PRENTICE HALL

core PYTHON APPLICATIONS programming

THIRD EDITION

- Already know Python but want to learn more? A *lot* more? Dive into a variety of topics used in practice for 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/Jython.
- 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

"The simplified yet deep level of detail, comprehensive coverage of material, and informative historical references make this book perfect for the classroom... An easy read, with complex examples presented simply, and great historical references rarely found in such books.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

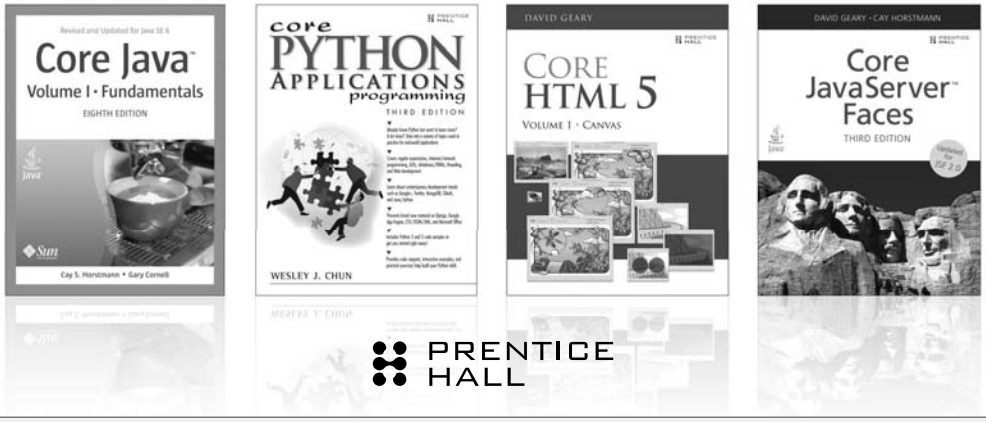
"If you like the Prentice Hall Core series, another good full-blown treatment to consider would be *Core Python Programming*. It addresses in elaborate concrete detail many practical topics that get little, if any, coverage in other books."

—Mitchell L. Model, MLM Consulting

Core PYTHON Applications Programming

Third Edition

The Core Series



Visit informit.com/coreseries for a complete list of available publications.

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.

Make sure to connect with us!
informit.com/socialconnect

Core PYTHON Applications Programming

Third Edition

Wesley J. Chun

PRENTICE HALL

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data
Chun, Wesley.
Core python applications programming / Wesley J. Chun. — 3rd ed.
p. cm.
Rev. ed. of: Core Python programming / Wesley J. Chun. c2007.
Includes index.
ISBN 0-13-267820-9 (pbk. : alk. paper)
1. Python (Computer program language) I. Chun, Wesley. Core Python programming. II. Title.
QA76.73.P98C48 2012
005.1'17—dc23 2011052903

ISBN-13: 978-0-13-267820-9
ISBN-10: 0-13-267820-9

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

This page intentionally left blank

PREFACE

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 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)

- LinuxJournal (linuxjournal.com)
  - Favorite Programming Language (2009–2011)
  - Favorite Scripting Language (2006–2008, 2010, 2011)
- LinuxQuestions.org Members Choice Awards
  - 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/Jython 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

2.5 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

Acknowledgments for the Third Edition

Reviewers and Contributors

Gloria Willadsen (lead reviewer)
Martin Omander (reviewer and also coauthor of Chapter 11, "Web Frameworks: Django," creator of the TweetApprover application, and coauthor of Section 15.2, "Google+," in Chapter 15, "Miscellaneous").
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.

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.

CHAPTER 4

Multithreaded Programming

> 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 of 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):

```
>>> import thread
>>>
```

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

```
>>> 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.

