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"The simplified yet deep level of detail, comprehensive coverage of material, and informative historical references make this book perfect for the classroom... An easy read, with complex examples presented simply, and great historical references rarely found in such books.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

Pearson

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.

PART I

General Application Topics

CHAPTER 1

Regular Expressions

Some people, when confronted with a problem, think, "I know, I'll use regular expressions." Now they have two *problems.*
—Jamie "jwz" Zawinski, August 1997

In this chapter...

- Introduction/Motivation
- Special Symbols and Characters
- Regexes and Python
- Some Regex Examples
- A Longer Regex Example

1.1 Introduction/Motivation

Manipulating text or data is a big thing. If you don't believe me, look very carefully at what computers primarily do today. Word processing, "fill-out-form" Web pages, streams of information coming from a database dump, stock quote information, news feeds—the list goes on and on. Because we might not know the exact text or data that we have programmed our machines to process, it becomes advantageous to be able to express it in patterns that a machine can recognize and take action upon.

If I were running an e-mail archiving company, and you, as one of my customers, requested all of the e-mail that you sent and received last February, for example, it would be nice if I could set a computer program to collate and forward that information to you, rather than having a human being read through your e-mail and process your request manually. You would be horrified (and infuriated) that someone would be rummaging through your messages, even if that person were *supposed* to be looking only at time-stamp. Another example request might be to look for a subject line like "ILOVEYOU," indicating a virus-infected message, and remove those e-mail messages from your personal archive. So this begs the question of how we can program machines with the ability to look for patterns in text.

Regular expressions provide such an infrastructure for advanced text pattern matching, extraction, and/or search-and-replace functionality. To put it simply, a regular expression (a.k.a. a "*regex*" for short) is a string that use special symbols and characters to indicate pattern repetition or to represent multiple characters so that they can "match" a set of strings with similar characteristics described by the pattern (Figure 1-1). In other words, they enable matching of multiple strings—a regex pattern that matched only one string would be rather boring and ineffective, wouldn't you say?

Python supports regexes through the standard library `re` module. In this introductory subsection, we will give you a brief and concise introduction. Due to its brevity, only the most common aspects of regexes used in everyday Python programming will be covered. Your experience will, of course, vary. We highly recommend reading any of the official supporting documentation as well as external texts on this interesting subject. You will never look at strings in the same way again!

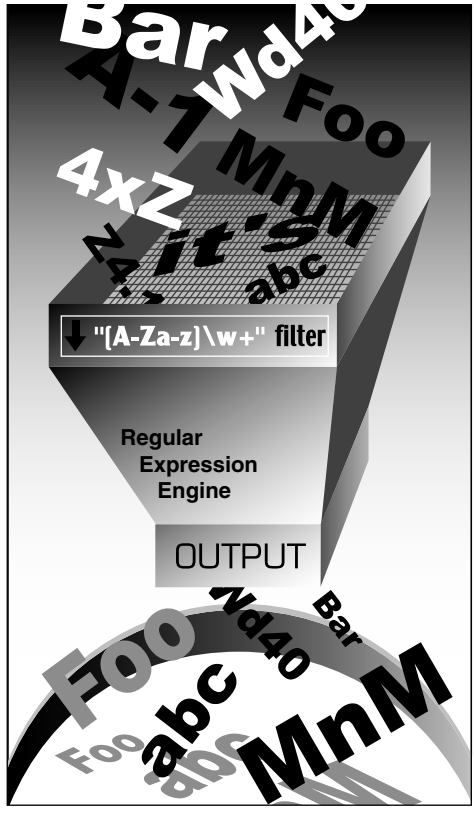

Figure 1-1 You can use regular expressions, such as the one here, which recognizes valid Python identifiers. `[A-Za-z]\w+` means the first character should be alphabetic, that is, either A–Z or a–z, followed by at least one (+) alphanumeric character (\w). In our filter, notice how many strings go into the filter, but the only ones to come out are the ones we asked for via the regex. One example that did not make it was "4xZ" because it starts with a number.

CORE NOTE: Searching vs. matching

Throughout this chapter, you will find references to searching and matching. When we are strictly discussing regular expressions with respect to patterns in strings, we will say "matching," referring to the term *pattern-matching*. In Python terminology, there are two main ways to accomplish pattern-matching: *searching*, that is, looking for a pattern match in any part of a string; and *matching*, that is, attempting to match a pattern to an entire string (starting from the beginning). Searches are accomplished by using the `search()` function or method, and matching is done with the `match()` function or method. In summary, we keep

the term "matching" universal when referencing patterns, and we differentiate between "searching" and "matching" in terms of how Python accomplishes pattern-matching.

1.1.1 Your First Regular Expression

As we mentioned earlier, regexes are strings containing text and special characters that describe a pattern with which to recognize multiple strings. We also briefly discussed a regular expression *alphabet*. For general text, the alphabet used for regular expressions is the set of all uppercase and lowercase letters plus numeric digits. Specialized alphabets are also possible; for instance, you can have one consisting of only the characters "0" and "1." The set of all strings over this alphabet describes all binary strings, that is, "0," "1," "00," "01," "10," "11," "100," etc.

Let's look at the most basic of regular expressions now to show you that although regexes are sometimes considered an advanced topic, they can also be rather simplistic. Using the standard alphabet for general text, we present some simple regexes and the strings that their patterns describe. The following regular expressions are the most basic, "true vanilla," as it were. They simply consist of a string pattern that matches only one string: the string defined by the regular expression. We now present the regexes followed by the strings that match them:

Regex Pattern	String(s) Matched
`foo`	`foo`
`Python`	`Python`
`abc123`	`abc123`

The first regular expression pattern from the above chart is "foo." This pattern has no special symbols to match any other symbol other than those described, so the only string that matches this pattern is the string "foo." The same thing applies to "Python" and "abc123." The power of regular expressions comes in when special characters are used to define character sets, subgroup matching, and pattern repetition. It is these special symbols that allow a regex to match a set of strings rather than a single one.

1.2 Special Symbols and Characters

We will now introduce the most popular of the special characters and symbols, known as *metacharacters*, which give regular expressions their power and flexibility. You will find the most common of these symbols and characters in Table 1-1.

Table 1-1 Common Regular Expression Symbols and Special Characters

Notation	Description	Example Regex
Symbols		
literal	Match literal string value *literal*	foo
re1|*re2*	Match regular expressions *re1* or *re2*	*foo*|*bar*
.	Match *any character* (except \n)	b.b
^	Match *start of string*	^Dear
$	Match *end of string*	/bin/*sh$
*	Match *0 or more* occurrences of preceding regex	[A-Za-z0-9]*
+	Match *1 or more* occurrences of preceding regex	[a-z]+\.com
?	Match *0 or 1* occurrence(s) of preceding regex	goo?
{*N*}	Match *N* occurrences of preceding regex	[0-9]{3}
{*M*,*N*}	Match from *M* to *N* occurrences of preceding regex	[0-9]{5,9}
[...]	Match any single character from *character* class	[aeiou]
[..*x*-*y*..]	Match any single character in the *range from* x to y	[0-9],[A-Za-z]

Notation	Description	Example Regex
Symbols		
`[^...]`	*Do not match* any character from character class, including any ranges, if present	`[^aeiou]`, `[^A-Za-z0-9_]`
`(*|+|?|{})?`	Apply "non-greedy" versions of above occurrence/repetition symbols (`*`, `+`, `?`, `{}`)	`.*?[a-z]`
`(...)`	Match enclosed regex and save as *subgroup*	`([0-9]{3})?`, `f(oo|u)bar`
Special Characters		
`\d`	Match any decimal *digit*, same as `[0-9]` (`\D` is inverse of `\d`: do not match any numeric digit)	`data\d+.txt`
`\w`	Match any *alphanumeric* character, same as `[A-Za-z0-9_]` (`\W` is inverse of `\w`)	`[A-Za-z_]\w+`
`\s`	Match *any whitespace* character, same as `[ \n\t\r\v\f]` (`\S` is inverse of `\s`)	`of\sthe`
`\b`	Match any *word boundary* (`\B` is inverse of `\b`)	`\bThe\b`
`\N`	Match saved *subgroup* `N` (see `(...)` above)	`price: \16`
`\c`	Match any *special character* `c` verbatim (i.e., without its special meaning, literal)	`\.`, `\\`, `\*`
`\A` (`\Z`)	Match *start (end) of string* (also see `^` and `$` above)	`\ADear`

(Continued)

Table 1-1 Common Regular Expression Symbols and Special Characters *(Continued)*

Notation	Description	Example Regex
Extension Notation		
`(?iLmsux)`	Embed one or more special "flags" parameters within the regex itself (vs. via function/method)	`(?x)`, `(?im)`
`(?:...)`	Signifies a group whose match is *not* saved	`(?:\w+\.)*`
`(?P<name>...)`	Like a regular group match only identified with `name` rather than a numeric ID	`(?P<data>)`
`(?P=name)`	Matches text previously grouped by `(?P<name>)` in the same string	`(?P=data)`
`(?#...)`	Specifies a comment, all contents within ignored	`(?#comment)`
`(?=...)`	Matches if `...` comes next without consuming input string; called *positive lookahead assertion*	`(?=.com)`
`(?!...)`	Matches if `...` doesn't come next without consuming input; called *negative lookahead assertion*	`(?!.net)`
`(?<=...)`	Matches if `...` comes prior without consuming input string; called *positive lookbehind assertion*	`(?<=800-)`
`(?<!...)`	Matches if `...` doesn't come prior without consuming input; called *negative lookbehind assertion*	`(?<!192\.168\.)`
`(?(`*id/name*`)`*Y*`|`*N*`)`	Conditional match of regex *Y* if group with given *id* or `name` exists else *N*; `|`*N* is optional	`(?(1)y|x`

1.2.1 Matching More Than One Regex Pattern with Alternation (|)

The pipe symbol (|), a vertical bar on your keyboard, indicates an alternation operation. It is used to separate different regular expressions. For example, the following are some patterns that employ alternation, along with the strings they match:

Regex Pattern	Strings Matched
`at|home`	`at, home`
`r2d2|c3po`	`r2d2, c3po`
`bat|bet|bit`	`bat, bet, bit`

With this one symbol, we have just increased the flexibility of our regular expressions, enabling the matching of more than just one string. Alternation is also sometimes called *union* or *logical OR*.

1.2.2 Matching Any Single Character (.)

The dot or period (`.`) symbol matches any single character except for `\n`. (Python regexes have a compilation flag [`S` or `DOTALL`], which can override this to include `\n`s.) Whether letter, number, whitespace (not including "`\n`"), printable, non-printable, or a symbol, the dot can match them all.

Regex Pattern	Strings Matched
`f.o`	Any character between "f" and "o"; for example, `fao`, `f9o`, `f#o`, etc.
`..`	Any pair of characters
`.end`	Any character before the string `end`

Q: What if I want to match the dot or period character?
A: To specify a dot character explicitly, you must escape its functionality with a backslash, as in "`\.`".

1.2.3 Matching from the Beginning or End of Strings or Word Boundaries (^, $, \b, \B)

There are also symbols and related special characters to specify searching for patterns at the beginning and end of strings. To match a pattern starting from the beginning, you must use the carat symbol (^) or the special character \A (backslash-capital "A"). The latter is primarily for keyboards that do not have the carat symbol (for instance, an international keyboard). Similarly, the dollar sign ($) or \Z will match a pattern from the end of a string.

