string), 780
starttls() method (SMTP object), 119
stat() method (NNTP object), 107
stat() method (POP3 object), 125, 127
stateless protocol
   HTTP as, 392
states, enumeration and definition, 579
static dispatch, 329
static PyObject* function, 371
stock quotes
downloading into Excel, 338–340
   Yahoo! server for, 685–689
stock.py script, 688
csv module for, 717
stockcsv.py script, 718
csv module for, 717
stop() function, 736
stock quotes
stock.csv module for, 717
stock.csv script, 718
str type, 800
str.format() method, 196, 207
str.__getslice__() method, 138
str.join() method, 137
str.startswith() method, 138
str.title() method, 460
str() function, 769, 775
   __str__() method, 192, 296, 306, 791
stream socket, 60
StreamRequestHandler class, 79, 81
string format operator (%)
   bytes objects and, 409
directives, 777
STRING type object (DB-API), 267
StringIO class, 475, 731
strings
   built-in methods, 778–781
   converting to Unicode, 189
   in Python 3, 815
   in regular expressions, Unicode vs. ASCII/
      bytes, 188
   matching, 44–45
      from start or end, 10
   multiple, 22
   obtaining Python release number as, 142
   parsing, csv module for, 686
   raw, 512
script to generate, 41–43
   searching for pattern in middle, 21
   splitting based on regex, 30–31
   “title-case formatter”, 285
   Unicode vs. ASCII, 800–801, 815
strip() function, 775
strip() method (string), 780
sub() function/method, 18, 29
   __sub__() method, 792
subclassing Thread(), 177–178
subgroup from regex
   matching saved, 7
   saving, 7
   searches, 27
subn() function/method, 29
subprocess module, 206, 209
sudo command, 500
sum() function, 771
summation function, thread for, 180–182
Sun Microsystems Java/Swing, 249
superuser
   creating, 513
   login as, 520
swapcase() function, 775
swapcase() method (string), 780
swhello.java program, 746, 747
swhello.py program, 747
SWIG (Simplified Wrapper and Interface Generator), 384
swing module, 249
Swing, GUI development and, 745–748
sybase, 317
symmetric_difference() function
   (set types), 785
symmetric_difference() function (set types), 784
syncdb command, 512, 579
   and database table creation, 518
   superuser creation, 513
   synchronization of threads, 166, 170
   synchronization primitives, 201
      shared resources and, 261
   synchronization primitives for threads,
      190–201
   context management, 196
   locking example, 190–196
   semaphore example, 197–201
SyntaxError, 789
SyntaxWarning, 790
sys module/package, 414
sys.stdout.write() function, 819
SystemError, 789
SystemExit, 161, 788
executing server and clients, 71–73
listener setup and start, 62
server creation, 64–68
SocketServer class for, 80–82
SocketServer class for client creation, 82–83
timeStamp server, 66–68
Twisted server creation, 84–85
TCP client socket (TCPClientSock), 70
TCP/IP socket, creating, 61
TCPServer class (SocketServer module), 79
tell() method (file object), 787
template module, 345
templates for blog application, 562
in Django
cross-site request forgery, 544
directory, 529
for input, 542
for interface, 528–529
for Web page, 527
for Web pages, 570
inheritance, 553
for TweetApprover
to display post status, 592
login.html, 595
pending tweet form, 595
Terms of Service (ToS)
for Google service, 731
ternary/conditional operator, 229
test-driven development (TDD) model, 528
test_home() method, 556
testing
auto-generating database records for, 538
database, 556
Django blog application code review, 557–563
extensions, 379–382
in Django, 554–557
ResultSetWrapper class, 706
user interface, 556
when porting code to Python 3, 818
test_obj_create() method, 555
tests.py file for Django app, 508
auto-generation, 554–557
text editors, for email editing in Outlook, 341
text file, converting to PowerPoint, 347–356
text font size on Label widget, 224
text in Python 3, 800
text processing, 3
comma-separated values (CSV), 715–719
JavaScript Object Notation (JSON), 719–724
related modules, 740