```
1  #!/usr/bin/env python
2
3  from time import sleep, ctime
4
5  def loop0():
6      print 'start loop 0 at:', ctime()
7      sleep(4)
```

```
8       print 'loop 0 done at:', ctime()
9
10  def loop1():
11      print 'start loop 1 at:', ctime()
12      sleep(2)
13      print 'loop 1 done at:', ctime()
14
15  def main():
16      print 'starting at:', ctime()
17      loop0()
18      loop1()
19      print 'all DONE at:', ctime()
20
21  if __name__ == '__main__':
22      main()
```

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 us 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.

3.x

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.

Table 4-1 thread Module and Lock Objects

Function/Method	Description
thread *Module Functions*	
start_new_thread(*function*, *args*, *kwargs*=None)	Spawns a new thread and executes function with the given args and optional *kwargs*
allocate_lock()	Allocates LockType lock object
exit()	Instructs a thread to exit
LockType Lock *Object Methods*	
acquire(*wait*=None)	Attempts to acquire lock object
locked()	Returns True if lock acquired, False otherwise
release()	Releases lock

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:

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.

```
1  #!/usr/bin/env python
2
3  import thread
4  from time import sleep, ctime
5
6  def loop0():
7      print 'start loop 0 at:', ctime()
```

(Continued)

Example 4-2 Using the `thread` Module (`mtsleepA.py`) *(Continued)*

```
8       sleep(4)
9       print 'loop 0 done at:', ctime()
10
11  def loop1():
12      print 'start loop 1 at:', ctime()
13      sleep(2)
14      print 'loop 1 done at:', ctime()
15
16  def main():
17      print 'starting at:', ctime()
18      thread.start_new_thread(loop0, ())
19      thread.start_new_thread(loop1, ())
20      sleep(6)
21      print 'all DONE at:', ctime()
22
23  if __name__ == '__main__':
24      main()
```

`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.

```
$ mtsleepA.py
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

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 `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 `mtsleepB.py`, which is presented in Example 4-3. Running it, we see that the output is similar to `mtsleepA.py`. The only difference is that we did not have to wait the extra time for `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 (`mtsleepB.py`)

Rather than using a call to `sleep()` to hold up the main thread as in `mtsleepA.py`, the use of locks makes more sense.

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

import thread
from time import sleep, ctime

loops = [4,2]

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

```

(Continued)

Example 4-3 Using `thread` and Locks (`mtsleepB.py`) *(Continued)*

```
14      def main():
15          print 'starting at:', ctime()
16          locks = []
17          nloops = range(len(loops))
18
19       for i in nloops:
20           lock = thread.allocate_lock()
21           lock.acquire()
22           locks.append(lock)
23
24    for i in nloops:
25        thread.start_new_thread(loop,
26            (i, loops[i], locks[i]))
27
28       for i in nloops:
29           while locks[i].locked(): pass
30
31       print 'all DONE at:', ctime()
32
33  if __name__ == '__main__':
34       main()
```

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.

Lines 14–34

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.

Table 4-2 `threading` Module Objects

Object	Description
`Thread`	Object that represents a single thread of execution
`Lock`	Primitive lock object (same lock as in `thread` module)
`RLock`	Re-entrant lock object provides ability for a single thread to (re)acquire an already-held lock (recursive locking)
`Condition`	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
`Event`	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
`Semaphore`	Provides a "counter" of finite resources shared between threads; block when none are available
`BoundedSemaphore`	Similar to a `Semaphore` but ensures that it never exceeds its initial value
`Timer`	Similar to `Thread`, except that it waits for an allotted period of time before running
`Barrier`[a]	Creates a "barrier," at which a specified number of threads must all arrive before they're all allowed to continue

a. New in Python 3.2.

3.2

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.

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

Attribute	Description
Thread object data attributes	
`name`	The name of a thread.
`ident`	The identifier of a thread.
`daemon`	Boolean flag indicating whether a thread is daemonic.
Thread object methods	
`__init__(`*`group`*`=None, `*`target`*`=None, `*`name`*`=None, `*`args`*`=(), `*`kwargs`*`={}, `*`verbose`*`=None, `*`daemon`*`=None)`[c]	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.
`start()`	Begin thread execution.
`run()`	Method defining thread functionality (usually overridden by application writer in a subclass).
`join(`*`timeout`*`=None)`	Suspend until the started thread terminates; blocks unless *`timeout`* (in seconds) is given.
`getName()`[a]	Return name of thread.
`setName(`*`name`*`)`[a]	Set name of thread.
`isAlive/is_alive()`[b]	Boolean flag indicating whether thread is still running.
`isDaemon()`[c]	Return `True` if thread daemonic, `False` otherwise.
`setDaemon(`*`daemonic`*`)`[c]	Set the daemon flag to the given Boolean *`daemonic`* value (must be called before thread `start()`.

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/setDaemon()` 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.