Patterns that use these symbols differ from most of the others we describe in this chapter because they dictate location or position. In the previous Core Note, we noted that a distinction is made between *matching* (attempting matches of entire strings starting at the beginning) and *searching* (attempting matches from anywhere within a string). With that said, here are some examples of "edge-bound" regex search patterns:

Regex Pattern	Strings Matched
`^From`	Any string that starts with `From`
`/bin/tcsh$`	Any string that ends with `/bin/tcsh`
`^Subject: hi$`	Any string consisting solely of the string `Subject: hi`

Again, if you want to match either (or both) of these characters verbatim, you must use an escaping backslash. For example, if you wanted to match any string that ended with a dollar sign, one possible regex solution would be the pattern `.*\$$`.

The special characters \b and \B pertain to word boundary matches. The difference between them is that \b will match a pattern to a word boundary, meaning that a pattern must be at the beginning of a word, whether there are any characters in front of it (word in the middle of a string) or not (word at the beginning of a line). And likewise, \B will match a pattern only if it appears starting in the middle of a word (i.e., not at a word boundary). Here are some examples:

Regex Pattern	Strings Matched
`the`	Any string containing `the`
`\bthe`	Any word that starts with `the`

Regex Pattern	Strings Matched
`\bthe\b`	Matches only the word `the`
`\Bthe`	Any string that contains but does not begin with `the`

1.2.4 Creating Character Classes ([])

Whereas the dot is good for allowing matches of any symbols, there might be occasions for which there are specific characters that you want to match. For this reason, the bracket symbols (`[]`) were invented. The regular expression will match any of the enclosed characters. Here are some examples:

Regex Pattern	Strings Matched
`b[aeiu]t`	`bat`, `bet`, `bit`, `but`
`[cr][23][dp][o2]`	A string of four characters: first is "c" or "r," then "2" or "3," followed by "d" or "p," and finally, either "o" or "2." For example, `c2do`, `r3p2`, `r2d2`, `c3po`, etc.

One side note regarding the regex `[cr][23][dp][o2]`—a more restrictive version of this regex would be required to allow only "r2d2" or "c3po" as valid strings. Because brackets merely imply logical OR functionality, it is not possible to use brackets to enforce such a requirement. The only solution is to use the pipe, as in `r2d2|c3po`.

For single-character regexes, though, the pipe and brackets are equivalent. For example, let's start with the regular expression "ab," which matches only the string with an "a" followed by a "b." If we wanted either a one-letter string, for instance, either "a" or a "b," we could use the regex `[ab]`. Because "a" and "b" are individual strings, we can also choose the regex `a|b`. However, if we wanted to match the string with the pattern "ab" followed by "cd," we cannot use the brackets because they work only for single characters. In this case, the only solution is `ab|cd`, similar to the `r2d2/c3po` problem just mentioned.

1.2.5 Denoting Ranges (–) and Negation (^)

In addition to single characters, the brackets also support ranges of characters. A hyphen between a pair of symbols enclosed in brackets is used to indicate a range of characters; for example A–Z, a–z, or 0–9 for uppercase letters, lowercase letters, and numeric digits, respectively. This is a lexicographic range, so you are not restricted to using just alphanumeric characters. Additionally, if a caret (^) is the first character immediately inside the open left bracket, this symbolizes a directive *not* to match any of the characters in the given character set.

Regex Pattern	Strings Matched
`z.[0-9]`	"z" followed by any character then followed by a single digit
`[r-u][env-y]` `[us]`	"r," "s," "t," or "u" followed by "e," "n," "v," "w," "x," or "y" followed by "u" or "s"
`[^aeiou]`	A non-vowel character (Exercise: why do we say "non-vowels" rather than "consonants"?)
`[^\t\n]`	Not a TAB or `\n`
`["-a]`	In an ASCII system, all characters that fall between '"' and "a," that is, between ordinals 34 and 97

1.2.6 Multiple Occurrence/Repetition Using Closure Operators (*, +, ?, {})

We will now introduce the most common regex notations, namely, the special symbols `*`, `+`, and `?`, all of which can be used to match single, multiple, or no occurrences of string patterns. The asterisk or star operator (`*`) will match zero or more occurrences of the regex immediately to its left (in language and compiler theory, this operation is known as the *Kleene Closure*). The plus operator (`+`) will match one or more occurrences of a regex (known as *Positive Closure*), and the question mark operator (`?`) will match exactly 0 or 1 occurrences of a regex.

There are also brace operators (`{}`) with either a single value or a comma-separated pair of values. These indicate a match of exactly *N* occurrences (for `{`*N*`}`) or a range of occurrences; for example, `{`*M*`,` *N*`}` will match from *M* to *N* occurrences. These symbols can also be escaped by using the backslash character; `\*` matches the asterisk, etc.

In the previous table, we notice the question mark is used more than once (overloaded), meaning either matching 0 or 1 occurrences, or its other meaning: if it follows any matching using the close operators, it will direct the regular expression engine to match as few repetitions as possible.

What does "as few repetitions as possible" mean? When pattern-matching is employed using the grouping operators, the regular expression engine will try to "absorb" as many characters as possible that match the pattern. This is known as being *greedy*. The question mark tells the engine to lay off and, if possible, take as few characters as possible in the current match, leaving the rest to match as many succeeding characters of the next pattern (if applicable). Toward the end of the chapter, we will show you a great example where non-greediness is required. For now, let's continue to look at the closure operators:

Regex Pattern	Strings Matched
`[dn]ot?`	"d" or "n," followed by an "o" and, at most, one "t" after that; thus, `do`, `no`, `dot`, `not`.
`0?[1-9]`	Any numeric digit, possibly prepended with a "0." For example, the set of numeric representations of the months January to September, whether single or double-digits.
`[0-9]{15,16}`	Fifteen or sixteen digits (for example, credit card numbers.
`</?[^>]+>`	Strings that match all valid (and invalid) HTML tags.
`[KQRBNP][a-h][1-8]-[a-h][1-8]`	Legal chess move in "long algebraic" notation (move only, no capture, check, etc.); that is, strings that start with any of "K," "Q," "R," "B," "N," or "P" followed by a hyphenated-pair of chess board grid locations from "a1" to "h8" (and everything in between), with the first coordinate indicating the former position, and the second being the new position.

1.2.7 Special Characters Representing Character Sets

We also mentioned that there are special characters that can represent character sets. Rather than using a range of "0–9," you can simply use `\d` to indicate the match of any decimal digit. Another special character, `\w`, can be used to denote the entire alphanumeric character class, serving as a shortcut for `A-Za-z0-9_`, and `\s` can be used for whitespace characters. Uppercase versions of these strings symbolize non-matches; for example, `\D` matches any non-decimal digit (same as `[^0-9]`), etc.

Using these shortcuts, we will present a few more complex examples:

Regex Pattern	Strings Matched
`\w+-\d+`	Alphanumeric string and number separated by a hyphen
`[A-Za-z]\w*`	Alphabetic first character; additional characters (if present) can be alphanumeric (almost equivalent to the set of valid Python identifiers [see exercises])
`\d{3}-\d{3}-\d{4}`	American-format telephone numbers with an area code prefix, as in 800-555-1212
`\w+@\w+\.com`	Simple e-mail addresses of the form *XXX@YYY.com*

1.2.8 Designating Groups with Parentheses (())

Now, we have achieved the goal of matching a string and discarding non-matches, but in some cases, we might also be more interested in the data that we did match. Not only do we want to know whether the entire string matched our criteria, but also whether we can extract any specific strings or substrings that were part of a successful match. The answer is yes. To accomplish this, surround any regex with a pair of parentheses.

A pair of parentheses (`()`) can accomplish either (or both) of the following when used with regular expressions:

- Grouping regular expressions
- Matching subgroups

One good example of why you would want to group regular expressions is when you have two different regexes with which you want to compare a string. Another reason is to group a regex in order to use a repetition operator on the entire regex (as opposed to an individual character or character class).

One side effect of using parentheses is that the substring that matched the pattern is saved for future use. These subgroups can be recalled for the same match or search, or extracted for post-processing. You will see some examples of pulling out subgroups at the end of Section 1.3.9.

Why are matches of subgroups important? The main reason is that there are times when you want to extract the patterns you match, in addition to making a match. For example, what if we decided to match the pattern `\w+-\d+` but wanted save the alphabetic first part and the numeric second part individually? We might want to do this because with any successful match, we might want to see just what those strings were that matched our regex patterns.

If we add parentheses to both subpatterns such as `(\w+)-(\d+)`, then we can access each of the matched subgroups individually. Subgrouping is preferred because the alternative is to write code to determine we have a match, then execute another separate routine (which we also had to create) to parse the entire match just to extract both parts. Why not let Python do it; it's a supported feature of the `re` module, so why reinvent the wheel?

Regex Pattern	Strings Matched
`\d+(\.\d*)?`	Strings representing simple floating-point numbers; that is, any number of digits followed optionally by a single decimal point and zero or more numeric digits, as in "0.004," "2," "75.," etc.
`(Mr?s?\. )?[A-Z] [a-z]* [ A-Za-z-]+`	First name and last name, with a restricted first name (must start with uppercase; lowercase only for remaining letters, if any), the full name, prepended by an optional title of "Mr.," "Mrs.," "Ms.," or "M.," and a flexible last name, allowing for multiple words, dashes, and uppercase letters

1.2.9 Extension Notations

One final aspect of regular expressions we have not touched upon yet include the extension notations that begin with the question mark symbol (`(? . . .)`). We are not going to spend a lot of time on these as they are generally used more to provide flags, perform look-ahead (or look-behind), or check conditionally before determining a match. Also, although parentheses are used with these notations, only `(?P<name>)` represents a grouping for matches. All others do *not* create a group. However, you should still know what they are because they might be "the right tool for the job."

Regex Pattern	Notation Definition
`(?:\w+\.)*`	Strings that end with a dot, like "google.", "twitter.", "facebook.", but such matches are neither saved for use nor retrieval later.
`(?#comment)`	No matching here, just a comment.
`(?=.com)`	Only do a match if ".com" follows; do not consume any of the target string.
`(?!.net)`	Only do a match if ".net" does *not* follow.
`(?<=800-)`	Only do a match if string is preceded by "800-", presumably for phone numbers; again, do not consume the input string.
`(?<!192\.168\.)`	Only do a match if string is not preceded by "192.168.", presumably to filter out a group of Class C IP addresses.
`(?(1)y|x)`	If a matched group 1 (`\1`) exists, match against `y`; otherwise, match against `x`.

1.3 Regexes and Python

Now that we know all about regular expressions, we can examine how Python currently supports regular expressions through the `re` module, which was introduced way back in ancient history (Python 1.5), replacing the deprecated `regex` and `regsub` modules—both modules were removed from Python in version 2.5, and importing either module from that release on triggers an `ImportError` exception.

2.5

The `re` module supports the more powerful and regular Perl-style (Perl 5) regexes, allows multiple threads to share the same compiled regex objects, and supports named subgroups.

1.3.1 The re Module: Core Functions and Methods

The chart in Table 1-2 lists the more popular functions and methods from the `re` module. Many of these functions are also available as methods of compiled regular expression objects (regex objects and regex match objects. In this subsection, we will look at the two main functions/methods, `match()` and `search()`, as well as the `compile()` function. We will introduce several more in the next section, but for more information on all these and the others that we do not cover, we refer you to the Python documentation.