resources, 738
XML (eXtensible Markup Langauge), 724–738
Text widget, 221
tformat() function, 285, 288
thank_you() method, 585
threaded widget sets, 244
thread module, 161, 163, 164–169, 209
avoiding use, 164
functions and methods, 165
Thread object (threading module), 170
thread_allocate_lock() function, 169
ThreadFunc class, 176
ThreadFunc object, 176
Threading MixIn class (SocketServer module), 79
threading module, 161, 163, 169, 209
bookrank.py script, 182–189
functions, 179
synchronization primitives, 201
Thread class, 171–179
vs. thread module, 164
threading.activeCount() method, 196
threading.currentThread() method, 195
threading.enumerate() method, 195
ThreadingTCPServer class (SocketServer module), 79
ThreadingUDPServer class (SocketServer module), 79
threads, 159
alternatives, 206–209
app to spawn random number, 191
creating object instance
passing in callable class instance, 175–176
passing in function, 173–175
subclass instance, 177–178
example without, 162–163
execution of single, 160
exiting, 161
extensions and, 384
for Fibonacci, factorial, summation functions, 180–182
loops executed by single, 162–163
modules supporting, 163
processes and, 158–159
Python access to, 163
set for names of running, 192
spawning to handle client requests, 65
synchronization primitives, 190–201
context management, 196
locking example, 190–196
semaphore example, 197–201
threadsafety attribute (DB-API), 260
thtpd, 428, 446
TIDE + module, 248
Tile/Ttk module, 244–246, 248
time module, 168
time type object (DB-API), 267
time.time() function, 43, 489
time.sleep() function, 162, 232, 354
TimeFromTicks type object (DB-API), 267
timeout exception, for socket module, 77
timeout, for FTP connections, 97
Timer object (threading module), 170
timestamp
   extracting days of week from, 44
   from server, 73
Timestamp type object (DB-API), 267
timestamp() function, 736
TimestampFromTicks type object (DB-API), 267
TIPC (Transparent Interprocess Communication) protocol, 59
Tipfy, 617, 618
   App Engine and, 676
title() function, 775
title() method (string), 460, 780
"title-case formatter", 285
Tix (Tk Interface eXtensions), 238
Tix module, 248
Tk GUI toolkit, 214
   geometry managers, 218
   widgets, 219–221
Tk Interface eXtensions (Tix), 238
Tk module, 248
Tk GUITk lift, 214
   geometry managers, 218
   widgets, 219–221
Tk Interface eXtensions (Tix), 238
Tk library, 244–246
tkhello1.py script, 221–222
tkhello2.py script, 222
tkhello3.py script, 223
tkhello4.py script, 224–225
Tkinter module, 214–215, 248
demo code, 235
examples
   Button widget, 222
directory tree traversal tool, 230–236
   Label and Button widgets, 223
   Label widget, 221–222
   Label1, Button, and Scale widgets, 224–225
   importing, 215
   installing, 215
   Python programming and, 216–221
Tk for GUI, 745
TkZinc module, 248
TLS (Transport Layer Security), 144, 146
Tool Command Language (Tcl), 214
toplevel widget, 221
topnews() function, 732
top_posts() function, 758
Tk for GUI, 745
Tornado, 496
ToscaWidgets, 495
traceback, 445
transactional counter, 643
transition plan, 817
translate() function, 775
translate() method (string), 781
Transmission Control Protocol (TCP), 60
   client creation, 68–71
SocketServer server and client execution, 83
timestamp server, 66–68
Transparent Interprocess Communication (TIPC) protocol, 59
   tree format, for HTML documents, parsing, 423
tree-based parsers for XML, 725
troubleshooting Twython library install, 572
_truediv_() method, 792
truncate() method (file object), 787
try-except statement, while loop inside
except clause, 72
Ts_ci_wrap class, 486
ts_simple_wsgi_app(), for wrapping apps, 485
tsTcIntV6.py script, 71
tsTcInt.py script, 69–71
tsTcIntW.py script, 85, 86
tsTserv.py script, 66–68
tsTserv3.py script, 67, 68
tsTserv55.py script, 80
tsTservW.py script, 84
tsUcInt.py script, 75
tsUer.py script, 73, 74
tuple() function, 775
TurboEntity, 295
TurboGears, 495
resources, 597
twapi module, 735