```
1   #!/usr/bin/env python
2
3   import threading
4   from time import sleep, ctime
5
6   loops = [4,2]
7
8   def loop(nloop, nsec):
9       print 'start loop', nloop, 'at:', ctime()
10      sleep(nsec)
11      print 'loop', nloop, 'done at:', ctime()
12
13  def main():
14      print 'starting at:', ctime()
15      threads = []
```

(Continued)

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

```
    nloops = range(len(loops))

    for i in nloops:
        t = threading.Thread(target=loop,
            args=(i, loops[i]))
        threads.append(t)

    for i in nloops:          # start threads
        threads[i].start()

    for i in nloops:          # wait for all
        threads[i].join()     # threads to finish

    print 'all DONE at:', ctime()

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

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.

Example 4-5 Using Callable Classes (mtsleepD.py)

In this example, we pass in a callable class (instance) as opposed to just a function. It presents more of an object-oriented approach than mtsleepC.py.

```
#!/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)

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

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

When we run mtsleepD.py, we get the expected output:

```
$ mtsleepD.py
starting at: Sun Aug 13 18:49:17 2006
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.

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.

```
1   #!/usr/bin/env python
2
3   import threading
4   from time import sleep, ctime
5
6   loops = (4, 2)
7
8   class MyThread(threading.Thread):
9       def __init__(self, func, args, name=''):
10          threading.Thread.__init__(self)
11          self.name = name
12          self.func = func
13          self.args = args
14
15      def run(self):
16          self.func(*self.args)
17
18  def loop(nloop, nsec):
19      print 'start loop', nloop, 'at:', ctime()
20      sleep(nsec)
21      print 'loop', nloop, 'done at:', ctime()
22
23  def main():
24      print 'starting at:', ctime()
25      threads = []
26      nloops = range(len(loops))
27
28      for i in nloops:
29          t = MyThread(loop, (i, loops[i]),
30              loop.__name__)
31          threads.append(t)
32
```

(Continued)

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

```
33      for i in nloops:
34          threads[i].start()
35
36      for i in nloops:
37          threads[i].join()
38
39      print 'all DONE at:', ctime()'
40
41  if __name__ == '__main__':
42      main()
```

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
5
```

```
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
13      def getResult(self):
14          return self.res
15
16      def run(self):
17          print 'starting', self.name, 'at:', \
18              ctime()
19          self.res = self.func(*self.args)
20          print self.name, 'finished at:', \
21              ctime()
```

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

Function	Description
activeCount/ active_count()[a]	Number of currently active Thread objects
currentThread()/ current_thread[a]	Returns the current Thread object
enumerate()	Returns list of all currently active Threads
settrace(*func*)[b]	Sets a trace *function* for all threads
setprofile(*func*)[b]	Sets a profile *function* for all threads
stack_size(*size*=0)[c]	Returns stack size of newly created threads; optional *size* can be set for subsequently created threads

a. CamelCase names deprecated and replaced starting in Python 2.6.
b. New in Python 2.3.
c. An alias to thread.stack_size(); (both) new in Python 2.5.

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.

```
#!/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 '\n*** MULTIPLE THREADS'
    threads = []
```

```
    for i in nfuncs:
        t = MyThread(funcs[i], (n,),
            funcs[i].__name__)
        threads.append(t)

    for i in nfuncs:
        threads[i].start()

    for i in nfuncs:
        threads[i].join()
        print threads[i].getResult()

    print 'all DONE'

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

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
```

```
fac finished at: Wed Nov 16 18:52:28 2011
sum finished at: Wed Nov 16 18:52:28 2011
fib finished at: Wed Nov 16 18:52:31 2011
233
479001600
78
all DONE
```

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 staightforward. 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.

```
#!/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)) # or str.format()
    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.