Table 1-2 Common Regular Expression Attributes

Function/Method	Description
`re` *Module Function Only*	
`compile(pattern, flags=0)`	Compile regex *`pattern`* with any optional *`flags`* and return a regex object
`re` *Module Functions and Regex Object Methods*	
`match(pattern, string, flags=0)`	Attempt to match *`pattern`* to *`string`* with optional *`flags`*; return match object on success, `None` on failure
`search(pattern, string, flags=0)`	Search for first occurrence of *`pattern`* within *`string`* with optional *`flags`*; return match object on success, `None` on failure
`findall(pattern, string[,flags])`[a]	Look for all (non-overlapping) occurrences of *`pattern`* in *`string`*; return a list of matches
`finditer(pattern, string[, flags])`[b]	Same as `findall()`, except returns an iterator instead of a list; for each match, the iterator returns a match object
`split(pattern, string, max=0)`[c]	Split *`string`* into a list according to regex *`pattern`* delimiter and return list of successful matches, splitting at most *`max`* times (split all occurrences is the default)

(Continued)

Table 1-2 Common Regular Expression Attributes *(Continued)*

Function/Method	Description
`re` *Module Functions and Regex Object Methods*	
`sub(pattern, repl, string, count=0)`c	Replace all occurrences of the regex *`pattern`* in *`string`* with *`repl`*, substituting all occurrences unless *`count`* provided (see also `subn()`, which, in addition, returns the number of substitutions made)
`purge()`	Purge cache of implicitly compiled regex patterns
Common Match Object Methods (see documentation for others)	
`group(num=0)`	Return entire match (or specific subgroup *`num`*)
`groups (default=None)`	Return all matching subgroups in a tuple (empty if there aren't any)
`groupdict (default=None)`	Return dict containing all matching named subgroups with the names as the keys (empty if there weren't any)
Common Module Attributes (flags for most regex functions)	
`re.I, re.IGNORECASE`	Case-insensitive matching
`re.L, re.LOCALE`	Matches via `\w`, `\W`, `\b`, `\B`, `\s`, `\S` depends on locale
`re.M, re.MULTILINE`	Respectively causes `^` and `$` to match the beginning and end of each line in target string rather than strictly the beginning and end of the entire string itself
`re.S, re.DOTALL`	The `.` normally matches any single character except `\n`; this flag says `.` should match them, too
`re.X, re.VERBOSE`	All whitespace plus `#` (and all text after it on a single line) are ignored unless in a character class or back-slash-escaped, allowing comments and improving readability

a. New in Python 1.5.2; *`flags`* parameter added in 2.4.
b. New in Python 2.2; *`flags`* parameter added in 2.4.
c. *`flags`* parameter added in version 2.7 and 3.1.

CORE NOTE: Regex compilation (to compile or not to compile?)

In the Execution Environment chapter of *Core Python Programming* or the forthcoming *Core Python Language Fundamentals*, we describe how Python code is eventually compiled into bytecode, which is then executed by the interpreter. In particular, we specified that calling `eval()` or `exec` (in version 2.x or `exec()` in version 3.x) with a code object rather than a string provides a performance improvement due to the fact that the compilation process does not have to be performed repeatedly. In other words, using precompiled code objects is faster than using strings because the interpreter will have to compile it into a code object (anyway) each time before execution.

The same concept applies to regexes—regular expression patterns must be compiled into *regex* objects before any pattern matching can occur. For regexes, which are compared many times during the course of execution, we highly recommend using precompilation because, again, regexes have to be compiled anyway, so doing it ahead of time is prudent for performance reasons. `re.compile()` provides this functionality.

The module functions do cache the compiled objects, though, so it's not as if every `search()` and `match()` with the same regex pattern requires compilation. Still, you save the cache lookups and do not have to make function calls with the same string, over and over. The number of compiled regex objects that are cached might vary between releases, and is undocumented. The `purge()` function can be used to clear this cache.

1.3.2 Compiling Regexes with `compile()`

Almost all of the `re` module functions we will be describing shortly are available as methods for regex objects. Remember, even though we recommend it, precompilation is not required. If you compile, you will use methods; if you don't, you will just use functions. The good news is that either way, the names are the same, whether a function or a method. (This is the reason why there are module functions and methods that are identical; for example, `search()`, `match()`, etc., in case you were wondering.) Because it saves one small step for most of our examples, we will use strings, instead. We will throw in a few with compilation, though, just so you know how it is done.

Optional flags may be given as arguments for specialized compilation. These flags allow for case-insensitive matching, using system locale settings for matching alphanumeric characters, etc. Please see the entries in

Table 1-2 and the official documentation for more information on these flags (`re.IGNORECASE`, `re.MULTILINE`, `re.DOTALL`, `re.VERBOSE`, etc.). They can be combined by using the bitwise OR operator (`|`).

These flags are also available as a parameter to most `re` module functions. If you want to use these flags with the methods, they must already be integrated into the compiled regex objects, or you need to use the `(?F)` notation directly embedded in the regex itself, where `F` is one or more of `i` (for `re.I/IGNORECASE`), `m` (for `re.M/MULTILINE`), `s` (for `re.S/DOTALL`), etc. If more than one is desired, you place them together rather than using the bitwise OR operation; for example, `(?im)` for both `re.IGNORECASE` plus `re.MULTILINE`.

1.3.3 Match Objects and the `group()` and `groups()` Methods

When dealing with regular expressions, there is another object type in addition to the regex object: the *match object*. These are the objects returned on successful calls to `match()` or `search()`. Match objects have two primary methods, `group()` and `groups()`.

`group()` either returns the entire match, or a specific subgroup, if requested. `groups()` simply returns a tuple consisting of only/all the subgroups. If there are no subgroups requested, then `groups()` returns an empty tuple while `group()` still returns the entire match.

Python regexes also allow for named matches, which are beyond the scope of this introductory section. We refer you to the complete `re` module documentation for a complete listing of the more advanced details we have omitted here.

1.3.4 Matching Strings with `match()`

`match()` is the first `re` module function and regex object (regex object) method we will look at. The `match()` function attempts to match the pattern to the string, starting at the beginning. If the match is successful, a match object is returned; if it is unsuccessful, `None` is returned. The `group()` method of a match object can be used to show the successful match. Here is an example of how to use `match()` [and `group()`]:

```
>>> m = re.match('foo', 'foo')   # pattern matches string
>>> if m is not None:           # show match if successful
...     m.group()
...
'foo'
```

The pattern "foo" matches exactly the string "foo." We can also confirm that `m` is an example of a match object from within the interactive interpreter:

```
>>> m                        # confirm match object returned
<re.MatchObject instance at 80ebf48>
```

Here is an example of a failed match for which `None` is returned:

```
>>> m = re.match('foo', 'bar')# pattern does not match string
>>> if m is not None: m.group() # (1-line version of if clause)
...
>>>
```

The preceding match fails, thus `None` is assigned to `m`, and no action is taken due to the way we constructed our `if` statement. For the remaining examples, we will try to leave out the `if` check for brevity, if possible, but in practice, it is a good idea to have it there to prevent `AttributeError` exceptions. (`None` is returned on failures, which does not have a `group()` attribute [method].)

A match will still succeed even if the string is longer than the pattern, as long as the pattern matches from the beginning of the string. For example, the pattern "foo" will find a match in the string "food on the table" because it matches the pattern from the beginning:

```
>>> m = re.match('foo', 'food on the table') # match succeeds
>>> m.group()
'foo'
```

As you can see, although the string is longer than the pattern, a successful match was made from the beginning of the string. The substring "foo" represents the match, which was extracted from the larger string.

We can even sometimes bypass saving the result altogether, taking advantage of Python's object-oriented nature:

```
>>> re.match('foo', 'food on the table').group()
'foo'
```

Note from a few paragraphs above that an `AttributeError` will be generated on a non-match.

1.3.5 Looking for a Pattern within a String with `search()` (Searching versus Matching)

The chances are greater that the pattern you seek is somewhere in the middle of a string, rather than at the beginning. This is where `search()` comes in handy. It works exactly in the same way as match, except that it searches

for the first occurrence of the given regex pattern anywhere with its string argument. Again, a match object is returned on success; None is returned otherwise.

We will now illustrate the difference between match() and search(). Let's try a longer string match attempt. This time, let's try to match our string "foo" to "seafood":

```
>>> m = re.match('foo', 'seafood')     # no match
>>> if m is not None: m.group()
...
>>>
```

As you can see, there is no match here. match() attempts to match the pattern to the string from the beginning; that is, the "f" in the pattern is matched against the "s" in the string, which fails immediately. However, the string "foo" does appear (elsewhere) in "seafood," so how do we get Python to say "yes"? The answer is by using the search() function. Rather than attempting a *match*, search() looks for the first occurrence of the pattern within the string. search() evaluates a string strictly from left to right.

```
>>> m = re.search('foo', 'seafood')   # use search() instead
>>> if m is not None: m.group()
...
'foo'                   # search succeeds where match failed
>>>
```

Furthermore, both match() and search() take the optional flags parameter described earlier in Section 1.3.2. Lastly, we want to note that the equivalent regex object methods optionally take pos and endpos arguments to specify the search boundaries of the target string.

We will be using the match() and search() regex object methods and the group() and groups() match object methods for the remainder of this subsection, exhibiting a broad range of examples of how to use regular expressions with Python. We will be using almost all of the special characters and symbols that are part of the regular expression syntax.

1.3.6 Matching More than One String (|)

In Section 1.2, we used the pipe character in the regex bat|bet|bit. Here is how we would use that regex with Python:

```
>>> bt = 'bat|bet|bit'              # regex pattern: bat, bet, bit
>>> m = re.match(bt, 'bat')         # 'bat' is a match
>>> if m is not None: m.group()
...
```

```
'bat'
>>> m = re.match(bt, 'blt')        # no match for 'blt'
>>> if m is not None: m.group()
...
>>> m = re.match(bt, 'He bit me!') # does not match string
>>> if m is not None: m.group()
...
>>> m = re.search(bt, 'He bit me!') # found 'bit' via search
>>> if m is not None: m.group()
...
'bit'
```

1.3.7 Matching Any Single Character (.)

In the following examples, we show that a dot cannot match a \n or a non-character; that is, the empty string:

```
>>> anyend = '.end'
>>> m = re.match(anyend, 'bend')    # dot matches 'b'
>>> if m is not None: m.group()
...
'bend'
>>> m = re.match(anyend, 'end')     # no char to match
>>> if m is not None: m.group()
...
>>> m = re.match(anyend, '\nend')   # any char except \n
>>> if m is not None: m.group()
...
>>> m = re.search('.end', 'The end.')# matches ' ' in search
>>> if m is not None: m.group()
...
' end'
```

The following is an example of searching for a real dot (decimal point) in a regular expression, wherein we escape its functionality by using a backslash:

```
 >>> patt314 = '3.14'              # regex dot
 >>> pi_patt = '3\.14'             # literal dot (dec. point)
>>> m = re.match(pi_patt, '3.14')  # exact match
>>> if m is not None: m.group()
...
'3.14'
>>> m = re.match(patt314, '3014')  # dot matches '0'
>>> if m is not None: m.group()
...
'3014'
>>> m = re.match(patt314, '3.14')  # dot matches '.'
>>> if m is not None: m.group()
...
'3.14'
```

1.3.8 Creating Character Classes ([])

Earlier, we had a long discussion about `[cr][23][dp][o2]` and how it differs from `r2d2|c3po`" In the following examples, we will show that `r2d2|c3po` is more restrictive than `[cr][23][dp][o2]`:

```
>>> m = re.match('[cr][23][dp][o2]', 'c3po')# matches 'c3po'
>>> if m is not None: m.group()
...
'c3po'
>>> m = re.match('[cr][23][dp][o2]', 'c2do')# matches 'c2do'
>>> if m is not None: m.group()
...
'c2do'
>>> m = re.match('r2d2|c3po', 'c2do')# does not match 'c2do'
>>> if m is not None: m.group()
...
>>> m = re.match('r2d2|c3po', 'r2d2')# matches 'r2d2'
>>> if m is not None: m.group()
...
'r2d2'
```