twapi.py script, 695–696, 698–706
Tweepy, 691
   compatibility library for Twython and,
   693–706
Tweet class, for TweetApprover, 578
TweetApprover, 564–596
approver app
   urls.py URLconf file, 576
data model, 576–582
   DATABASES variable, 570
   installing Twython library for,
   571–572
poster and approver apps, 565
poster app
   data models file, 578
   urls.py URLconf file, 575
views.py file, 582
project file structure, 565–571
Project URLconf file, 573–575
reviewing tweets, 587–596
settings file, 566–571
submitting tweets for review, 582–586
templates
   for pending tweet form, 595
   login.html, 595
   to display post status, 592
URL structure, 572–576
user creation, 380
workflow, 565
tweet_auth.py file, 699
TweetForm, definition, 583
tweets, 690
Twisted framework, 84–87
   executing TCP server and client, 87
TCP client creation, 85–87
TCP server creation, 84–85
Web site, 89
URLs (Uniform Resource Locators) (continued)
GET request variables and values in, 452
variables in, 392
URLs file, 531
for Django app, 504, 508
urlunparse() function, 398, 399
URNs (Uniform Resource Names), 396
USE statement (SQL), 256
Usenet News System, 104–114
User Datagram Protocol (UDP), 61
user input
Django and, 542–546
cross-site request forgery, 544
templates, 542
URLconf entry, 543
views, 543
Web services processing of, 442
user interface
for blog, 527–533
for databases, 255
for searching posts, 758
testing, 556
user profile in Google+, 750
USER setting, for Django database, 510
user() method (POP3 object), 125
username for anonymous FTP, 97
UserRequestThread, 158
Users service, 616
adding in App Engine, 652–654
users, creating in TweetApprover, 580
user_timeline command (Twitter API), 695
user_timeline() function, 704
UserWarning, 790
ushuffle_*.py application, porting to use
MongoDB, 312
ushuffle_db.py application, 276–279
ushuffle_mongo.py application, 312–315
ushuffle_sad.py application, 292–301
output, 299–301
vs. ushuffle_sae.py application, 304
ushuffle_sae.py application, 301–304
ushuffle_so.py application, 304–309
UTF-8 encoding, 464
V
validating parsers, 725
value comparisons, 769
ValueError, 790
values() function, 804
values() method (dictionary), 783
van Rossum, Guido, 799
variables
hidden, in form, 454, 467
in URLs, 392
tags in Django templates, 528
vendor lock-in, 675
verify_credentials command (Twitter API), 695
verify_credentials() function (Twitter), 704
view functions, 543
create_blogpost(), 562
for blog application, 533–537
for Web page from Django, 527
in Django app, 532
views
fake, 533
for TweetApprover approver app, 587–592
for TweetApprover poster app, 582
generic, 537
in Django, 551–553
for user input, 543
generic views, 552–553
semi-generic views, 551
views.py file
for blog application, 560
for Django app, 508
virtual circuit, 60
virtualenv, 500
resources, 597
VSTO, 325
W
\w alphanumeric character set, 34
\w alphanumeric character set, 34
\w special character, for alphanumeric character class, 14
warming requests, in Google App Engine, 673
WarmUp service/API, 616
Warning, 790
Warning exception (DB-API), 263
Watters, Aaron, 258
Web addresses. See URLs (Uniform Resource Locators)
Web applications
Google App Engine and, 605
model-view controller (MVC) pattern, 514
Web browsers
as FTP client, 103
cookie management, 476
Web clients, 391–392, 394
parsing Web content, 418–424
programmatic browsing, 424–428
Python tools, 396–410
porting urllib2 HTTP authentication example, 407–410
urllib2 module/package, 399
urllib2 HTTP authentication example, 405–407
urlparse module/package, 398–404
simple Web crawler/spider/bot, 410–418
Web connection, opening, 400
Web forms, adding database entry from, 523
Web frameworks, 487, 494–496
App Engine vs., 609
resources on, 597
Web page templates in Django, 527
Web programming
real world development, 487
related modules, 433, 488–489
Web resources
concurrent.futures module, 209
DB-API, 268
list of supported databases, 270
on App Engine, 676
on Appstats, 672