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
all DONE at: Wed Mar 30 22:11:25 2011
```

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 `mtsleepC.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 `mtsleep`*X*`.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

3.x

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()
RefactoringTool: Files that were modified:
RefactoringTool: bookrank.py
```

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 (Unicode) 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:

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

Now let's try it again:

```
$ 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:

```
>>> 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:

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

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

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

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

```
$ 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:

```
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 `mtsleep`*X*`.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 mtsleepF.py
[Sat Apr  2 11:37:26 2011] Started Thread-1
[Sat Apr  2 11:37:26 2011] Started Thread-2
[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 *un*lucky, you might get strange output such as this pair of example executions:

```
$ python mtsleepF.py
[Sat Apr  2 11:37:09 2011] Started Thread-1
 [Sat Apr  2 11:37:09 2011] Started Thread-2
[Sat Apr  2 11:37:09 2011] Started Thread-3
[Sat Apr  2 11:37:12 2011] Completed Thread-1 (3 secs)
 [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 mtsleepF.py
[Sat Apr  2 11:37:56 2011] Started Thread-1
[Sat Apr  2 11:37:56 2011] Started Thread-2
 [Sat Apr  2 11:37:56 2011] Started Thread-3
[Sat Apr  2 11:37:56 2011] Started Thread-4

[Sat Apr  2 11:37:58 2011] Completed Thread-2 (2 secs)
 [Sat Apr  2 11:37:58 2011] Completed Thread-4 (2 secs)
    (remaining: Thread-3, Thread-1)
    (remaining: Thread-3, Thread-1)
```

```
[Sat Apr  2 11:38:00 2011] Completed Thread-1 (4 secs)
    (remaining: Thread-3)
[Sat Apr  2 11:38:00 2011] Completed Thread-3 (4 secs)
    (remaining: NONE)
all DONE at: Sat Apr  2 11:38:00 2011
```

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:

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

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

```
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-2
[Sun Apr  3 23:16:59 2011] Started Thread-3
[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)
```

```
[Sun Apr  3 23:17:02 2011] Completed Thread-1 (3 secs)
    (remaining: Thread-2)
[Sun Apr  3 23:17:03 2011] Completed Thread-2 (4 secs)
    (remaining: NONE)
all DONE at: Sun Apr  3 23:17:03 2011
```

The modified (and final) version of `mtsleepF.py` is shown in Example 4-10.

Example 4-10 Locks and More Randomness (`mtsleepF.py`)

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

```
#!/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

2.5 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

3.x 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 23:29:39 2011] Started Thread-1
[Sun Apr  3 23:29:39 2011] Started Thread-2
[Sun Apr  3 23:29:39 2011] Started Thread-3
[Sun Apr  3 23:29:41 2011] Completed Thread-3 (2 secs)
    (remaining: Thread-2, Thread-1)
[Sun Apr  3 23:29:42 2011] Completed Thread-2 (3 secs)
    (remaining: Thread-1)
[Sun Apr  3 23:29:43 2011] Completed Thread-1 (4 secs)
    (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 situation is more complex, you might need a more powerful synchronization primitive, instead. For applications with finite resources, using semaphores 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.