1.3.9 Repetition, Special Characters, and Grouping

The most common aspects of regexes involve the use of special characters, multiple occurrences of regex patterns, and using parentheses to group and extract submatch patterns. One particular regex we looked at related to simple e-mail addresses (`\w+@\w+\.com`). Perhaps we want to match more e-mail addresses than this regex allows. To support an additional hostname that precedes the domain, for example, `www.xxx.com` as opposed to accepting only `xxx.com` as the entire domain, we have to modify our existing regex. To indicate that the hostname is optional, we create a pattern that matches the hostname (followed by a dot), use the `?` operator, indicating zero or one copy of this pattern, and insert the optional regex into our previous regex as follows: `\w+@(\w+\.)?\w+\.com`. As you can see from the following examples, either one or two names are now accepted before the `.com`:

```
>>> patt = '\w+@(\w+\.)?\w+\.com'
>>> re.match(patt, 'nobody@xxx.com').group()
'nobody@xxx.com'
>>> re.match(patt, 'nobody@www.xxx.com').group()
'nobody@www.xxx.com'
```

Furthermore, we can even extend our example to allow any number of intermediate subdomain names with the following pattern. Take special note of our slight change from using ? to *. : \w+@(\w+\.)*\w+\.com:

```
>>> patt = '\w+@(\w+\.)*\w+\.com'
>>> re.match(patt, 'nobody@www.xxx.yyy.zzz.com').group()
'nobody@www.xxx.yyy.zzz.com'
```

However, we must add the disclaimer that using solely alphanumeric characters does not match all the possible characters that might make up e-mail addresses. The preceding regex patterns would not match a domain such as xxx-yyy.com or other domains with \W characters.

Earlier, we discussed the merits of using parentheses to match and save subgroups for further processing rather than coding a separate routine to manually parse a string after a regex match had been determined. In particular, we discussed a simple regex pattern of an alphanumeric string and a number separated by a hyphen, \w+-\d+, and how adding subgrouping to form a new regex, (\w+)-(\d+), would do the job. Here is how the original regex works:

```
>>> m = re.match('\w\w\w-\d\d\d', 'abc-123')
>>> if m is not None: m.group()
...
'abc-123'

>>> m = re.match('\w\w\w-\d\d\d', 'abc-xyz')
>>> if m is not None: m.group()
...
>>>
```

In the preceding code, we created a regex to recognize three alphanumeric characters followed by three digits. Testing this regex on abc-123, we obtained positive results, whereas abc-xyz fails. We will now modify our regex as discussed before to be able to extract the alphanumeric string and number. Note how we can now use the group() method to access individual subgroups or the groups() method to obtain a tuple of all the subgroups matched:

```
>>> m = re.match('(\w\w\w)-(\d\d\d)', 'abc-123')
>>> m.group()                       # entire match
'abc-123'
>>> m.group(1)                      # subgroup 1
'abc'
>>> m.group(2)                      # subgroup 2
'123'
>>> m.groups()                      # all subgroups
('abc', '123')
```

As you can see, `group()` is used in the normal way to show the entire match, but it can also be used to grab individual subgroup matches. We can also use the `groups()` method to obtain a tuple of all the substring matches.

Here is a simpler example that shows different group permutations, which will hopefully make things even more clear:

```
>>> m = re.match('ab', 'ab')          # no subgroups
>>> m.group()                         # entire match
'ab'
>>> m.groups()                        # all subgroups
()
>>>
>>> m = re.match('(ab)', 'ab')        # one subgroup
>>> m.group()                         # entire match
'ab'
>>> m.group(1)                        # subgroup 1
'ab'
>>> m.groups()                        # all subgroups
('ab',)
>>>
>>> m = re.match('(a)(b)', 'ab')      # two subgroups
>>> m.group()                         # entire match
'ab'
>>> m.group(1)                        # subgroup 1
'a'
>>> m.group(2)                        # subgroup 2
'b'
>>> m.groups()                        # all subgroups
('a', 'b')
>>>
>>> m = re.match('(a(b))', 'ab')      # two subgroups
>>> m.group()                         # entire match
'ab'
>>> m.group(1)                        # subgroup 1
'ab'
>>> m.group(2)                        # subgroup 2
'b'
>>> m.groups()                        # all subgroups
('ab', 'b')
```

1.3.10 Matching from the Beginning and End of Strings and on Word Boundaries

The following examples highlight the positional regex operators. These apply more for searching than matching because `match()` always starts at the beginning of a string.

```
>>> m = re.search('^The', 'The end.')          # match
>>> if m is not None: m.group()
...
'The'
>>> m = re.search('^The', 'end. The')       # not at beginning
>>> if m is not None: m.group()
...
>>> m = re.search(r'\bthe', 'bite the dog') # at a boundary
>>> if m is not None: m.group()
...
'the'
>>> m = re.search(r'\bthe', 'bitethe dog')  # no boundary
>>> if m is not None: m.group()
...
>>> m = re.search(r'\Bthe', 'bitethe dog')  # no boundary
>>> if m is not None: m.group()
...
'the'
```

You will notice the appearance of raw strings here. You might want to take a look at the Core Note, "Using Python raw strings," toward the end of this chapter for clarification on why they are here. In general, it is a good idea to use raw strings with regular expressions.

There are four other `re` module functions and regex object methods that we think you should be aware of: `findall()`, `sub()`, `subn()`, and `split()`.

1.3.11 Finding Every Occurrence with `findall()` and `finditer()`

`findall()` looks for all non-overlapping occurrences of a regex pattern in a string. It is similar to `search()` in that it performs a string search, but it differs from `match()` and `search()` in that `findall()` always returns a list. The list will be empty if no occurrences are found, but if successful, the list will consist of all matches found (grouped in left-to-right order of occurrence).

```
>>> re.findall('car', 'car')
['car']
>>> re.findall('car', 'scary')
['car']
>>> re.findall('car', 'carry the barcardi to the car')
['car', 'car', 'car']
```

Subgroup searches result in a more complex list returned, and that makes sense, because subgroups are a mechanism with which you can extract specific patterns from within your single regular expression, such as matching an area code that is part of a complete telephone number, or a login name that is part of an entire e-mail address.

For a single successful match, each subgroup match is a single element of the resulting list returned by `findall()`; for multiple successful matches, each subgroup match is a single element in a tuple, and such tuples (one for each successful match) are the elements of the resulting list. This part might sound confusing at first, but if you try different examples, it will help to clarify things.

2.2

The `finditer()` function, which was added back in Python 2.2, is a similar, more memory-friendly alternative to `findall()`. The main difference between it and its cousin, other than the return of an iterator versus a list (obviously), is that rather than returning matching strings, `finditer()` iterates over match objects. The following are the differences between the two with different groups in a single string:

```
>>> s = 'This and that.'
>>> re.findall(r'(th\w+) and (th\w+)', s, re.I)
[('This', 'that')]
>>> re.finditer(r'(th\w+) and (th\w+)', s,
...   re.I).next().groups()
('This', 'that')
>>> re.finditer(r'(th\w+) and (th\w+)', s,
...   re.I).next().group(1)
'This'
>>> re.finditer(r'(th\w+) and (th\w+)', s,
...   re.I).next().group(2)
'that'
>>> [g.groups() for g in re.finditer(r'(th\w+) and (th\w+)',
...   s, re.I)]
[('This', 'that')]
```

In the example that follows, we have multiple matches of a single group in a single string:

```
>>> re.findall(r'(th\w+)', s, re.I)
['This', 'that']
>>> it = re.finditer(r'(th\w+)', s, re.I)
>>> g = it.next()
>>> g.groups()
('This',)
>>> g.group(1)
'This'
>>> g = it.next()
>>> g.groups()
('that',)
>>> g.group(1)
'that'
>>> [g.group(1) for g in re.finditer(r'(th\w+)', s, re.I)]
['This', 'that']
```

Note all the additional work that we had to do using `finditer()` to get its output to match that of `findall()`.

Finally, like `match()` and `search()`, the method versions of `findall()` and `finditer()` support the optional `pos` and `endpos` parameters that control the search boundaries of the target string, as described earlier in this chapter.

1.3.12 Searching and Replacing with sub() and subn()

There are two functions/methods for search-and-replace functionality: `sub()` and `subn()`. They are almost identical and replace all matched occurrences of the regex pattern in a string with some sort of replacement. The replacement is usually a string, but it can also be a function that returns a replacement string. `subn()` is exactly the same as `sub()`, but it also returns the total number of substitutions made—both the newly substituted string and the substitution count are returned as a 2-tuple.

```
>>> re.sub('X', 'Mr. Smith', 'attn: X\n\nDear X,\n')
'attn: Mr. Smith\012\012Dear Mr. Smith,\012'
>>>
>>> re.subn('X', 'Mr. Smith', 'attn: X\n\nDear X,\n')
('attn: Mr. Smith\012\012Dear Mr. Smith,\012', 2)
>>>
>>> print re.sub('X', 'Mr. Smith', 'attn: X\n\nDear X,\n')
attn: Mr. Smith

Dear Mr. Smith,

>>> re.sub('[ae]', 'X', 'abcdef')
'XbcdXf'
>>> re.subn('[ae]', 'X', 'abcdef')
('XbcdXf', 2)
```

As we saw in an earlier section, in addition to being able to pull out the matching group number using the match object's `group()` method, you can use `\N`, where *N* is the group number to use in the replacement string. Below, we're just converting the American style of date presentation, MM/DD/YY{,YY} to the format used by all other countries, DD/MM/YY{,YY}:

```
>>> re.sub(r'(\d{1,2})/(\d{1,2})/(\d{2}|\d{4})',
...     r'\2/\1/\3', '2/20/91') # Yes, Python is...
'20/2/91'
>>> re.sub(r'(\d{1,2})/(\d{1,2})/(\d{2}|\d{4})',
...     r'\2/\1/\3', '2/20/1991') # ... 20+ years old!
'20/2/1991'
```

1.3.13 Splitting (on Delimiting Pattern) with `split()`

The `re` module and regex object method `split()` work similarly to its string counterpart, but rather than splitting on a fixed string, they split a string based on a regex pattern, adding some significant power to string splitting capabilities. If you do not want the string split for every occurrence of the pattern, you can specify the maximum number of splits by setting a value (other than zero) to the `max` argument.

If the delimiter given is not a regular expression that uses special symbols to match multiple patterns, then `re.split()` works in exactly the same manner as `str.split()`, as illustrated in the example that follows (which splits on a single colon):

```
>>> re.split(':', 'str1:str2:str3')
['str1', 'str2', 'str3']
```

That's a simple example. What if we have a more complex example, such as a simple parser for a Web site like Google or Yahoo! Maps? Users can enter city and state, or city plus ZIP code, or all three? This requires more powerful processing than just a plain 'ol string split:

```
>>> import re
>>> DATA = (
...     'Mountain View, CA 94040',
...     'Sunnyvale, CA',
...     'Los Altos, 94023',
...     'Cupertino 95014',
...     'Palo Alto CA',
... )
>>> for datum in DATA:
...     print re.split(', |(?= (?:\d{5}|[A-Z]{2})) ', datum)
...
['Mountain View', 'CA', '94040']
['Sunnyvale', 'CA']
['Los Altos', '94023']
['Cupertino', '95014']
['Palo Alto', 'CA']
```

The preceding regex has a simple component, split on comma-space (", "). The harder part is the last regex, which previews some of the extension notations that you'll learn in the next subsection. In plain English, this is what it says: also split on a single space if that space is immediately followed by five digits (ZIP code) or two capital letters (US state abbreviation). This allows us to keep together city names that have spaces in them.