on building extensions, 366
on Cython, 385
on database-related modules/packages, 316
on DRY, 591
on extensions, 387
on FTP, 104
on GUIs, 250
on JOINs, 298
on JSON, 719
on lython, 744
on MongoDB, 311
on NNTP, 114
on non-relational databases, 319
on NoSQL, 310
on Office applications, 357
on Pyco, 386
on PyPy, 387
on Pyrex, 385
on Python versions, 806
on Query methods, 298
on receiving e-mail, 660
on SMTP, 120
on SWIG, 384
on text processing, 738
on Twitter, 704, 707
on Twitter API libraries, 691
on Web frameworks, 597
on XML-RPC, 736
on Yahoo! Finance Server, 707
on race conditions, 190
on ur17182, 407
Web server farm, 395
Web server Gateway Interface (WSGI), 480–482
reference server, 483
Web servers, 55, 391–392, 428–433
implementing simple base, 430–431
in Django, 505
scaling, 487
setup for CGI, 446–448
typical modern-day components, 444
Web services
basics, 685
microblogging with Twitter, 690–707
Yahoo! Finance Stock Quotes Server, 685–689
Web sites
CGI for fully interactive, 457–463
downloading latest version of file, 101
Web surfing, 391–392
web.py, 496
web2py, 496, 618
App Engine and, 676
web2py framework, 619
webapp framework, 617, 619
webapp2 framework, 617, 627
Web-based SaaS cloud services, 135
webbrowser module/package, 433, 489
WebWare MiddleKit, 289
well-known port numbers, 59
whitespace characters
\s in regex for, 14
matching in regex, 7
removing, 113
who command (POSIX), regular expression for
output, 36–38
who variable (Python), 451
widgets, 217
default arguments, 221
in top-level window object, 219
WIDTH variable for Google+ program, 754
win32com.client module, 327
win32ui module, 249
windowing object, 216
top-level, 217
defining size, 225
widgets in, 219
Windows Extensions for Python, 326
windows servers, 55
WindowsError, 789
with statement, 38
context manager and, 196
g et Ranking() use of, 208
withdraw() function, 329
word boundaries
matching and, 7, 10, 26
matching from start or end, 10
Word, COM programming with, 331
word.pw script, 331
workbook in Excel, 329
wrappers, listing for Python interpreter, 376
wrapping apps, 485
write() function, WSGI standard and, 481
write() method, 81, 102
write() method (file object), 787
writelines() method (file object), 787
writeQ() function, 205
writer() function, 205
writer() method, 717
WSGI (Web Server Gateway Interface), 496
middleware and wrapping apps, 485
sample apps, 484
servers, 482
updates in Python 3, 486
wsgi.* environment variables, 483
wsgiref module, 435, 489
demonstration app, 484
wsgiref.simple_server.demo_app(), 484
wsgiref.simple_server.WSGIServer, 483
wxGlade module, 248
wxPython module, 248
wxWidgets, animalWx.pyw application, 240–242

X
xhdr() method (NNTP object), 107, 112
xist tool, 462
XML (eXtensible Markup Language), 724–738
converting Python dict to, 725–729
vs. JSON, 719
in practice, 729–733
xml package, 434, 725
xml.dom module/package, 434, 740
xml.dom.minidom, 725
xml.etree module/package, 434
xml.etree.ElementTree module/package, 740
xml.parsers.expat package, 434, 740
xml.sax module/package, 434, 740
xml11b module, 434, 725
XML-RPC, 733–738
client code, 737–738
resources, 736
xmlrpc.client package, 733
xmlrpc.server package, 733
xmlrpcclnt.py script, 737–738
xmlrpclib module, 148, 434, 733, 737, 740
xmlrpcsrvr.py script, 734–737
XMPP (eXtensible Messaging and Presence Protocol), 614
XMPP (eXtensible Messaging and Presence Protocol) API, 616, 660
__xor__() method, 793
xreadlines() method (file object), 786

Y
Yahoo! Finance Stock Quotes Server, 685–689
code interface with, 736
csv module for, 717–719
parameters, 687, 695
resources, 707
Yahoo! Mail, 135, 138–144
Yahoo! Mail Plus, 135, 139
YAML (yet another markup language), 622
yielding, 159
ymail.py script, 140–144

Z
\Z special character, for matching from end of string, 10
ZeroDivisionError, 788
zf11() function, 775
zf11() method (string), 781
Zip files
for App Engine SKD, 620
Google App Engine and, 613
zip() function, 731, 804
iterator version, 820
Zope, 496