```
#!/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()

```

```
32  def producer(loops):
33      for i in xrange(loops):
34          refill()
35          sleep(randrange(3))
36
37  def consumer(loops):
38      for i in xrange(loops):
39          buy()
40          sleep(randrange(3))
41
42  def _main():
43      print 'starting at:', ctime()
44      nloops = randrange(2, 6)
45      print 'THE CANDY MACHINE (full with %d bars)!' % MAX
46      Thread(target=consumer, args=(randrange(
47          nloops, nloops+MAX+2),)).start() # buyer
48      Thread(target=producer, args=(nloops,)).start() #vndr
49
50  @register
51  def _atexit():
52      print 'all DONE at:', ctime()
53
54  if __name__ == '__main__':
55      _main()
```

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.

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 nondeterministic, as is the consumer consuming the goods produced by the producer.

3.x

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

Attribute	Description
`Queue`/`queue` *Module Classes*	
`Queue(`*`maxsize`*`=0)`	Creates a FIFO queue of given *`maxsize`* where inserts block until there is more room, or (if omitted), unbounded
`LifoQueue(`*`maxsize`*`=0)`	Creates a LIFO queue of given *`maxsize`* where inserts block until there is more room, or (if omitted), unbounded
`PriorityQueue(`*`maxsize`*`=0)`	Creates a priority queue of given *`maxsize`* where inserts block until there is more room, or (if omitted), unbounded
`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

Attribute	Description
Queue/queue *Object Methods*	
`qsize()`	Returns queue size (approximate, whereas queue may be getting updated by other threads)
`empty()`	Returns `True` if queue empty, `False` otherwise
`full()`	Returns `True` if queue full, `False` otherwise
`put(`*`item`*`, `*`block`*`=True, `*`timeout`*`=None)`	Puts *`item`* in queue; if *`block`* True (the default) and *`timeout`* is `None`, blocks until room is available; if *`timeout`* is positive, blocks at most `timeout` seconds or if *`block`* False, raises the `Empty` exception
`put_nowait(`*`item`*`)`	Same as `put(item, False)`
`get(`*`block`*`=True, `*`timeout`*`=None)`	Gets *item* from queue, if *block* given (not 0), block until an item is available
`get_nowait()`	Same as `get(False)`
`task_done()`	Used to indicate work on an enqueued item completed, used with `join()` below
`join()`	Blocks until all items in queue have been processed and signaled by a call to `task_done()` above

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 0
producing object for Q... size now 1
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
reader finished at: Sun Jun 18 20:27:25 2006
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.

```
#!/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]
nfuncs = range(len(funcs))

def main():
    nloops = randint(2, 5)
    q = Queue(32)

    threads = []
    for i in nfuncs:
        t = MyThread(funcs[i], (q, nloops),
            funcs[i].__name__)
        threads.append(t)

    for i in nfuncs:
        threads[i].start()

    for i in nfuncs:
        threads[i].join()

    print 'all DONE'

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.

Lines 31–47

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

2.4

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

2.6

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.

3.2

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

```
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.

Example 4-13 Higher-Level Job Management (`bookrank3CF.py`)

Our friend, the book rank screenscraper, but this time using `concurrent.futures`.

```
1   #!/usr/bin/env python
2
3   from concurrent.futures import ThreadPoolExecutor
4   from re import compile
5   from time import ctime
6   from urllib.request import urlopen as uopen
7
8   REGEX = compile(b'#([\d,]+) in Books ')
9   AMZN = 'http://amazon.com/dp/'
10  ISBNs = {
11      '0132269937': 'Core Python Programming',
12      '0132356139': 'Python Web Development with Django',
13      '0137143419': 'Python Fundamentals',
14  }
15
16  def getRanking(isbn):
17      with uopen('{0}{1}'.format(AMZN, isbn)) as page:
18          return str(REGEX.findall(page.read())[0], 'utf-8')
19
20  def _main():
21      print('At', ctime(), 'on Amazon...')
22      with ThreadPoolExecutor(3) as executor:
23          for isbn, ranking in zip(
24                  ISBNs, executor.map(getRanking, ISBNs)):
25              print('- %r ranked %s' % (ISBNs[isbn], ranking)
26      print('all DONE at:', ctime())
27
28  if __name__ == '__main__':
29      main()
```

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 4-6 Threading-Related Standard Library Modules

Module	Description
`thread`[a]	Basic, lower-level thread module
`threading`	Higher-level threading and synchronization objects
`multiprocessing`[b]	Spawn/use subprocesses with a "threading" interface
`subprocess`[c]	Skip threads altogether and execute processes instead
`Queue`	Synchronized FIFO queue for multiple threads
`mutex`[d]	Mutual exclusion objects

(Continued)

Table 4-6 Threading-Related Standard Library Modules *(Continued)*

Module	Description
`concurrent.futures`[e]	High-level library for asynchronous execution
`SocketServer`	Create/manage threaded TCP or UDP servers

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+).

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

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:

```
$ 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
```

INDEX

C

D

E

F

G

M

S

T

V

W