Naturally, this is just a simplistic regex that could be a starting point for an application that parses location information. It doesn't process (or fails)

lowercase states or their full spellings, street addresses, country codes, ZIP+4 (nine-digit ZIP codes), latitude-longitude, multiple spaces, etc. It's just meant as a simple demonstration of `re.split()` doing something `str.split()` can't do.

As we just demonstrated, you benefit from much more power with a regular expression split; however, remember to always use the best tool for the job. If a string split is good enough, there's no need to bring in the additional complexity and performance impact of regexes.

1.3.14 Extension Notations (?...)

There are a variety of extension notations supported by Python regular expressions. Let's take a look at some of them now and provide some usage examples.

With the `(?iLmsux)` set of options, users can specify one or more flags directly into a regular expression rather than via `compile()` or other `re` module functions. Below are several examples that use `re.I/IGNORECASE`, with the last mixing in `re.M/MULTILINE`:

```
>>> re.findall(r'(?i)yes', 'yes? Yes. YES!!')
['yes', 'Yes', 'YES']
>>> re.findall(r'(?i)th\w+', 'The quickest way is through this
tunnel.')
['The', 'through', 'this']
>>> re.findall(r'(?im)(^th[\w ]+)', """
... This line is the first,
... another line,
... that line, it's the best
... """)
['This line is the first', 'that line']
```

For the previous examples, the case-insensitivity should be fairly straightforward. In the last example, by using "multiline" we can perform the search across multiple lines of the target string rather than treating the entire string as a single entity. Notice that the instances of "the" are skipped because they do not appear at the beginning of their respective lines.

The next pair demonstrates the use of `re.S/DOTALL`. This flag indicates that the dot (`.`) can be used to represent `\n` characters (whereas normally it represents all characters except `\n`):

```
>>> re.findall(r'th.+', '''
... The first line
... the second line
... the third line
```

```
... ''')
['the second line', 'the third line']
>>> re.findall(r'(?s)th.+', '''
... The first line
... the second line
... the third line
... ''')
['the second line\nthe third line\n']
```

The `re.X/VERBOSE` flag is quite interesting; it lets users create more human-readable regular expressions by suppressing whitespace characters within regexes (except those in character classes or those that are backslash-escaped). Furthermore, hash/comment/octothorpe symbols (`#`) can also be used to start a comment, also as long as they're not within a character class backslash-escaped:

```
>>> re.search(r'''(?x)
...     \((\d{3})\) # area code
...     [ ]         # space
...     (\d{3})     # prefix
...     -           # dash
...     (\d{4})     # endpoint number
... ''', '(800) 555-1212').groups()
('800', '555', '1212')
```

The `(?:...)` notation should be fairly popular; with it, you can group parts of a regex, but it does not save them for future retrieval or use. This comes in handy when you don't want superfluous matches that are saved and never used:

```
>>> re.findall(r'http://(?:\w+\.)*(\w+\.com)',
 ...     'http://google.com http://www.google.com http://
 code.google.com')
 ['google.com', 'google.com', 'google.com']
 >>> re.search(r'\((?P<areacode>\d{3})\) (?P<prefix>\d{3})-(?:\d{4})',
 ...     '(800) 555-1212').groupdict()
{'areacode': '800', 'prefix': '555'}
```

You can use the `(?P<name>)` and `(?P=name)` notations together. The former saves matches by using a name identifier rather than using increasing numbers, starting at one and going through *N*, which are then retrieved later by using `\1, \2, ... \N`. You can retrieve them in a similar manner using `\g<name>`:

```
>>> re.sub(r'\((?P<areacode>\d{3})\) (?P<prefix>\d{3})-(?:\d{4})',
 ...     '(\g<areacode>) \g<prefix>-xxxx', '(800) 555-1212')
'(800) 555-xxxx'
```

Using the latter, you can reuse patterns in the same regex without specifying the same pattern again later on in the (same) regex, such as in this example, which presumably lets you validate normalization of phone

numbers. Here are the ugly and compressed versions followed by a good use of `(?x)` to make things (slightly) more readable:

```
>>> bool(re.match(r'\((?P<areacode>\d{3})\) (?P<prefix>\d{3})-
(?P<number>\d{4}) (?P=areacode)-(?P=prefix)-(?P=number)
1(?P=areacode)(?P=prefix)(?P=number)',
...     '(800) 555-1212 800-555-1212 18005551212'))
True
>>> bool(re.match(r'''(?x)
...
...     # match (800) 555-1212, save areacode, prefix, no.
...     \((?P<areacode>\d{3})\)[ ](?P<prefix>\d{3})-(?P<number>\d{4})
...
...     # space
...     [ ]
...
...     # match 800-555-1212
...     (?P=areacode)-(?P=prefix)-(?P=number)
...
...     # space
...     [ ]
...
...     # match 18005551212
...     1(?P=areacode)(?P=prefix)(?P=number)
...
... ''', '(800) 555-1212 800-555-1212 18005551212'))
True
```

You use the `(?=...)` and `(?!...)` notations to perform a lookahead in the target string without actually consuming those characters. The first is the positive lookahead assertion, while the latter is the negative. In the examples that follow, we are only interested in the first names of the persons who have a last name of "van Rossum," and the next example let's us ignore e-mail addresses that begin with "noreply" or "postmaster."

The third snippet is another demonstration of the difference between `findall()` and `finditer()`; we use the latter to build a list of e-mail addresses (in a more memory-friendly way by skipping the creation of the intermediary list that would be thrown away) using the same login names but on a different domain.

```
>>> re.findall(r'\w+(?= van Rossum)',
... '''
...     Guido van Rossum
...     Tim Peters
...     Alex Martelli
...     Just van Rossum
...     Raymond Hettinger
... ''')
['Guido', 'Just']
>>> re.findall(r'(?m)^\s+(?!noreply|postmaster)(\w+)',
```

```
... '''
...     sales@phptr.com
...     postmaster@phptr.com
...     eng@phptr.com
...     noreply@phptr.com
...     admin@phptr.com
... ''')
['sales', 'eng', 'admin']
>>> ['%s@aw.com' % e.group(1) for e in \
re.finditer(r'(?m)^\s+(?!noreply|postmaster)(\w+)',
... '''
...     sales@phptr.com
...     postmaster@phptr.com
...     eng@phptr.com
...     noreply@phptr.com
...     admin@phptr.com
... ''')]
['sales@aw.com', 'eng@aw.com', 'admin@aw.com']
```

The last examples demonstrate the use of conditional regular expression matching. Suppose that we have another specialized alphabet consisting only of the characters 'x' and 'y,' where we only want to restrict the string in such a way that two-letter strings must consist of one character followed by the other. In other words, you can't have both letters be the same; either it's an 'x' followed by a 'y' or vice versa:

```
>>> bool(re.search(r'(?:(x)|y)(?(1)y|x)', 'xy'))
True
>>> bool(re.search(r'(?:(x)|y)(?(1)y|x)', 'xx'))
False
```

1.3.15 Miscellaneous

There can be confusion between regular expression special characters and special ASCII symbols. We can use `\n` to represent a NEWLINE character, but we can use `\d` meaning a regular expression match of a single numeric digit.

Problems can occur if there is a symbol used by both ASCII and regular expressions, so in the following Core Note, we recommend the use of Python raw strings to prevent any problems. One more caution: the `\w` and `\W` alphanumeric character sets are affected by the `re.L`/`LOCALE` and Unicode (`re.U`/`UNICODE`) flags.

CORE NOTE: Using Python raw strings

You might have seen the use of raw strings in some of the previous examples. Regular expressions were a strong motivation for the advent of raw strings. The reason lies in the conflicts between ASCII characters and regular expression special characters. As a special symbol, `\b` represents the ASCII character for backspace, but `\b` is also a regular expression special symbol, meaning "match" on a word boundary. For the regex compiler to see the two characters `\b` as your string and not a (single) backspace, you need to escape the backslash in the string by using another backslash, resulting in `\\b`.

This can get messy, especially if you have a lot of special characters in your string, adding to the confusion. We were introduced to raw strings in the Sequences chapter of *Core Python Programming* or *Core Python Language Fundamentals*, and they can be (and are often) used to help keep regexes looking somewhat manageable. In fact, many Python programmers swear by these and only use raw strings when defining regular expressions.

Here are some examples of differentiating between the backspace `\b` and the regular expression `\b`, with and without raw strings:

```
>>> m = re.match('\bblow', 'blow') # backspace, no match
>>> if m: m.group()
...
>>> m = re.match('\\bblow', 'blow') # escaped \, now it works
>>> if m: m.group()
...
'blow'
>>> m = re.match(r'\bblow', 'blow') # use raw string instead
>>> if m: m.group()
...
'blow'
```

You might have recalled that we had no trouble using `\d` in our regular expressions without using raw strings. That is because there is no ASCII equivalent special character, so the regular expression compiler knew that you meant a decimal digit.

1.4 Some Regex Examples

Let's look at a few examples of some Python regex code that takes us a step closer to something that you would actually use in practice. Take, for example, the output from the POSIX (Unix-flavored systems like Linux, Mac OS X, etc.) `who` command, which lists all the users logged in to a system:

```
$ who
wesley       console      Jun 20 20:33
wesley       pts/9        Jun 22 01:38    (192.168.0.6)
wesley       pts/1        Jun 20 20:33    (:0.0)
wesley       pts/2        Jun 20 20:33    (:0.0)
wesley       pts/4        Jun 20 20:33    (:0.0)
wesley       pts/3        Jun 20 20:33    (:0.0)
wesley       pts/5        Jun 20 20:33    (:0.0)
wesley       pts/6        Jun 20 20:33    (:0.0)
wesley       pts/7        Jun 20 20:33    (:0.0)
wesley       pts/8        Jun 20 20:33    (:0.0)
```

Perhaps we want to save some user login information such as login name, the teletype at which the user logged in, when the user logged in, and from where. Using `str.split()` on the preceding example would not be effective because the spacing is erratic and inconsistent. The other problem is that there is a space between the month, day, and time for the login timestamps. We would probably want to keep these fields together.

You need some way to describe a pattern such as "split on two or more spaces." This is easily done with regular expressions. In no time, we whip up the regex pattern `\s\s+`, which means at least two whitespace characters.

Let's create a program called `rewho.py` that reads the output of the `who` command, presumably saved into a file called `whodata.txt`. Our `rewho.py` script initially looks something like this:

```
import re
f = open('whodata.txt', 'r')
for eachLine in f:
    print re.split(r'\s\s+', eachLine)
f.close()
```

The preceding code also uses raw strings (leading "r" or "R" in front of the opening quotes). The main idea is to avoid translating special string characters like `\n`, which is not a special regex pattern. For regex patterns that do have backslashes, you want them treated verbatim; otherwise, you'd have to double-backslash them to keep them safe.

We will now execute the `who` command, saving the output into `whodata.txt`, and then call `rewho.py` to take a look at the results:

```
$ who > whodata.txt
$ rewho.py
['wesley', 'console', 'Jun 20 20:33\012']
['wesley', 'pts/9', 'Jun 22 01:38\011(192.168.0.6)\012']
['wesley', 'pts/1', 'Jun 20 20:33\011(:0.0)\012']
['wesley', 'pts/2', 'Jun 20 20:33\011(:0.0)\012']
['wesley', 'pts/4', 'Jun 20 20:33\011(:0.0)\012']
['wesley', 'pts/3', 'Jun 20 20:33\011(:0.0)\012']
['wesley', 'pts/5', 'Jun 20 20:33\011(:0.0)\012']
['wesley', 'pts/6', 'Jun 20 20:33\011(:0.0)\012']
['wesley', 'pts/7', 'Jun 20 20:33\011(:0.0)\012']
['wesley', 'pts/8', 'Jun 20 20:33\011(:0.0)\012']
```

It was a good first try, but not quite correct. For one thing, we did not anticipate a single TAB (ASCII \011) as part of the output (which looked like at least two spaces, right?), and perhaps we aren't really keen on saving the \n (ASCII \012), which terminates each line. We are now going to fix those problems as well as improve the overall quality of our application by making a few more changes.

First, we would rather run the `who` command from within the script instead of doing it externally and saving the output to a `whodata.txt` file—doing this repeatedly gets tiring rather quickly. To accomplish invoking another program from within ours, we call upon the `os.popen()` command. Although `os.popen()` has now been made obsolete by the `subprocess` module, it's still simpler to use, and the main point is to illustrate the functionality of `re.split()`.

We get rid of the trailing \ns (with `str.rstrip()`) and add the detection of a single TAB as an additional, alternative `re.split()` delimiter. Example 1-1 presents the final Python 2 version of our `rewho.py` script:

Example 1-1 Split Output of the POSIX `who` Command (`rewho.py`)

This script calls the `who` command and parses the input by splitting up its data along various types of whitespace characters.

```
1  #!/usr/bin/env python
2
3  import os
4  import re
5
6  f = os.popen('who', 'r')
7  for eachLine in f:
8     print re.split(r'\s\s+|\t', eachLine.rstrip())
9  f.close()
```

Example 1-2 presents `rewho3.py`, which is the Python 3 version with an additional twist. The main difference from the Python 2 version is the

3.x

2.5-2.6

`print()` function (vs. a statement). This entire line is italicized to indicate critical Python 2 versus 3 differences. The `with` statement, available as experimental in version 2.5, and official in version 2.6, works with objects built to support it.

Example 1-2 Python 3 Version of `rewho.py` Script (`rewho3.py`)

This Python 3 equivalent of `rewho.py` simply replaces the **`print`** statement with the `print()` function. When using the **`with`** statement (available starting in Python 2.5), keep in mind that the `file` (Python 2) or `io` (Python 3) object's context manager will automatically call `f.close()` for you.

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

import os
import re

with os.popen('who', 'r') as f:
  for eachLine in f:
      print(re.split(r'\s\s+|\t', eachLine.rstrip()))
```

Objects that have context managers implemented for them makes them eligible to be used with **`with`**. For more on the **`with`** statement and context management, please review the "Errors and Exceptions" chapter of *Core Python Programming* or *Core Python Language Fundamentals*. Don't forget for either version (`rewho.py` or `rewho3.py`) that the `who` command is only available on POSIX systems unless you're using Cygwin on a Windows-based computer. For PCs running Microsoft Windows, try `tasklist` instead, but there's an additional tweak you need to do. Keep reading to see a sample execution using *that* command.

Example 1-3 merges together both `rewho.py` and `rewho3.py` into `rewhoU.py`, with the name meaning "`rewho` universal." It runs under both Python 2 and 3 interpreters. We cheat and avoid the use of `print` or `print()` by using a less than fully-featured function that exists in both version 2.x and version 3.x: `distutils.log.warn()`. It's a one-string output function, so if your display is more complex than that, you'll need to merge it all into a single string, and then make the call. To indicate its use within our script, we'll name it `printf()`.

We also roll in the **`with`** statement here, too. This means that you need at least version 2.6 to run this. Well, that's not quite true. We mentioned earlier that it's experimental in version 2.5. This means that you need to include this additional statement if you wish to use it: `from __future__ import with_statement`. If you're still using version 2.4 or older, you have no access to this import and must run code such as that in Example 1-1.

Example 1-3 Universal Version of `rewho.py` Script (`rewhoU.py`)

This script runs under both Python 2 and 3 by proxying out the **`print`** statement and the `print()` function with a cheap substitute. It also includes the **`with`** statement available starting in Python 2.5.

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

import os
from distutils.log import warn as printf
import re

with os.popen('who', 'r') as f:
  for eachLine in f:
      printf(re.split(r'\s\s+|\t', eachLine.strip()))
```

The creation of `rewhoU.py` is one example of how you can create a universal script that helps avoid the need to maintain two versions of the same script for both Python 2 and 3.

Executing any of these scripts with the appropriate interpreter yields the corrected, cleaner output:

```
$ rewho.py
['wesley', 'console', 'Feb 22 14:12']
['wesley', 'ttys000', 'Feb 22 14:18']
['wesley', 'ttys001', 'Feb 22 14:49']
['wesley', 'ttys002', 'Feb 25 00:13', '(192.168.0.20)']
['wesley', 'ttys003', 'Feb 24 23:49', '(192.168.0.20)']
```

Also don't forget that the `re.split()` function also takes the optional `flags` parameter described earlier in this chapter.

A similar exercise can be achieved on Windows-based computers by using the `tasklist` command in place of `who`. Let's take a look at its output on the following page.

```
C:\WINDOWS\system32>tasklist

Image Name                     PID Session Name     Session#    Mem Usage
========================= ====== ================ ======== ============
System Idle Process              0 Console                0         28 K
System                           4 Console                0        240 K
smss.exe                       708 Console                0        420 K
csrss.exe                      764 Console                0      4,876 K
winlogon.exe                   788 Console                0      3,268 K
services.exe                   836 Console                0      3,932 K
. . .
```

As you can see, the output contains different information than that of `who`, but the format is similar, so we can consider our previous solution by performing an `re.split()` on one or more spaces (no TAB issue here).

The problem is that the command name might have a space, and we (should) prefer to keep the entire command name together. The same is true of the memory usage, which is given by "NNN K," where NNN is the amount of memory K designates kilobytes. We want to keep this together, too, so we'd better split off of *at least* one space, right?

Nope, no can do. Notice that the process ID (PID) and Session Name columns are delimited only by a single space. This means that if we split off at least one space, the PID and Session Name would be kept together as a single result. If we copied one of the preceding scripts and call it `retasklist.py`, change the command from `who` to `tasklist /nh` (the `/nh` option suppresses the column headers), and use a regex of `\s\s+`, we get output that looks like this:

```
Z:\corepython\ch1>python retasklist.py
['']
['System Idle Process', '0 Console', '0', '28 K']
['System', '4 Console', '0', '240 K']
['smss.exe', '708 Console', '0', '420 K']
['csrss.exe', '764 Console', '0', '5,028 K']
['winlogon.exe', '788 Console', '0', '3,284 K']
['services.exe', '836 Console', '0', '3,924 K']
. . .
```

We have confirmed that although we've kept the command name and memory usage strings together, we've inadvertently put the PID and Session Name together. We have to discard our use of split and just do a regular expression match. Let's do that and filter out both the Session Name and Number because neither add value to our output. Example 1-4 shows the final version of our Python 2 `retasklist.py`:

Example 1-4 Processing the DOS `tasklist` Command Output (`retasklist.py`)

This script uses a regex and `findall()` to parse the output of the DOS `tasklist` command, displaying only the data that's interesting to us. Porting this script to Python 3 merely requires a switch to the `print()` function.

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

import os
import re

f = os.popen('tasklist /nh', 'r')
for eachLine in f:
    print re.findall(
        r'([\w.]+(?: [\w.]+)*)\s\s+(\d+) \w+\s\s+\d+\s\s+([\d,]+ K)',
        eachLine.rstrip())
f.close()
```

If we run this script, we get our desired (truncated) output:

```
Z:\corepython\ch1>python retasklist.py
[]
[('System Idle Process', '0', '28 K')]
[('System', '4', '240 K')]
[('smss.exe', '708', '420 K')]
[('csrss.exe', '764', '5,016 K')]
[('winlogon.exe', '788', '3,284 K')]
[('services.exe', '836', '3,932 K')]
. . .
```

The meticulous regex used goes through all five columns of the output string, grouping together only those values that matter to us: the command name, its PID, and how much memory it takes. It uses many regex features that we've already read about in this chapter.

Naturally, all of the scripts we've done in this subsection merely display output to the user. In practice, you're likely to be processing this data, instead, saving it to a database, using the output to generate reports to management, etc.

1.5 A Longer Regex Example

We will now run through an in-depth example of the different ways to use regular expressions for string manipulation. The first step is to come up with some code that actually generates random (but not too random) data on which to operate. In Example 1-5, we present `gendata.py`, a script that generates a data set. Although this program simply displays the generated set of strings to standard output, this output could very well be redirected to a test file.

Example 1-5 Data Generator for Regex Exercises (`gendata.py`)

This script creates random data for regular expressions practice and outputs the generated data to the screen. To port this to Python 3, just convert **`print`** to a function, switch from `xrange()` back to `range()`, and change from using `sys.maxint` to `sys.maxsize`.

```
1  #!/usr/bin/env python
2
3  from random import randrange, choice
4  from string import ascii_lowercase as lc
5  from sys import maxint
6  from time import ctime
7
```

(Continued)

Example 1-5 Data Generator for Regex Exercises (`gendata.py`) *(Continued)*

```
8   tlds = ('com', 'edu', 'net', 'org', 'gov')
9
10  for i in xrange(randrange(5, 11)):
11      dtint = randrange(maxint)      # pick date
12      dtstr = ctime(dtint)           # date string
13      llen = randrange(4, 8)         # login is shorter
14      login = ''.join(choice(lc) for j in range(llen))
15      dlen = randrange(llen, 13)   # domain is longer
16      dom = ''.join(choice(lc) for j in xrange(dlen))
17      print '%s::%s@%s.%s::%d-%d-%d' % (dtstr, login,
18          dom, choice(tlds), dtint, llen, dlen)
```

This script generates strings with three fields, delimited by a pair of colons, or a double-colon. The first field is a random (32-bit) integer, which is converted to a date. The next field is a randomly generated e-mail address, and the final field is a set of integers separated by a single dash (-).

Running this code, we get the following output (your mileage will definitely vary) and store it locally as the file `redata.txt`:

```
Thu Jul 22 19:21:19 2004::izsp@dicqdhytvhv.edu::1090549279-4-11
Sun Jul 13 22:42:11 2008::zqeu@dxaibjgkniy.com::1216014131-4-11
Sat May  5 16:36:23 1990::fclihw@alwdbzpsdg.edu::641950583-6-10
Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::1171590364-6-8
Thu Jun 26 19:08:59 2036::ugxfugt@jkhuqhs.net::2098145339-7-7
Tue Apr 10 01:04:45 2012::zkwaq@rpxwmtikse.com::1334045085-5-10
```

You might or might not be able to tell, but the output from this program is ripe for regular expression processing. Following our line-by-line explanation, we will implement several regexes to operate on this data as well as leave plenty for the end-of-chapter exercises.

Line-by-Line Explanation

Lines 1–6

In our example script, we require the use of multiple modules. Although we caution against the use of **`from-import`** because of various reasons (e.g., it's easier to determine where a function comes from, possible local module conflict, etc.), we choose to import only specific attributes from these modules to help you focus on those attributes only as well as shortening each line of code.

Line 8

`tlds` is simply a set of higher-level domain names from which we will randomly pick for each randomly generated e-mail address.

Lines 10–12

Each time `gendata.py` executes, between 5 and 10 lines of output are generated. (Our script uses the `random.randrange()` function for all cases for which we desire a random integer.) For each line, we choose a random integer from the entire possible range (0 to $2^{31} - 1$ [`sys.maxint`]), and then convert that integer to a date by using `time.ctime()`. System time in Python and most POSIX-based computers is based on the number of seconds that have elapsed since the "epoch," which is midnight UTC/GMT on January 1, 1970. If we choose a 32-bit integer, that represents one moment in time from the epoch to the maximum possible time, 2^{32} seconds *after* the epoch.

Lines 13–16

The login name for the fake e-mail address should be between 4 and 7 characters in length (thus `randrange(4, 8)`). To put it together, we randomly choose between 4 and 7 random lowercase letters, concatenating each letter to our string, one at a time. The functionality of the `random.choice()` function is to accept a sequence, and then return a random element of that sequence. In our case, the sequence is the set of all 26 lowercase letters of the alphabet, `string.ascii_lowercase`.

We decided that the main domain name for the fake e-mail address should be no more than 12 characters in length, but at least as long as the login name. Again, we use random lowercase letters to put this name together, letter by letter.

Lines 17–18

The key component of our script puts together all of the random data into the output line. The date string comes first, followed by the delimiter. We then put together the random e-mail address by concatenating the login name, the "`@`" symbol, the domain name, and a randomly chosen high-level domain. After the final double-colon, we put together a random integer string using the original time chosen (for the date string), followed by the lengths of the login and domain names, all separated by a single hyphen.

1.5.1 Matching a String

For the following exercises, create both permissive and restrictive versions of your regexes. We recommend that you test these regexes in a short application that utilizes our sample `redata.txt`, presented earlier (or use your own generated data from running `gendata.py`). You will need to use it again when you do the exercises.

To test the regex before putting it into our little application, we will import the `re` module and assign one sample line from `redata.txt` to a string variable `data`. These statements are constant across both illustrated examples.

```
>>> import re
>>> data = 'Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::1171590364-6-8'
```

In our first example, we will create a regular expression to extract (only) the days of the week from the timestamps from each line of the data file `redata.txt`. We will use the following regex:

"^Mon|^Tue|^Wed|^Thu|^Fri|^Sat|^Sun"

This example requires that the string start with ("^" regex operator) any of the seven strings listed. If we were to "translate" the above regex to English, it would read something like, "the string should start with "Mon," "Tue,". . . , "Sat," or "Sun."

Alternatively, we can bypass all the caret operators with a single caret if we group the day strings like this:

"^(Mon|Tue|Wed|Thu|Fri|Sat|Sun)"

The parentheses around the set of strings mean that one of these strings must be encountered for a match to succeed. This is a "friendlier" version of the original regex we came up with, which did not have the parentheses. Using our modified regex, we can take advantage of the fact that we can access the matched string as a subgroup:

```
>>> patt = '^(Mon|Tue|Wed|Thu|Fri|Sat|Sun)'
>>> m = re.match(patt, data)
>>> m.group()                          # entire match
'Thu'
>>> m.group(1)                         # subgroup 1
'Thu'
>>> m.groups()                         # all subgroups
('Thu',)
```

This feature might not seem as revolutionary as we have made it out to be for this example, but it is definitely advantageous in the next example or anywhere you provide extra data as part of the regex to help in the

string matching process, even though those characters might not be part of the string you are interested in.

Both of the above regexes are the most restrictive, specifically requiring a set number of strings. This might not work well in an internationalization environment, where localized days and abbreviations are used. A looser regex would be: `^\w{3}`. This one requires only that a string begin with three consecutive alphanumeric characters. Again, to translate the regex into English, the caret indicates "begins with," the `\w` means any single alphanumeric character, and the `{3}` means that there should be 3 consecutive copies of the regex which the `{3}` embellishes. Again, if you want grouping, parentheses should be used, such as `^(\w{3})`:

```
>>> patt = '^(\w{3})'
>>> m = re.match(patt, data)
>>> if m is not None: m.group()
...
'Thu'
>>> m.group(1)
'Thu'
```

Note that a regex of `^(\w){3}` is not correct. When the `{3}` was inside the parentheses, the match for three consecutive alphanumeric characters was made first, and then represented as a group. But by moving the `{3}` outside, it is now equivalent to three consecutive single alphanumeric characters:

```
>>> patt = '^(\w){3}'
>>> m = re.match(patt, data)
>>> if m is not None: m.group()
...
'Thu'
>>> m.group(1)
'u'
```

The reason why only the "u" shows up when accessing subgroup 1 is that subgroup 1 was being continually replaced by the next character. In other words, `m.group(1)` started out as "T," then changed to "h," and then finally was replaced by "u." These are three individual (and overlapping) groups of a single alphanumeric character, as opposed to a single group consisting of three consecutive alphanumeric characters.

In our next (and final) example, we will create a regular expression to extract the numeric fields found at the end of each line of `redata.txt`.

1.5.2 Search versus Match... and Greediness, too

Before we create any regexes, however, we realize that these integer data items are at the end of the data strings. This means that we have a choice of using either search or match. Initiating a search makes more sense because we know exactly what we are looking for (a set of three integers), that what we seek is not at the beginning of the string, and that it does not make up the entire string. If we were to perform a match, we would have to create a regex to match the entire line and use subgroups to save the data we are interested in. To illustrate the differences, we will perform a search first, and then do a match to show you that searching is more appropriate.

Because we are looking for three integers delimited by hyphens, we create our regex to indicate as such: `\d+-\d+-\d+`. This regular expression means, "any number of digits (at least one, though) followed by a hyphen, then more digits, another hyphen, and finally, a final set of digits." We test our regex now by using `search()`:

```
>>> patt = '\d+-\d+-\d+'
>>> re.search(patt, data).group()            # entire match
'1171590364-6-8'
```

A match attempt, however, would fail. Why? Because matches start at the beginning of the string, the numeric strings are at the end. We would have to create another regex to match the entire string. We can be lazy, though, by using `.+` to indicate just an arbitrary set of characters followed by what we are really interested in:

```
patt = '.+\d+-\d+-\d+'
>>> re.match(patt, data).group()             # entire match
'Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::1171590364-6-8'
```

This works great, but we really want the number fields at the end, not the entire string, so we have to use parentheses to group what we want:

```
>>> patt = '.+(\d+-\d+-\d+)'
>>> re.match(patt, data).group(1)            # subgroup 1
'4-6-8'
```

What happened? We should have extracted `1171590364-6-8`, not just `4-6-8`. Where is the rest of the first integer? The problem is that regular expressions are inherently greedy. This means that with wildcard patterns, regular expressions are evaluated in left-to-right order and try to "grab" as many characters as possible that match the pattern. In the preceding case, the `.+` grabbed every single character from the beginning of the string, including most of the first integer field that we wanted. The `\d+` needed only

a single digit, so it got "4," whereas the .+ matched everything from the beginning of the string up to that first digit: "Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::117159036," as indicated in Figure 1–2.

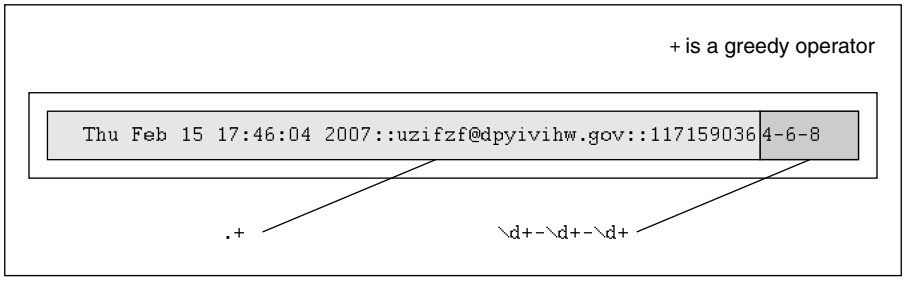

Figure 1-2 Why our match went awry: + is a greedy operator.

One solution is to use the "don't be greedy" operator: ?. You can use this operator after *, +, or ?. It directs the regular expression engine to match as few characters as possible. So if we place a ? after the .+, we obtain the desired result, as illustrated in Figure 1–3.

```
>>> patt = '.+?(\d+-\d+-\d+)'
>>> re.match(patt, data).group(1)            # subgroup 1
'1171590364-6-8'
```

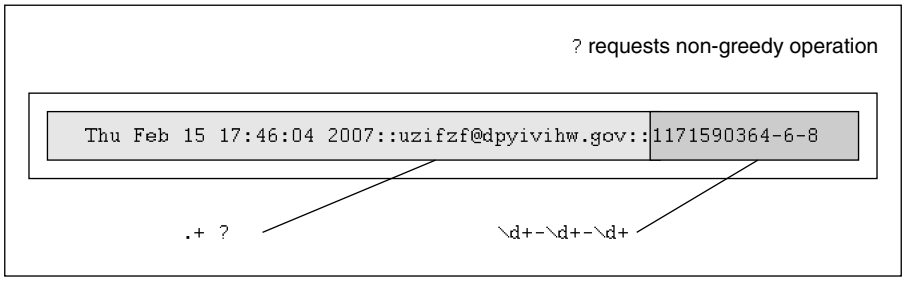

Figure 1-3 Solving the greedy problem: ? requests non-greediness.

Another solution, which is actually easier, is to recognize that "::" is our field separator. You can then just use the regular string `strip('::')` method to get all the parts, and then take another split on the dash with `strip('-')` to obtain the three integers you were originally seeking. Now, we did not choose this solution first because this is how we put the strings together to begin with using `gendata.py`!

One final example: suppose that we want to pull out only the middle integer of the three-integer field. Here is how we would do it (using a search so that we don't have to match the entire string): `-(\d+)-`. Trying out this pattern, we get:

```
>>> patt = '-(\d+)-'
>>> m = re.search(patt, data)
>>> m.group()                          # entire match
'-6-'
>>> m.group(1)                         # subgroup 1
'6'
```

We barely touched upon the power of regular expressions, and in this limited space we have not been able to do them justice. However, we hope that we have given an informative introduction so that you can add this powerful tool to your programming skills. We suggest that you refer to the documentation for more details on how to use regexes with Python. For a more complete immersion into the world of regular expressions, we recommend *Mastering Regular Expressions* by Jeffrey E. F. Friedl.

1.6 Exercises

Regular Expressions. Create regular expressions in Exercises 1-1 to1-12 that:

1-1. Recognize the following strings: "bat," "bit," "but," "hat," "hit," or "hut."
1-2. Match any pair of words separated by a single space, that is, first and last names.
1-3. Match any word and single letter separated by a comma and single space, as in last name, first initial.
1-4. Match the set of all valid Python identifiers.
1-5. Match a street address according to your local format (keep your regex general enough to match any number of street words, including the type designation). For example, American street addresses use the format: 1180 Bordeaux Drive. Make your regex flexible enough to support multi-word street names such as: 3120 De la Cruz Boulevard.
1-6. Match simple Web domain names that begin with "www." and end with a ".com" suffix; for example, www.yahoo.com. Extra Credit: If your regex also supports other high-level domain names, such as .edu, .net, etc. (for example, www.foothill.edu).

1-7. Match the set of the string representations of all Python integers.
1-8. Match the set of the string representations of all Python longs.
1-9. Match the set of the string representations of all Python floats.
1-10. Match the set of the string representations of all Python complex numbers.
1-11. Match the set of all valid e-mail addresses (start with a loose regex, and then try to tighten it as much as you can, yet maintain correct functionality).
1-12. Match the set of all valid Web site addresses (URLs) (start with a loose regex, and then try to tighten it as much as you can, yet maintain correct functionality).

1-13. *`type()`*. The `type()` built-in function returns a type object, which is displayed as the following Pythonic-looking string:

```
>>> type(0)
<type 'int'>
>>> type(.34)
<type 'float'>
>>> type(dir)
<type 'builtin_function_or_method'>
```

Create a regex that would extract the actual type name from the string. Your function should take a string like this `<type 'int'>` and return `int`. (Ditto for all other types, such as `'float'`, `'builtin_function_or_method'`, etc.) Note: You are implementing the value that is stored in the `__name__` attribute for classes and some built-in types.

1-14. *Processing Dates*. In Section 1.2, we gave you the regex pattern that matched the single or double-digit string representations of the months January to September (`0?[1-9]`). Create the regex that represents the remaining three months in the standard calendar.
1-15. *Processing Credit Card Numbers*. Also in Section 1.2, we gave you the regex pattern that matched credit card (CC) numbers (`[0-9]{15,16}`). However, this pattern does not allow for hyphens separating blocks of numbers. Create the regex that allows hyphens, but only in the correct locations. For example, 15-digit CC numbers have a pattern of 4-6-5, indicating four digits-hyphen-six digits-hyphen-five digits; and 16-digit CC numbers have a 4-4-4-4 pattern. Remember to "balloon"

the size of the entire string correctly. Extra Credit: There is a standard algorithm for determining whether a CC number is valid. Write some code that not only recognizes a correctly formatted CC number, but also a valid one.

Playing with `gendata.py`. The next set of Exercises (1-16 through 1-27) deal specifically with the data that is generated by `gendata.py`. Before approaching Exercises 1-17 and 1-18, you might want to do 1-16 and all the regular expressions first.

1-16. Update the code for `gendata.py` so that the data is written directly to `redata.txt` rather than output to the screen.
1-17. Determine how many times each day of the week shows up for any incarnation of `redata.txt`. (Alternatively, you can also count how many times each month of the year was chosen.)
1-18. Ensure that there is no data corruption in `redata.txt` by confirming that the first integer of the integer field matches the timestamp given at the beginning of each output line.

Create Regular Expressions That:

1-19. Extract the complete timestamps from each line.
1-20. Extract the complete e-mail address from each line.
1-21. Extract only the months from the timestamps.
1-22. Extract only the years from the timestamps.
1-23. Extract only the time (HH:MM:SS) from the timestamps.
1-24. Extract only the login and domain names (both the main domain name and the high-level domain together) from the e-mail address.
1-25. Extract only the login and domain names (both the main domain name and the high-level domain) from the e-mail address.
1-26. Replace the e-mail address from each line of data with your e-mail address.
1-27. Extract the months, days, and years from the timestamps and output them in "Mon, Day, Year" format, iterating over each line only once.

Processing Telephone Numbers. For Exercises 1-28 and 1-29, recall the regular expression introduced in Section 1.2, which matched telephone numbers but allowed for an optional area code prefix: `\d{3}-\d{3}-\d{4}`. Update this regular expression so that:

1-28. Area codes (the first set of three-digits and the accompanying hyphen) are optional, that is, your regex should match both 800-555-1212 as well as just 555-1212.

1-29. Either parenthesized or hyphenated area codes are supported, not to mention optional; make your regex match 800-555-1212, 555-1212, and also (800) 555-1212.

Regex Utilities. The final set of exercises make useful utility scripts when processing online data:

1-30. *HTML Generation*. Given a list of links (and optional short description), whether user-provided on command-line, via input from another script, or from a database, generate a Web page (`.html`) that includes all links as hypertext anchors, which upon viewing in a Web browser, allows users to click those links and visit the corresponding site. If the short description is provided, use that as the hypertext instead of the URL.

1-31. *Tweet Scrub*. Sometimes all you want to see is the plain text of a tweet as posted to the Twitter service by users. Create a function that takes a tweet and an optional "meta" flag defaulted `False`, and then returns a string of the scrubbed tweet, removing all the extraneous information, such as an "RT" notation for "retweet", a leading `.`, and all "#hashtags". If the meta flag is `True`, then also return a dict containing the metadata. This can include a key "RT," whose value is a tuple of strings of users who retweeted the message, and/or a key "hashtags" with a tuple of the hashtags. If the values don't exist (empty tuples), then don't even bother creating a key-value entry for them.

1-32. *Amazon Screenscraper*. Create a script that helps you to keep track of your favorite books and how they're doing on Amazon (or any other online bookseller that tracks book rankings). For example, the Amazon link for any book is of the format, http://amazon.com/dp/ISBN (for example, http://amazon.com/dp/0132678209). You can then change the domain name to check out the equivalent rankings on Amazon sites in other countries, such as Germany (.de), France (.fr), Japan (.jp), China (.cn), and the UK (.co.uk). Use regular expressions or a markup parser, such as BeautifulSoup, lxml, or html5lib to parse the ranking, and then let the user pass in a command-line argument that specifies whether the output should be in plain text, perhaps for inclusion in an e-mail body, or formatted in HTML for Web consumption.

CHAPTER 2

Network Programming

So, IPv6. You all know that we are almost out of IPv4 address space. I am a little embarrassed about that because I was the guy who decided that 32-bit was enough for the Internet experiment. My only defense is that that choice was made in 1977, and I thought it was an experiment. The problem is the experiment didn't end, so here we are.

—Vint Cerf, January 2011[1]
(verbally at linux.conf.au conference)

In this chapter...

- Introduction
- What Is Client/Server Architecture?
- Sockets: Communication Endpoints
- Network Programming in Python
- *The `SocketServer` Module
- *Introduction to the Twisted Framework
- Related Modules

1. Dates back to 2004 via http://www.educause.edu/EDUCAUSE+Review/EDUCAUSEReviewMagazineVolume39/MusingsontheInternetPart2/157899

2.1 Introduction

In this section, we will take a brief look at network programming using sockets. But before we delve into that, we will present some background information on network programming, how sockets apply to Python, and then show you how to use some of Python's modules to build networked applications.

2.2 What Is Client/Server Architecture?

What is client/server architecture? It means different things to different people, depending on whom you ask as well as whether you are describing a software or a hardware system. In either case, the premise is simple: the *server*—a piece of hardware or software—provides a "service" that is needed by one or more *clients* (users of the service). Its sole purpose of existence is to wait for (client) requests, respond to those clients (provide the service), and then wait for more requests.

Clients, on the other hand, contact a server for a particular request, send over any necessary data, and then wait for the server to reply, either completing the request or indicating the cause of failure. The server runs indefinitely, continually processing requests; clients make a one-time request for service, receive that service, and thus conclude their transaction. A client might make additional requests at some later time, but these are considered separate transactions.

The most common notion of the client/server architecture today is illustrated in Figure 2-1, which depicts a user or client computer retrieving information from a server across the Internet. Although such a system is indeed an example of a client/server architecture, it isn't the only one. Furthermore, client/server architecture can be applied to computer hardware as well as software.

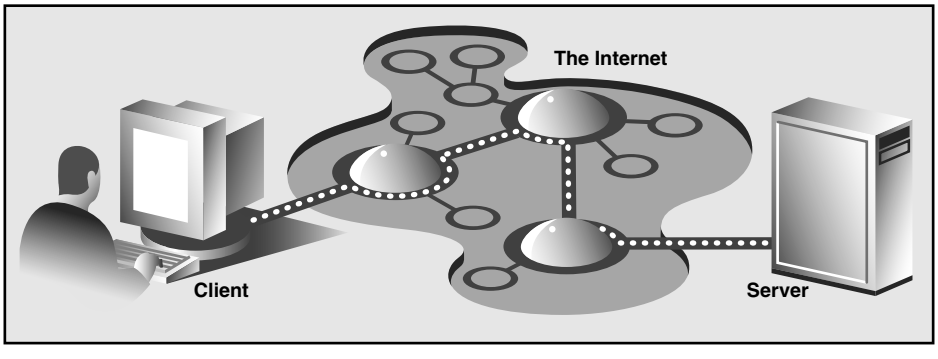

Figure 2-1 Typical conception of a client/server system on the Internet.

2.2.1 Hardware Client/Server Architecture

Print(er) servers are examples of hardware servers. They process incoming print jobs and send them to a printer (or some other printing device) attached to such a system. Such a computer is generally network-accessible and client computers would send it print requests.

Another example of a hardware server is a file server. These are typically computers with large, generalized storage capacity, which is remotely accessible to clients. Client computers mount the disks from the server computer as if the disk itself were on the local computer. One of the most popular network operating systems that support file servers is Sun Microsystems' Network File System (NFS). If you are accessing a networked disk drive and cannot tell whether it is local or on the network, then the client/server system has done its job. The goal is for the user experience to be exactly the same as that of a local disk—the abstraction is normal disk access. It is up to the programmed implementation to make it behave in such a manner.

2.2.2 Software Client/Server Architecture

Software servers also run on a piece of hardware but do not have dedicated peripheral devices as hardware servers do (i.e., printers, disk drives, etc.). The primary services provided by software servers include program execution, data transfer retrieval, aggregation, update, or other types of programmed or data manipulation.

One of the more common software servers today is the Web server. Individuals or companies desiring to run their own Web server will get one or more computers, install the Web pages and or Web applications they wish to provide to users, and then start the Web server. The job of such a server is to accept client requests, send back Web pages to (Web) clients, that is, browsers on users' computers, and then wait for the next client request. These servers are started with the expectation of running forever. Although they do not achieve that goal, they go for as long as possible unless stopped by some external force such as being shut down, either explicitly or catastrophically (due to hardware failure).

Database servers are another kind of software server. They take client requests for either storage or retrieval, act upon that request, and then wait for more business. They are also designed to run forever.

The last type of software server we will discuss are windows servers. These servers can almost be considered hardware servers. They run on a

computer with an attached display, such as a monitor of some sort. Windows clients are actually programs that require a windowing environment in which to execute. These are generally considered graphical user interface (GUI) applications. If they are executed without a window server, meaning, in a text-based environment such as a DOS window or a Unix shell, they are unable to start. Once a windows server is accessible, then things are fine.

Such an environment becomes even more interesting when networking comes into play. The usual display for a windows client is the server on the local computer, but it is possible in some networked windowing environments, such as the X Window system, to choose another computer's window server as a display. In such situations, you can be running a GUI program on one computer, but have it displayed at another!

2.2.3 Bank Tellers as Servers?

One way to imagine how client/server architecture works is to create in your mind the image of a bank teller who neither eats, sleeps, nor rests, serving one customer after another in a line that never seems to end (see Figure 2-2). The line might be long or it might be empty on occasion, but at any given moment, a customer might show up. Of course, such a teller was fantasy years ago, but automated teller machines (ATMs) seem to come close to such a model now.

The teller is, of course, the server that runs in an infinite loop. Each customer is a client with a need that must be addressed. Customers arrive and are handled by the teller in a first-come-first-served manner. Once a transaction has been completed, the client goes away while the server either serves the next customer or sits and waits until one comes along.

Why is all this important? The reason is that this style of execution is how client/server architecture works in a general sense. Now that you have the basic idea, let's adapt it to network programming, which follows the software client/server architecture model.

2.2.4 Client/Server Network Programming

Before a server can respond to client requests, some preliminary setup procedures must be performed to prepare it for the work that lies ahead. A *communication endpoint* is created which allows a server to listen for requests. One can liken our server to a company receptionist or switchboard operator who answers calls on the main corporate line. Once the phone number and equipment are installed and the operator arrives, the service can begin.